TOWARDS SELF-OPTIMIZING FRAMEWORKS FOR COLLABORATIVE SYSTEMS

Sasa Junuzovic

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2010

Prasun Dewan

James Anderson

Nicholas Graham

Saul Greenberg

Jasleen Kaur

Ketan Mayer-Patel

ABSTRACT

Sasa Junuzovic: Towards Self-Optimizing Frameworks for Collaborative Systems
(Under the direction of Prasun Dewan)

Two important performance metrics in collaborative systems are local and remote response times. For certain classes of applications, it is possible to meet response time requirements better than existing systems through a new system without requiring hardware, network, or user-interface changes. This self-optimizing system improves response times by automatically making runtime adjustments to three aspects of a collaborative application.

One of these aspects is the collaboration architecture. Previous work has shown that dynamically switching architectures at runtime can improve response times; however, no previous work performs the switch automatically.

The thesis shows that (a) another important performance parameter is whether multicast or unicast is used to transmit commands, and (b) response times can be noticeably better with multicast than with unicast when transmission costs are high. Traditional architectures, however, support only unicast – a computer that processes input commands must also transmit commands to all other computers. To support multicast, a new bi-architecture model of collaborative systems is introduced in which two separate architectures govern the processing and transmission tasks that each computer must perform.

The thesis also shows that another important performance aspect is the order in which a computer performs these tasks. These tasks can be scheduled sequentially or concurrently on a single-core, or in parallel on multiple cores. As the thesis shows, existing single-core

policies trade-off noticeable improvements in local (remote) for noticeable degradations in remote (local) response times. A new lazy policy for scheduling these tasks on a single-core is introduced that trades-off an unnoticeable degradation in performance of some users for a much larger noticeable improvement in performance of others. The thesis also shows that on multi-core devices, the tasks should always be scheduled on separate cores.

The self-optimizing system adjusts the processing architecture, communication architecture, and scheduling policy based on response time predictions given by a new analytical model. Both the analytical model and the self-optimizing system are validated through simulations and experiments in practical scenarios.

To my mom, dad, and sister. Without you, this is not possible.

ACKNOWLEDGEMENTS

There are many people who have helped me during my graduate student career, and I offer my gratitude to all of them. For brevity, I have kept the list short here: my advisor Prasun Dewan for his support; members of my dissertation committee for their insightful feedback; Kori Inkpen, Yong Rui, Zhengyou Zhang, Rajesh Hegde, and everyone else in Microsoft Research for the opportunity to work with them.

This dissertation was funded in part by a Natural Science and Engineering Research Council of Canada scholarship, a Microsoft Research fellowship, a Scholars for Tomorrow fellowship, and NSF grants ANI 0229998, IIS 0312328, IIS 0712794, and IIS 0810861.

Thank you all for your help and support.
Sasa

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Performance can be the difference between life and death in the world of collaborative systems. To illustrate, consider the following scenario. During candidates' day at UNC, the computer science department invites a number of students for demos of the research projects within the department. Unfortunately, some of these candidates are also invited to the candidates' days at Duke and MIT, both of which happen on the same day as UNC's candidates' day. To give all of the invited students a taste of research at UNC, UNC shares the demo applications using an application-sharing system. Therefore, the students visiting the other schools can remotely try the applications as long as they have with them Internet-connected portable devices, such as laptops, netbooks, and smart phones. The interactivity of the demo is of the utmost importance. The shared application must respond to the operations by the students at Duke and MIT quickly and notify the students of any operations by other users in a timely fashion; otherwise, a student may get bored and quit the demo, which could result in the student not coming to UNC. Even worse, the student may end up going to Duke!

In general, in computer science, the performance of a system is a function of the available resources. If resources are abundant, then the system always performs well. On the other hand, if resources are insufficient, then the system never performs well. These two boundary cases bracket the case in which resources are sufficient but scarce, called the

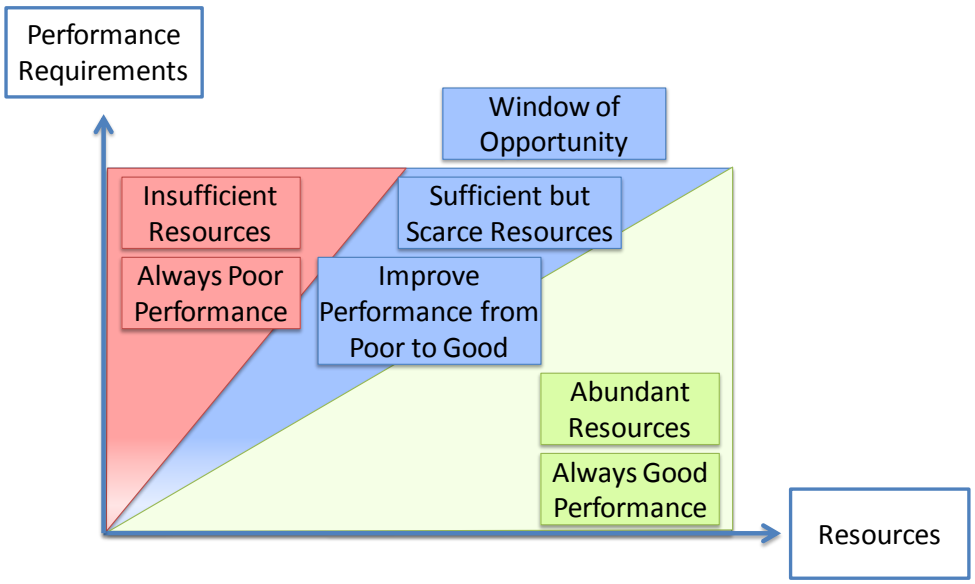

Figure 1-1. Window of opportunity for improving performance.

window of opportunity [55] (Figure 1-1). In the window of opportunity, it is possible for a system to have good performance, although new algorithms and implementations may be necessary to achieve it. To illustrate this idea, consider the multimedia networking discipline and its window of opportunity shown in Figure 1-2. When network bandwidth utilization is

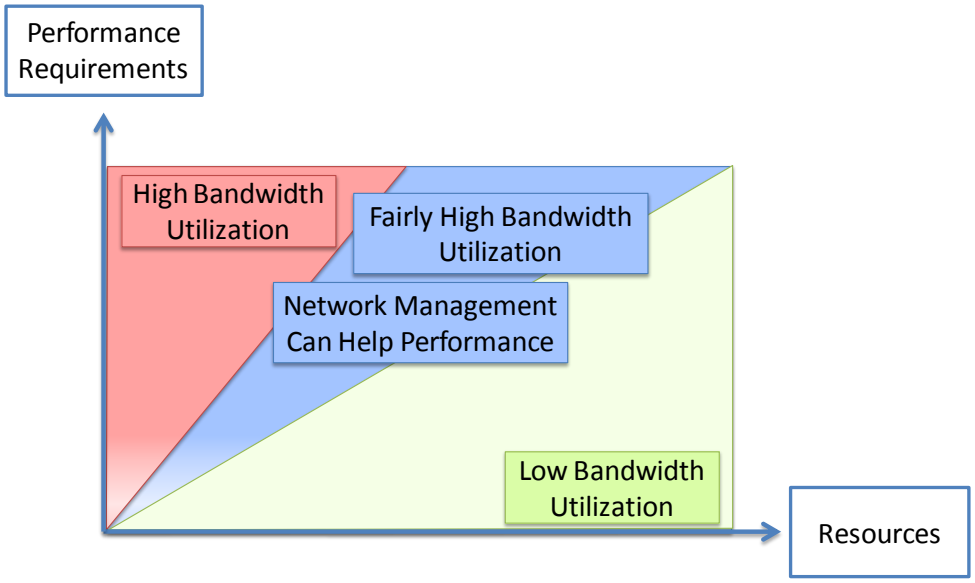

Figure 1-2. Window of opportunity in multimedia networking.

below some threshold, multimedia network systems perform well. When, on the other hand, the network utilization is above some other threshold, these systems suffer from poor performance. And if the maximum bandwidth utilization is between these thresholds, then clever network management techniques, such as jitter buffers and congestion control mechanisms, can make a poorly performing system perform well without requiring hardware or network changes.

As in multimedia networking systems, a window of opportunity also exists for improving the performance of collaborative systems. Our work focuses on improving performance of collaborative systems when resources are sufficient but scarce, which brings us to our thesis:

THESIS

For certain classes of applications, it is possible to meet performance criteria better than existing systems through a new collaborative framework without requiring hardware, network, or user-interface changes.

The thesis raises four questions:

1. How is the performance of a shared application measured?
2. What does it mean for one system to better meet performance criteria than another?
3. What are the parameters of performance?
4. How can the performance of a system be improved?

The answers to the first two questions are relatively brief; they are given in the next two subsections. The remainder of this chapter answers the third question. Finally, the remainder of this thesis is needed to answer to the fourth and final question.

1.1 Performance Metrics

One important performance metric illustrated in the above candidates' day scenario is the local response time for an input command [80], which is defined as the time that elapses from the moment a user enters an input command to the moment that user sees the output for the command. Another, related performance metric illustrated is the remote response time for an input command [33], which is defined as the time that elapses from the moment a user enters an input command to the moment a different user sees the output for the command. Previous work has identified several other performance metrics, such as jitter [50], throughput [42], task completion time [22], and bandwidth consumption [51]. While all of these metrics are important, in this thesis, we focus on response times.

Our Focus:

While response times, jitter, throughput, task completion time, and bandwidth consumption are all important performance metrics, we focus on response times.

1.2 Performance Comparisons

Human-perception studies have shown that users cannot distinguish between response times below a certain threshold. For instance, Shneiderman [80] has shown that users cannot distinguish between local response times below 50ms. Jay et al. [54] complement these

results by showing that remote response times for visual and haptic operations above 50ms and 25ms, respectively, are noticeable. In addition, they found that 50ms increments in remote response times are noticeable for both kinds of operations.

While no study has directly addressed noticeable changes in local response times, one can derive them indirectly from the study of acceptable local response times by Youmans [95]. To find acceptable response time thresholds, Youmans created a system in which users can request system speed-ups if the system appears slow. Each time a speed-up is requested, the response times improve by one eighth. For example, if a user requests a speed-up when the response times are 800ms, the response times decrease to 700ms. Youmans found that users requested system speed-ups until the response times reached the 300-500ms range. By working backwards from this range, it seems that participants requested system speed-ups when response times were in the 343-571ms range. While these results could imply that users can notice response time decreases between 343-300=43ms and 571-500=71ms, they could also imply that users did not notice the effect of the final speed-up and as a result stopped asking for further speed-ups. Therefore, we take another step backwards from the 343-571ms range and find that participants requested system speed-ups when the response times were in the 392-653ms range. These values imply that users can notice decreases in local response times between 392-343=49ms and 653-571=82ms. Based on these results of this study and studies by Shneiderman [80] and Jay et al. [54], we assume that a 50ms change in local or remote response times is noticeable to users.

Assumption:

A 50ms change in local or remote response times is noticeable to users.

Using the noticeable response time thresholds, we can compare the performances of two systems. A simplistic approach is to define the performance of system A to be better than that of system B if the number of users whose response times are noticeably better with system A than with system B is higher than the number of users whose response times are noticeably worse. One issue with this approach is that there is no way to distinguish between local and remote response times, which may be important in some scenarios. To illustrate, consider the candidates' day scenario in which the professor is demoing an application to students at Duke and MIT. Since the professor is familiar with the application, the professor knows in advance what the result of performing a particular command will be. On the other hand, the students cannot anticipate exactly what will happen when a command is entered. In this case, it is more important for the students to see result of a professor's command quickly than it is for the professor because the professor knows what the result will be while the students do not. In other words, the remote response times may be more important than remote response times. Therefore, if the remote response times are better with system B than with system A, then system B is better than system A even if system A is better than system B according to the simplistic definition.

The problem with the simplistic definition arises because the response times are inherently partially ordered and external criteria must be used to create a total order. As our example shows, one useful external criterion is the users who input commands. Another useful criterion may be the identity of the users. For instance, if in our example, the professor is interested in recruiting a particular student more so than the other students, the remote response times of that student may be more critical than those of other students. In general,

infinitely many external criteria exist. To make the thesis tractable, we consider only two: the list of inputting users and the identities of the collaborators.

The exact application of the external criteria depends on the users' response time requirements. Therefore, what is needed is a user-defined function that accepts as parameters the response times of systems, the list of inputting users, and the identities of all users, and returns a total performance order of the systems. Therefore, the total performance order function is an expression of the users' response time requirements. Given such a function, we define the performance of system A to be better than that of system B if the function ranks A as better than B.

Definition:

We say that system A better meets users' response time requirements than system B if system A is given a higher rank than system B by a user-defined total performance order function expressing the requirements.

1.3 Performance Factors

In order to improve performance, we first need to identify the parameters that impact performance. Previous work has shown that the collaboration architecture is a parameter [21][27][42]. Our thesis is that multicast and scheduling are also performance parameters. In this section, we derive six sub-theses regarding these three parameters. The proofs of the sub-theses, when combined, serve as a proof of the main thesis.

1.3.1 Collaboration Architecture

One factor that impacts response times is the collaboration architecture used by the system [27]. A collaboration architecture defines the logical system components, their physical distribution, and the interaction between them. A well-known collaboration architecture model is Dewan's general model [27], which represents an application as a stack of N layered components as shown in Figure 1-3. Each component in the stack provides an abstraction to and services requests from the component at the next lower layer. To share an application, a layer is selected to be logically shared. The part of the application contained in shared (non-shared) layers is called the program (user-interface) component. A program component may perform user-interface related tasks. For example, in a shared window system, all layers above the window system, such as the toolkit layer, are part of the program component. These two terms denote the fact that the shared (non-shared) layers are closer to the program (user-interface). The program component manages the object that is shared by all of the users of the application. The user-interface component allows interaction with the

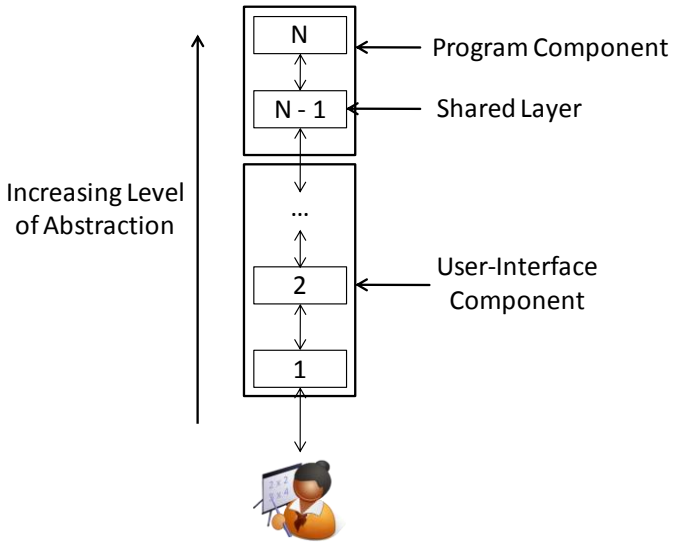

Figure 1-3. Program and user-interface component definition.

Table 1-1. Systems employing different collaborative architectures.

Mapping	System
Centralized	NetMeeting Application Sharing
	Live Meeting
	Webex Application Sharing
	VNC
	Google Wave
Replicated	NetMeeting Whiteboard
	Grove PowerPoint
	Webex PowerPoint
	GroupKit [21]

shared object by manipulating state that is not shared by the users. A separate user-interface component runs on each user's machine. The program component, on the other hand, may or may not be replicated. Regardless of the number of program component replicas, each user-interface component must be mapped to a program component to which it sends input commands and from which it receives outputs [21]. To keep the program component replicas synchronized in this many-to-one mapping of user-interfaces to program components, each program component must send input commands it receives from the user-interfaces mapped to it to all of the other program components.

Three popular mappings have been used in the past: centralized, replicated, and hybrid (Table 1-1). The centralized mapping maps all of the user-interface components to a single program component. The replicated mapping maps each user-interface to its local program component. All other mappings are hybrid mappings. To illustrate the three kinds of mappings, consider the above candidates' day scenario in which six students, three from Duke and three from MIT, are using smart phones to remotely join the application-sharing session. The professor demoing the application is also in the session and is using a desktop computer. A centralized mapping in which the program component on the professor's

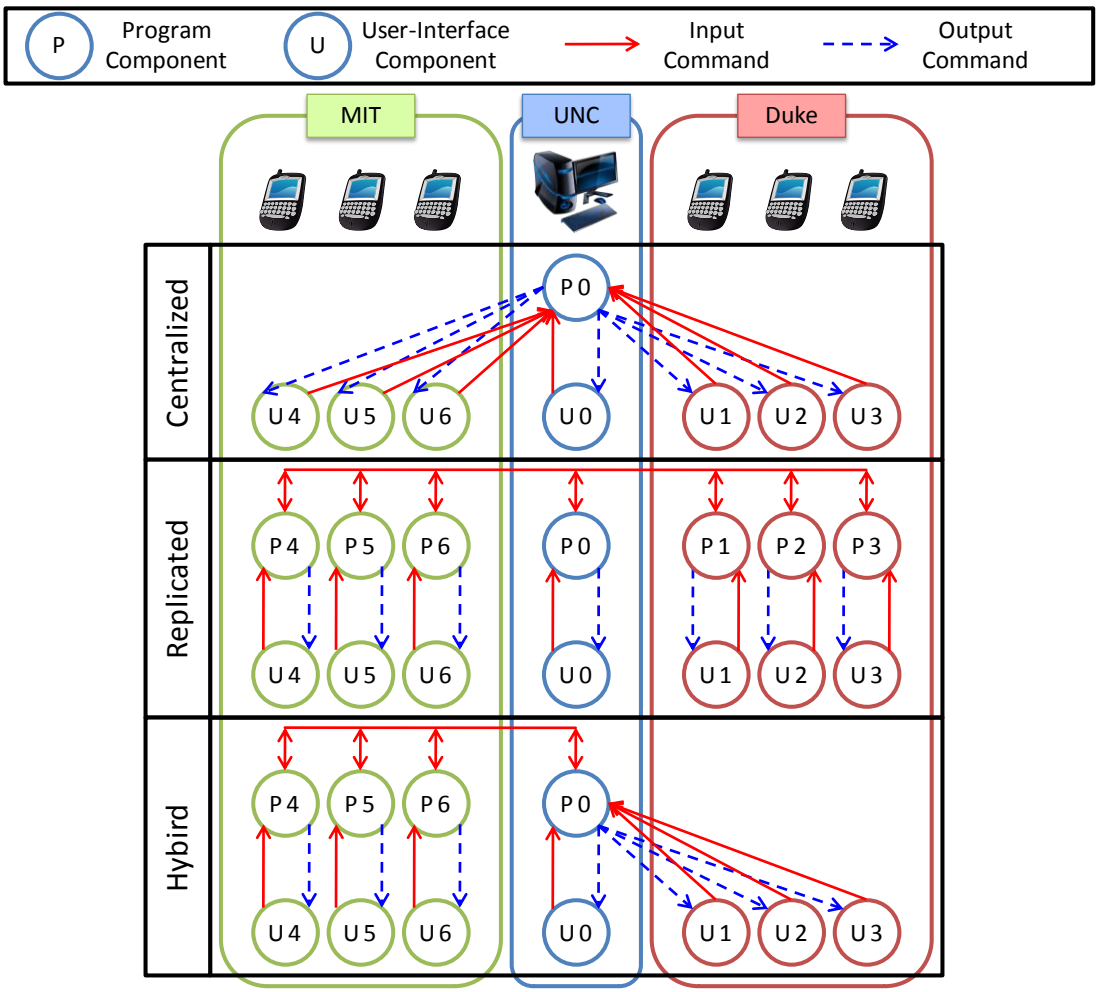

Figure 1-4. Centralized, replicated, and hybrid mappings.

computer is used is shown in Figure 1-4 (top). The replicated mapping is shown in Figure 1-4 (middle). A hybrid mapping in which the user-interfaces on the Duke smart phones are mapped to the program component on the UNC desktop and all of the other user-interfaces are mapped to their local program components is shown in Figure 1-4 (bottom). This thesis focuses on the centralized and replicated mappings. As the figure shows, the hybrid mapping consists of a mix of centralized and replicated mappings. Hence our centralized and

Our Focus:

We focus on the centralized and replicated mapping cases.

replicated results can be extended to cover the hybrid mapping case. We defer those arguments until the discussions chapter.

The choice of mapping can impact the interactivity of the shared application. Suppose that the professor is demonstrating an exceptionally good but computationally expensive Checkers AI and is inviting the students at Duke and MIT to challenge it. What mapping should be used to provide good local response times to Checkers moves entered by the students? As UNC and Duke are near each other, the network latency between them is low. Suppose that the network latency between UNC and MIT is also low. Assume that the professor's desktop is much more powerful than the smart phones. Since the AI algorithm is computationally heavy, a centralized mapping (Figure 1-4 (top)) in which the UNC desktop runs the program component may offer the best local response times. The reason is that it pays for the smart phones to incur the round-trip costs between them and UNC for using the desktop as a high-speed computation server. If the network latencies were high, then the high round-trip times between the smart phones and the desktop would annul the benefit of using the desktop as a high-speed server. In this case, the replicated mapping shown (Figure 1-4 (middle)) could give optimal local response times.

In general, there are infinitely many collaboration scenarios and choosing the architecture that best satisfies the users' response time requirements on a case-by-case basis is a difficult task. This leads us to our first sub-thesis:

SUB-THESIS I

It is possible to develop a system that automatically switches to the architecture that satisfies any user-specified response time criteria better than existing approaches.

1.3.2 Unicast and Multicast

In the centralized and replicated architectures, a computer running the program component is responsible for delivering input commands or outputs to multiple other computers: in the centralized architecture, it sends outputs to the user-interfaces running on all of the other computers, and in the replicated architecture, it sends input commands to the program components running on all of the other computers. One question left unanswered is whether the input commands and outputs are unicast or multicast.

Previous work in operating systems and networking has shown that multicast can reduce the worst case remote response times in content-streaming systems [18][56]. The remote response times are improved by 1) minimizing the maximum sum of transmission delays and network latencies on a path from the source to any receiver (maximum end-to-end latency) and 2) respecting the bandwidth capabilities of the devices. In collaboration systems, optimizing these conditions can actually increase remote response times. To illustrate, consider again the candidates' day scenario in which the professor is demonstrating the Checkers AI to the three students at Duke and three students at MIT. Suppose that the students at MIT have invited another student at MIT to remotely join the demo. As a result, there are now four remote participants from MIT and three from Duke. The new student is using a laptop which is as powerful as the professor's desktop. All of the other students are still using smart phones. The smart phones are not only less powerful than the laptop and the desktop but also have slower network connections than the laptop and the desktop. Assume that the centralized architecture is used, in which the professor's desktop is running the program component. Finally, suppose that the network latencies are low.

With unicast, the professor's desktop must transmit commands to all of the students' devices, as shown in Figure 1-5 (top). The figure shows the transmission of an output for an input command entered by the professor. Therefore, the student to whose device the professor's desktop transmits last has the highest remote response time. The student's response time includes the time the desktop requires to transmit the output to all of the other devices first.

An example of a multicast overlay that optimizes end-to-end latency and respects bandwidth capabilities, which are the two traditionally optimized conditions, is shown in Figure 1-5 (bottom). In this multicast overlay, the desktop first sends the output to the laptop at MIT, which then forwards the output to the smart phones at MIT. As Figure 1-5 (bottom) shows, multicast divides the transmission task between the desktop and the laptop and, hence, effectively parallelizes it. The net effect is that once the laptop receives the output, the multicast transmission speed becomes twice that of unicast. Since the output is transmitted to six of the seven destinations at the multicast speed, the maximum multicast remote response times will be approximately half of the maximum unicast remote response time. It is not exactly half because before multicast speed is achieved, the desktop must first transmit the output to the laptop.

The construction of this multicast overlay ignores the impact of collaboration specific parameters, in particular, the order in which a device processes and forwards the output. To improve the response times to the local user, a device may first process the output and then forward it. In this case, the multicast remote response time to the smart phones at MIT

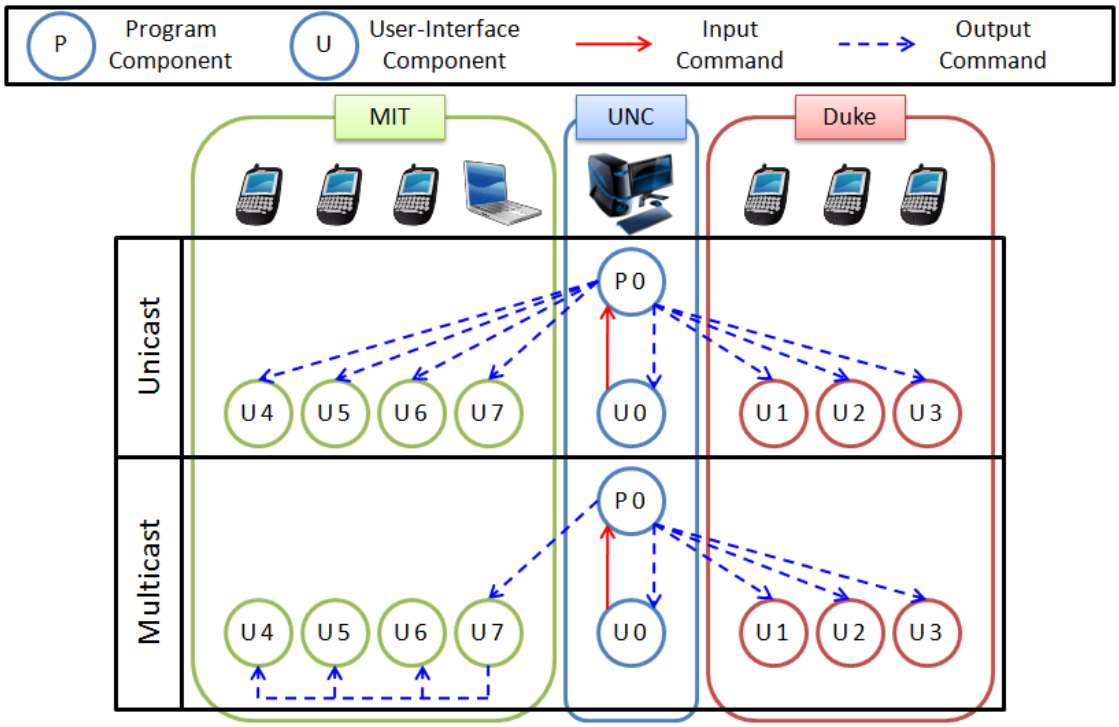

Figure 1-5. Unicast and multicast overlays.

contain both the times that the desktop and the laptop require to process the output. On the other hand, the unicast remote response times do not contain the time the laptop requires to process the output. Hence, if the time the laptop requires to process the output is high, then the multicast overlay may actually increase remote response times compared to unicast.

As Figure 1-5 (bottom) illustrates, multicast requires the formation of communication overlays. But as discussed above, all of the existing collaboration architectures couple the processing of input commands with the distribution of input commands and outputs. As a result, these architectures do not support the distribution of input commands and outputs along arbitrary paths, which is required for multicast. To support multicast, a new bi-architecture model of collaborative systems is needed, which models collaboration systems as two separate sub-architectures: the processing architecture,

which governs the mapping of user-interface and program components, and the communication architecture, which governs the maintenance of communication overlays.

This leads to our second and third sub-theses:

SUB-THESIS II

The communication architecture impacts response times.

SUB-THESIS III

It is possible to develop a system that automatically switches to the communication architecture that satisfies any user-specified response time criteria better than existing approaches.

Definition:

The bi-architecture collaborative systems model represents a system through two separate sub-architectures: the processing architecture, which governs the mapping of user-interface and program components, and the communication architecture, which governs the maintenance of communication overlays.

1.3.3 Scheduling of Tasks

The systems proposed in the previous two sections automatically maintain the processing and communication architectures. Each of these architectures mandates specific tasks that the users' devices must perform. The processing architecture dictates which computers process input commands in addition to processing outputs, while the

communication architecture dictates the destinations to which a computer transmits commands. One question left unanswered is the relationship between the execution of the processing and transmission tasks, in particular, how these independent tasks are scheduled on the users' devices.

The implementation and evaluation of scheduling schemes for these tasks depend on how many cores are available for scheduling. For example, if two or more cores are available for scheduling, it is possible to carry out processing and transmission tasks in parallel. We will consider both single-core and multi-core scheduling policies.

Single-Core Scheduling Policies

One way of scheduling the processing and transmission tasks on a single core is to execute them sequentially. There are two sequential policies possible in which either the processing or the transmission task is performed first. The process-first policy provides better local response times than the transmit-first policy because, unlike the transmit-first policy, it does not delay the processing of a command until the transmission task completes. When the transmit-first policy is used, the local response time includes, in addition to the time required for the processing task, the time required to perform the transmission task. Comparing the remote response times of the two policies is more complicated. Transmitting first from a source seems to improve the remote response times of the destinations. However, as each destination may also forward commands to other computers, delaying processing of the received command can increase the response time seen by its user. In particular, if the time the destination computer requires for performing the transmission task is higher than the sum of the processing times of the computers on the path from the source to the destination, then the transmit-first policy provides worse response times than the process-first policy. Based

on our experience with state-of-the-art multicast schemes, usually a small number of computers actually forward commands. Moreover, an even smaller number of computers forward commands to many destinations. Therefore, we expect the transmit-first policy to provide better response times than the process-first policy to most destinations when a state-of-the-art multicast scheme is used.

An alternative to sequentially scheduling the processing and transmission tasks is to create separate threads for these tasks and schedule the threads using a round-robin policy. Intuitively, such a concurrent policy may seem to give response times in between those supported by the two sequential policies. In fact, with this policy, it is possible to get local response times that are as bad as those of the transmit-first policy. The reason is that when two tasks are scheduled concurrently, the shorter of the two tasks completes earlier than the longer task. Therefore, the longer task completes at the same time with the concurrent policy as it does with the sequential policy that schedules the longer task second. Thus, if the time the source computer requires to perform the processing task is larger than the time it requires to perform the transmission task, then the concurrent local response time is as bad as the transmit-first local response time. Similarly, the concurrent remote response times can be as bad as the process-first remote response times. Consider a destination computer which does not forward commands to other computers. If each computer on the path from the source to the destination, including the source but excluding the destination, completes the processing task before transmitting to the next computer down the path, then the concurrent remote response time is as bad as the process-first remote response time.

In summary, processing first tends to give the best local response times, transmitting first tends to give the best remote response times, and concurrent execution tends to give

local and remote response times that are in between those supported by the other two policies.

One issue with the previous policies is that there is no way to control the tradeoff between local and remote response times. Controlling the tradeoff is particularly attractive when an unnoticeable increase in one metric can result in a noticeable decrease in the other metric. Research in human-computer interaction has shown certain increases are indeed unnoticeable – users cannot distinguish between local response times below 50ms [80]. Therefore, we designed a new lazy scheduling policy that, like the process-first policy, gives precedence to the processing task, but delays its execution if the resulting increases in response times cannot be noticed by humans. By performing a part of the transmission task while the processing task is delayed, such a policy should be able to improve the remote response times of some users. More importantly, the reduction in remote response times does not result in a noticeable increase in the local response times.

Definition:

The lazy scheduling policy gives precedence to the processing task but delays its execution if the resulting increases in response times cannot be noticed by humans.

Multi-Core Scheduling Policies

Each single-core policy also has a multi-core equivalent. The multi-core transmit-first policy first uses all available cores for the transmission task. Once the transmission task is completed, the policy assigns a core for the processing task. The multi-core process-first policy initially uses one core to perform the processing task and the remaining core for the transmission task. Once the processing task completes, all cores perform the remainder of the

transmission task. The multi-core concurrent policy runs the two tasks in parallel, using one core for the processing task and the rest of the available cores for the transmission task. Finally, the multi-core lazy policy initially uses all of the available cores for the transmission task. Once the processing task cannot be delayed further without the user noticing the delay, the policy assigns a core to perform the processing task and uses the remaining cores for the transmission task.

The above multi-core policies do not account for the case when the processing task is parallelizable. A parallelizable processing task is an attractive idea for the lazy policy. The reason is that by spreading the total processing time across multiple cores, the policy can delay the start of the processing task longer without the users noticing the delay. However, the above multi-core policies parallelize only the transmission task. The reason is that the nature of the task makes it easy to do so. For example, two cores can transmit to two different destinations at the same time. From a general framework perspective, the processing task is opaque – hence, the policies cannot parallelize it explicitly. Nevertheless, all of the above policies can be modified to handle a parallel processing task. With the transmit-first (process-first) policy, all of the cores can be made available to the processing (transmission) task once the transmission (processing) task completes. In the concurrent policy, the cores can be evenly shared by the processing and transmission tasks. Finally, in the lazy policy, once the processing task cannot be delayed any longer without the user noticing, all of the cores can be made available to the processing task. To actually delay the processing task longer because it is parallelizable, the lazy policy will require a processing task time that takes into account the number of available cores.

We evaluate the impact on response times of the process-first, transmit-first, concurrent, and lazy scheduling policies for both the single-core and multi-core cases. This leads to our fourth and fifth sub-theses:

SUB-THESIS IV

The scheduling policy used for executing the processing and transmission tasks impacts response times.

SUB-THESIS V

It is possible to develop a system that automatically switches to the scheduling policy that satisfies any user-specified response time criteria better than existing systems.

1.3.4 Analytical Model

So far, we have proposed a framework that can automatically switch to the processing architecture, communication architecture, and scheduling policy that best meet users' response time requirements. In order to decide when to make a switch, the framework must invoke the user-defined total performance order function. As described earlier, that function takes as parameters the response times of the systems which it is supposed to order, the list of inputting users, and the identities of the users. Each combination of the processing architecture, communication architecture, and the scheduling policy defines a system. Therefore, to invoke the total order function, our framework must first be able to predict the response times for any combination of these factors. Therefore, what is needed is a model

that can analytically evaluate the impact on performance for any of these combinations, which leads to our sixth and final sub-thesis:

SUB-THESIS VI

It is possible to develop a model that analytically evaluates the impact on response times of different processing architectures, communication architectures, and scheduling policies to the degree necessary to automate their maintenance.

In order to apply the model, we need to develop algorithms that can perform runtime measurements and predictions of the values of collaboration conditions that impact response times. These algorithms can be based on the mechanisms used for similar purposes in other computer science disciplines. For example, in multi-processor systems, an algorithm is used to estimate the resource requirements of a job before the job is scheduled on a particular processor. This algorithm can serve as a basis for estimating the amount of work required to process an input command on a particular computer in the collaboration. Moreover, in server-farm systems, load balancing algorithms continuously estimate server loads so that when a request arrives, it is forwarded to the server with the lowest load. This algorithm can be reused to estimate the non-collaboration load on a computer. The system load and input command work estimates can be combined to get an estimate of how much time a computer requires to process an input command.

The parameter estimation algorithms described will have to extend to the multi-core case. In particular, the algorithms will have to consider the fact that even though multiple cores can transmit to different destinations in parallel, the device typically has only one network connection. Hence, all of the commands are actually transmitted serially on the

physical link. Thus, using N cores to perform the transmission task may not actually result in a factor of N reduction in the total transmission time.

New algorithms for measuring collaboration-specific parameters will also need to be developed. For example, think time, which is the amount of time a user thinks for about the last output before entering the next input command, can have an impact on performance: low think times can result in queuing and temporary system overload as new input commands arrive before the system has a chance to process those already in the system. Accurate think time predictions are therefore a key element for providing optimal performance. For example, they can help with preemptive architecture transitions if the current architecture is expected to be overloaded soon.

1.4 Scope

We have given one collaborative scenario so far, namely, the Checkers scenario in our example, but we have not qualitatively described the types of scenarios we consider. In this section, we present three driving problems that motivated the thesis and outline the design space of applications that the thesis considers. We carve the design space by specifying points along three dimensions of collaborative applications: program component type, user interface type, and collaborative functionality. As new points in the application design space are added to the thesis scope, we relate them to the applications in our three driving problems.

1.4.1 Driving Problems

In general, collaborative scenarios can be classified along the time and place design space [33]. The thesis focuses on the same-time different-place part of the design space. Three driving problems from this part of the design space motivated the thesis:

1. A distributed PowerPoint presentation
2. A collaborative Checkers computer game
3. Instant messaging

These three problems are important examples of real collaborative scenarios. Distributed presentations are becoming common, instant messaging is pervasive, and collaborative games, such as checkers, chess, and online poker, are extremely popular. In fact, by itself, distributed PowerPoint presentations are an important scenario as an entire industry has been created around it. Microsoft LiveMeeting and Webex are just two of many commercial applications that allow users to give presentations to distributed audiences. The company whose product offers the best performance is going to have an advantage over all of the other companies.

In addition to being important practical scenarios, a nice feature of the driving problems behind our work is that we have available to us a collaborative Checkers game, an instant messaging application, and a way to programmatically interact with PowerPoint. Moreover, we have been able to integrate these applications with our proposed systems, and hence we can perform experiments with the applications to evaluate the systems.

While the driving problems are based on only three collaborative applications, the thesis applies to a much wider range of applications. In the next three sub-sections, we outline the parts of the design space of collaborative applications to which our work applies.

Scope:

Three driving problems that motivated the thesis are 1) a distributed PowerPoint presentation, 2) a collaborative Checkers computer game, and 3) instant messaging.

The discussion focuses on three orthogonal dimensions in the application design space. The first two are architecture related and the last one is implementation related.

1.4.2 Program Component Type

As mentioned above, collaboration architectures model a shared application as a set of layers as shown in Figure 1-3. To share the application, a layer is selected to be shared. The shared layer and the layers above it, which are also shared, form the program component. The layers below the shared layer, which are not shared, form the user-interface component. Here we focus on the program component. In the next sub-section, we focus on the user-interface component.

The program component manages the object that is shared by all of the users of the application. Program components can be classified by whether or not they process computationally heavy input commands and whether or not they distribute large amounts of data. Based on these distinctions, four types of program components can be derived: logic-centric: processes computationally heavy input commands; data-centric: distributes large amounts of data; logic-and-data-centric: both processes computationally heavy input commands and distributes large amounts of data; and stateless: neither processes computationally heavy input commands nor distributes large amounts of data.

The type of program component affects the degree to which changing the processing architecture, communication architecture, and scheduling policies will impact response times.

For example, in a logic-centric application, the choice of processing architecture matters more than the choice of communication architecture or scheduling policy. In a data-centric application, the communication architecture matters the most. And in a logic-and-data centric application, not only do the processing and communication architecture matter but so does the scheduling policy. Finally, in stateless applications, the potential performance improvements that can be achieved by automating these three aspects of the application are lower than in the other cases. Therefore, it is important to consider applications from each of these categories when investigating performance.

The thesis considers all four types of program components from a theoretical perspective. In particular, the analytical model proposed in sub-thesis VI applies to all four program component types. From an experimental perspective, which is used to validate the theoretical analysis, the types of program components considered are determined by the applications in our driving problems. As a result, the thesis considers three of the four types of program components from an experimental perspective: instant messaging, which is stateless; Checkers, which is logic-centric; and PowerPoint, which is data-centric.

Scope:

We theoretically analyze response times for applications with all four program component types, and we experimentally validate the theoretical analysis for three of the four program component types: instant messaging, which is stateless; checkers, which is logic-centric; and PowerPoint, which is data- centric.

1.4.3 User Interface Type

While the program component manages the object that is shared by all of the users of the application, the user-interface component allows interaction with the shared object by manipulating state that is not shared by the users. Not all user commands need to be sent from the user-interface to the program component, that is, not all commands need to be shared. In particular, if a user command interacts with only the state on the local user's device that is not shared with other users, then there is no reason to distribute the command to other users. The number and frequency of non-shared commands determines the thickness of the user-interface [42].

Ideally, the analysis of response times should account for both the non-shared and shared commands. After all, non-shared commands should have good local response times. Moreover, these commands can impact the response times of shared commands and vice versa. To correctly analyze the interplay between the response times of these two types of commands, we need a model of the event flow of the non-shared commands. This event flow is application dependent. Therefore, we need a separate model for each application. As there are infinitely many applications, creating a model for each one of them is beyond the scope of a single work. In order to make the thesis tractable, the analysis presented does not account for non-shared commands.

In the applications that we consider, non-shared commands either do not exist or their costs are negligible. For example, in PowerPoint and Checkers such commands do not exist. In instant messaging, such commands may exist if incremental sharing is not supported. In this case, when a user types a new character, a non-shared command is issued that updates the user's display by showing the character. The cost of displaying text as it is being typed is

negligible on any device in use today. When incremental sharing is supported, as in Google Wave [39], then the typing commands are actually shared commands because the typed text is displayed incrementally to all collaborators.

Of course, many popular applications, such as some massively multiplayer online games [42], have thick user-interface components in which non-shared commands have high processing costs. We return to the potential impact on response times of these commands in the discussions chapter.

Scope:

We consider only the response times of shared commands, which interact with the shared state, in isolation of non-shared commands, which interact with only the local state. The impact of non-shared commands on the response times shared commands are not analyzed.

1.4.4 Collaborative Functionality

In addition to classifying collaborative applications according to the type of program and user-interface components they have, we can also classify these applications by the type of collaborative functionality they provide. Some functionality, which we refer to as mandatory functionality, are common to all collaborative applications. For instance, the mandatory functions required to share an object are the processing of user commands invoked on the object and distributing the results of these commands to all users. In addition to the mandatory functions, there are optional collaborative functions that an application can provide which can increase response times [27]. Some of these functions, such as concurrency control, access control, merging (i.e. consistency maintenance), and awareness,

can increase response times. Concurrency control impacts performance because it can delay the execution of an action until it determines that the action does not conflict with other actions. Access control can increase response times because before the program component executes commands from a user, it must first verify that the commands are authorized. Merging algorithms impact performance because the merge result must be calculated when a command is merged with previous commands, which effectively increases the processing cost, and hence, the response time of the command. Finally, awareness mechanisms can increase response times as they increase the amount of data that has to be communicated between users' machines, which can impact the transmission costs of the user commands.

Ideally, analyzing the impact on response times of changing the processing architecture, communication architecture, and scheduling policies should account for all of collaborative functions that impact response times. However, each of these functions has a large design space. Moreover, each point in a function's design space can have several implementations. The matter is further complicated by the fact the impact depends on the type of communication and collaboration architecture that is used [42]. To make the thesis tractable, we consider only the mandatory functions. The analysis can later be extended to support some of the optional functions.

There are many applications that do not implement any of the optional functions. For instance, with the exception of the telepointers awareness mechanism in our Checkers application, the PowerPoint, Checkers, and instant messaging applications we consider do not provide these functionalities. Moreover, in many collaborative scenarios, these functions are not necessary. For example, in the PowerPoint scenarios we analyzed, there was only a single presenter. Hence, conflicting actions never occurred. In the Checkers scenarios we

analyzed, the users employed a social-protocol to prevent entering conflict commands. Finally, in instant messaging, simultaneous actions can occur, but no merging of commands is done. The result is that the local message histories for two users may be out of order. Thus, the users may have an inconsistent view of the chat history. Fortunately, in this and some other cases [43], the users do not care that their views are inconsistent.

The lack of optional collaborative functions in the applications we consider does not mean that these functions are not important. On the contrary, these mechanisms are important in many applications. For example, concurrency control and merging are important in massively multiplayer online games [42] and multi-user editors [7][35]. Moreover, numerous user studies have shown the benefits of having awareness mechanisms [29][48][75]. We will return to the discussion regarding the impact on response times of these and other optional functions in the discussion chapter.

Scope:

We consider only the response times of mandatory commands in isolation of commands created by optional collaborative functions, such as concurrency control, access control, merging, and awareness.

1.4.5 Assumptions

So far, we have outlined the scope of our thesis by pointing out the points in the design space of collaborative applications that the thesis considers. For the considered applications, the thesis makes one additional implementation assumption:

- When an idle computer receives a command, it immediately begins performing the tasks for the command. In particular, there are no delays added

in performing these tasks, such as the delays necessary for atomic broadcast or causal consistency.

We make the assumption to keep response times low. As mentioned above, users can notice 50ms increments in response time and hence the processing of the commands cannot be delayed for long. This assumption can lead to several problems in practical scenarios. We will discuss these issues in the discussions chapter.

Assumption:

When an idle computer receives a command, it immediately begins performing the tasks for the command. In particular, the computer does not delay these tasks to accommodate atomic broadcast or causal consistency requirements.

In addition, the thesis makes two user assumptions:

- Users interactively discuss each other's actions.
- Users communicate verbally through a pre-established channel, such as a telephone.

The user-related assumptions are consistent with our driving problems. The interactive discussion assumption is satisfied by definition in a distributed PowerPoint presentation as the presenter is discussing the presentation while the presentation is being advanced. Moreover, other users may discuss among themselves the topic the presenter is currently presenting. The external communication channel assumption holds true in commercial in systems, such as LiveMeeting.

The user-related assumptions dictate the types of scenarios considered by the thesis. Examples of scenarios that do not satisfy these criteria are streaming multimedia applications

that playback pre-recorded audio and video streams, videoconferencing systems without a shared workspace, and shared file repositories and versioning systems. Our criteria also exclude systems which capture workspace interactions as a video stream and then stream the video to all of the other users. While such a system is interesting, it does not satisfy our criteria of allowing all users to interact with the shared workspace. In addition, the criteria exclude adding a large delay to commands before either sending them or when they are received. Such delays can be used to ensure that the changes to the shared state are shown at approximately the same time to all users or that all users have reached a consistent state when the inputting user changes. Unfortunately, delaying the display of the result to the inputting user may noticeably increase local response times. Thus, the delay in displaying the result should only be done on the receiver side. In this case, the user who performs an action will see the result of the action sooner than the other users. In fact the user may perform several other actions before the action is seen by others. As a result, these delays prevent other users from reacting in a timely fashion to the initial action. Timely reactions can be extremely important. Consider an online Checkers game that is being followed by a large audience. When the players make a good move against the computer, the audience will want to congratulate them by sending them a happy face or thumbs up in an instant message. If the

Scope:

We focus on collaboration scenarios in which users interactively discuss each other's actions and communicate verbally through a pre-established channel, such as a telephone. Moreover, we consider scenarios in which an idle computer executes tasks for a command as soon as the command is received.

players receive such feedback much after they had performed the action, the feedback may not make any sense.

1.5 Applicability of Results

This dissertation shows that under certain conditions, specifically when processor and network resources are neither over-stressed nor under-stressed, the self-optimizing system can have clear performance benefits over traditional groupware systems. While we speculate that these conditions occur in actual scenarios, further analysis of network and processor bottlenecks is needed to determine whether they actually occur. At the same time, the thesis clearly states several simplifying assumptions. Moreover, the self-optimizing system is a complex prototype that may have high development costs. Therefore, the question is whether the costs of using the system are worth the benefits in actual scenarios. The simple answer is that the system is a realistic alternative to existing groupware; however, it cannot be applied blindly as it involves a series of tradeoffs and unknowns. We will return to the cost-benefit analysis in the discussion section.

1.6 Summary

In this chapter, we have presented 1) our thesis, 2) its six sub-theses, 3) its scope, and 4) its assumptions. Thus, we can state our thesis statement in full:

Thesis

For certain classes of applications, it is possible to meet user-specified response time requirements better than existing systems through a new collaborative framework without requiring hardware, network, or user-interface changes. In particular,

1. Given the result shown in previous work that the processing architecture impacts response times, it is possible to develop a system that automatically switches to the processing architecture that satisfies any user-specified response time criteria better than existing approaches.
2. In addition to the processing architecture, the choice of communication architecture also impacts response times.
3. It is possible to develop a system that automatically switches to the communication architecture that satisfies any user-specified response time criteria better than existing approaches.
4. Not only do the processing and communication architectures impact response times but so does the scheduling policy used for executing the processing and transmission tasks.
5. It is possible to develop a system that automatically switches to the scheduling policy that satisfies any user-specified response time criteria better than existing approaches.
6. It is possible to develop a model that analytically evaluates the impact on response times of different processing architectures, communication architectures, and scheduling policies to the degree necessary to automate their maintenance.

The analytical performance model proposed in the sixth sub-thesis is an important contribution of its own. First, like analytical models in other computer science fields, it increases our understanding of the subject analyzed. In the case of collaboration architectures, it helps us better understand and compare the event flow and performance of the centralized and replicated architectures with and without multicast using the transmit-first, process-first, concurrent, lazy, and parallel scheduling policies. Moreover, it provides

guidance for users with varying degrees of choice regarding the combination of the processing architecture, communication architecture, and scheduling policy in the collaboration systems they use. It can be used by (a) users of adaptive systems to decide when to reconfigure the systems, (b) users who have a choice of systems with different configurations to choose the system most suited for a particular collaboration mode (defined by the values of the collaboration parameters), and (c) users locked into a specific configuration to decide how to change the hardware and other collaboration parameters to improve performance. The model is also necessary prove the first five sub-theses. As a result, we will develop the model incrementally as we prove the first five sub-theses.

In the first five sub-theses, we consider three performance factors of collaborative systems: 1) processing architecture, 2) communication architecture, and 3) scheduling policies. Systems are proposed that automate the maintenance of each of the three factors. These three factors are independent of each other. If one or two of them are fixed, then those that are not fixed can still be changed. For instance, for any processing architecture, we can use either a unicast or multicast architecture. And for any processing and communication architecture pair, multiple scheduling policies can be used. Therefore, to prove the thesis, it is sufficient to show automating the maintenance of one of these factors in isolation of the others can improve response times. We first show that automating the processing architecture maintenance can improve response times when the communication architecture and scheduling policy are fixed. We then show that when the scheduling policy is fixed, automating the communication architecture maintenance can improve response times for any processing architecture. Finally, we show that automating the scheduling policy selection can improve response times for any combination of processing and communication architectures.

The rest of this thesis is organized as follows. Chapter 2 proves sub-thesis I. Chapter 3 proves sub-theses II and III. Chapter 4 proves sub-theses IV and V. The proof of sub-thesis VI is given incrementally in chapters 2 through 4. Chapter 5 presents related work. Chapter 6 discusses the applicability and limitations of our work. Finally, Chapter 7 presents conclusions and future work.

CHAPTER 2

AUTOMATING PROCESSING ARCHITECTURE MAINTENANCE

2.1 Overview

As mentioned in the introduction chapter, previous work has shown that the choice of the architecture impacts the response times of the shared program. In general, choosing the architecture that provides the best response times is a difficult task. Therefore, Chung [21] developed an analytical model that can predict the architecture that gives the best response times. The model assumes two-user collaborations, a constant cost of processing each input command, zero cost of processing each output command, zero cost of transmitting inputs and outputs, constant think times before each command, and no type-ahead (no type-ahead means that users do not enter a command until they see the output for the previous commands). Chung showed both analytically and experimentally that (a) low network latency favors the centralized architecture and (b) asymmetric processing powers favor the centralized architecture. As these conditions can change dynamically, he developed a system that supports architecture changes at runtime. He also performed experiments showing that when a user with a powerful computer joins the collaboration, it is useful to dynamically centralize the shared program to the new user's computer.

One issue with Chung's system is that the users have to select the architecture to use at start time and decide when to switch architectures at runtime. A system that automatically

maintains the architecture would be useful because it would relieve the users of performing these tasks. In this chapter, we present such a self-optimizing system for small-scale collaboration scenarios, where by small-scale, we mean two or three users.

In the process of creating the self-optimizing system, we extended Chung's work in two ways. First, we present a three-user version of Chung's two-user model that relaxes three of the six assumptions made by the original model. In particular, the new model does not assume a constant cost of processing each input command, zero cost of processing each output command, or constant think time before each input command. The updated model still assumes negligible transmission costs and no type-ahead. Second, we present a system that can automatically gather the parameters of the three-user model and apply the model to decide which architecture should be used. By combining this new system with our own version of Chung's system that can dynamically switch architectures at runtime, we create a self-optimizing system that automates the maintenance of the architecture. Therefore, our system consists of two-sub systems. The first is our version of Chung's system, which we call the sharing sub-system to denote the fact that it is responsible for sharing the application. The second is the new system we develop, which we call the optimization sub-system to denote to the fact that it is responsible for improving response times.

Chapter Scope:

We analyze the impact of the collaboration architecture on response times. We consider small-scale collaboration scenarios involving two or three users in which the cost of transmitting commands is negligible and there is no type-ahead.

Chapter Goals:

We present our self-optimizing system that better meets response time requirements than existing systems by automating the maintenance of the architecture.

Each of the systems of the self-optimizing system raises several issues that must be addressed in order for the system to function correctly. A fundamental issue raised by the sharing sub-system is how it shares an application among the users. The system must somehow intercept users' input commands and the corresponding outputs and send them to the appropriate computers. Ideally, it should do so in a manner transparent to the application. Moreover, the system needs to be configurable so that the same application can be shared using both centralized and replicated architectures.

Another fundamental issue is how the sharing sub-system switches between replicated and centralized architectures dynamically at runtime. Whenever the system switches from a centralized to a replicated architecture, the system must bring the program components on the new master computers up to date; otherwise, these program components may be out of sync with the program component running on the computer which was the master in the centralized architecture. Moreover, if an architecture switch is not performed atomically with respect to user commands, then the shared application may be shared in a manner that is inconsistent with the notion of centralized and replicated architectures. To illustrate, suppose that the system switches from the replicated (centralized) to the centralized (replicated) architecture. Suppose that during the switch, an input (output) command was en-route to the computer that is (was) a slave in the centralized architecture. Therefore, a slave

(master) user in the new architecture will receive an input (output) command from a remote user which is inconsistent with the notion of centralized (replicated) architectures.

A related issue is how the system accommodates late-comers. When a late-comer joins as a master, the system must as above, bring the program components on the late-comer's computer up to date; otherwise, these program components may not be synchronized with the program components on other computers. Similarly, when a late-comer joins as a slave, the system must bring the user-interface component on the late-comer's computer up to date; otherwise, future outputs may not make sense.

Chung's framework has provided solutions to all of these issues. Since our sharing sub-system is a version of Chung framework, for each issue, we will describe Chung's solution. If in our system we use a different approach than Chung, then we will state the difference between ours and Chung's systems. If we do not state a difference, it means that we have reused Chung's solution.

The optimization sub-system raises a different set of issues. One issue is how the system is organized. Implementations in which the system is organized in a client-server or peer-to-peer fashion on the users' machines both seem promising. However, the former can overload the machine on which it is running and hence degrade the response times of the local user, while the latter must reach an agreement among the peer components, which is a version of the distributed consensus problem commonly found in distributed systems.

Another important issue is how the optimization sub-system gathers values of the parameters in the analytical model, which it must do in order to apply the model. In particular, it must measure the input and output processing costs on each user's computer and the network latencies among all of the users' computers.

Once the system gathers the parameter values, it can use the analytical model to predict replicated and centralized architecture response times. The next issue is how it passes the predicted response times to the total order function and then invoke the functionality in the sharing sub-system for switching architectures to change to the architecture returned by the total order function.

We address the issues raised by both the sharing and optimization sub-systems system when we describe them below. The rest of this chapter is organized as follows. We first derive our three-user version of Chung's analytical model. We then describe our version of Chung's system that can dynamically switch architectures at runtime. Following this, we discuss the upgrades to the system necessary to turn it into our self-optimizing system. Then, we describe experiments conducted with the self-optimizing system. Finally, we present discussions and a brief summary.

2.2 Formal Analysis

Developing an analytical model is a complex task. Therefore, like Chung [21], we make certain assumptions in our response time model of collaboration architectures: the collaboration involves at most three users, who we denote as $user_1$, $user_2$, and $user_3$; the transmission costs are negligible; and there is no type-ahead. Because of the second assumption, whenever a computer must both send a command to the other computers and process the command, we assume that it first sends the command and then processes it.

Unlike Chung, we do not assume a constant cost of processing each input command, zero cost of processing each output command, or constant think times. It is important to consider processing times because a computer must process commands before displaying

outputs to the local user. The impact of think times on response times is somewhat less apparent. Think times can impact response times because when think times of commands are low, slower computers may fall behind the faster computers in processing the commands. In particular, when a command reaches slower computers, these computers may not be ready to begin processing the command immediately if they are still busy processing previous commands. The delays in processing the commands increase the response times of the commands.

In all of the small-scale collaborations we logged, the users' think times were high enough that computers were always ready to process the next command whenever it arrived. These collaborations included commands that were generated fairly rapidly, such as those of telepointers. Telepointing actions are typically generated thirty times each second, that is, they have think times of 33ms. Even though these think times appear low, they are much higher than the processing times of telepointing commands, which were 1.5ms on a P3 desktop, the slowest computer we used in our experiments. Thus, the think times are effectively high even for telepointing actions. Therefore, in this chapter, we present the analysis only for the case when think times are high.

Analysis Scope:

In this chapter, we present the analysis only for the case when think times are high because in all of the small-scale collaborations we observed, think times were always high. Separate analysis is needed for the case when think times are low.

We will introduce the remaining parameters of our model as we develop the equations. We start by considering the equations for the local response times in centralized architectures.

2.2.1 Local Response Times in Centralized Architectures

In a centralized architecture, the shared program executes on a computer belonging to one of the collaborators, receiving input from and broadcasting output to all users. The computer running the shared program is called the master while all other computers are called slaves.

In cases where a master user inputs commands, the commands are processed by the local computer. It processes each input command immediately after the local user provides it, computes the output for the command, transmits the output to the slave users, and process the output to display the result to the local user. Let $p_{i,j}^{IN}$ ($p_{i,j}^{OUT}$) denote the time user_j's computer requires to process input (output) command i. Hence, if user_m is the master in a centralized architecture, user_m's local response time is $p_{i,m}^{IN} + p_{i,m}^{OUT}$.

Now, consider the case where a slave user inputs command. The slave user's computer must first transmit the command to the master, which must then travel to the master computer. When the master computer receives the command, it then processes the command without interruption, computes an output message, and transmits the output back to the slave user, which must then travel back to the slave's computer. When the slave's computer receives the output, it processes the output. Let $d(j,k)$ denote the network latency a command experiences as it travels from user_j's to user_k's computer. Therefore, if user_{s1} (user_{s2}) is a slave user in a centralized architecture in which user_m is the master, then user_{s1}'s

($user_{s2}$'s) local response time is $d(s1,m) + p_{i,m}^{IN} + d(m,s1) + p_{i,s1}^{OUT}$ $(d(s2,m) + p_{i,m}^{IN} + d(m,s2) + p_{i,s2}^{OUT})$. Hence, if $user_m$ is the master and $user_{s1}$ and $user_{s2}$ are slaves, we have

$$local_{CENT,i,m}u_m = p_{i,m}^{IN} + p_{i,m}^{OUT} \quad \text{if } user_m \text{ is master}$$

$$local_{CENT,i,s1}u_m = d(s1,m) + p_{i,m}^{IN} + d(m,s1) + p_{i,s1}^{OUT} \quad \text{if } user_m \text{ is master}$$

$$local_{CENT,i,s2}u_m = d(s2,m) + p_{i,m}^{IN} + d(m,s2) + p_{i,s2}^{OUT}$$

We next derive the equations for the local response times in the replicated architecture.

2.2.2 Local Response Times in Replicated Architectures

In the replicated architecture, a separate replica of the program executes on the computer of each user, receiving input from all users and producing output for only the local user. Hence, all computers are masters in the replicated architecture. Thus, $user_j$'s local response time in the replicated case is $p_{i,j}^{IN} + p_{i,j}^{OUT}$. Hence,

$$local_{REP,i,j} = p_{i,j}^{IN} + p_{i,j}^{OUT}$$

2.2.3 Remote Response Times in Centralized Architectures

Recall that a non-inputting user's remote response time for a command is defined as the time that elapses from the moment the command is input to the moment the user sees its output. In centralized architectures, a user's remote response time depends on whether the non-inputting and inputting users are both slaves or if one of them is the master. If the master user provides the input, a slave user will see the output once the input command is processed, the output traverses the network, and the slave user's computer processes the output. Thus, we have

$$remote_{CENT,i,s1}u_m = p_{i,m}^{IN} + d(m,s1) + p_{i,s1}^{OUT} \quad \text{if master } user_m \text{ inputs}$$

$$remote_{CENT,i,s2}u_m = p_{i,m}^{IN} + d(m,s2) + p_{i,s1}^{OUT} \quad \text{if master } user_m \text{ inputs}$$

If a slave user provides the input, the master user will see the output once the input command reaches the master computer and the computer processes it and the corresponding output. The other slave user will see the output once the input command reaches the master computer, the master computer processes and transmits the output to the other slave, the output travels to the other slave's computer, and finally the other slave's computer processes it. Thus,

$$remote_{CENT,i,m}u_m = d(s1,m) + p_{i,m}^{IN} + p_{i,m}^{OUT}$$

$$remote_{CENT,i,s2}u_m = d(s1,m) + p_{i,m}^{IN} + d(m,s2) + p_{i,s2}^{OUT}$$

if slave $user_{s1}$ inputs, and

$$remote_{CENT,i,m}u_m = d(s2,m) + p_{i,m}^{IN} + p_{i,m}^{OUT}$$

$$remote_{CENT,i,s1}u_m = d(s2,m) + p_{i,m}^{IN} + d(m,s1) + p_{i,s2}^{OUT}$$

if slave $user_{s2}$ inputs.

2.2.4 Remote Response Times in Replicated Architectures

In the replicated architecture, the remote response time of a command includes the time the command requires to travel from the inputting user's computer to the receiving user's computer and the time the receiving user's computer requires to process the command and its output. Thus, the $user_k$'s remote response time of command i entered by $user_j$ is given by

$$remote_{REP,i,k} = d(j,k) + p_{i,k}^{IN} + p_{i,k}^{OUT}$$

2.3 Self-Optimizing System Implementation

Our self-optimizing collaboration framework can apply the analytical model to automatically switch the architecture at runtime to improve the performance. As mentioned above, the framework consists of two sub-systems. One sub-system, called the sharing sub-

system, is a version of Chung's system that provides the functionality necessary to switch architectures at runtime. The other sub-system, called the optimization sub-system, provides the functionality that decides when to invoke the architecture change functionality of the first system. In this section, we describe how we implemented these systems. Since our sharing sub-system is a version of Chung framework, we describe Chung's solutions to any issues it raises. If there is a difference between how Chung system and our version of it handle an issue, we present the difference.

2.3.1 Sharing Applications

The fundamental issue with any collaboration framework is how it shares applications. As mentioned above, collaborative systems assume that the application is logically divided into a program and a user-interface component as shown in Figure 2-1 (top-left). Chung's framework makes use of this property to share the application. A part of the system, which we refer to as the sharing client-side component, is logically situated between the user-interface component and the (potentially inactive) local program component, as show in Figure 2-1 (top-right). The local program component is inactive if it is not mapped to any user-interface component, which happens, for instance, when the hosting computer is a slave. The actual application is not aware of the client-side component. From the user-interface component's perspective, the client-side component appears to be the program component, and from the program component's perspective, the client-side component appears to be the user-interface component. In particular, (a) the user-interface sends input commands to and receives outputs from it, while (b) the program component receives input commands from and sends outputs to it. Meanwhile, the client-side component forwards received (a) input commands to the local program component, and (b) output commands to

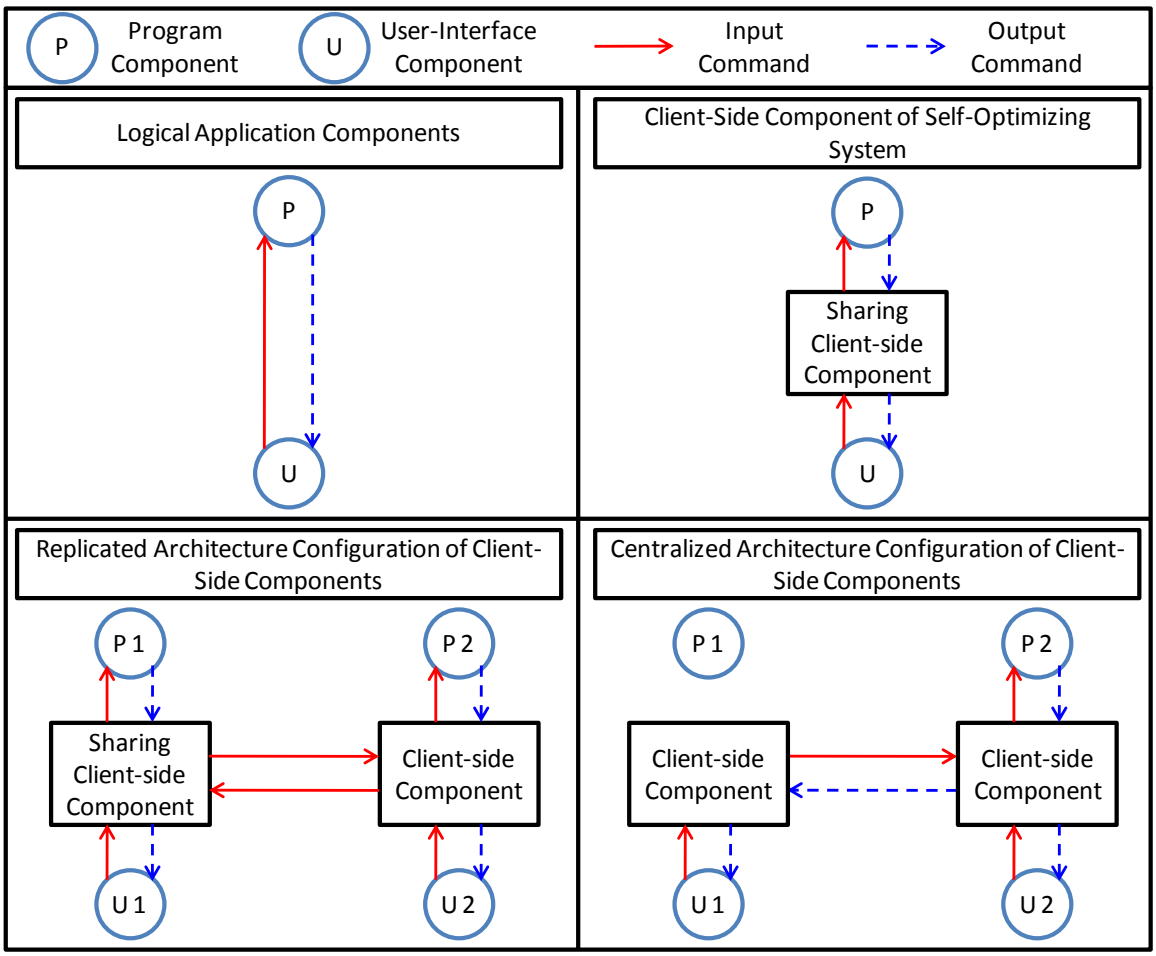

Figure 2-1. Client-Side Component of Self-Optimizing System.

the local user-interface component. The client-side components on the users' computers communicate directly with each other. They act as a communication switch that can be configured to enforce replicated or centralized mappings of user-interface to program components shown in Figure 2-1 (bottom-left) and Figure 2-1 (bottom-right), respectively.

2.3.2 Changing Architectures of Shared Applications

To change an architecture dynamically at runtime, the framework must reconfigure the sharing client-side component mappings. To illustrate, consider a two-user scenario in which $user_1$ and $user_2$ are sharing an application using the replicated architecture shown in

Figure 2-1 (bottom-left). To switch from the replicated architecture to the centralized architecture in which $user_2$'s computer is the master (Figure 2-1 (bottom-right)), the framework configures 1) the client-component on $user_1$'s computer to forward input commands to the client-side component on $user_2$'s computer and 2) the client-side component on $user_2$'s computer to forward outputs to not only the local user-interface component but also to the client-side component on $user_1$'s computer. Similarly, to switch back to the replicated architecture shown in Figure 2-1 (bottom-left), the framework configures 1) the client-component on $user_1$'s computer to forward input commands to both the local program component and the client-side component on $user_2$'s computer and 2) the client-side component on $user_2$'s computer to forward inputs to not only the local program component but also to the client-side component on $user_1$'s computer.

One issue that arises when changing the mappings of the client-side components is that the computers may temporarily be configured in an inconsistent manner. In particular, the switch takes some time, and each computer may switch over at a different rate depending on the processing power of the computer. Therefore, if the switch is not performed atomically with respect to users' input commands, then during a switch, an input command and its output may be distributed to some users' computers using mappings in the old and to others using mappings in the new architecture. Without extra precautionary measures, it could happen that during the switch, slave computers receive input commands and master computer receive output commands from other computers. Such messages are inconsistent with the notion of centralized and replicated architectures. One way of ensuring that this problem does not arise is to temporarily pause the processing and transmission of commands, perform the architecture switch, and then resume inputs. An issue with this approach is that

the response times of commands during the switch may appear high if the switch takes a long time. An approach that does not have this issue is to run the old and new architectures in parallel and then seamlessly switch all users to the new architecture once it is fully deployed. This is the approach taken by Chung. Since the focus of this thesis is not optimizing the architecture switch mechanism, we use the first approach as it easier. We leave the implementation of Chung's solution in our system as future work.

Another architecture switch mechanism issue is bringing the program component running on a new master to a state that is consistent with the other program components. This is an issue when a slave computer becomes a master because of an architecture switch. The reason is that the slave was not receiving input commands and, therefore, the shared object was not updated in the slave's inactive program component. One way of bringing a new master up to date is to replay all of the input commands to the master. Unfortunately, such an approach is unbounded in terms of memory and time as the number of commands increases continually throughout a session. Chung presented an approach that does not suffer from this problem. He created a compressed logging approach which requires O(number of objects in shared state) memory space and time to bring a new master's shared state up to date. However, this approach is more complicated to implement than the previous approach. Once again, as the focus of this thesis is not optimizing the time it takes to bring a latecomer or a new master up to speed, we use the easier approach. The client-side component of our system records all input commands which it has forwarded to its local program component. When there is a new master user, our system randomly picks one of the client-side components running on a master computer to send all of the input commands entered so far to the client-

side component on the new master. In the future, we plan to implement Chung's solution into our system.

2.3.3 Accommodating Late-comers

An issue that is related to problem of bringing the program component on a new master's computer up to date is bringing a late-comers computer up to date. When a late-comer joins, Chung maps the sharing client-side component running on the late-comers computer to the sharing client-side components of the users already in the system. The mapping is determined by the architecture currently being used. If the centralized architecture is currently being used, the late-comer joins as slave. Therefore, (a) the late-comer's client-side component is configured to forward input commands to the client-side component on the master computer and (b) the master user's client-side component is configured to forward outputs to the late-comer's client-side component. If the replicated architecture is currently being used, the late-comer joins as a master. Therefore, (a) the late-comer's client-side component is configured to forward input commands not only to the local program component but also to the client-side components running on all of the user's computers and (b) the client-side component of each user that was already in the session is configured to forward inputs to the late-comer's client-side component.

Since the late-comer joins into a particular architecture, there is no issue of temporarily configuring the client-side components in an inconsistent manner. However, if the late-comer joins as a master, then the framework must still bring the late-comer's program component up to date. We use the same mechanism as above. In addition, when the late-comer joins as a slave, the framework must bring the user-interface of a latecomer to a state that is consistent with all of the other user-interfaces. In this case, Chung replay a

compressed log of output commands while our system replays all of the output commands to the latecomer. In particular, the client-side component of our system records all outputs which it has forwarded to the local user-interface component. When a latecomer joins as a slave, our system randomly picks one of the client-side components on a different computer to send all of the output commands observed so far to the client-side component on the latecomer's computer. In the case when a master switches to a slave because of an architecture change, the user-interface is up-to-date and no replay is necessary.

We have addressed the issues regarding the sharing sub-system. We next address the issues regarding the optimization sub-system, starting with its architecture.

2.3.4 Optimization System Architecture

The optimization sub-system is composed of two separate components. These components work together to gather the values of the response time parameters identified by the model and apply the model. The optimization client-side component runs alongside the sharing client-side component on each user's computer as shown in Figure 2-2. The optimization client-side components are responsible creating performance reports for the local user and sending them to the server-side component of the system as shown in Figure 2-2. The server-side component collects all of the reports. It uses the reports to estimate the parameter values, apply the analytical model, and reconfigure the sharing client-side components if necessary. In this section, we justify the architectural decisions of the system. In the next two sub-sections, we describe how the optimization client-side and server-side components generate performance reports and apply the analytical model, respectively.

The server component can be centralized or distributed. One centralized approach is to run it on a computer belonging to one of the user's in the session. We do not take this

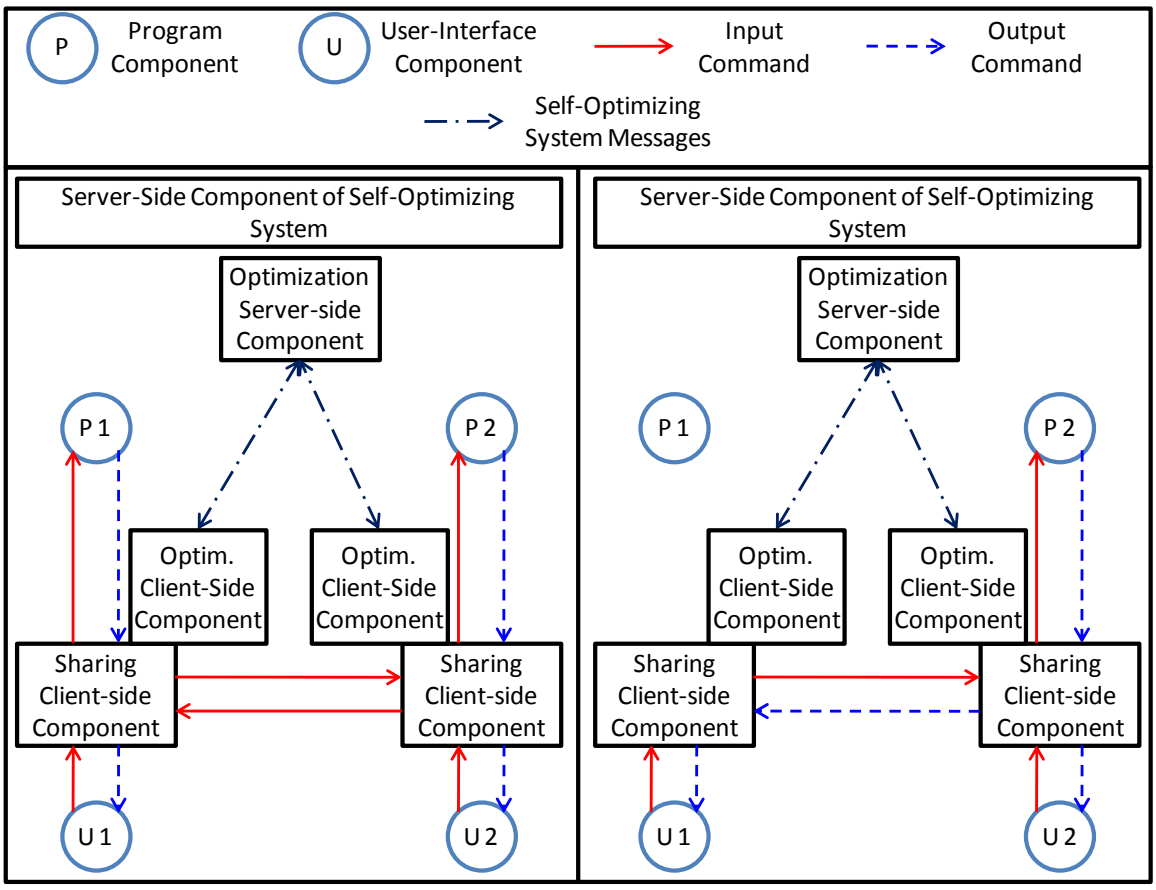

Figure 2-2. Server-Side Component of the Self-Optimizing System.

approach, however, primarily because it can impact the response times on the user's computer. The processing and networking resources required by our system can be high, especially in large-scale scenarios that we consider in the next chapter. An alternative approach to centralizing the system on one of the user's machines is to use a distributed approach in which each user's computer runs a server side component. The component on each user's computer receives reports from a subset of all of the client-side components. It may or may not share reports with the component on other users' systems. If components share performance data, they do not share all of it with all other components. Otherwise, this distributed approach does not offer any benefit over the earlier centralized approach. The distributed approach reduces the impact on response times of our system on any particular

user. In general, however, such approaches suffer from a consensus issue. In particular, in can be difficult for all computers to agree on a particular architecture as each computer is making decisions based on potentially different data, that is, to reach distributed consensus. The approach we actually take is a version of the centralized approach. Our system runs on a dedicated (infrastructure) computer that is not being used by the users' machines, as shown in Figure 2-2. For instance, the computer running the session manager can also run our system for all sessions. Such a computer exists even in most highly distributed peer-to-peer systems. It is typically called the bootstrapping node [90].

One issue with centralizing our system on a single machine is that it may take a long time to decide which architecture should be used when there are many sessions running in parallel. Fortunately, poor response times by our system do not degrade the response times of user commands. The worst that can happen if it does not decide on a new architecture quickly is that the response times of the users' commands stay the same. Moreover, it system can be configured to periodically calculate whether an architecture switch would better satisfy user-provided response time requirements. As a result, it is unlikely that architecture calculations would be taking place for all sessions at exactly the same moment in time. The period can be triggered after either some user-specified amount of time elapses or the system receives a user-specified number of performance reports. The only exception is when the system configuration changes because of an external event, such as a new user joining the collaboration. In such cases, the system restarts the period. Whenever the period elapses, the system calculates whether or not a switch to a new architecture would be beneficial to response times. The system switches to a new architecture only if the architecture is different from the current architecture and is predicted to improve response times. By default, our

system is set to calculate whether or not an architecture switch would improve response times after every five user commands. Future studies are needed to determine what the best default value is, which may likely be application and scenario dependent. For now, the arbitrary value of five seems to work well in our experiments.

2.3.5 Gathering Parameter Values

In order to apply the equations in the model, the optimization sub-system must first gather the values of the response time parameters identified by the model:

- The input and output processing times on each user's computer
- the network latencies between the users' computers

Processing Times

As users join the session, each user's computer registers with the server-side component of the optimization system. The registration is performed by the client-side component of the system. It sends the system a description of the local computer. This description consists of the name, family, and speed of the processor of the computer. To uniquely identify the type of computer processor, all three of these values are necessary. The name typically gives a description, such as "Intel(R) Core(TM)2 Duo CPU E4400 @ 2.00Ghz." Two CPUs can have the same name but belong to different families, such as "x86" or "Itanium." Therefore, the CPU family is included in the description. Finally, the speed gives the clock speed of the CPU which may not necessarily be included in the name, although it may be as this example shows. The optimization system gathers parameters on a per computer description rather than a per user basis in order to be able to reuse of values collected during the current session. To illustrate, consider a latecomer user who has the same kind of computer as some user who has been in the session from the start. If the system

collected values on a per user basis, the system would have no information regarding the latecomer and would have to wait for more data in order to decide if an architecture switch would improve response times. But because the system collects values on a per computer description basis, it can reuse the parameter values gathered for the computer belonging to the user who has been in the session from the start as the parameter values of the latecomer's computer since the two users' computers are the same. Thus, the system can immediately apply the model and check if an architecture switch is needed.

The optimization system gathers new processing times as users input commands during the session. Each time a computer completes processing a command, the client-side component sends a performance report to the server-side component. The report includes the time the computer required to process the command. By default, a report is sent after each command is processed; however, the users can configure the system to buffer the reports and send a cumulative report after a user-specified number of messages have been processed. The buffering option is provided to reduce communication overhead if the extra communication adversely affects response times.

Network Latencies

The network latency collection approach is the dual of the one used to collect processing and think times. Instead of waiting for users to report network latencies to hosts with which their computers have communicated, our system requests network latency measurements from the users' computers to hosts specified by the system. The reason for this design decision is that the users' computers may not necessarily be aware of all other computers. For example, in a centralized architecture, a slave user's computer is aware of only the master's computer because it sends commands to and receives commands from only

the master computer. In order to make informed architecture optimization decisions, our system needs to know the latencies between each pair of user's computers. Therefore, it needs the latencies between the slave's computer and all of the other slave computers in the session. Because each user's computer registers with the optimization system when the user logs in, the system is aware of all computers. Therefore, when a new user registers, the system sends a list of IP addresses of all of the currently registered users' computers to the client-side component on the new user's computer. The client-side component then estimates the network latencies to the IP addresses in the list. It does so using the ubiquitous ping tool. In order to expedite the process, it creates a separate thread pings each destination and reports the latency measured by each thread back to our system. Since network latencies can change during the session, our system periodically asks each computer to repeat the measurements and report up-to-date network latency values. The polling period can be configured by users. By default, our system polls for new latency measurements every sixty seconds. We use the default polling period in our experiments.

One issue with frequently asking the users' computers to measure network latencies is that the measuring process may cause response times of collaborative commands to increase. To reduce the impact, the threads on the computer that perform the latency measurements are assigned a low priority. Moreover, our system can be configured to ask each computer to measure the network latencies to a random subset of the other computers. While this further reduces the potential impact of these measurements on response times, it increases the chances that our system has out-of-date parameter values of network latencies between some computers. In our experiments, a computer always measured latencies to all of the other

computers. Since we are focusing on two and three user scenarios, the maximum number of latencies a computer had to measure each time was two, which is low.

2.3.6 Applying the Analytical Model

Using the collected parameter values, the server-side component of the optimization sub-system could apply the analytical model as follows. First, estimate the values of processing times and network latencies based on the values reported by the client-side components. Second, use the estimates in the model and calculate the estimated response times of all users for a command entered by each inputting user in all possible centralized architectures and the replicated architecture. The number of centralized architectures depends on the number of users as each user can be a master in a centralized architecture. Since we are considering two and three user scenarios, there are at least two and at most three centralized architectures possible. Third, invoke the user-defined total order function passing the estimated response times, the list of inputting users, and the user identities as the parameters to the function. Finally, switch to the architecture that is ranked as the best by the total order function.

The procedure requires that the server-side component is aware of who the inputting users are. One approach for determining the inputting users is to assume that a user is an inputting user only if the user has entered some command so far. An issue with this approach is that a user who may input in the future may not have yet input a command. An alternative approach, which is the one our system uses, is to require that the users specify who the inputting users will be at the start of the session.

To illustrate the procedure, consider a three user scenario in which each user inputs commands. After estimating the values of the model parameters, our system crates the 4 x 3 x

3 response time matrix. The [x, y, z] entry in the matrix gives $user_y$'s response for a command entered by $user_z$ when architecture x is used. Since there are four possible architectures, the first dimension is of size four. Since there are three users in the session, the second dimension is of size three. And since each of the three users is an inputting user, the third dimension is also of size three. The third dimension is sorted based on the inputting user index from lowest to highest. Our system then invokes the total order function, passing the response time matrix, the list of inputting user indices sorted from lowest to highest, and a list of the identities of the users as parameters. Our system keeps an architecture index map that maps an architecture to an index in the first dimension of the matrix. When the total order function returns, it simply returns an index of the architecture that best satisfied the users' response time requirements. Our system uses the architecture index map to look up what architecture is the best and deploys it.

Unfortunately, the procedure does not work in all cases. In particular, the values reported by the client-side component may not provide sufficient data to estimate the response times for all architectures. For example, in a centralized architecture, a client-side component running on a slave machine will report output but not input processing costs. The reason is that slave users do not process input commands. Therefore, if a slave user's computer type is different from the master user's computer type, then our system does not have input processing times for the computer type used by the slave. As a result, it cannot estimate the response times in the replicated architecture or a centralized architecture in which the slave's computer is the master.

There are several ways of handling the case when some parameter values are missing. One way of handling this issue is to simply not attempt any architecture switches. Under this

approach, our system cannot improve response times during the current session. As we will see in the next section, it can still help improve response times in future sessions.

An alternative approach is to temporarily switch from a centralized architecture to the replicated architecture without considering the impact on response times. After receiving several input processing times from each computer, our system would have all of the parameters values it needs to predict the response times for all architectures. At that point, it can switch to the architecture that best satisfies the response time requirements. One issue with this approach is that the replicated architecture may provide worse response times than the current centralized architecture. In this case, the users' could experience poor performance until our system eventually switches architectures again.

A third approach, which is the one our system employs, is to estimate the missing parameter values and predict response times of potential architectures using the combined estimated and measured parameter values. The system estimates input processing times of a slave computer by calculating the ratio of the output processing costs on the master and the slave. It is always possible to compute this ratio because all computers process output commands. Then, assuming that the ratios of the master's and slave's input processing times is the same as the ratio of their output processing times, the system calculates the slave's input processing times based on the master's input processing times. One issue with this approach is that that the ratio of output processing times for two different types of computers may not be equal to the ratio of their input processing times. In fact, in our experience, these ratios are not identical. For example, we have found that the input to output processing cost of Checkers commands is 21.8 on a P3 desktop, 19.5 on a P4 desktop, and 40.7 on a Core2 desktop. In the worst case, the incorrect estimates would result in incorrect response time

predictions, and thus, the architecture that the total order function ranks as the best may not actually be the best. Fortunately, the ratios do not have to be perfectly equal for the estimate to be useful. In general, a computer that takes longer to process an output to a command than another computer also takes longer to process the corresponding input command. Hence, by using the estimating procedure, the system should at least correctly deduce which computer takes less time to process input commands. If the estimate causes a switch to a suboptimal architecture, our approach reduces to the second approach above. The degradation in response times is only temporary as eventually our system will choose the optimal architecture. For example, suppose that the system switches from a centralized architecture to either the replicated architecture or a centralized architecture in which a user for whose computer the input processing times had to be estimated is the master. Once the switch is made, the former slave computer will report input processing times and our system will soon receive performance reports with input processing times for the computer. Then, our system can more accurately predict response times of all architectures and as a result, eventually, switch to a more optimal architecture. In our experiments, a suboptimal architecture switch never occurred.

We can now summarize how our optimization system assigns values of the processing time parameters in the equation. It first calculates the average processing times for any computer type for which processing times have been reported. It can use either all of the reports or some user-specified number of the most recent reports to calculate the average processing times. By default, it uses all of them. When it is configured to use a subset of the reports, it can be further configured to apply a running average calculation using user-specified weights for the average processing time calculated for the latest set of reports and

the average calculated during the previous iteration. Default values are 0.7 and 0.3, respectively. Further experiments are needed to determine the optimal values instead of using magic numbers. For now, we use these default values as they seemed to work well in our experiments. Once it has calculated average processing times based on received reports, it then estimates the remaining processing times.

2.3.7 Re-using Parameter Values Across Sessions

Whenever a session ends, our system saves the estimate values of the processing times for each computer type. Using saved values from previous sessions, it can make initial architecture decisions at the start of the next sessions as long as 1) the application shared in the next session is that same as the application that was shared in the earlier sessions, and 2) the computer types used in the next session were used during the previous sessions.

As mentioned earlier, re-using parameter values across sessions makes the optimization system useful even if it is configured to not make architecture switches when measurements for some parameters are missing. In this case, our system still saves any parameter measurements it does receive. Saved values recorded for different collaborative sessions in which the same application was shared using different architectures are merged into one set of parameter values. The merged set is what is used as the initial values in future sessions.

When initial parameter values are available, calculating the average values and estimating missing parameters values at runtime is done using the same approach as described above, with one small difference. The first time the values are computed or estimated during the session, the previous iteration values are set to be the estimate provided

to the system at start time. Thus, the running average calculation will include the parameter measurements from previous sessions.

Of course, if any of the computers in the next session is a type our system has not encountered before, then accurate initial predictions are not possible even with merged parameter value sets. In this case, our waits to receive reports from the new computer type before deciding whether to switch away from the initial architecture.

2.4 Evaluation

We present two sets of experiments that illustrate the benefit of the self-optimizing system. In the first set of experiments, we will use the model to decide at runtime when to perform dynamic architecture changes and better satisfy user provided response time requirements. In the second set of experiments, we will use the model to pick an architecture at the start of the collaboration. Ideally, both sets of experiments should also show the practicality of our system. Therefore, the experimental data needs to be realistic. Our approach to gathering realistic experimental data is described in detail in Appendix A. For the results we present next, it suffices to say that we are using experimental data that is based on actual logs collected as collaborators used the distributed Checkers program used by Chung [21], which allows a group of users to play against the computer. We chose this program for two reasons. First, it is a computer-intensive task, allowing us to validate the effect of processing time differences. Second, the user study participants we used to collect logs knew the game rules, so no user training was needed. In the experiments we conducted to gather these logs, two users played together against the computer and both users made Checkers moves.

In Checkers, both the users and the computer make moves. The users perform a move when one of them moves a piece on the board. The computer then responds with its own move. The computer calculation of the next move depends on the piece positions, and is hence not constant. Thus, we report the average local and remote response times over all the moves in a single game.

We used three computers, a Core2 2.0GHz desktop, a P4 1.7GHz desktop, and a P3 500MHz desktop, which have processing power differences that can be expected when users collaborate. We use the P3 desktop to simulate next-generation mobile devices and current generation netbooks. The computers are connected on a local LAN. Based on Chung's and Dewan's experiments, we added 72, 162 and 370 ms to the LAN delays to estimate half the round-trip time from a U.S. East Coast LAN-connected computer to a German LAN-connected computer, German modem-connected computer, and Indian LAN-connected computer, respectively. As LAN delays vary during an experiment, we performed it ten times and report the average performances for these ten trials.

2.4.1 Dynamically Switching Architecture at Runtime

Our three-user version of Chung's analytical model makes several predictions regarding choice of architecture. It is important to test that our self-optimizing system picks the architecture that the model predicts.

Analytical Model Predictions

One prediction made by the model, which agrees with Chung's original results, is that when processing powers of collaborators computers are asymmetric and the network latencies are low, it is useful to centralize the application on the fastest computer. To illustrate, suppose that two users are in the session, and that $user_1$'s computer is more

powerful than user$_2$'s computer. Suppose also that the computers are on the same LAN and hence the network latencies between them are low (i.e., $d{\sim}0ms$). In this case, user$_1$'s and user$_2$'s local response time differences between 1) the centralized architecture in which user$_1$'s computer is the master and 2) the centralized architecture in which user$_2$'s computer is the master and the replicated architecture are given by

$$local_{CENT,i,1}u_1 - local_{CENT,i,1}u_2 = p_{i,1}^{IN} + p_{i,1}^{OUT} - \left(d(1,2) + p_{i,2}^{IN} + d(1,2) + p_{i,1}^{OUT}\right)$$
$$= p_{i,1}^{IN} - p_{i,2}^{IN} < 0$$

$$local_{CENT,i,1}u_1 - local_{REP,i,1} = p_{i,1}^{IN} + p_{i,1}^{OUT} - \left(p_{i,1}^{IN} + p_{i,1}^{OUT}\right) = 0$$

$$local_{CENT,i,2}u_1 - local_{CENT,i,2}u_2 = p_{i,1}^{IN} - p_{i,2}^{IN} < 0$$

$$local_{CENT,i,2}u_1 - local_{REP,i,2} = p_{i,1}^{IN} - p_{i,2}^{IN} < 0$$

As these equations show, the centralized architecture in which user$_1$'s computer is the master provides user$_1$ with local response times equal to those of the replicated architecture and better than those of the centralized architecture in which user$_2$'s computer is the master. The equations also show that user$_2$'s local response times are the lowest in the centralized architecture in which user$_1$'s computer is the master.

Consider now user$_1$'s and user$_2$'s remote response times differences between 1) the centralized architecture in which user$_1$'s computer is the master and 2) the centralized architecture in which user$_2$'s computer is the master and the replicated architecture. The response time differences are given by

$$remote_{CENT,i,1}u_1 - remote_{CENT,i,1}u_2 = d(2,1) + p_{i,1}^{IN} + p_{i,1}^{OUT} - \left(p_{i,2}^{IN} + d(2,1) + p_{i,2}^{OUT}\right)$$
$$= p_{i,1}^{IN} - p_{i,2}^{IN} < 0$$

$$remote_{CENT,i,1}u_1 - remote_{REP,i,1} = d(2,1) + p_{i,1}^{IN} + p_{i,1}^{OUT} - \left(d(2,1) + p_{i,1}^{IN} + p_{i,1}^{OUT}\right) = 0$$

$$remote_{CENT,i,2}u_1 - remote_{CENT,i,2}u_2 = p_{i,1}^{IN} - p_{i,2}^{IN} < 0$$

$$remote_{CENT,i,2}u_1 - remote_{REP,i,2} = p_{i,1}^{IN} - p_{i,2}^{IN} < 0$$

As these equations show, the centralized architecture in which $user_1$'s computer is the master provides both $user_1$ and $user_2$ with better remote response times than those of than those of the centralized architecture in which $user_2$'s computer is the master and those of the replicated architecture. Hence, overall, the centralized architecture in which $user_1$'s computer hosts the application should be used to provide the users with the lowest response times.

Another prediction by the model is that when a third user with a computer more powerful than $user_1$'s and $user_2$'s computers joins the collaboration late, the centralized architecture in which $user_3$'s computer is the master should be used. The difference analysis used in the two-user case to deduce that the centralized architecture in which $user_1$'s computer is the master should be used to provide the lowest response times to both $user_1$ and $user_2$ can be extended to the three-user case. In this case, the analysis would show that the centralized architecture in which $user_3$'s computer is the master should be used. We will not present the analytical analysis for this case, however, because it similar to the difference analysis we used above and it is tedious to repeat it.

Self-Optimization Experiments

To verify that our self-optimizing system indeed selects 1) the centralized architecture in which $user_1$'s computer hosts the application when only $user_1$ and $user_2$ are present and 2) the centralized architecture in which $user_3$'s computer hosts the application when $user_3$ joins late, we performed the following experiment with our distributed Checkers game. Initially, $user_1$ and $user_2$ play together against the computer. $User_1$ is using the P4 desktop and $user_2$ is using the P3 desktop. The P3 desktop is less powerful than the P4 desktop. Fifteen moves into the game (approximately one third of the game), $user_3$ joins with the Core2 desktop,

which is more powerful than the P4 and the P3 desktops. $User_1$ and $user_2$ make all of the moves; $user_3$ observes their moves. In total, $user_1$ made 37 moves and $user_2$ made 11 moves.

The total order function used ranks one system better than another if the number of response times that were better with the first system is higher than the number of response times that were better with the second system. In the case of a tie, such as when one architecture provides better response times to one user while a different architecture provides better response times to the other user, the function randomly picked one of the architectures.

We performed experiments with and without our system. For both cases, we performed three sets of tests. In the first set, the initial architecture was the one in which $user_1$'s computer hosts the application. In the second set, the initial architecture was the one in which $user_2$'s computer hosts the application. Finally, in the third set, the initial architecture was the replicated architecture. We added 0ms to the LAN latencies in all of these tests.

For this experiment, we configured our system to check whether an architecture switch is useful every five user's Checkers moves. Therefore, in the experiment cases in which our system is running, we expect an optimization to occur five moves into the game. Moreover, we expect another optimization to occur at the twentieth move as this is five moves after $user_3$ joins. Other optimizations are also possible, although ideally, the system will only perform the two we expect. Any other optimization suggests some instability in our system.

The benefit of using our system with each of the three initial architectures is shown in Figure 2-3 through 2-11. In all experiment runs, the optimizations happened twice, except when initial architecture was the centralized architecture in which $user_1$'s computer hosts the

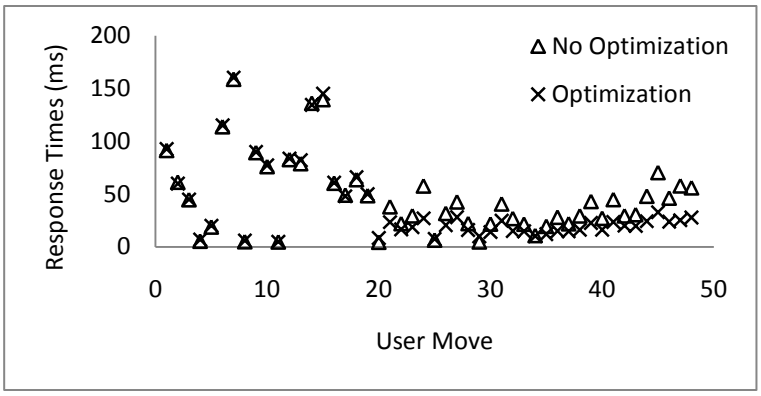

Figure 2-3. User$_1$'s response times starting with user$_1$ hosting the application.

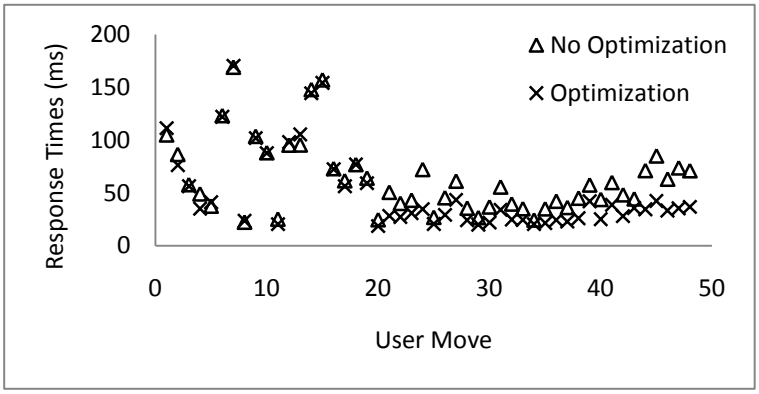

Figure 2-4. User$_2$'s response times starting with user$_1$ hosting the application.

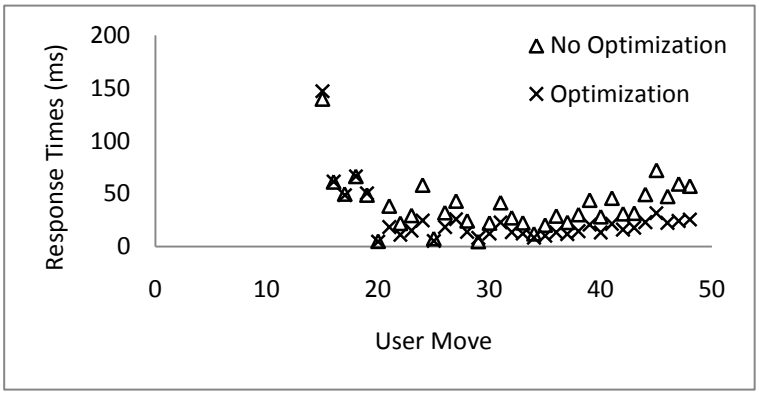

Figure 2-5. User$_3$'s response times starting with user$_1$ hosting the application.

application. This is expected because the first optimization happens before user$_3$ joins, and according to the model, user$_1$'s computer should host the application in this case. If user$_1$ is already hosting the application, no optimization is needed. Therefore, the current experiment did not reveal any instability in our system.

Figures 2-3, 2-4, and 2-5 show $user_1$'s, $user_2$'s, and $user_3$'s response times, respectively, when the initial architecture is the centralized architecture in which $user_1$'s computer hosts the application. As mentioned above, we will consider a change of 50ms in response times significant. At the same time, we will attribute a change of 5ms or less, which is an order of magnitude less than 50ms, to fluctuation in traffic in the network and processes running on the computers that are not related to our experiments. Thus, we consider two response times different by 5ms or less to be the same. For each user, the response times are the same for the first twenty moves regardless of whether or not our system is running. As mentioned above, this is expected as the optimal architecture until $user_3$ joins is the centralized architecture in which $user_1$'s computer hosts the application. Since $user_3$ does not join until fifteenth move and our system requires five moves to occur before suggesting a new architecture, we do not expect to see any benefit of our system until after the twentieth move. After the twentieth move, the response times improve for all users when our system is running. The reason is that our system suggested that the centralized architecture in which $user_3$'s computer hosts the application should be used and then switched to that architecture. The average response times before and after the twentieth move are given in Table 2-1. As we can see, while our system does improve response times, the improvements on average are less than 50ms, which is not noticeable to users. For the rest of the experiments, our system is able to improve response times significantly.

Consider first the case when the initial architecture is the replicated architecture. The response times for experiment runs in which that was the case are shown in Figures 2-6, 2-7, and 2-8. In these experiment runs, we can see a decrease in response times after not only the twentieth move but also after the fifth move. The reason is that after the fifth move,

Table 2-1. Response times of users with and without the self-optimizing system.

User	Moves	Initial Architecture					
		Centralized on $user_1$'s computer		Centralized on $user_2$'s computer		Replicated	
		No Opt	With Opt	No Opt	With Opt	No Opt	With Opt
1	1 to 5	44.262	44.8494	127.6556	126.1352	43.835	43.9156
	6 to 20	74.01147	75.40013	217.1244	75.58087	74.70327	75.33527
	21 to 48	34.92927	23.415	104.116	22.77615	35.75545	23.68752
2	1 to 5	67.0922	64.0212	131.4488	130.4956	135.8504	133.4872
	6 to 20	88.41227	87.372	220.7997	87.75207	222.5123	88.87407
	21 to 48	50.52882	33.69236	109.761	34.18915	112.5713	33.65991
3	1 to 5						
	6 to 20	61.8045	63.022	174.7638	62.03383	26.67233	62.56733
	21 to 48	35.914	21.36661	103.2056	21.12027	16.64103	21.58079

our system switches the architecture from the replicated architecture to the centralized architecture in which $user_1$'s computer hosts the application. Later, five moves after $user_3$ joins, the system performs another switch, this time to the centralized architecture in which $user_3$'s computer hosts the application.

The average response times for Checkers moves one through five, five through twenty, and twenty one and on are given in Table 2-1. As Table 2-1 shows, our system significantly improves response times for $user_2$ after the first optimization because $user_2$'s computer is now using $user_1$'s computer as a fast computation server. $User_1$'s response times do not improve since both before and after the first optimization, $user_1$'s computer generated responses for $user_1$. After the second optimization, the response times improve for all users although not significantly.

Finally, consider the case when the initial architecture is the one in which $user_2$'s computer hosts the application. The response times for these experiment runs are shown in Figure 2-9, 2-10, and 2-11. Note that we can analyze these graphs the same way as we analyzed the previous case. In this case, the response times are significantly improved for

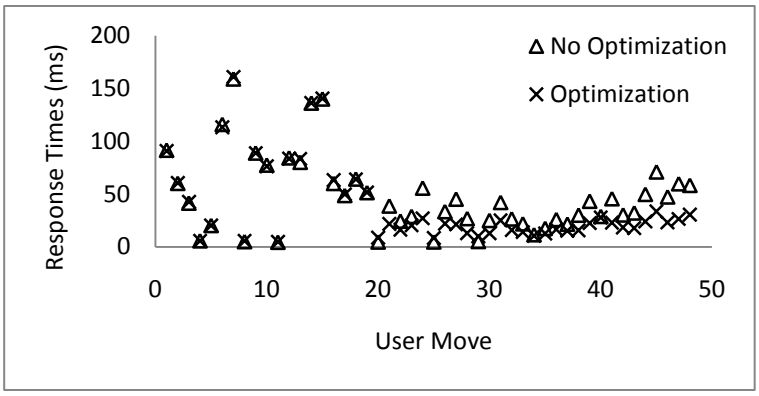

Figure 2-6. User$_3$'s response times starting with the replicated architecture.

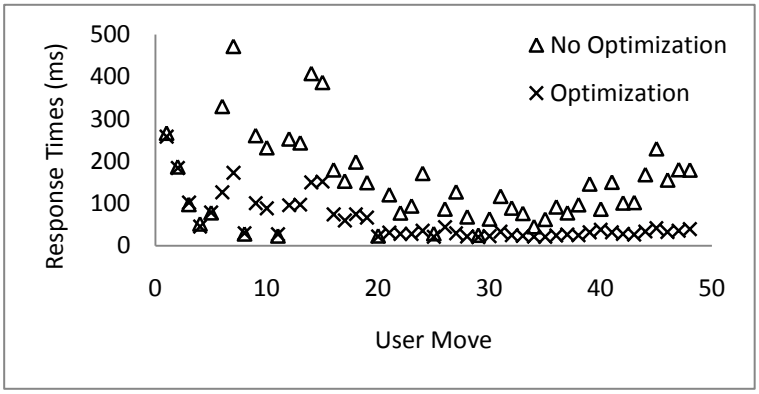

Figure 2-7. User$_3$'s response times starting with the replicated architecture.

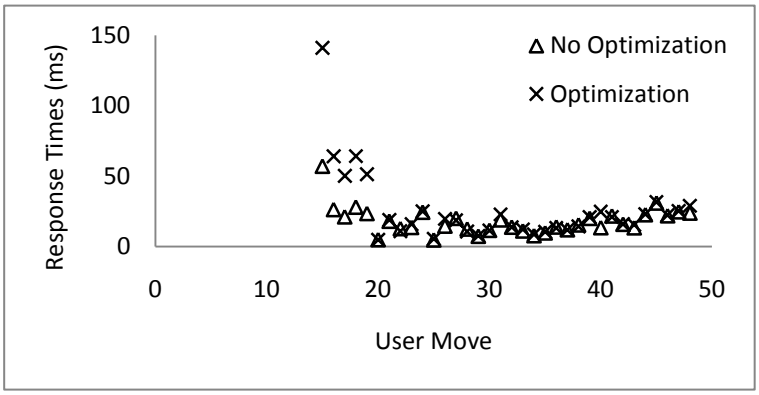

Figure 2-8. User$_3$'s response times starting with the replicated architecture.

both user$_1$ and user$_2$ after the first optimization. They are improved again after the second optimization, although not significantly, as before.

Interestingly, the response times for user$_3$ were worse for moves six through twenty when the initial architecture used was the replicated architecture. While this may seem odd, it

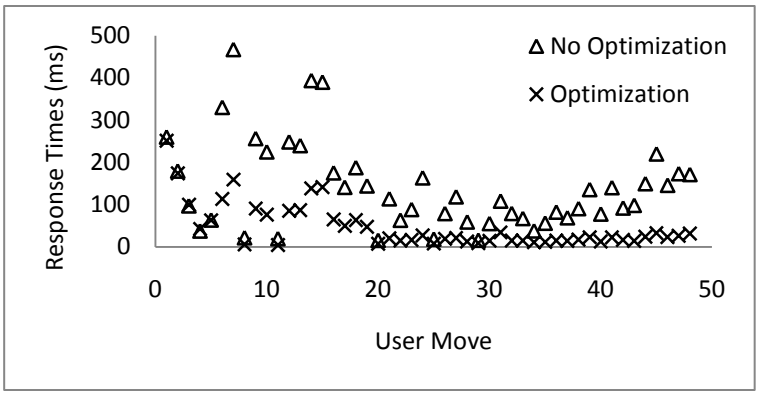

Figure 2-9. User$_3$'s response times starting with the replicated architecture.

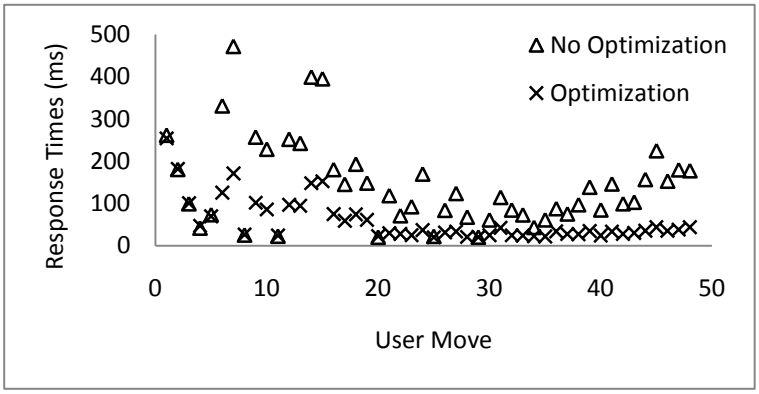

Figure 2-10. User$_3$'s response times starting with the replicated architecture.

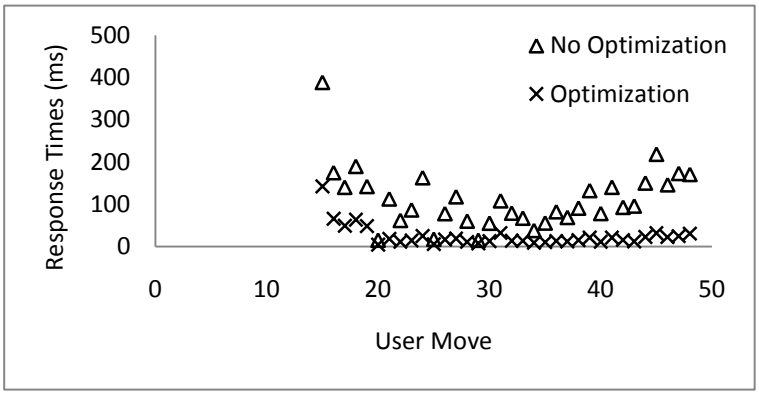

Figure 2-11. User$_3$'s response times starting with the replicated architecture.

is actually expected. By the time user$_3$ joins, the self-optimizing system had switched the architecture from replicated to the centralized architecture in which user$_1$'s computer hosts the application. When user$_3$ joins, user$_3$'s computer joins as a slave. Since user$_1$'s computer is less powerful than user$_3$'s computer, until the system switches the architecture to the

centralized architecture in which $user_3$'s computer hosts the application, $user_3$ is going to have worse with than without the self-optimizing system. The reason is that without the system, the architecture would have not changed from replicated to centralized before $user_3$ joined. Hence, $user_3$'s computer would have joined as a master.

2.4.2 Choosing Architecture at Start Time

The analytical model can be used not only to drive dynamic architecture changes at runtime, but also to select the initial architecture at start time. In order to use the model in this fashion, our system needs to have values of processing costs and network delays before the collaboration starts. Network delays can be gathered just before the collaboration starts as users join the session. As for initial estimates of the processing costs, we can use the values our system saved at the end of the previous set of experiments. Given the values of the processing costs of and network latencies among collaborators' computers at session start time, our system can apply our equations to deduce what architecture to use at start time. If the processing costs of the collaborators' computers are asymmetric and network latencies are low, then the equations inform us to the centralized architecture in which the user with the most powerful computer hosts the application.

We repeated the above experiments with the initial processing cost values. At start time, our system chose the centralized architecture in which $user_1$'s computer is the master. Eventually, when $user_3$ joined, the system switched to the centralized architecture in which $user_3$'s computer is the master.

The processing cost estimates we used are given in Table 2-2. To check the accuracy of the estimates, we compare them to the processing times measured for each computer type in a separate set of experiments, which are given in Table 2-3. As Table 2-2 and

Table 2-2. Average estimated Checkers processing costs.

	P3 Desktop	P4 Desktop	Core2 Desktop
Estimated Input Processing Costs	106.278	61.447	11.147
Estimated Output Processing Costs	8.054	2.445	0.411

Table 2-3. Average measured Checkers processing costs.

	P3 Desktop	P4 Desktop	Core2 Desktop
Measured Input Processing Costs	122.219	42.238	17.158
Measured Output Processing Costs	5.597	2.161	0.421

Table 2-3 show the estimated costs are different from the actual measured costs. On the one hand, the relative order of the fastest to slowest computers is the same in both tables, which is one reason why our system was able to choose the correct architecture at start time. However, the absolute values of the costs in the two tables are different. The difference in the absolute values is an artifact of the fact that the costs of processing any two commands in a single Checkers game are different. For instance, the cost of processing the command depends on the number of pieces on the board and their positions, which changes throughout the game. In each of the cases in which our system is running, the three types of computers actually calculate a different set of computer moves. Since we use the costs of processing these sets of moves as a basis for our estimate, the estimate is going to be different than the value given by calculating all of the moves. However, as our system is able to gather costs from more scenarios, the cost estimates should also become more accurate. One way to gather more accurate estimates is to perform measurements during sessions in which the replicated architecture is used the entire time. After few such sessions, the estimated costs recorded by our system should be closer to the actual processing costs because when the

replicated architecture is used, all computers report both input and output processing costs for all commands. Hence, the initial architecture picked based on these costs is more likely to be the best architecture to use.

2.4.3 Impact on Response Times of Self-Optimizing System Overheads

One issue left unaddressed in the discussion of the results is the impact on response times of the self-optimizing system as it gathers parameter values and switches architectures. As mentioned earlier, when switching between architectures, the system pauses the execution of processing and transmission tasks. As a result, if users enter commands during architecture switch time, they may notice an increase in response times for these commands. However, if the architecture switch can be completed during think time, then the users will not notice an increase in response times. In our experiments, the average and maximum times required to perform the first optimization (switch to centralized architecture in which $user_1$'s computer hosts the application) were 286ms and 308ms, respectively. The average and maximum times required to perform the second optimization (switch to centralized architecture in which $user_3$'s computer hosts the application) were 280ms and 360ms, respectively. On the other hand, the lowest think time we observed in the logs was 2134ms. Moreover, think times were below 4000ms a total of only 8 out of 48 moves. Since the maximum architecture switch time, 360ms, is less than the minimum think time, 2134ms, an architecture switch happened either in between two moves (during think time) or it overlapped with at most one move. Hence, in the above scenarios, users could notice increased response times for at most one command because of a switch. Finally, the average and maximum times required to bring $user_3$ up to speed when $user_3$ joins the session were 632ms and 638ms, respectively. The reason this time is higher than the architecture switch times is because the it includes not only

the time required to replay commands entered so far to $user_3$'s computer but also setting up all of the initial data structures for $user_3$.

In general, the architecture switch time could take longer than it did in our experiments. In this case, response times may suffer for several commands. One way to handle this problem is to notify users that an architecture switch could be made, and then wait for the users to notify the system that it can go ahead with the switch. Users often need to stretch out, go to the bathroom, etc., and during these pauses in new input commands, the architecture switch could be performed.

So far, we have only considered the impact on response times of the architecture maintenance mechanism. Response times can also be impacted by the mechanism that collects performance and network latency reports from the users. To check whether or not these mechanisms impacted response times, we must compare the response times of each users with and without our system. We compare the response times in the first set of experiments above. The response times for the first five commands should betray any impact of the parameter collection mechanisms on response times. The reason is that in the case when our system is running, it does not change the architecture during the first five commands but it is still collecting data. As Table 2-1 shows, the response times for the first five commands with and without our system are the same (within 5ms).

2.4.4 Other Results

In the presented experiments, we have considered only the case when the collaborators are all on the same LAN. While this is realistic in some scenarios, it is also realistic that they are on different LANs. In this case, network latencies among them will be higher than LAN latencies, that is, they will be higher than 0. Unfortunately, we do not have

access to computers on different LANs. As a result, we simulate delays between users to simulate them being on different LANs. Each user's computer delays the processing of the command by the simulated network latency between the user's computer and the sending user's computer to simulate latencies between them.

We have performed experiments with latencies other than LAN latencies, such as those between a U.S. East Coast LAN-connected computer to a German LAN-connected computer, German modem-connected computer, and Indian LAN-connected computer, respectively. The corresponding round-trip times are 144, 324, and 740ms. All of these round-trip times are larger than the processing cost differences of the machines used in our experiment. As a result, it does not pay for any computer to act as a high-speed computation server for another. Thus, the replicated architecture should always be used. When we ran these experiments, our system indeed switched to the replicated architecture regardless of the starting architecture in our experiments with non-LAN network latencies.

2.5 Summary

To summarize, we have presented an analytical response time model for centralized and replicated architectures in small-scale scenarios. The model extends the previous small-scale model by Chung and relaxes some of its assumptions. We have presented a system that can automatically apply our model to automate the maintenance of the collaboration architecture. We have also presented experiments that show that our system can noticeably improve response times. Moreover, we have identified new implementation and policy issues that must be addressed by any system that automatically maintains the architecture. These results combined prove sub-thesis I and a part of sub-thesis VI, which we re-state here.

SUB-THESIS I

It is possible to develop a system that automatically switches to the architecture that satisfies any user-specified response time criteria better than existing approaches.

SUB-THESIS VI

It is possible to develop a model that analytically evaluates the impact on response times of different processing architectures, communication architectures, and scheduling policies to the degree necessary to automate their maintenance.

2.6 Relevant Publications

1. Junuzovic, S., Chung, G., and Dewan, P. Formally analyzing two-user centralized and replicated architectures. *European Conference on Computer Supported Cooperative Work (ECSCW)*. 2005. pp: 83-102.

CHAPTER 3

AUTOMATING COMMUNICATION ARCHITECTURE MAINTENANCE

3.1 Overview

In the previous chapter, we focused on small-scale collaboration scenarios with two or three users. We presented an analytical model that can predict response times in such scenarios for both centralized and replicated architectures when transmission costs are negligible and there is no type-ahead. We also presented a new system that can automate the architecture maintenance under these conditions and better meet response time requirements than existing systems.

An important limitation with our earlier results is that they do not apply to scenarios in which transmission costs of commands are high. When transmission costs are high, they impact remote response times because it takes time to send commands to the remote users. This occurs when commands carry large amounts of data. For example, the size of a PowerPoint command can be several megabytes if it contains a slide from a presentation or even an entire presentation.

The limitation also implies that the results do not apply to large-scale scenarios involving hundreds or thousands of users. The reason is that when the number of users is high, the cost of transmitting a command to all of them can be high. This is true even if the command is small. It is especially true if the command is large. Some scenarios indeed

involve a large number of users. For instance, online chat rooms can have hundreds of people participating. Company-wide presentations given over Webex or LiveMeeting can have hundreds and even thousands of users. Online games can have audiences that reach even higher numbers. For example, the popular Starcraft game tournaments in South Korea can have millions of fans following the games. One can also imagine an online Checkers game that is being watched by thousands of people.

An important issue that arises when commands are transmitted to many destinations is whether commands are unicast or multicast. Multicast has long been advocated as a more efficient data distribution scheme than unicast – and justifiably so. In particular, it can better utilize network resources, such as routers and physical links, by reducing the degree of packet duplication. However, there has been little work done in applying this idea to distributed collaboration. The T 120 protocol [24] advocated the use of a multicast tree to reduce the amount of data (audio, video, bitmaps) transmitted on the network. How such a tree was built or the improvement in network usage in different kinds of collaboration scenarios was not studied. RMX [19] and SRM [35] have studied the use of multicast to improve the packet loss handling and fault tolerance of shared whiteboards. In this chapter, we extend this research by focusing on performance of the collaborative application rather than its reliability or network usage.

The idea of multicast requires the construction, for each source of messages, a multicast overlay that defines the paths a message takes to reach the destinations. Intuitively, such multicast overlays can degrade performance. They can increase remote response times as data must pass through additional computers to reach the collaborators. However, we

show that because of the cost of transmission costs, multicast can actually improve remote response times.

Existing collaboration architectures, however, do not support multicast. In particular, they couple the processing of input commands with the distribution of input commands and outputs. Master computers unicast input (output) commands to all other computers in the replicated (centralized) architecture. Thus, more than one computer is not involved in distributing data, which is inconsistent with the notion of multicast. Hence, we present a new bi-architecture model of collaborative applications that explicitly supports multicast. It represents a collaborative application using two separate sub-architectures: the processing sub-architecture dictates master-slave relationships, and the communication sub-architecture governs how the commands are distributed. The system presented in the previous chapter automates the maintenance of the processing architecture. In this chapter, we present a system that automates the maintenance of the communication sub-architecture.

In the process of creating the system that automates the communication architecture maintenance, we extended our earlier work in two ways. First, we present a new N-user response time model that supports both unicast and multicast. The model handles any-scale scenarios, that is, both small-scale and large-scale scenarios. Moreover, it relaxes all of the assumptions made by our earlier model. In particular, it does not assume that transmission costs are negligible or that there is no type-ahead. Second, we extend our self-optimization framework to collect transmission costs and apply the new N-user model.

Chapter Scope:

We analyze the impact on response times of the communication architecture in isolation of the impact of the processing architecture. We consider any-scale collaboration scenarios involving both small and large numbers of users. We relax our earlier assumptions that transmission costs are negligible and that there is no type-ahead.

Chapter Goals:

We show that the communication architecture impacts response times. We present an extension of our self-optimizing system that better meets response time requirements than existing systems by automating the maintenance of the communication architecture.

The system that automates the communication architecture maintenance raises several issues. One issue is how the system measures transmission costs. These costs depend on both the CPU and the network card: the CPU must queue a message for transmission by the network card and the network card must actually transmit the message onto the physical link. Measuring the CPU transmission costs can be done simply by timing how long the transmission loop takes. The same technique works for measuring the network card transmission costs when blocking communication is used. However, it does not work when non-blocking communication is used because the network card does not notify the application when transmission is completed. Hence, new techniques for measuring the network card transmission costs are needed when non-blocking communication is used.

Once the transmission costs are measured, the next issue is whether the system calculates a shared communication overlay rooted used to distribute the commands of all users or a different communication overlay rooted at each inputting user. The latter can improve response times more than the former since the messages from a user do not have to reach the root of the shared overlay first. On the other hand, the overlay calculation takes time and thus, the calculation of the former is quicker than that of the latter. Therefore, with the latter, the response time improvements are more immediate than with the former.

One the communication architecture is computed, the next issue is how the system deploys it. In the previous chapter, the system that automates the processing architecture maintenance pauses the processing of new commands during an architecture change. This has the undesirable effect of increasing the response times of commands entered during the change. Although we do not implement it, Chung presented a solution that does not require such a pause. His solution relies on running the old and the new processing architecture in parallel. The equivalent of such a system for changing the communication architecture would be desirable for the same reason.

Another, related issue is how the system maintains the communication architecture as users join and leave the session. Although we have addressed this issue in the previous chapter, we must return to it because it poses new problems when multicast is used. In particular, in a multicast overlay, users become disconnected if the user who was forwarding messages to them in the multicast overlay leaves. Hence, these users must someone be reinserted into the overlay. Similarly, the system must insert late-comers into the overlay.

As we describe the self-optimizing system, we will address all of the issues it raises. The rest of this chapter is organized as follows. We first present the bi-architecture model of

collaborative systems. Then, we develop the N-user analytical model for this architecture. Following this, we describe the self-optimizing system that automates the maintenance of the communication architecture. Then, we present experiments conducted with the extended version of our self-optimization system. Finally, we end with discussions and a brief summary.

3.2 Bi-Architecture Model

As mentioned above, multicast requires the formation of communication overlays. But all of the existing collaboration architectures couple the processing of input commands with the distribution of input commands and outputs. For example consider a collaborative scenario in which there are six people. The centralized and replicated architectures for this scenario are shown in Figure 3-1. As Figure 3-1 shows, in the centralized architecture, the

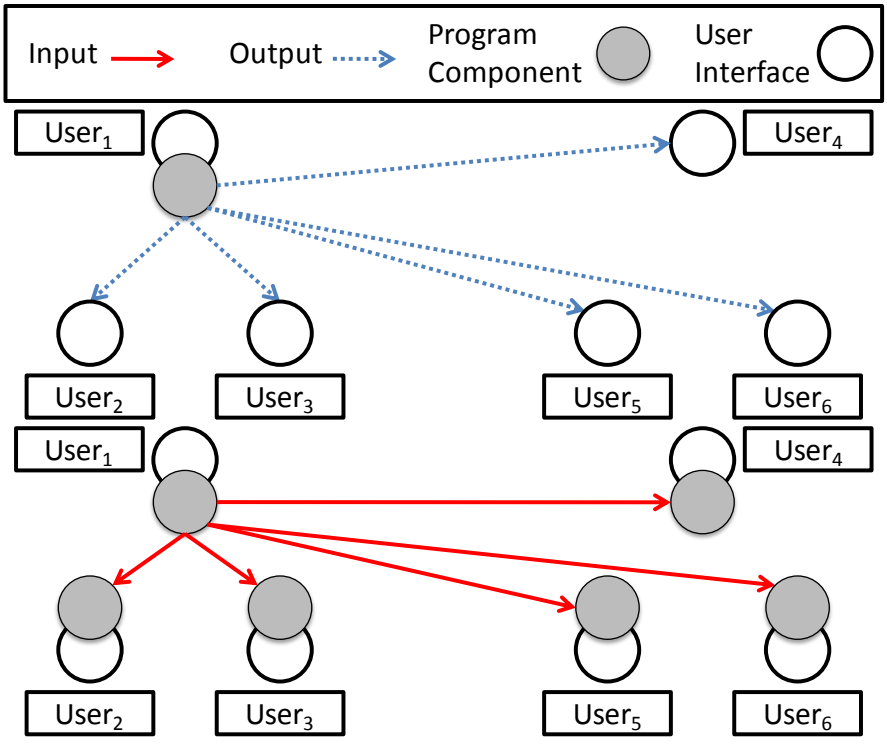

Figure 3-1. Centralized (top) and replicated (bottom) architectures with six users.

master computer unicasts output commands to all slave computers, while in the replicated architecture, a master computer unicasts all input commands it receives from its local user to all other master computers. In other words, more than one computer is not involved in distributing data, which is inconsistent with the notion of multicast.

To support multicast, we define a new bi-architecture collaborative systems model that decouples the processing and distribution tasks. The processing architecture governs the master-slave relationships and the communication architecture dictates how input (output) commands are distributed from one master computer to other master (slave) computers. In the model, every computer may perform some part of the transmission task. For example, if multicast is used in the centralized architecture, then a slave computer, in addition to processing any outputs that it receives, may also need to forward the outputs to other slaves

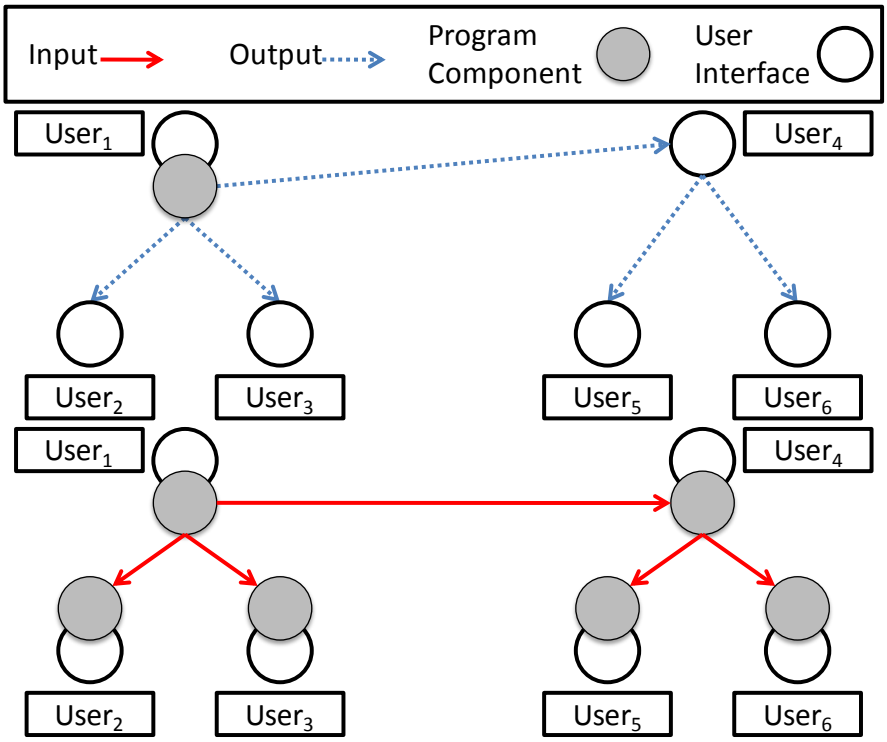

Figure 3-2. Bi-architecture model with a multicast communication architecture a (top) centralized and (bottom) the replicated processing architectures with six users.

as illustrated in Figure 3-2 (top). Figure 3-2 (top) shows the transmission after $user_1$'s computer, which is the master, computes the output for a command entered by $user_1$. The master transmits the output only to (computers belonging to) $user_2$, $user_3$, and $user_4$. $User_4$'s computer, which is a slave, then forwards the output to $user_5$ and $user_6$. Similarly, if multicast is used in the replicated architecture, a master computer that receives an input command from another master, may, in addition to processing the command, forward it to other masters as shown in Figure 3-2 (bottom). Figure 3-2 (bottom) shows the transmission of an input command entered by $user_1$. $User_1$'s computer transmits the command only to $user_2$, $user_3$, and $user_4$. $User_4$'s computer forwards the command entered by $user_1$ to $user_5$ and $user_6$. By definition, when a unicast communication architecture is used, the bi-architecture model degenerates to that of traditional collaboration architectures.

3.2.1 Implementation Issues

The main question of in this chapter is whether a unicast or a multicast communication architecture should be used to optimize the performance of a given processing architecture. To answer the question, we must first define certain implementation aspects of the bi-architecture model.

Scheduling of Transmission and Processing Tasks

The first of these aspects is whether processing and transmission tasks are carried out in a single thread, or in separate threads. In this chapter, we assume that both processing and data distribution tasks are carried out by a single thread. Multi-threaded implementations of program components, of course, have the potential of improving performance, especially on a multi-core or a multiprocessor computer. We leave the nature and impact on performance of such implementations for the next chapter.

When all operations are carried out by a single thread, one must determine the order in which they are carried out. Two scheduling policies, based on the operation kind, are 1) process-first, which favors local response times by postponing transmission tasks until pending processing tasks complete and the output produced by these tasks is displayed to the local user, and 2) transmit-first, which favors remote response times by giving precedence to transmission over processing and display. In this chapter, we assume a transmit-first scheduling policy. The process-first policy will be discussed in the next chapter when we consider the impact of scheduling. By fixing the scheduling policy, we are able to focus, in this chapter, on the difference between unicast and multicast communication.

In the unicast architectures, the scheduling policies are relevant only to master computers. The reason is that the slave computers do not participate in the data distribution task. When multicast is used, this is no longer true; in particular, a slave computer may be responsible for forwarding output commands that it receives to other computers. Thus, the scheduling policies must distinguish between masters and slaves. Table 3-1 shows the steps

Table 3-1. Order of processing and transmission tasks.

Centralized Processing Architecture	Replicated Processing Architecture

If (master computer)
  Wait for next input cmd
  Process input command
Else
  Wait for next output cmd
Transmit output cmd to (zero or more) slave users
Process output cmd
Repeat
	Wait for next input cmd
	Transmit input cmd to (zero or more) master users
	Process input cmd
	Process output cmd
	Repeat

taken by master and slave computers with multicast and transmit-first scheduling. The key difference between these steps and those taken in the unicast case is as follows. In the unicast case, a master computer transmits an input (output) command to *all* other computers in the replicated (centralized) architecture, respectively. In the multicast case, on the other hand, the set of destinations to which the inputting user's master computer transmits a command is determined by the multicast overlay. There must be at least one destination in the set; otherwise the system cannot be collaborative. Moreover, the multicast overlay determines the set of destinations to which a non-inputting user's computer transmits received commands. Depending on the multicast overlay, the destination set is empty for some computers and non-empty for others.

Blocking vs. Non-Blocking Communication

The final issue we must address is the whether a computer forwards commands using blocking or non-blocking communication. The difference between the two communication modes is whether or not the thread that makes a send call blocks until the network card transmits the command on the physical wire. Since typically the network card is much slower than the CPU, it seems that non-blocking communication should always be used, especially when the transmission task is given priority over the processing task. The reason is that the processing task is delayed less with non-blocking (than with blocking) communication, and hence the response times to the local user are better with non-blocking communication. However, a subtle but important issue arises when a large command is transmitted to a large number of destinations in a non-blocking fashion – the network card buffers can overflow! Hence, there are scenarios in which blocking communication should be used instead of non-blocking communication.

As both blocking and non-blocking communication are useful, our system supports both. From an implementation point of view, supporting both communication modes is a matter of setting a flag at system start time. From an analytical point of view, however, supporting both complicates the analysis as we will see when we develop our formal model.

3.2.2 Choice of Multicast Algorithm

In this first-cut effort at investigating the bi-architecture model, we did not want to develop a new algorithm for creating a multicast tree. Instead, we wanted to analyze the performance of an existing algorithm. There are two classes of such algorithms, namely, IP and application-layer multicast. IP multicast assumes that network level routers support multicast and can be organized into multicasting overlays. Hence, the source host sends only a single copy of a message and the routers make sure that the message reaches the desired destinations. In other words, the routers perform the actual packet duplication and forwarding of messages. In contrast, application-layer multicast assumes no multicast support at the network layer; instead, it organizes the end-user hosts into multicast overlays. In such overlays, the hosts are connected by logical links, which map to physical paths in the underlying network. Unfortunately, even though multicasting is a mature field, because of a lack of a practical approach to upgrading legacy backbone routers to include multicast functionality, a lack of a scalable inter-domain routing protocol, and other deployment issues [28], IP multicast is not widely available. For this reason, we analyze an existing application-layer multicast scheme.

In general, there are many approaches to create application-layer multicast overlays. Most of these approaches model the network as a graph in which the hosts are the vertices and the logical links between these hosts are the edges. Each host is assigned a set of

constraints, which acts as knobs for controlling resource usage. For example, the degree constraints can specify the available outgoing bandwidth of each host. Each link is assigned a cost, which is incurred each time the link is traversed. For instance, the cost can specify the latency of a link. Using this network model, traditional multicast schemes focus on minimizing the diameter of the multicast overlay while satisfying the host constraints. A typical optimization function that is used 1) minimizes the maximum sum of transmission delays, which is a function of bandwidth, and network latencies, which are also known as propagation delays, on a path from the source to any receiver (maximum end-to-end latency) and 2) respects the bandwidth capabilities of the devices. The implicit assumption in this approach is that the diameter of the overlay determines the largest end-to-end cost.

Recently, Brosh and Shavitt [13] argued that traditional delay and bandwidth oriented optimization functions are valid for network-layer but not application-layer overlays. They state that such functions assume network-layer data distribution capabilities at the application-layer, even though the data distribution capabilities at the two layers are fundamentally different. This difference can make the cost of the transmitting data to multiple destinations significant at the application layer (What we call transmission costs, they called processing costs. We use a more specific term as there are other kinds of processing tasks in our domain, such as those for processing input and output commands). For instance, network level routers are optimized for duplicating packets. Moreover, they have multiple physical outgoing links and can therefore transmit packets in parallel. Therefore, the key aspect of a router's transmission costs is its bandwidth, and the transmission cost is simply calculated as "size of message/bandwidth." Bandwidth, however, is not the only transmission cost factor on end-user machines. In particular, these machines

are neither optimized for duplicating packets nor do they typically have multiple network connections. Before the command reaches the network interface, the operating system must traverse the network stack and copy data buffers along the way, which takes time. Moreover, the operating system must perform these steps for each destination. Study by Abdelkhalek et al. [1] of the server for Quake, a popular multi-player first-person shooter game, found that these costs can be significant in practice. They found that the server spent 50% of CPU time on transmitting commands to clients. In addition, end-user machines typically do not have multiple network connections. Therefore, the network-layer transmission cost calculation is invalid at the application layer because it accounts only for the transmission costs at the network interface. It must also account for the speed at which the CPU can duplicate messages and write them to a memory location from which the network card can read them. Hence, both CPU transmission costs and network card transmission costs should be accounted for in the transmission cost of an end-user computer. As a result, Brosh and Shavitt define a new algorithm for creating application-layer multicast trees, which explicitly considers application-layer transmission times. They showed that the optimal multicast problem is NP-Complete for their network model and developed a heuristic multicast algorithm, called HMDM. They compared its end-to-end delays with the end-to-end delays produced by Dijkstra's Shortest Path Tree algorithm, which does not consider transmission times. Their simulations of these two algorithms show that the HMDM scheme provides better end-to-end delays than Dijsktra's scheme.

In summary, while other application-layer multicast schemes exist [8], HMDM is the only approach that considers the time end-hosts require for duplicating and transmitting messages on the network in the building of such a tree. As our motivation for applying

multicast to collaboration was based on the assumption that transmission costs are significant, we decided to use HMDM as the basis for our multicast architecture.

3.3 Formal Analysis

In the previous chapter, we presented a three-user formal model of response times in collaborative systems. The model assumed three-user collaborations, a zero cost of transmitting inputs and outputs, unicast communication, and no type-ahead. As a result, the model does not handle the impact on response times of transmission costs or a multicast communication architecture. We next present a formal model that (a) relaxes all of the assumptions of the previous chapter model and (b) supports both unicast and multicast communication. Regardless of whether unicast or multicast is used to distributed commands, either blocking or non-blocking communication can be used. As mentioned earlier, it makes more sense to use non-blocking than blocking communication because the CPU does not block waiting for the network card with the former while it does block with the latter. Thus, we present the analytical model for the case when non-blocking communication is used.

Analysis Scope:

Although both blocking and non-blocking communication can be used to transmit messages, we focus only on the non-blocking communication case.

We present the model starting with remote response time equations for the replicated architecture. The rest of the model can be derived directly from these equations. We start by considering the response times of commands entered once all of the computers have completed processing and transmission tasks for all of the previous commands – that is,

when the collaborative system has entered a quiescent state. We then analyze the response times of a command entered by a user before the output for the user's previous command has been processed on all machines. Finally, we consider the response times of simultaneous commands entered by different users.

3.3.1 Quiescent State Commands

Let us start by considering the remote response times of commands entered during a quiescent state.

Replicated Remote Response Time

We first consider an input command entered by a master user. To reach a particular user's computer, which we refer to as the destination computer, the command must travel from the source computer to the destination computer along some path. The path may consist of additional computers as shown in Figure 3-3. We refer to the source computer and these additional computers as intermediate computers. The terms destination and intermediate are relative to a particular path. An intermediate computer on one path is a destination computer on a different path as all users see the output of an input command. Let π denote the path from the source to the destination, m denote the number of computers on the path including the source and destination computers, and $\pi_k, 1 \leq k \leq m$, denote the k^{th}computer on the path π, where π_1 is the source and π_m the destination computer.

The replicated remote response time of command i to computer j along path π is given by

$$remote_{REP,i,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{REP}(i, \pi_k) + dest_{REP}(i, \pi_m)$$

where $d(\pi_k, \pi_{k+1})$ is the network latency between the k^{th} and $k + 1^{st}$ computers on path π, $int_{REP}(i, \pi_k)$ is the delay command i experiences on the k^{th} intermediate computer on the

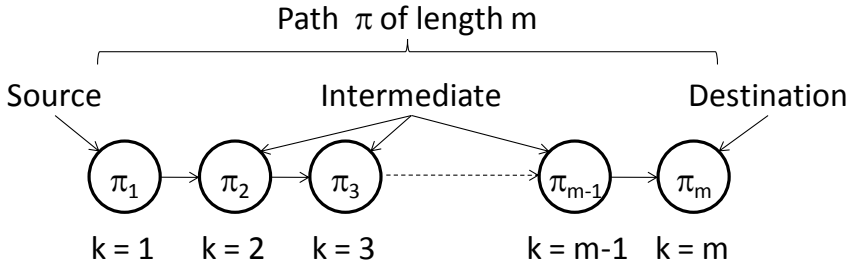

Figure 3-3. An example path of length m from source to destination.

path, and $dest_{REP}(i, \pi_m)$ is the delay command i experiences on the destination computer. The destination and intermediate computers contribute different delays because the former contributes to the remote response time of the local user while the latter contribute to the remote response time of a remote user. This results in a fundamental difference between the equations for the intermediate and destination computers. In the case of an intermediate computer, we must determine when the computer transmits to the downstream computer. In the case of the destination computer, we must determine when the input and output processing complete.

The first component of the remote response time equation is independent of the processing and transmission tasks, as it is a sum of the network latencies on the path from the source to the destination.

Consider first the delay of π_k, the k^{th} intermediate computer on path π. Its delay is equal to the time that it requires to transmit the command to the next computer along the path, π_{k+1}. By transmit, we mean to have placed the entire message destined to π_{k+1} on the physical wire. The transmit time depends on two factors: 1) the CPU transmit time, which is

the amount of time the CPU requires to copy the message data buffers from application space to the memory location from which the network card reads data for transmission, and 2) the network card transmit time, which is the amount of time the network card requires to read the message data buffers from memory and send it on the physical wire. We denote the CPU transmit time by $x_{CPU}{}_{i,\pi_k}^{IN}$ and the network card transmit time by $x_{NIC}{}_{i,\pi_k}^{IN}$. Note that $x_{NIC}{}_{i,\pi_k}^{IN}$ is not simply the inverse of the bandwidth of the system because it includes the time the network card requires to read data from the memory.

In general, computer π_k may have to transmit to more than one destination. Therefore, its delay depends on the number of other computers to which it transmits before transmitting to computer π_{k+1}. Consider the case when computer π_k transmits first to computer π_{k+1} To transmit a message to the first destination, computer π_k's CPU must queue the message for transmission, which requires $x_{CPU}{}_{i,\pi_k}^{IN}$ time, and then computer π_k's network card must transmit the message, which requires $x_{NIC}{}_{i,\pi_k}^{IN}$ time. Therefore, the delay is equal to $x_{CPU}{}_{i,\pi_k}^{IN} + x_{NIC}{}_{i,\pi_k}^{IN}$, as shown in Figure 3-4.

When computer π_{k+1} is not the first destination to which computer π_k forwards commands, the analysis is slightly more complicated. Let $pos(\pi_k, \pi_{k+1})$ denote the position of computer π_{k+1} in computer π_k's list of destinations. The delay is not simply equal to $pos(\pi_k, \pi_{k+1})$ times the delay to the first destination, that is, $pos(\pi_k, \pi_{k+1}) * (x_{CPU}{}_{i,\pi_k}^{IN} + x_{NIC}{}_{i,\pi_k}^{IN})$. The reason is that once the CPU transmits to the first destination, the CPU and the network card work in parallel, and the equation does not capture the parallelism. Since they are performing the transmission in parallel, the slower one determines the delay. In theory, either of the two can be the bottleneck. However, in all of our experiments, we found it was

always the network card. In particular, in our experiments, $x_{NIC}{}^{IN}_{i,\pi_k} > 2 * x_{CPU}{}^{IN}_{i,\pi_k}$.

Therefore, we consider only the case when the network card is the bottleneck. In this case,

Analysis Scope:

Although in theory both the CPU and the network card can be transmission bottlenecks, we found that the network card was always the bottleneck in all of the collaboration logs we recorded. Therefore, we focus on the case when the network card is the bottleneck.

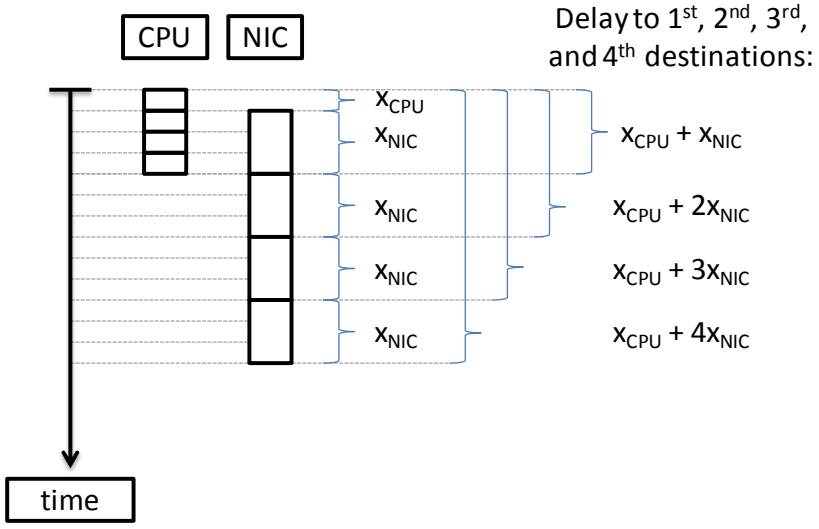

Figure 3-4. Non-blocking transmission of four messages in which the network card (left) and the CPU (right) is the bottleneck.

the delay is equal to the time the network card requires to transmit the command to $pos(\pi_k, \pi_{k+1})$ destinations plus the time the CPU requires to queue the message for a single transmission, as shown in Figure 3-4.

We can generalize the delay to the first and non-first destinations as follows:

$$int_{REP}(i, \pi_k) = x_{CPU}{}^{IN}_{i,\pi_k} + pos(\pi_k, \pi_{k+1}) * x_{NIC}{}^{IN}_{i,\pi_k}$$

The delay of the destination computer π_m also depends on the number of computers to which it forwards commands because it must first transmit the command to all of them before processing the input command and its output. Unlike for intermediate computers, where the delay depended on the network card, in this case, the delay depends only on the CPU. The reason is that once the CPU queues messages in the network card's transmission queue, the work done by the network card does not take any CPU time. Let p_{i,π_m}^{IN} (p_{i,π_m}^{OUT}) denote the time computer π_m requires to process input (output) command i, and let fan_{π_m} denote the number of destinations to which the computer forwards commands, that is, the fan out from the computer. Thus, the delay of the destination computer equals

$$dest_{REP}(i,\pi_m) = fan_{\pi_m} * x_{CPU}{}_{i,\pi_m}^{IN} + p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}$$

Replicated Local Response Time

So far, we have presented only the equations for the replicated remote response times. We next present the equations for replicated local response times for commands entered by a master user. Recall that the local response time is the time that elapses from the moment a user enters an input command to the moment the user sees the output for the command, which is equivalent to the time that elapses from the moment the inputting user's computer receives the command to the moment the computer completes processing the output of the command. Therefore, the local response time is exactly the delay of the destination computer defined above. This makes sense because the inputting user's computer is both the source and the destination. Thus, the local response time equations for command i entered by user$_j$ are given by

$$local_{REP,i,j} = dest_{REP}(i,j) = fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT}$$

Centralized Architecture

The equations we have presented have considered the case in which the processing architecture is replicated. As a result, each input command is entered by a master user. Let us next consider the centralized architecture, with both master and slave commands.

We can obtain the centralized architecture equations for commands entered by master users from the above replicated architecture equations by adjusting them for the two main differences in the two architectures. First, in the centralized architecture, only the master computer processes input commands, while all computers process output commands. Therefore, when calculating the delays of the computers on the path from the source to the destination, the processing times in the delays are equal to the time needed to process only output commands. Second, instead of transmitting input commands, the computers transmit output commands. Based on these two differences, the centralized architecture general remote response time equation is given by

$$remote_{CENT,i,j} = p_{i,\pi_1}^{IN} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{CENT}(i, \pi_k) + dest_{CENT}(i, \pi_m)$$

The first term, p_{i,π_1}^{IN}, for which there is no equivalent in the replicated case, accounts for the fact that the master computer must process the input command before it can send the output. The second term$\sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1})$,is the exactly the same as in the replicated case. The last two terms are the delays created by the computers on the path from a source to a destination. The centralized delay on an intermediate computer π_k is

$$int_{CENT}(\pi_k) = x_{CPU}{}_{i,\pi_k}^{OUT} + \sigma(\pi_k, \pi_{k+1}) * x_{NIC}{}_{i,\pi_k}^{OUT}$$

And, similarly, the centralized delay on the destination computer is

$$dest_{CENT}(\pi_m) = fan_{\pi_m} * x_{CPU}{}_{i,\pi_m}^{OUT} + p_{i,\pi_m}^{OUT}$$

These equations capture the fact that in the centralized architecture, the destination and intermediate computers do not process input commands. Moreover, they transmit outputs instead of inputs.

The centralized architecture local response time equation can be derived in a similar manner. Since, the master user's computer is both the source and the destination, the local response time is given by

$$local_{CENT,i,j} = p_{i,j}^{IN} + dest_{CENT}(i,j)$$

As was the case for the remote response time equation, the first term in the local response time equation, $p_{i,j}^{IN}$, accounts for fact that the master must accounts for the fact that the master computer must process the input command before it can send and process the output.

We can also obtain the equations for input commands entered by slave users by morphing the above equations for input commands entered by master users. The only difference between the two kinds of input commands is that a command entered by a slave must first reach the master computer. Once the command reaches the master, the problem reduces to that of calculating the remote response time from the master to the slave, which we have already done above. The time the command takes to reach the master computer is equal to the time the slave computer requires to transmit the command to a single destination (i.e. the master) plus the time the command takes to traverse the network between the slave and master computers. Therefore, we can obtain the equations for the local and remote response time of command i entered by slave user_a whose master is user_b by adding the term $x_{CPU}{}_{i,a}^{IN} + x_{NIC}{}_{i,a}^{IN} + d(a,b)$ to the response time equations.

3.3.2 Implications for Quiescent State Commands

The analysis above helps us better understand the nature of multicast and how it differs from unicast. It also helps us formally confirm intuitive expectations and, more interesting, derive some unintuitive results about multicast.

Local Response Times

So far, our motivation for using multicast has been to reduce remote response times. Our equations predict that multicast can also improve local response times. Consider first the centralized architecture local response times of a master user. The difference in the local response times with and without multicast is given by

$$local_{CENT,i,j}^{unicast} - local_{CENT,i,j}^{multicast} = p_{i,j}^{IN} + dest_{CENT}^{unicast}(i,j) - \left(p_{i,j}^{IN} + dest_{CENT}^{multicast}(i,j)\right)$$

$$= dest_{CENT}^{unicas\ t}(i,j) - dest_{CENT}^{multicast}(i,j)$$

$$= \left(fan_j^{unicast} - fan_j^{multicast}\right) * x_{CPU}{}_{i,j}^{OUT}$$

The number of destinations to which a master transmits when unicast (multicast) is used, $fan_j^{unicast}$ ($fan_j^{multica\ st}$,), is equal (less than or equal) to the number of users minus onethe number of users minus one. Therefore, the above equation is always less than or equal to zero. As a result, the local response time of the master user in the centralized architecture is no worse with multicast than with unicast. In fact, since the nature of multicast is to distribute the transmission task among multiple computers, usually the number of computers to which the master transmits with unicast is larger than with multicast, that is, $fan_j^{multicast} < fan_j^{unicast}$. Thus, the local response time of a master user in a centralized architecture is expected to be better with multicast.

Similarly, the difference in the local response times with and without multicast for the replicated architecture is given by the following equation

$$local_{REP,i,j}^{unicast} - local_{REP,i,j}^{multicast} = dest_{REP}^{unicast}(i,j) - dest_{REP}^{unicast}(i,j)$$

$$= \left(fan_j^{unicast} - fan_j^{multicast}\right) * x_{CPU}{}_{i,j}^{IN}$$

As with the centralized case, $fan_j^{multicast} \leq fan_j^{unicast}$. Moreover, usually, $fan_j^{multicast} < fan_j^{unicast}$. Therefore, the local response time of a master user in the replicated architecture should be lower with multicast than with unicast.

To illustrate the difference equations, consider user$_1$'s local response time in the replicated-unicast and replicated-multicast architectures shown in Figure 3-1 (bottom) and Figure 3-2 (bottom), respectively, with the following additional properties: (a) with unicast, user$_1$'s computer transmits commands first to user$_4$, then to user$_2$, user$_3$, user$_5$, and finally user$_6$; (b) with multicast, user$_1$'s computer transmits commands first to user$_4$, then to user$_2$, and finally to user$_3$, while user$_4$'s computer forwards the commands first to user$_5$ and then to user$_6$; (c) user$_1$ enters all of the commands; and (d) the users all have the same computers. Whenever we use the example, we omit the user index terms from the transmission and processing parameters because we assume that all of the computers are the same.

The local response time for user$_1$ is given by $local_{REP,i,1} = fan_1 * x_{CPU}{}_i^{IN} + p_i^{IN} + p_i^{OUT}$. When unicast is used, user$_1$'s computer transmits the command to five destinations. When multicast is used, it transmits commands to only three destinations. Therefore, the unicast local response time is $2 * x_{CPU}{}_i^{IN}$ higher than the multicast local response time, which agrees with the difference equation, $\left(fan_1^{unicast} - fan_1^{multicast}\right) * x_{CPU}{}_i^{IN} = (5-3) * x_{CPU}{}_i^{IN} = 2 * x_{CPU}{}_i^{IN}$.

Remote Response Times

As the above analysis shows, our model shows that multicast can only improve response times. On the other hand, our model predicts that multicast may improve or degrade remote response times. Consider the remote response time of slave user, b, in the centralized architecture for a command i entered by master j. The difference in the remote response times with and without multicast is given by the following equation

$$remote_{CENT,i,j}^{unicast} - remote_{CENT,i,j}^{multicast}$$

$$= \left(p_{i,j}^{IN} + d(j,b) + int_{CENT}^{unicast}\ (i,j) + dest_{CENT}^{unicast}\ (i,b)\right)$$

$$- \left(p_{i,j}^{IN} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{CENT}^{multicast}\ (i,\pi_k) + dest_{CENT}^{multicast}\ (i,\pi_m)\right)$$

$$= \left(d(j,b) + int_{CENT}^{unicast}\ (i,j) + dest_{CENT}^{unicast}\ (i,b)\right)$$

$$- \left(\sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{CENT}^{multic\ ast}(i,\pi_k) + dest_{CENT}^{multicast}\ (i,\pi_m)\right)$$

The first and second brackets give the remote response times with unicast and multicast, respectively. The first sub-term in each bracket gives the network latency delays an output experiences as it travels from the source to the destination. With unicast, it includes only the delay from the source to the destination. With multicast, on the other hand, it includes all of the network latencies on the path from the source to the destination in the multicast overlay. The second sub-term in each equation gives the total delay an output experiences on the intermediate computers as it travels from the source to the destination. With unicast, it experiences intermediate delays only on the source computer. With multicast, it experiences intermediate delays on all computers on the path from the source to the destination other than the destination computer. The two sub-terms combined give the time at

which the output reaches the destination. By design, the HMDM scheme reduces this time compared to unicast, that is,

$$d(j,b) + int_{CENT}^{unicast}\ (i,j) < \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{CENT}^{multicast}\ (i, \pi_k)$$

The third and final sub-term gives the delay the output experiences at the destination computer. With unicast, a destination never has to forward commands to other computers. Thus, the delay includes only the time the destination computer requires to process the output and is equal to $p_{i,b}^{OUT}$. With multicast, on the other hand, each destination may have to forward outputs to other computers. Thus, the delay includes the time the destination's CPU requires to transmit the output plus the time it takes to process the output, which is equal to $fan_b * x_{CPU}{}_{i,b}^{OUT} + p_{i,b}^{OUT}$.

Comparing all of the sub-terms in above difference equation shows that if the time CPU on the destination computer requires to forward the output with multicast, $fan_b^{multicast} * x_{CPU}{}_{i,b}^{OUT}$, is greater than the reduction in the time at which the output reaches the destination with multicast compared to unicast, then multicast increases remote response times. Otherwise, multicast decreases remote response times. This analysis illustrates how the that fact that multicast algorithms, such as HMDM, do not consider collaboration specific parameters can lead increases in response times when multicast is used. In particular, because they do not consider neither the order in which a computer performs the processing and transmission tasks nor the processing costs, multicast algorithms can increase remote response times in collaborative systems.

The same reasoning can be applied to the replicated remote response times. Therefore, we simply state the replicated remote response time difference equation with unicast and multicast.

$$remote_{REP,i,j}^{unicast} - remote_{REP,i,j}^{multicast}$$

$$= \left(d(j,b) + int_{REP}^{unicast}\ (i,j) + dest_{REP}^{unicast}\ (i,b)\right)$$

$$-\left(\sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{REP}^{multi\ cast}\ (i,\pi_k) + dest_{REP}^{multicast}\ (i,\pi_m)\right)$$

To illustrate both how multicast can improve and degrade remote response times compared to unicast, consider $user_4$'s and $user_6$'s remote response times in Figure 3-1 (bottom) and Figure 3-2 (bottom). The reason we consider $user_4$ and $user_6$ instead of other users is because $user_4$ is the only non-source intermediate computer in our example and $user_6$ is the "farther" from the source than any other user. As Figure 3-2 (bottom) shows, the path from $user_1$ to $user_6$ is longer than the path from $user_1$ to any other user, except $user_5$. The paths from $user_1$ to $user_5$ and $user_6$ both go through $user_4$. Since $user_4$ transmits first to $user_5$ and then to $user_6$, we consider $user_6$ to be farther away than $user_5$ is from $user_1$.

Consider the remote response time of $user_6$. When unicast is used, the path π from $user_1$ to $user_6$ computer is of length m=2 and π_1 and π_2 are $user_1$'s and $user_6$'s computers, respectively. According to our replicated architecture intermediate delay equation, $user_1$'s computer's delay is equal to the time it requires to transmit the command to five destinations, ${x_{CPU}}_i^{OUT} + 5 * {x_{NIC}}_i^{OUT}$, (since it transmits to $user_6$'s computer last). According to our destination delay equation, $user_6$'s computer's delay is equal to the time it requires to process the input and the corresponding output command, $p_i^{IN} + p_i^{OUT}$. The total network delay is equal to the time the command requires to traverse the network from $user_1$ to $user_6$, $d(1,6)$.

Thus, user$_6$'s unicast remote response time is equal to $d(1,6) + {x_{CPU}}_i^{OUT} + 5 * {x_{NIC}}_i^{OUT} + p_i^{IN} + p_i^{OUT}$. When multicast is used, the path π from user$_1$ to user$_6$ is of length m=3 and π_1, π_2, and π_3 are user$_1$'s, user$_4$'s, and user$_6$'s computers, respectively. According to our model, user$_1$'s computer's delay is equal to the time it requires to transmit the command to a single destination, ${x_{CPU}}_i^{OUT} + {x_{NIC}}_i^{OUT}$, since it transmits to user$_4$ first. Similarly, user$_4$'s computer's delay is equal to the time it requires to transmit the command to two destinations, ${x_{CPU}}_i^{OUT} + 2 * {x_{NIC}}_i^{OUT}$, since it transmits to user$_6$ last. User$_6$'s computer's delay is equal to the time the computer requires to process the input and the corresponding output command, $p_i^{IN} + p_i^{OUT}$. Finally, the network delay is equal to the amount of time the command requires to traverse the network from user$_1$ to user$_4$ and then from user$_4$ to user$_6$, $d(1,4) + d(4,6)$. Since all of the users have the same computers, user$_6$'s remote response time is equal to $d(1,4) + d(4,6) + {x_{CPU}}_i^{OUT} + {x_{NIC}}_i^{OUT} + {x_{CPU}}_i^{OUT} + 2 * {x_{NIC}}_i^{OUT} + p_i^{IN} + p_i^{OUT} = d(1,4) + d(4,6) + 2 * {x_{CPU}}_i^{OUT} + 3 * {x_{NIC}}_i^{OUT} + p_i^{IN} + p_i^{OUT}$. Subtracting the unicast response times from those of multicast gives $d(1,4) + d(4,6) - d(1,6) + {x_{CPU}}_{i,1}^{OUT} - 2 * {x_{NIC}}_{i,1}^{OUT}$. Thus, if the difference in the multicast and unicast network delays plus the time the CPU requires to transmit to a single destination, $d(1,4) + d(4,6) - d(1,6) + {x_{CPU}}_i^{OUT}$, is greater than time the network card requires to transmit the command to two destinations, $2 * {x_{NIC}}_i^{OUT}$, then user$_6$'s response times are better with unicast than with multicast, and vice versa.

Consider now the remote response time of user$_4$. When unicast is used, the path π from user$_1$ to user$_4$ computer is of length m=2 and π_1 and π_2 are user$_1$'s and user$_4$'s computers, respectively. According to our model, user$_1$'s computer's delay is equal to the

time it requires to transmit the command to a single destination, $x_{CPU}{}_i^{OUT} + x_{NIC}{}_i^{OUT}$. Also, user$_4$'s computer's delay is equal to the time it requires to process the command, $p_i^{IN} + p_i^{OUT}$. The total network delay is equal to the time the command requires to traverse the network from user$_1$ to user$_4$, $d(1,4)$. Thus, user$_4$'s unicast remote response time is equal to $d(1,4) + x_{CPU}{}_i^{OUT} + x_{NIC}{}_i^{OUT} + p_i^{IN} + p_i^{OUT}$. When multicast is used, the path π from user$_1$ to user$_4$ is of length m=2 and π_1 and π_2 are user$_1$'s and user$_4$'s computers, respectively. According to our model, user$_1$'s computer's delay is the same as with unicast. It is equal to the time it requires to transmit the command to a single destination, $x_{CPU}{}_i^{OUT} + x_{NIC}{}_i^{OUT}$. User$_4$'s computer's delay is different than with unicast, however. Based on our equations, the delay with multicast is equal to the time it requires to transmit the command two destinations, $x_{CPU}{}_i^{OUT} + 2 * x_{NIC}{}_i^{OUT}$, plus the time it requires to process the command, $p_i^{IN} + p_i^{OUT}$. The total network delay is equal to the time the command requires to traverse the network from user$_1$ to user$_4$, $d(1,4)$. Thus, user$_4$'s multicast remote response time is equal to $d(1,4) + x_{CPU}{}_i^{OUT} + x_{NIC}{}_i^{OUT} + x_{CPU}{}_i^{OUT} + 2 * x_{NIC}{}_i^{OUT} + p_i^{IN} + p_i^{OUT}$. Thus, user$_4$'s multicast remote response times are $x_{CPU}{}_i^{OUT} + 2 * x_{NIC}{}_i^{OUT}$ higher than those of unicast. This analysis illustrates how collaboration-specific factors, such as the processing time of commands, can actually make remote response times for a user higher with multicast than with unicast.

3.3.3 Consecutive Commands by a Single User

So far, we have assumed that a user enters a command when the system is in a quiescent state. In this section, we relax this assumption by considering the case when a user enters a command before all users' devices have completed the processing and transmission tasks of all commands entered previously by the user. Thus, we are considering the case

when think times are low or there is type-ahead and only one user is entering commands. In this case, a command may arrive at a computer early, that is, before the computer has completed the processing and transmission tasks of a previous command. When this happens, the new command must wait until the computer completes the transmission and processing task of the previous commands. Hence, the analysis presented so far does not apply because a command can be delayed on a user's computer for longer than the time accounted for in the above response time equations. We next present an analysis that accounts for this additional delay. We present first the analysis for input commands entered by a master user in the replicated architecture.

Replicated Local Response Time

Consider first the local response times of commands entered by a master user. The response time equation for the first command i entered by master j user when the system is in a quiescent state is repeated from above

$$local_{REP,i,j} = fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT}$$

Suppose that user j enters command $i+1$ t_{i+1} time after entering command i and that $t_{i+1} < fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT}$. Therefore, after command $i+1$ arrives, the CPU on user$_j$'s computer still requires $fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT} - t_{i+1}$ time to complete the transmission and processing tasks of command i. The response time of command $i+1$ includes this additional delay, and is given by

$$local_{REP,i+1,j} = fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT} - t_{i+1} + fan_j * x_{CPU}{}_{i+1,j}^{IN} + p_{i+1,j}^{IN} + p_{i+1,j}^{OUT}$$

Suppose that user j enters command $i+2$ t_{i+2} time after entering command i. If the command is entered after user$_j$'s computer completes all tasks for command i, this is the same case as for the local response time of command $i+1$. If, on the other hand, user$_j$'s

computer is still carrying out tasks for command i, then the local response time of command $i+2$ includes the time the computer requires to complete the tasks for command i, $fan_j * {x_{CPU}}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT} - t_{i+2}$, plus the time it requires to complete the tasks for command $i+1$, $fan_j * {x_{CPU}}_{i+1,j}^{IN} + p_{i+1,j}^{IN} + p_{i+1,j}^{OUT}$, plus the time it requires to perform the task for command $i+2$, $fan_j * {x_{CPU}}_{i+2,j}^{IN} + p_{i+2,j}^{IN} + p_{i+2,j}^{OUT}$. More generally, the local response time of command $i+z$ entered t_{i+z} time after the CPU on user$_j$'s computer begins transmitting command i but before it completes processing command i is given by

$$local_{REP,i+z,j} = \sum_{c=i}^{i+z} \left(fan_j * {x_{CPU}}_{c,j}^{IN} + p_{c,j}^{IN} + p_{c,j}^{OUT} \right) - t_{i+z}$$

As described above, user$_j$'s computer's destination delay for command i entered in a quiescent state is equal to $fan_j * {x_{CPU}}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT}$. Therefore, the above equation can be stated equivalently as

$$local_{REP,i+z,j} = \sum_{c=i}^{i+z} dest(c,j) - t_{i+z}$$

Replicated Remote Response Times

The remote response time analysis is more complicated. The reason is that a command may arrive early at more than one computer from the path to the destination. In particular, even if user$_j$ enters command $i+1$ after the computer j completed the tasks for command i, an intermediate or the destination computer may not have completed tasks for command i by the time command $i+1$ reaches it. To illustrate how this can happen, consider the remote response times of user$_b$ for command $i+z$ entered by user$_j$. Let computer π_s, the s^{th} computer on the path from j to b, be the computer on the path that

takes the longest to perform tasks of commands. We call π_s the critical computer on the path. If π_s is idle when command $i + z$ reaches it, then all computers must be idle when the command reaches them. Therefore, response times reduce to those of commands entered in a quiescent state. Thus, we consider the case when π_s is busy performing tasks for previous commands when command $i + z$ reaches it. Let command i entered by user_j be the last

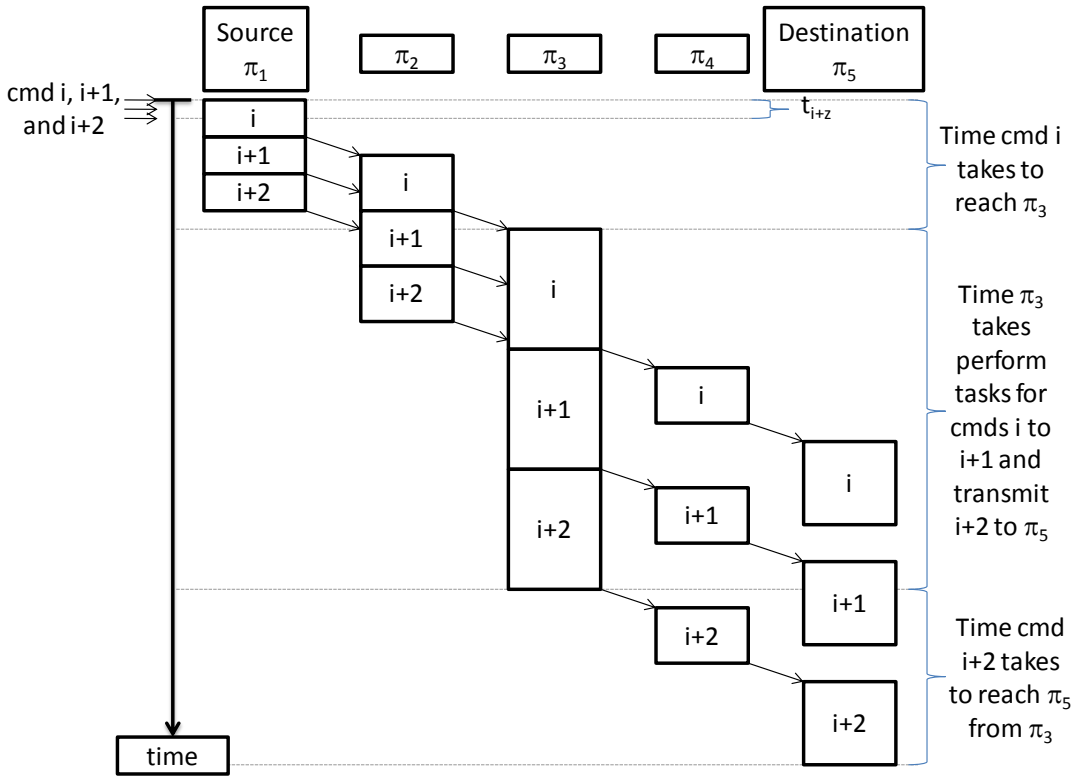

Figure 3-5. Critical computer on path from source to destination.

command that π_s received when it was in an idle state. Since it was idle when command i reached it, then all computers were idle when the command reached them; otherwise, π_s would not be the critical computer on the path. Moreover, command $i + z$ reaches all computers downstream from π_s when they are in an idle state; otherwise, again, π_s would not be the critical computer on the path.

We illustrate the notion of a critical computer through an example of a path of length five shown in Figure 3-5. As Figure 3-5 shows, three commands, i, $i + 1$, and $i + 2$, are entered consecutively. The total transmission and processing time for each for each command on the five computers is shown by the sizes of the boxes in the diagram. The total times spent on a command by each computer ordered from lowest to highest are $\pi_1 < \pi_2 = \pi_4 < \pi_5 < \pi_3$. As Figure 3-5 shows, the source user inputs commands quicker than the source computer π_1 can complete the transmission and processing tasks. Hence, commands arrive early at the source computer. Since π_2, the next computer on the path, takes longer to complete the tasks for each command, they arrive early also at π_2. Since π_3 is takes even longer than π_2 the second computer, they also arrive early at π_3. However, they do not arrive early at π_4 and π_5, the destination computer. The reason is that these computers are able to complete the tasks for each command faster than π_3 the third computer. Hence, in this case, π_3 is the critical computer.

We can use the notion a critical computer to derive the remote response time of command $i + z$. The critical computer may be an intermediate computer or the destination computer. When it is an intermediate (destination) computer, then user_b's remote response time of command $i + z$ is equal to 1) the amount of time that elapses from the moment command i is entered to the moment it reaches the critical computer, 2) plus the amount of time that elapses from the moment the critical computer receives command i to the it transmits command $i + z$ to the next computer on the path (completes processing of command $i + z$), 3) plus the amount of time that elapses from the moment the critical computer transmits command $i + z$ to the next downstream computer to the moment user_b sees its output, 4) minus the amount of time that elapses from the moment command i is

entered to the moment command $i + z$ is entered. The case when an intermediate computer other than the source is the critical computer is illustrated in Figure 3-5. Since the critical computer is idle when command i reaches it, then by the definition of the critical computer, all computers are idle when command i reaches them, as shown in Figure 3-5. Thus, the first is term is equal to the sum of network latencies on the part of the path from the source to the critical computer plus the intermediate delays for command i of computers on this part of the path, that is, $\sum_{k=1}^{s-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int_{REP}(i, \pi_k)$. Note that if the critical computer is the same as the source computer, then the first term is equal to zero. The second term is more complicated and we return to it momentarily. The third term is similar to the first term. By the definition of the critical computer, all computers downstream from the critical computer are idle when command $i + z$ reaches them, as shown in Figure 3-5. Thus, the third term is equal to sum of the network latencies along the part of the path from the critical to the destination computer, plus the intermediate delays for command $i + z$ of all non-destination computers on this part of the path, plus the delay command $i + z$ on the destination computer, that is, $\sum_{k=s}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=s+1}^{m-1} int_{REP}(i + z, \pi_k) + dest_{REP}(i + z, \pi_m)$. Note that if the critical computer is the same as the destination computer, then the third term is equal to zero. The fourth term is a constant. Therefore, the first, third, and fourth terms are independent of the number of consecutive commands that have been entered.

The second term, which we refer to as the critical computer time, is more complex than the others because it depends on the number of commands that have been entered consecutively. When the critical computer is an intermediate computer, then the time it requires to complete the tasks of consecutive commands is a function of both the network card and CPU costs. It is a function of CPU costs because until the CPU completes tasks for

a command, the network card cannot begin transmitting the next command even if it is idle because the CPU must first transmit it. It is a also function of transmission times because until the network card completes transmitting a command to all destinations, it cannot begin transmitting the next command even if the CPU has completed the processing and transmission tasks for it. In one extreme case, the total network card transmission time is higher than the CPU processing time for each command. In this case, the network card falls

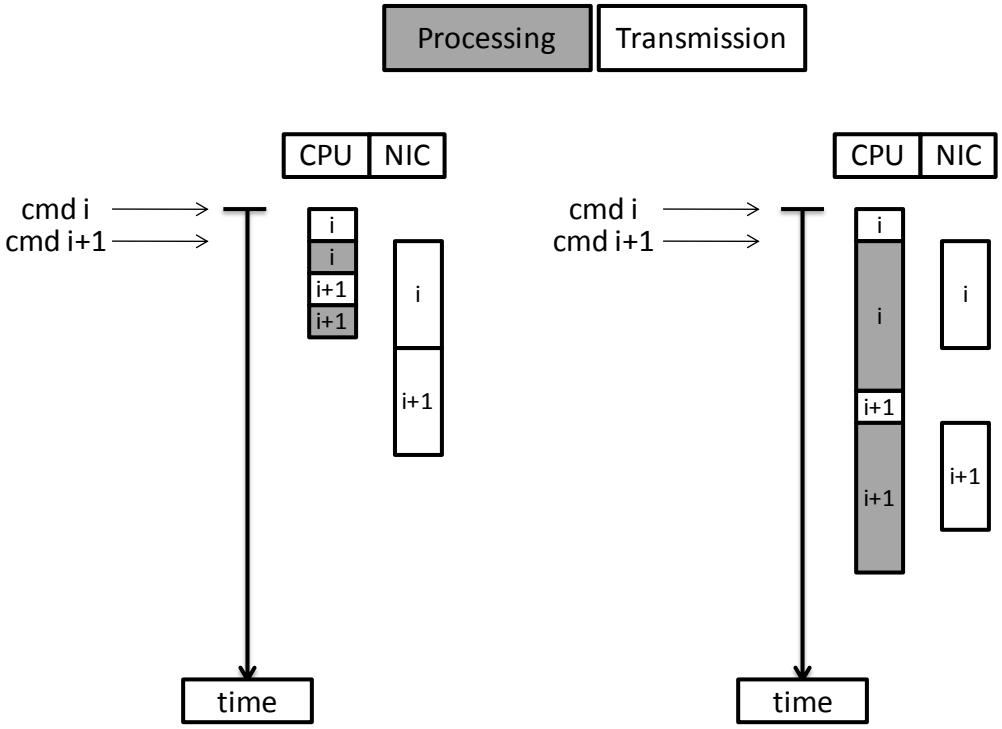

Figure 3-6. Critical computer time for completing tasks of consecutive commands.

further behind the CPU with each command, as shown in Figure 3-6 (left). The other extreme case is that the CPU processing time is higher than the total network card transmission time for each command. In this case, the network card keeps up with the CPU, as shown in Figure 3-6 (right). In all other cases, when neither the total network card transmission time nor the CPU processing time dominate each other, then for any one command, the network card either falls further behind or makes up ground on the CPU. Therefore, on average, the

network card keeps up with the CPU. Hence, all non-extreme cases are equivalent to the second extreme case. Thus, we analyze the response times only for the two extreme cases.

Let us first consider the extreme case when the total network card transmission times dominate the CPU processing times. The critical computer time is equal to (1) the amount of time that elapses from the moment the critical computer receives command i to the moment the CPU transmits i to the first destination, plus (2) the amount of time that elapses from the moment the network card begins to transmit i to the first destination to the moment it transmits $i + z - 1$ to the last destination, plus (3) the amount of time that elapses from the moment the network card begins transmitting $i + z$ to the moment it transmits it to the next downstream computer. The first term is equal to ${x_{CPU}}_{i,\pi_s}^{IN}$. The second term is equal to $\sum_{k=i}^{i+z-1}\left(fan_{\pi_s} * {x_{NIC}}_{k,\pi_s}^{IN}\right)$ because once the network card begins transmitting i, it does not stop until it finishes transmitting $i + z$. The third term is equal to $pos(\pi_s, \pi_{s+1}) * {x_{NIC}}_{i+z,\pi_s}^{IN}$.

Thus, we can finally state user_b's remote response time of command $i + z$ entered by user_j by adding the critical computer time to the other three components of the remote response time, which we had derived earlier. In particular, when an intermediate computer is the critical computer and the total network transmission times dominate the CPU processing time the remote response time is given by

$$remote_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int_{REP}(i, \pi_k)$$
$$+ {x_{CPU}}_{i,\pi_s}^{IN} + \sum_{c=i}^{i+z-1}\left(fan_{\pi_s} * {x_{NIC}}_{c,\pi_s}^{IN}\right) + pos(\pi_s, \pi_{s+1}) * {x_{NIC}}_{i+z,\pi_s}^{IN}$$
$$+ \sum_{k=s+1}^{m-1} int_{REP}(i+z, \pi_k) + dest_{REP}(i+z, \pi_m) - t_{i+z}$$

At first, this equation may appear quite different from the general remote response time equation for commands entered in a quiescent state. There are two differences in these two equations. The first difference is that there are two intermediate delay sums instead of one because in the consecutive command case, we have to separately consider the delays of computers upstream and downstream from the critical computer. The other, major difference is the critical computer time, $x_{CPU}{}_{i,\pi_s}^{IN} + \sum_{c=i}^{i+z-1}\left(fan_{\pi_s} * x_{NIC}{}_{c,\pi_s}^{IN}\right) + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}_{i+z,\pi_s}^{IN}$, that is included in the equation for consecutive commands. However, a closer look at the two differences shows that the consecutive command equation "stretches out" the delay of the intermediate computer π_s in the quiescent state equation. Therefore, the remote response time equation of consecutive commands indeed resembles the response time equation of commands entered in a quiescent state.

The other extreme case is when the CPU processing time dominates the total network card transmission for each command. In this case, the critical computer time is equal to (1) the amount of time that elapses from the moment the CPU begins to perform tasks for command i to the moment the CPU completes that task for command $i + z - 1$, plus (2) the amount of time that elapses from the moment the CPU begins to perform tasks for command $i + z$ to the moment the network transmits it to the next downstream computer. The first term is equal to $\sum_{c=i}^{i+z-1}\left(fan_{\pi_s} * x_{CPU}{}_{c,\pi_s}^{IN} + p_{c,\pi_s}^{IN} + p_{c,\pi_s}^{OUT}\right)$. The second term is equal to $x_{CPU}{}_{i+z,\pi_s}^{IN} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}_{i+z,\pi_s}^{IN}$. Thus, we can state user_b's remote response time of command $i + z$ entered by user_j when an intermediate computer is the critical computer and the CPU processing time dominate the total network transmission times as follows

$$remote_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int_{REP}(i, \pi_k)$$
$$+ \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}_{c,\pi_s}^{IN} + p_{c,\pi_s}^{IN} + p_{c,\pi_s}^{OUT} \right)$$
$$+ x_{CPU}{}_{i+z,\pi_s}^{IN} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}_{i+z,\pi_s}^{IN}$$
$$+ \sum_{k=s+1}^{m-1} int_{REP}(i+z, \pi_k) + dest_{REP}(i+z, \pi_m) - t_{i+z}$$

This equation, like the one before it, also resembles the general remote response time equation for commands entered in a quiescent state. The differences between this and the quiescent state equations are the same as the differences between the previous and quiescent state equations.

When the critical computer is the destination, the critical computer time is not a function of the network card transmission times. Hence, the second term on the destination computer is equal to the time the CPU requires to transmit and process all of the commands, that is, to $\sum_{c=i}^{i+z} \left(fan_{\pi_m} * x_{CPU}{}_{c,\pi_m}^{IN} + p_{c,\pi_m}^{IN} + p_{c,\pi_m}^{OUT} \right)$. Thus, when the destination computer is the critical computer, the response time equations are given by

$$remote_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{REP}(i, \pi_k)$$
$$+ \sum_{c=i}^{i+z} \left(fan_{\pi_m} * x_{CPU}{}_{c,\pi_m}^{IN} + p_{c,\pi_m}^{IN} + p_{c,\pi_m}^{OUT} \right) - t_{i+z}$$

This equation is similar the general remote response time equation of quiescent state commands. This makes sense because command i must traverse the path from the source to the destination regardless of whether the user enters consecutive commands after it. The only

difference is that the destination delay term is more complex in the consecutive command equation because of the extra delay command $i + z$ experiences at the destination.

Centralized Response Times

So far, we have presented only the response times of consecutive commands entered by master users in the replicated architecture. Next, we give the equations for the response times of such commands entered by master and slave users in the centralized architecture. Consider first the remote response times of commands entered by the master user. As in the analysis of commands entered in the quiescent state, we can derive these equations from the replicated architecture equations as long as we account for the two main differences in the centralized and replicated architectures. First, in the centralized architecture, only the master computer processes both input and output commands, while other computers process only output commands. Therefore, when calculating the delays of the computers on the path from the source to the destination, the processing times in the delays are equal to the time needed to process only output commands. Second, instead of transmitting input commands, the computers transmit output commands.

When the destination is the critical computer, the response time is given by

$$remote_{CENT,i+z,j} = p_{i,\pi_1}^{IN} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{CENT}(i, \pi_k) + \sum_{c=i}^{i+z} \left(fan_{\pi_m} * x_{CPU}{}_{c,\pi_m}^{OUT} + p_{c,\pi_m}^{OUT} \right) - t_{i+z}$$

The first term, p_{i,π_1}^{IN}, for which there is no equivalent in the replicated case, accounts for the fact that the master computer must process input command i before it can send its output. This processing time must be included in the amount of time that elapses from the moment command i is entered to the moment it reaches the critical computer, which as described in

the replicated discussion above, is the first component of the remote response time equation for command $i + z$. As mentioned above, the same term had to be included in the centralized equations for commands entered in a quiescent state.

Obtaining the centralized equations when the critical computer is not the destination is slightly more complicated. The reason is that when processing times dominate the total network card transmission times, we have to separately consider the source and non-source critical computer cases because the source computer processes inputs while other computers do not. If the total network card transmission times dominate the processing times, then it does not matter that the source computer performs extra processing (i.e. processes the input) than the other computers. Thus, in this case, the centralized response time equations can be obtained from the corresponding replicated equations by adjusting for the two differences between the centralized and replicated architectures. Thus,

$$remote_{REP,i+z,j} = p^{IN}_{i,\pi_1} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int_{CENT}(i, \pi_k)$$

$$+ x_{CPU}{}^{OUT}_{i,\pi_s} + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{NIC}{}^{OUT}_{c,\pi_s} \right) + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{OUT}_{i+z,\pi_s}$$

$$+ \sum_{k=s+1}^{m-1} int_{CENT}(i+z, \pi_k) + dest_{CENT}(i+z, \pi_m) - t_{i+z}$$

As above, the first term, p^{IN}_{i,π_1}, accounts for the fact that the master computer must process input command i before it can send its output, which contributes to the remote response time of command $i + z$.

However, when the processing times dominate the total network card transmission time, then we need separate equations for the case when the critical computer is or is not the source. When the critical computer is neither the source nor the destination, another

computer must exist on the path from the source to the destination, which implies multicast communication. It also implies that the critical computer is a slave. Hence, we obtain the centralized equation from the replicated equation by adjusting for the differences in the two architectures

$$remote_{CENT,i+z,j} = p_{i,\pi_1}^{IN} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int_{CENT}(i, \pi_k)$$
$$+ \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}_{c,\pi_s}^{OUT} + p_{c,\pi_s}^{OUT} \right) + x_{CPU}{}_{i+z,\pi_s}^{OUT} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}_{i+z,\pi_s}^{OUT}$$
$$+ \sum_{k=s+1}^{m-1} int_{CENT}(i+z, \pi_k) + dest_{CENT}(i+z, \pi_m) - t_{i+z}$$

On the other hand, when the critical computer is the source, then the critical computer is the master. Thus, we can obtain the centralized equation from the replicated equation by adjusting for the fact that outputs are transmitted instead of inputs. Hence,

$$remote_{CENT,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}_{c,\pi_s}^{OUT} + p_{c,\pi_s}^{IN} + p_{c,\pi_s}^{OUT} \right)$$
$$+ x_{CPU}{}_{i+z,\pi_s}^{OUT} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}_{i+z,\pi_s}^{OUT}$$
$$+ \sum_{k=s+1}^{m-1} int_{CENT}(i+z, \pi_k) + dest_{CENT}(i+z, \pi_m) - t_{i+z}$$

Unlike in the previous centralized architecture equations, the first term is not p_{i,π_1}^{IN}. The reason is that the time required to process the inputs is accounted for in, $\sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}_{k,\pi_s}^{OUT} + p_{c,\pi_s}^{IN} + p_{c,\pi_s}^{OUT} \right)$, in particular, the $\sum_{c=i}^{i+z-1} \left(p_{c,\pi_s}^{IN} \right)$ part of the sum. It must be a sum of all of the input processing times instead of just the processing time of input command i through $i+z$ because the critical computer is the master, and the master processes all input commands.

Obtaining the centralized local response time equation is simpler than deriving the remote response time equations. The reason is that there is only one case for the local equation. Moreover, we can derive it directly from the replicated local response time equation

$$local_{CENT,i+z,j} = \sum_{c=i}^{i+z} p_{i,\pi_1}^{IN} + \sum_{c=i}^{i+z} \left(fan_j * x_{CPU}{}_{c,j}^{OUT} + p_{c,j}^{OUT} \right) - t_{i+z}$$

$$= \sum_{c=i}^{i+z} p_{i,\pi_1}^{IN} + \sum_{c=i}^{i+z} dest_{CENT}(c,j) - t_{i+z}$$

As in the final remote response time case, the first term, $\sum_{c=i}^{i+z} p_{i,\pi_1}^{IN}$, accounts for the fact that the master computer must process all input commands in addition to outputs.

Finally, we can also obtain the equations for input commands entered by slave users by morphing the equations for input commands entered by master users. The only difference between the two kinds of input commands is that a command entered by a slave must first reach the master computer. Therefore, as was the case with the case with commands entered by slave commands in a quiescent state, we can obtain the equations for the local and remote response time of consecutive commands entered by slave $user_a$ whose master is $user_b$ by adding the term $x_{CPU}{}_{i,a}^{IN} + x_{NIC}{}_{i,a}^{IN} + d(a,b)$ to the response time equations.

3.3.4 Implications for Consecutive Commands by a Single User

We can compare the response times of consecutive commands with unicast and multicast by analyzing the difference in the respective equations. We can make two comparisons. We can compare the absolute response times of consecutive commands with unicast and multicast. We can also compare how much worse do response times of consecutive commands become with unicast and multicast. The result of either one

comparison can be used to derive the results for the other. In particular, we can obtain the former by adding the latter to the response times of commands entered in a quiescent state, and we can obtain the latter by subtracting the response times of commands entered in a quiescent state from the former. We do the former because the difference equations are cleaner: we have to consider only the terms in the equations that are a function of the number of consecutive commands.

Local Response Times

Let us start with the local response times of commands in the replicated architecture. The difference is given by

$$
\begin{aligned}
& local_{REP,i+z,j}^{unicast} - local_{REP,i+z,j}^{multicast} \\
& \quad = \left(\sum_{c=i}^{i+z} \left(fan_j^{unicast} * {x_{CPU}}_{c,j}^{IN} + p_{c,j}^{IN} + p_{c,j}^{OUT} \right) - t_{i+z} \right) \\
& \quad - \left(\sum_{c=i}^{i+z} \left(fan_j^{multicast} * {x_{CPU}}_{c,j}^{IN} + p_{c,j}^{IN} + p_{c,j}^{OUT} \right) - t_{i+z} \right) \\
& \quad = \sum_{c=i}^{i+z} \left(fan_j^{unicast} - fan_{\pi_j}^{multicast} \right) * {x_{CPU}}_{c,j}^{IN}
\end{aligned}
$$

Recall from above that the number of destinations to which the source computer transmits with multicast is less than or equal to that with unicast. For this reason, multicast was beneficial to the local response times of quiescent state commands. As the above difference equation shows, the benefit of multicast is additive for consecutive commands. In fact, with multicast, the local response times may not increase at all! For instance, if command $i+1$ is entered at time t_{i+1} and $fan_j^{multicast} * {x_{CPU}}_{c,j}^{IN} + p_{c,j}^{IN} + p_{c,j}^{OUT} \leq t_{i+1} < fan_j^{unicast} * {x_{CPU}}_{c,j}^{IN} + p_{c,j}^{IN} + p_{c,j}^{OUT}$, then with multicast, the local computer is ready to begin

carrying out tasks for command $i + 1$ as soon as the command is entered, while with unicast, it is not ready.

The equation for the centralized architecture difference for master and slave commands, when simplified, is

$$local_{CENT,i+z,j}^{unicast} - local_{CENT,i+z,j}^{multicast} = \sum_{c=i}^{i+z} \left(fan_j^{unicast} - fan_j^{multicast} \right) * x_{CPU}{}_{c,j}^{OUT}$$

The same reasoning applies for this difference as for the replicated difference. Hence, multicast provides the same benefit over unicast to local response times in the centralized and the replicated architectures.

Remote Response Times

We can also compare the remote response times of consecutive commands with unicast and multicast by analyzing the difference in the respective equations. The only terms in the equations that depend on the number of consecutive commands are those that give the critical computer time. Thus, as we will see, the difference in the equations boils down to the differences in the critical computer terms.

Let us start with the response times of commands in the replicated architecture. Consider first the case when the critical computer is not the destination computer. There are two cases to consider: when the total network transmission time for each command dominates the processing time of the command, and vice versa. Consider first the case when the network card transmission tasks are high and processing times are low. The difference in the response times is given by

$$remote^{unicast}_{REP,i+z,j} - remote^{multicast}_{REP,i+z,j} = \left(d(j,b) + {x_{CPU}}^{IN}_{i,j} + \sum_{c=i}^{i+z-1} \left(fan^{unicast}_{j} * {x_{NIC}}^{IN}_{c,j} \right) \right.$$

$$\left. + pos(j,b) * {x_{NIC}}^{IN}_{i+z,j} + p^{IN}_{i+z,b} + p^{OUT}_{i+z,b} - t_{i+z} \right) - \left(\sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) \right.$$

$$+ \sum_{k=1}^{s-1} int_{REP}(i,\pi_k) + {x_{CPU}}^{IN}_{i,\pi_s} + \sum_{c=i}^{i+z-1} \left(fan^{multicast}_{\pi_s} * {x_{NIC}}^{IN}_{c,\pi_s} \right)$$

$$\left. + pos(\pi_s, \pi_{s+1}) * {x_{NIC}}^{IN}_{i+z,\pi_s} + \sum_{k=s+1}^{m-1} int_{REP}(i+z,\pi_k) + dest_{REP}(i+z,\pi_k) - t_{i+z} \right)$$

In the difference equation, all of the terms except the $\sum_{c=i}^{i+z-1} \left(fan^{unicast}_{j} * {x_{NIC}}^{IN}_{c,j} \right)$ term in the first bracket and the $\sum_{c=i}^{i+z-1} \left(fan^{multicast}_{\pi_s} * {x_{NIC}}^{IN}_{c,\pi_s} \right)$ term in the second bracket are independent of the number of consecutive commands. Therefore, the difference equation for response times of consecutive commands reduces to

$$remote^{unicast}_{REP,i+z,j} - remote^{multicast}_{REP,i+z,j} = \sum_{c=i}^{i+z-1} \left(fan^{unicast}_{j} - fan^{multicast}_{\pi_s} \right) * {x_{NIC}}^{IN}_{c,j}$$

Since we are comparing non-destination computers, the unicast critical computer must be the source. With multicast, it can be the source or any other intermediate computer. If the multicast critical computer is not the source, the difference equation does not inform us of anything since we are comparing transmission times of different computers. Hence, we consider only the case when the multicast critical computer is the source. In this case, we can expect the difference to be positive. The reason is that multicast divides the transmission task across multiple computers, while with unicast, the source computer must transmit commands to all other computers, and thus, $fan^{multicast}_{\pi_s} \leq fan^{unicast}_{j}$. Therefore, the multicast remote response times increase less than those of unicast. Moreover, the benefit of multicast is

additive. As the number of consecutive commands increases, the multicast response times are increasingly better than those of unicast. One implication of this result is that deep multicast trees will have a larger additive benefit than shallow trees, which in turn will have larger benefits than unicast, which is effectively the shallowest multicast tree possible. The reason is that the transmission task is divided among more computers in deep than in shallow trees. Of course, deep multicast trees imply more network hops from the source to a destination computer, which can increase absolute remote response times.

When processing times are higher than the network card transmission times, the difference in unicast and multicast response times is given by

$$remote_{REP,i+z,j}^{unicast} - remote_{REP,i+z,j}^{multicast} = \left(d(j,b) + \sum_{c=i}^{i+z-1} \left(fan_j^{unicast} * {x_{CPU}}_{c,j}^{IN} + p_{c,j}^{IN} + p_{c,j}^{OUT} \right) \right.$$

$$\left. + {x_{CPU}}_{i+z,j}^{IN} + pos(j,b) * {x_{NIC}}_{i+z,j}^{IN} + p_{i+z,b}^{IN} + p_{i+z,b}^{OUT} - t_{i+z} \right)$$

$$- \left(\sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int_{REP}^{multicast}\ (i, \pi_k) \right.$$

$$+ \sum_{c=i}^{i+z-1} \left(fan_{\pi_s}^{multicast} * {x_{CPU}}_{c,\pi_s}^{IN} + p_{c,\pi_s}^{IN} + p_{c,\pi_s}^{OUT} \right) + {x_{CPU}}_{i+z,\pi_s}^{IN}$$

$$+ pos(\pi_s, \pi_{s+1}) * {x_{NIC}}_{i+z,\pi_s}^{IN} + \sum_{k=s+1}^{m-1} int_{REP}^{multicast}\ (i+z, \pi_k)\, z$$

$$\left. + dest_{REP}^{multicas\ t}(i+z, \pi_m) - t_{i+z} \right)$$

As was the case with the previous difference equation, there is only a single term in each of the bracketed terms that depends on the number of consecutive commands. Hence, the difference equation boils down to the difference in these terms

$$remote^{unicast}_{REP,i+z,j} - remote^{multicast}_{REP,i+z,j}$$

$$= \sum_{c=i}^{i+z-1} \left(fan^{unicast}_{j} * x_{CPU}{}^{IN}_{c,j} + p^{IN}_{c,j} + p^{OUT}_{c,j} \right) - \sum_{c=i}^{i+z-1} \left(fan^{multicast}_{\pi_s} * x_{CPU}{}^{IN}_{c,\pi_s} + p^{IN}_{c,\pi_s} + p^{OUT}_{c,\pi_s} \right)$$

As above, we can discuss the difference only when the multicast critical computer is the source. In this case, the difference equation further reduces to

$$remote^{unicast}_{REP,i+z,j} - remote^{multicast}_{REP,i+z,j} = \sum_{c=i}^{i+z-1} \left(fan^{unicast}_{j} - fan^{multicast}_{\pi_s} \right) * x_{CPU}{}^{IN}_{c,j}$$

Thus, the differences is greater than or equal to zero because with multicast the number of destinations to which the source computer transmits to is less than or equal to that with unicast. Hence, multicast remote response times can be better than those of unicast.

The final case to consider is when the destination computer is the both the multicast and unicast critical computer. The response time difference is given by

$$remote^{unicast}_{REP,i+z,j} - remote^{multicast}_{REP,i+z,j}$$

$$= \left(d(j,b) + x_{CPU}{}^{IN}_{i,j} + pos(j,b) * x_{NIC}{}^{IN}_{i,j} + \sum_{c=i}^{i+z} \left(p^{IN}_{c,b} + p^{OUT}_{c,b} \right) - t_{i+z} \right)$$

$$- \left(\sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int^{multicast}_{REP} (i, \pi_k) \right.$$

$$\left. + \sum_{c=i}^{i+z} \left(fan^{multicast}_{\pi_m} * x_{CPU}{}^{IN}_{c,\pi_m} + p^{IN}_{c,\pi_m} + p^{OUT}_{c,\pi_m} \right) - t_{i+z} \right)$$

Once again, there is only a single term in each of the bracketed terms that depends on the number of consecutive commands. Hence, the difference equation boils down to the difference in these terms

$$remote^{unicast}_{REP,i+z,j} - remote^{multicast}_{REP,i+z,j}$$

$$= \sum_{c=i}^{i+z} \left(p^{IN}_{c,b} + p^{OUT}_{c,b}\right) - \sum_{c=i}^{i+z} \left(fan^{multicast}_{\pi_m} * x_{CPU}{}^{IN}_{c,\pi_m} + p^{IN}_{c,\pi_m} + p^{OUT}_{c,\pi_m}\right)$$

This difference can never be positive. In fact, whenever the destination computer also forwards commands with multicast, the differences is negative. Hence, multicast remote response times are worse than those of unicast when the destination computer is the critical computer.

We can repeat the above replicated difference analysis for the centralized equations. As in the replicated case, each difference equation reduces to the differences in the unicast and multicast critical computer times. Consider first the case when both the multicast and unicast critical computer is the source. When the network card transmission tasks are high and processing times are low, the difference in the response times is given by

$$remote^{unicast}_{CENT,i+z,j} - remote^{multicast}_{CENT,i+z,j} = \sum_{c=i}^{i+z-1} \left(fan^{unicast}_{j} - fan^{multicast}_{\pi_s}\right) * x_{NIC}{}^{OUT}_{c,j}$$

Otherwise, the difference in response times is given by

$$remote^{unicast}_{CENT,i+z,j} - remote^{multicast}_{CENT,i+z,j} = \sum_{c=i}^{i+z-1} \left(fan^{unicast}_{j} - fan^{multicast}_{\pi_s}\right) * x_{CPU}{}^{OUT}_{c,j}$$

Finally, if both the unicast and multicast critical computers are the destination, the difference in response times is given by

$$remote^{unicast}_{CENT,i+z,j} - remote^{multicast}_{CENT,i+z,j} = \sum_{c=i}^{i+z} \left(p^{OUT}_{c,b}\right) - \sum_{c=i}^{i+z} \left(fan^{multicast}_{\pi_m} * x_{CPU}{}^{OUT}_{c,\pi_m} + p^{OUT}_{c,\pi_m}\right)$$

As we see, the centralized difference equations look the same as the replicated difference equations only adjusted for the two differences in the replicated and centralized architectures: in the centralized architecture, 1) only the master computer processes inputs and 2) the

computers distribute outputs instead of inputs. Therefore, the replicated architecture results hold for the centralized architecture.

In summary, when type-ahead is allowed or think times are low, the response time of each consecutive command from the same user can be worse than the response time of the previous command. With multicast, the amount of time the response time increases with each consecutive command is lower than with unicast. Moreover, in some cases, local response times with multicast may not increase at all even if they increase with unicast. The effect of multicast on the remote response times of these commands is less obvious. When the destination is the critical computer with both multicast and unicast, the remote response times increase more with multicast than with unicast. On the other hand, when the source is the critical computer with both multicast and unicast, we expect that the remote response times increase less with each consecutive command when multicast is used instead of unicast. However, when the multicast and unicast critical computers are not the same computer, the remote response times may increase more with either multicast or unicast.

3.3.5 Response Times of Simultaneous Commands

We have considered the remote response times of commands entered by the same user in a quiescent or a non-quiescent state. The final case to consider is the remote response times of simultaneous commands entered by different users.

When simultaneous commands are entered by different users, the worst possible remote response times for i command entered by user_j occurs when (a) it arrives at the critical computer at the same time as commands entered by other users, and (b) the critical computer performs the tasks for command i last. Therefore analysis of the remote response times of command i is almost the same as the analysis of the commands entered

consecutively by the same user. The only difference is that the critical computer term includes the time required to perform tasks of commands entered by all of the other users.

3.4 Self-Optimizing System Implementation

Recall from the previous chapter that we have developed a system that can select at start time or dynamically switch at runtime to a processing architecture that best meets user-provided response time requirements. We have extended that system to do the same for the communication architecture.

3.4.1 Gathering Parameter Values

To make decisions regarding the communication architecture, the system applies the analytical model of response times for large-scale scenarios. In order to apply the model, the system must collect values of all of the response time parameters identified by the model. The system described in the previous for optimizing the processing architecture collects some of these parameters, such as the network latencies and the input and output processing times. Therefore, to optimize the communication architecture, the system must also collect:

- Transmission times of input and output commands for each user's computer, including the CPU transmit times and network card transmit times
- Think times of all commands

CPU Transmission Times

As described in the previous chapter, our system gathers processing times by collecting performance reports that contain these times from each user's computer. More specifically, whenever a user's computer completes the transmission and processing tasks for a command, the client-side component of the optimization system sends a performance

report, which includes the time required to process the command, to the server-side component of the system. Our system uses the same mechanism to collect CPU transmission costs. Whenever a client-side component sends a report, the component includes the CPU transmission costs of the command in addition to the processing costs. When the server-side component receives a performance report, it looks up the number of destinations to which the user's computer transmitted the command. This information is available to our system since it controls the processing and communication architectures. Then, our system divides the transmission time reported by the computer by the number of destinations to which the computer transmitted to obtain the cost of transmitting the command to a single destination.

Network Card Transmission Times

Collecting network card transmission times is more complicated. In particular, when non-blocking communication is used, it is difficult to tell from the application layer when the network card actually completes transmitting a message. One approach is to use a network monitoring application, such as WireShark [91], to monitor packets as they leave the network card. Such applications can inform the client-side component of optimization system when the last packet of the message destined to a particular destination leaves the network card. Since the component is aware of when the CPU finished the transmission of the message to the destination, it can subtract this time from the time at which the last packet was transmitted as reported by the network monitoring application and get the network card's transmission time.

One issue with using a network monitoring application to compute network card transmission times is that running the network monitoring application together with the collaborative application can impact the timing information reported by the application. To

reduce the perturbation caused by measurements, the approach we take instead relies on the final destination to which a computer transmits to report to our system when it receives the message. Then, our system can subtract the time at which the final destination received the message from the time the network card at the source computer started transmitting the message to the first destination, divide that time by the number of destinations to which the source computer transmitted, and finally obtain the network card transmission time of the source computer.

For our approach to work, the three computers, the source, the final destination, and the one on which our optimization system is running, must have a common notion of time. One solution is 1) for the source computer to send a start transmission report to our system just before the transmission actually starts, 2) for the final destination to send a received transmission report as soon as it receives a command (before the destination begins to perform the processing and transmission tasks for the command), and 3) our system to timestamp each of these reports with the local timestamp as soon as these reports are received. Then our system could compute how much time the source computer's network card required to transmit the message to all destinations (assuming that our system can adjust for network latencies between the three computers). Unfortunately, this approach does not work with non-blocking communication because the start and receive transmission reports may be queued up behind other messages, such as transmissions of user commands. When such queuing occurs, our system cannot deduce when exactly the reports were sent.

An approach that does not have this problem is to synchronize the clocks of our system and the users' computers. With this approach, the source and the final destination computers record the local time at which the message is sent and received, respectively, and

include these times in the performance report that contains the processing and CPU transmission times. Since the clocks on all machines are synchronized, our system can subtract the start transmission time in the source's report from the receive transmission time in the destination's report to calculate how long the network card on the source computer required to transmit the command to all the destinations. For this approach to work, we need to synchronize the clocks.

In general, distributed clock synchronization has been solved to within ten milliseconds over the Internet and within a few milliseconds on a local LAN [65]. Unfortunately, the setup in our department does not allow us to use NTP on machines running Windows. Since all of our machines are Windows-based, we used a custom approach. Whenever a user registers with our system, the user's computer sends thirty messages to the server-side component of the optimization system containing the current value of the clock on the user's machine. The messages are sent at one hundred millisecond intervals. The server-side component timestamps each message as soon as it is received with the local timestamp. When all messages are received, our system 1) computes the average difference between the remote and local timestamps in the middle ten messages, and 2) records the final value as the difference between the local clock and the clock on the user's machine. The clocks are still not synchronized, however, because the difference calculation does not account for the network latencies the messages experience. To account for the latencies, as soon as the client-side component finishes sending the messages with the timestamp, it immediately measures network latency between itself and the computer on which the server-side component is running using the ping tool and then sends the measured value to the server-side component. When the server-side component receives the network

latency measurement, it adjusts the calculated clock difference between itself and the computer which sent the report to account for the latency. Based on our experience, the standard deviation of the timestamp differences for these ten messages is less than 2 milliseconds. Since 2ms is more than an order of magnitude smaller than 50ms, the value we consider significant, we consider the clocks synchronized when this approach is taken. This approach makes two assumptions. First, it assumes that the clock drift on each user's machine is the same for the duration of the collaborative session. Second, it assumes that the network latency value was the same during the period the timestamp messages are sent and the period immediately after during which the latency was measured. These assumptions were true in our experiments.

Think Times

The final parameter left to collect are the think times, which are gathered as follows. Whenever a user enters a command, the user's processing report for the command also includes its think time. Other computers do not report think times for it in their performance reports because it was not entered by their users. The system collects think times on a per user basis. Thus, whenever a user's processing report includes a think time value, the system adds the think time to the think time values reported so far by the user.

3.4.2 Applying the Analytical Model

Using the collected values of network latencies and processing and transmission costs, our system could apply the analytical model as follows. First, for each computer type, calculate the average processing and transmission times. Second, use the calculated values and the network latencies in the model equations to calculate the estimated response times of commands by each inputting user for the current processing architecture with and without

multicast. Third, invoke the user-defined total order function using the estimated response times for each system and the user identities as the parameters of the function. Fourth, switch to the processing and communication architecture that is ranked as the best by the total order function.

To illustrate the procedure, consider a three user scenario in which each user inputs commands. After estimating the values of the model parameters, our system crates the 2 x 3 x 3 response time matrix. The [1, y, z] and [2,y,z] entries in the matrix gives $user_y$'s response for a command entered by $user_z$ when unicast and multicast is used, respectively. Since there are three users in the session, the second dimension is of size three. It then invokes the total order function, passing the response time matrix, the list of inputting user indices sorted from lowest to highest, and a list of the identities of the users as parameters. When the total order function returns, it simply returns an index of the communication architecture that best satisfied the users' response time requirements.

Unfortunately, as described in the previous chapter, values of some parameters may never be reported. In the previous chapter, there was just one parameter for which our system may not have any reported values – the input processing times of slave computers. Now, input and output transmission times may not be reported for some computer types. In particular, our system will not receive any transmission related reports for any computer that does not forward messages. Therefore, as before, we have to estimate the missing parameter values.

Consider CPU output transmission costs and suppose that for some computer A, the CPU input transmission but not CPU output transmission reports were received. In this case, to estimate the CPU output transmission cost for computer A, our system checks if both CPU

input and output transmission reports have been received for any other computer. Such a computer may or may not exist. Consider first the case when such a computer exists. Let B denote the computer. First, the system uses the reports to calculate the CPU input and output transmission costs for computer B by averaging the values that have been reported. As described in the previous chapter, the system can use some number of the most recent reports or all of the reports in the average. Then, it computes the input-output ratio of costs for computer B. Finally, it calculates the input transmission cost of A, and divides the calculated value by B's ratio to estimate the CPU output transmission costs of A. In the case there is no computer for which both CPU input and output transmission reports have been received, the system calculates the input transmission cost for A and estimates that the CPU output transmission cost for A is the same.

When neither CPU input nor CPU output transmission costs have been received for A, a different procedure must be used. In this case, the system finds some computer C for which either the CPU output transmission have been reported or can be estimated using the above procedure. Note that some computer must have reported either CPU input or CPU output transmission costs; otherwise, there would be no collaborative session. The above procedure can be used to estimate whichever one is missing. Hence, computer C exists. The system then calculates the output processing costs for A and C. Output processing costs are reported by all machines so the calculation is possible. It then computes the A-C output processing cost ratio. It uses the ratio and the calculated or estimated value of the CPU output transmission costs for C to estimate the CPU output transmission costs for A.

The procedure used to estimate missing CPU transmission costs can be repeated for missing network card transmission costs. Moreover, in the previous chapter, we have

described a procedure for estimating missing processing costs. Hence, the system can estimate any missing processing or transmission costs. Whether or not these estimates are correct is a separate issue. As mentioned earlier, if they lead to an architecture switch that degrades response times, the degradation is temporary because the system will eventually receive true values for the parameters it had to estimate. Therefore, the system will eventually choose the optimal processing and communication architecture that the total order function ranks as the best.

3.4.3 Multiple Multicast Trees

An important performance issue when switching to a multicast is the number of multicast trees that are created. There are three reasonable choices for the number of trees. First, a tree can be created for every inputting user. This approach has the benefit of optimizing the communication architecture for each inputting user. When the number of inputting users is small this is the optimal approach. However, when many users can input commands, calculating and deploying multicast trees for all of them could take a long time. In particular, the HMDM algorithm runtime is $O((\text{number of users})^3)$. An alternative approach is to create a single multicast tree that is shared by all inputting users. Which user is at the root of the tree depends on the response time requirements. It could be rooted at the user who is inputting the most commands or the user who is the most important. Regardless of which user is the root of the multicast tree, the commands from all other inputting users must either be unicast or first have to travel from the inputting user's computer to the user's computer who is at the root of the tree. Thus, in this case, the response times of these commands may not be as good as when each inputting user has a dedicated multicast tree.

Yet another approach is to combine the first two approaches and deploy a separate multicast tree for each "important" user.

In general, the approach that should be used depends on the response time specifications. Our system deploys as many multicast trees as returned by the total order function. If the function does not return a multicast tree rooted at each inputting user, then it must specify whether the user's commands are to be unicast or the index of the user who is at the root of the multicast tree that should be used. In our experiments, we used total order functions that return a multicast tree for each inputting user since the number of inputting users was at most two. We need to modify the return value of total order function to support multiple multicast trees. For each inputting user, the function returns a pair of values. The first value indicates whether the user's input commands (the corresponding output commands) in the replicated (centralized) processing architecture should be unicast or multicast. If it indicates multicast, then the second value gives the index of the user who is at the root of the multicast tree that should be used. In a centralized processing architecture, this could only be the master computer.

One issue that arises when multiple multicast trees are deployed is that some computers may be intermediate computers in more than one tree. In this case, they decide which multicast tree to use based on the user who input the command.

3.4.4 Communication Architecture Switch Mechanism

Once the total order function returns the communication overlay that should be used, our system must deploy it. Recall from our discussion in the previous chapter that when our self-optimizing system switches processing architectures dynamically, it temporarily pauses the acceptance of new input commands. As a result, if users entered input commands during

the pause, the response times of these commands would be higher than usual. The same approach works for dynamically switching communication architectures. However, we use a more clever approach, one that does not require any pauses in acceptance of new input commands. Our system deploys the new communication architecture in the background using low priority threads on each user's computer. During the deployment, commands are distributed using the communication architecture that was used before the switch.

The communication architecture change is performed in two steps. First, our system sends to each computer the (a) destinations to which the computer will need to forward commands in the new communication architecture, and (b) the version number of that communication architecture. As mentioned above, each computer may be an intermediate computer on multiple multicast trees in which case it must know which tree to use for commands entered by each user. The destination data sent to each computer includes this information. Once this data is sent, our system then waits for each computer to report back that it has established the connections needed for the new communication architecture before it begins the second step. When all computers report back, it sends a command to each computer that switches the current communication architecture version number from the previous version to the new version.

With this dual stage approach, any commands entered during the communication architecture change are distributed using the old communication architecture. Eventually, all commands are distributed using the new communication architecture. After a user's computer receives commands only with the new version number for some time, it can close the connections required for the old communication architecture. The amount of time to wait can be configured in our system. The amount of time needs to be no higher than the largest

possible response time of a command. In our experience, response times are usually on the order of. Therefore, several minutes is likely sufficient. If the user's are unsure what value to use, they can configure our system to close the connections at the end of the session.

Another issue with our approach is how the users' computers decide whether to use the current or new communication architecture. To solve the problem, we tag each communication architecture with a version number. The version number of the initial communication architecture created at start time is zero and the version number of each new communication architecture deployed dynamically is incremented by one. Each user's computer is told the current communication architecture version, which is zero initially. When a computer receives an input command from the local user, it tags the input command with the current version of the communication architecture. Whenever a computer needs to forward a command to other computers, it first checks the version number with which the command is tagged and then it forwards the command according to the communication architecture with that version number.

3.4.5 Users Leaving and Joining

The final issue with communication architecture maintenance is how our system handles the joining and leaving of users in a collaborative session. When unicast is used for communication, the connections for a leaving (joining) user can simply be torn down (established) between the user's computer and the computers belonging to all other users. When multicast is used for communication, the process is more complicated, especially when a user leaves. To illustrate, suppose that a user whose computer is an intermediate computer on some path leaves. In this case, all computers downstream from the leaving user become

disconnected from upstream computers. To reconnect the users, our system calculates and deploys a new communication architecture. A joining user is handled in the same fashion.

3.5 Evaluation

We have presented an analytical model of response times for large-scale collaborations. Using the model, we have shown theoretical results about the benefit of multicast in collaborative systems. In addition, we have created a self-optimizing system that uses the model to automatically select the communication architecture that best meets response-time requirements. While these results are a contribution on their own, it is important to see whether or not (a) the theoretical differences given by the model can be significant in practical scenarios and (b) the self-optimizing system can better meet response time requirements than existing systems in these scenarios. Since the self-optimizing system applies the analytical model, we can accomplish both evaluation goals by checking whether or not the system significantly improves response times in practical scenarios.

We verify only the part of the model that predicts absolute response times for commands entered in a quiescent state. Similarly, we verify only the part of our system that automatically improves the response times of such commands. We do not evaluate the other parts of the model or the system – specifically the parts regarding the response time predictions of consecutive and simultaneous commands – because in our logs, commands were always entered in a quiescent state. This was somewhat surprising to us since our logs contained telepointer actions and these actions are generated at a much faster pace than Checkers, PowerPoint, and IM commands. But even these commands were always entered in a quiescent state. To illustrate, consider a telepointer motion. The motion will appear smooth

if the telepointer commands are generated and processed at a rate of thirty per second (or one every 33ms). We have found that on a P4 desktop, the CPU processing and network card transmission times of a telepointer command are 0.83 and 0.06ms, respectively. Thus, if the application generates a telepointer command every 33ms, then a P4 desktop has 32.17ms to perform the transmission task. When unicast is used, the command can be transmitted to as many as 536 destinations. When multicast is used, the number of users supported is even higher because each P4 computer that forwards commands can forward to as many as 536 destinations. Although the telepointer costs, as well as costs of other continuous motions (such as a drag-and-drop), are going to be different on other processors, we expect that they are still low. Thus, even telepointer actions are entered in a quiescent state in our logs. Of course, in some collaborations, users may actually enter consecutive and simultaneous commands. Therefore, an important future work direction for us is to verify our model and system for such application.

In general, in computer science, there are two possible ways to evaluate a system in a particular scenario: simulations and experiments. Simulations estimate the performance of a system in the scenario by using an analytical model of the system. Compared to a pure theoretical application of the model, which gives trends and implications, simulations provide quantitative theoretical performance results in practical scenarios. Experiments, on the other hand, measure the actual performance of the system while it is being used.

Whenever it is practical to do so, experiments should be used instead of simulations because simulations estimate performance while experiments measure actual performance. Unfortunately, it is not always practical to run experiments. There are two issues with running experiments. The first issue is repeatability. In some cases, it may be impossible to

Note:

Simulations estimate the performance of a system in a particular scenario by using an analytical model of the system. Compared to a pure theoretical application of the model, which gives trends and implications, simulations provide quantitative theoretical performance results in practical scenarios.

ensure the same conditions across experiments. For example, in network systems research, if an anomaly happens while measuring the impact of some network congestion algorithm on real Internet traffic, that anomaly may never happen again. Thus, if the algorithm is updated to handle the anomaly, it may be impossible to test the update. The other issue is the resources required to run experiments. For instance, network systems researchers may not have resource available to them to setup a large networking experiment with many routers and computers.

When it is not practical to run experiments, simulations are used instead. For instance, in order to test network management schemes, network researchers often rely on the popular network simulator (NS) application that can simulate the impact of these schemes on network traffic. In this case, it is important to validate that the analytical model used in the simulations accurately represents the system being evaluated. Otherwise, simulated performance may not reflect actual performance. Once the simulations are validated, they may in fact completely replace experiments even when experiments are practical. The reason is that in general it is easier and quicker to setup and run simulations than it is experiments.

It turns out that to evaluate the self-optimizing system, it is not practical for us to run experiments. In the discussion that follows, we describe why this is the case. Following this,

we explain how we use simulations instead. Finally, we show how we validated our simulations.

3.5.1 Experiments

To evaluate the self-optimizing system through experiments, we also have to handle the repeatability and resource issues that arise when running experiments. Consider the repeatability issue first. In ideal experiments, we would measure performance with and without our system under different collaboration conditions while users collaborate live. However, users cannot be counted on performing the same sequences of actions in different experiments. Thus, we do not have repeatability. As a result, accurate comparison of performance across experiments is not possible. For this reason, we replay logs of previously recorded actions as explained in Appendix A.

When we replay recorded logs, ideally, each user in the experiment is running on a separate machine. The reason is that running multiple instances of the shared application on a single machine 1) is unrealistic, given that users no longer share time-sharing systems, and 2) can affect the timing information due to the operating system's scheduling of tasks. Since we are focusing on large-scale collaborations, we would require a large number of computers. Unfortunately, the total number of computers we have available for an experiment is ten. Hence, it is not possible for us to run ideal experiments with hundreds or thousands of users on our computers. Instead of using our computers, we could use computers in a public cluster such as PlanetLab [69]. It is a network of several hundred computers distributed around the world that can be used by researchers for free. However, there are several reasons why we do not use it. One reason is that a PlanetLab machine can be used by multiple researchers at the same time. Since we are collecting timing information, it would be difficult to understand

results because the machines we use may be overloaded with tasks other researchers are performing. Another reason we do not use PlanetLab is because it does not provide any control over the computers used in a particular experiment – it chooses them at random. Therefore, it would be difficult to perform multiple experiments in which we vary the processing and communication architecture only, which is necessary to measure their impact on response times. Also, PlanetLab machines are Linux-based and we are restricted to Windows-based computers because Windows the only platform on which we know how to interact programmatically with PowerPoint. Another cluster of machines is Amazon's EC2 service [5]. It suffers from the same issues as PlanetLab and in addition, it is not free. Therefore, we have to use our equipment, which means that we must run simulations instead of experiments.

3.5.2 Simulations

As mentioned above, from a performance perspective, multicast has been used to reduce end-to-end delays. Also as mentioned above, in collaboration systems, end-to-end delays are related to remote response times. Therefore, we expect that in at least some practical scenarios, the self-optimizing system can improve remote response times by automating the maintenance of the communication architecture. To theoretically verify that the self-optimizing system can improve response times in a practical scenario, we consider a PowerPoint scenario in which the presentation is being given to 100, 200, 300, 400, and 500 audience members around the world.

The self optimizing system decides which communication architecture to use for this scenario in five steps. First, it gathers the parameters of the analytical model presented in this chapter. These parameters include (a) the PowerPoint processing and transmission costs

contained in performance reports sent by the users' computers and (b) the network latency measurements performed by each computer. Second, the system must create the multicast tree using the HMDM algorithm. Third, the system applies the model. In particular, it plugs the measured parameter values into the model to calculate the response times to commands by each inputting user with and without multicast. Fourth, it passes these response times to the total order function which then informs it which communication architecture to deploy. Fifth, it deploys the architecture.

In a simulated scenario, the first step is not possible because there are no computers that send performance reports and network latency measurements. Therefore, the simulation must provide these values. Based on the published network latency data between 1740 computers [67], we set the network latencies between all users equal to those between a random subset of 100, 200, 300, 400, 500 of the 1740 computers. One issue with randomly selecting the subset is whether the subset preserves properties, such as triangle inequality and latency distributions, of the entire set. Zhang et al. [96] analyzed random subsets taken from latencies measured between 3997 computers and found that they were representative of the overall measurements. Moreover, we have measured realistic PowerPoint processing and transmission costs for the netbook, P3 desktop, P4 desktop, Core2 desktop as described in Appendix A. Hence, in our scenario, the lecturer is assigned a netbook, and we randomly assign the computers used by the remaining users to be a P3 desktop, P4 desktop, Core2 desktop, or a netbook. Moreover, the users are organized in a centralized architecture in which the lecturer's computer is the master. The presenter's computer is using unicast to distribute messages to all users. Finally, we use a total order function that ranks the system with lowest maximum remote response times as the best.

Once the values are provided, the next step is to calculate the HMDM multicast tree. To perform this calculation, the values of all parameters used by the HMDM model must be known. It turns out that these parameters are the network card transmission costs and the network latencies, both of which are provided by the simulation. Hence, the HMDM algorithm can be simulated. The next two steps are executed in the same manner regardless of how the analytical model parameter values provided and the multicast tree is calculated. Therefore, these steps are the same whether the system is being simulated or used in an experiment. The fourth and final step, which deploys the optimal communication architecture, like the first step, needs actual computers. The reason is that the architecture is deployed by sending messages to the users' computers informing them how to establish communication channels with each other. For now, however, we do need the fourth step; our goal is to show that in theory, choosing the communication architecture can better meet response time requirements, which can be done in the first three steps.

For each scenario, we ran 40 simulations, for which we report the average results along with a 95% confidence interval. In most cases, we omitted graphing the interval because the interval was two or three orders of magnitudes less than the average.

Results

The the maximum remote response time of an audience member and the local response time for the lecturer are shown in Figures 3-7 and 3-8, respectively. The maximum remote response times increase much faster with unicast than with multicast as the number of collaborators grows as shown in Figure 3-7. For example, as the number of collaborators increased from 100 to 500, the maximum unicast remote response time grows by 5104.53s. The multicast remote response times grow only by 54.31ms – a two orders of

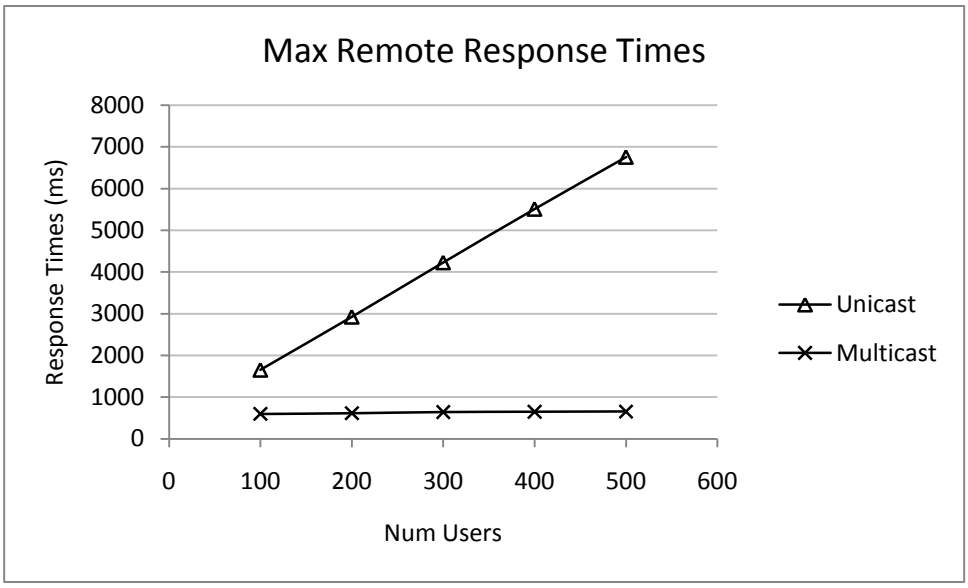

Figure 3-7. Max remote response time of observer of PowerPoint presentation.

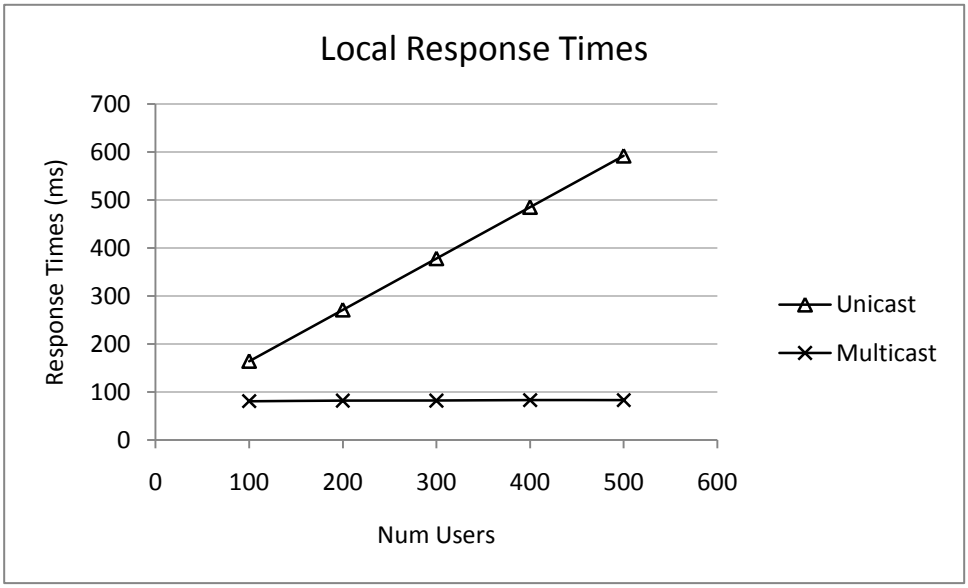

Figure 3-8. Local Response Times of PowerPoint presenter.

magnitude difference. As was the case with local response times, the remote response times are significantly better with multicast than with unicast for all sizes of collaborations: with 100 collaborators, they are 1052.25ms better; with 500 users, they are 6102.47ms better. Therefore, the self-optimizing system would have switched from unicast to multicast and significantly lowered the maximum remote response times.

As Figure 3-8 shows, the presenter's local response time increases linearly with the number of users when unicast is used (427.6ms), while when multicast is used, it increases slightly (2.03ms). This is expected as multicast relieves the source computer from transmitting to all the destinations. More importantly, local response times are significantly better with multicast than with unicast for all sizes of collaborations: with 100 collaborators, they are 82.41ms better; with 500 collaborators, they are 508.98ms better. Therefore, the self-optimizing system would have also significantly lowered the local response times.

The simulation shows that our purely theoretical results that multicast will help reduce remote response times apply to practical scenarios. Moreover, it also helps reduce local response times in such scenarios.

3.5.3 Validating Simulations

As mentioned above, if we are going to accept our simulation results, we must validate our simulations. For this, we need to run experiments and simulations for the same scenario and compare the results. Unfortunately, as mentioned above, we cannot run experiments for large scenarios because we do not have enough machines. This was the reason to use simulations in the first place. It seems that this is a-chicken-and-an-egg problem.

Fortunately, even with as few as ten machines, we can, in fact, run some limited large-scale experiments. To do so, we use a virtualization-like approach in which we treat each user's computer as a virtual computer that is mapped to one of the physical computers. One physical computer may have multiple virtual computers mapped to it. In fact, in experiments involving hundreds of users, there can be a large number of virtual computers mapped to a single physical computer since we have only ten of them. We could use off-the-

shelf virtualization software to accomplish this. However, since we are using Windows machines and we may need to virtualize many users on a single computer, we cannot do this since a single Core2 desktop with 1Gb of memory can run only several virtual Windows machines because of memory restrictions. Instead, we use added virtualization-like functionality in our collaborative framework approach that is able to map up to one hundred users onto a single computer before memory becomes an issue.

To illustrate how virtualization-like solutions can help us run experiments, consider an experiment with 100 users. Suppose that (a) $user_1$ inputs all the commands and the remaining 99 users observe them, (b) suppose that unicast is used to transmit commands, (c) $user_1$'s computer transmits to $user_{100}$'s computer last, and (d) we are interested in $user_1$'s local response times and $user_{100}$'s remote response times. As explained above, to properly measure the response times we are interested in, we must run $user_1$ and $user_{100}$ on dedicated machines. As a result, we are left with eight machines on which to run the remaining 98 users. We can evenly divide, to within a few users, these 98 users on eight machines. Therefore, a machine may run as many as 13 users. While we cannot accurately measure the response times of these users, $user_1$'s computer will still require time to transmit to them. Therefore, we are able to run certain near-ideal large-scale experiments.

Experiments when multicast is used instead of unicast are more difficult. The reason is that our system dynamically measures performance parameters based on which it dynamically deploys multicast overlays. As a result, the multicast overlay that our system deploys may be one in which the path from $user_1$ to $user_{100}$ contains one or more of the 98 users running on 8 machines as intermediate users. For the same reason we could not rely on response times reported by these 98 users, we cannot rely on response times reported by any

user to which these users forward commands. Once again, it seems impossible to validate through experiments the predictions made by our analytical model.

Fortunately, a work-around is possible that will allow us to run some large-scale experiments with multicast. Unfortunately, the workaround requires us to impose even more limits on the scenario we can use. Recall that our system can use previously measured parameter values to make architectural decisions at start time. Because of this functionality, we can force our system to deploy a multicast tree in which the path from $user_1$ to $user_{100}$ contains only users running on separate dedicated physical machines. We do this by fixing the values of some performance parameters that are provided to our system at start time. Moreover, we configure our self-optimizing system to not measure the values of the fixed parameters at runtime. Then, we compare the actual performance measurements with simulated performance. Note that because we are fixing some values and ignoring their actual values at runtime, the experiment is less realistic than one in which the values are measured dynamically. However, fixing these values is the only way we can run experiments using the resources available us. We next describe how we used this approach to validate our simulations.

Simulation

Consider the same 100 user scenario above, in which one user is giving a PowerPoint presentation to 99 other users in which we fix the parameter values as follows. Suppose that $user_1$, $user_2$, and $user_{100}$ are using a netbook, a P4 desktop, and a Core2 desktop, respectively, and that all of the other users are using Core2 desktops. Moreover, suppose that 1) $user_1$ can communicate directly with all users except $user_{100}$, 2) $user_2$ can communicate directly with $user_1$ and $user_{100}$, and 3) no other direct communication between any users is possible. Since

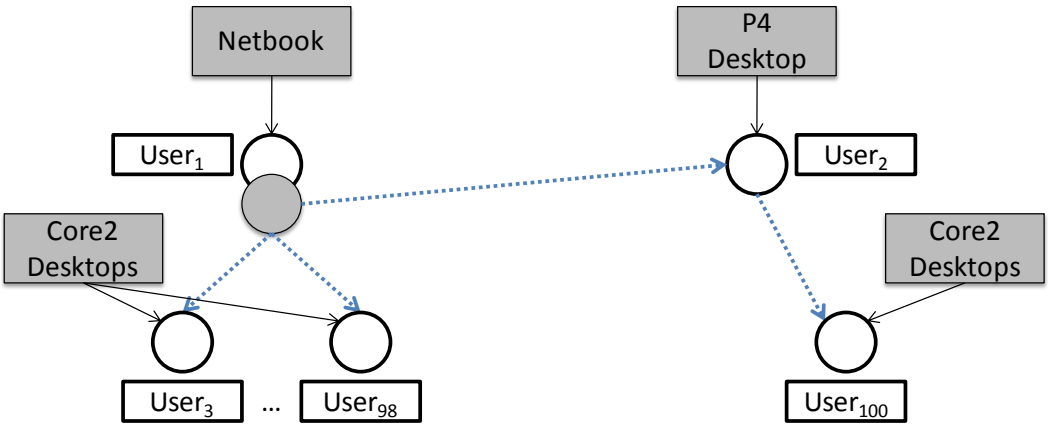

Figure 3-9. Local Response Times of PowerPoint presenter.

we are running all of our computers on the same LAN, these communication restrictions do not reflect reality in our experiments as all computers, and hence all users, can communicate directly with each other. Nevertheless, we can trick our system into believing in these communication restrictions by providing it with the following network latencies: 0ms latencies from $user_1$ to all other users except $user_{100}$; 0ms latencies from $user_{99}$ to $user_1$ and $user_{100}$; 1000000ms (effectively infinite) network delays between from any other users to all other users. By doing this, we force the HMDM multicast scheme to create a multicast tree in which (a) $user_1$ transmits first to $user_2$ and then to the $user_3$ through $user_{99}$ and (b) $user_2$ transmits only to $user_{100}$ as shown in Figure 3-9. The rest of the parameters values provided to the system at start time, such as processing and transmission costs, are assigned realistic values using the approach described in Appendix A. This scenario will enable us to validate our simulations because we have a path from $user_1$ to $user_{100}$ that has only one intermediate node, $user_2$. Moreover, we have a netbook, a P4 desktop, and a Core2 desktop on which we can run $user_1$, $user_2$, and $user_{100}$, respectively. Hence, we have 1) isolated a path in the

multicast overlay that our system would create if the conditions matched those we setup and 2) an overlay which we can map onto our ten computers.

We performed both simulations and an experiment for this scenario. Since we were interested in not only whether or not the simulations predict absolute performance correctly but also whether or not they predict trends correctly, we actually performed two sets of simulations and experiments with 3 and 100 users, respectively We report $user_1$'s local response times for both sets, and $user_3$'s and $user_{100}$'s remote response times for the 3 and 100 user set, respectively.

We configured our system to calculate a new communication architecture only once, immediately after the presenter advances the presentation to the second slide. Moreover, since we are interested in whether or not our system can perform communication architecture changes dynamically, we configured our system to optimize only the communication architecture. In particular, we configured the system to keep the initial processing architecture - the centralized architecture in which the presenter's computer is the master - throughout the session. Finally, we used a total order function that optimizes response times for as many users as possible.

The simulated and experiment response times for $user_1$ and the $user_3$ and $user_{100}$ with unicast are shown in Figure 3-10. The simulated and experiment response times for these users with multicast are shown in Figure 3-11. We ran the experiment ten times and report the average response times. The simulations needed to be run only once since the values of all parameters are constant. This was not the case in our previous result in which we simulated the performance of our system without fixing any parameter values. In that result, the computer types used by users other than the presenter were randomly assigned for each

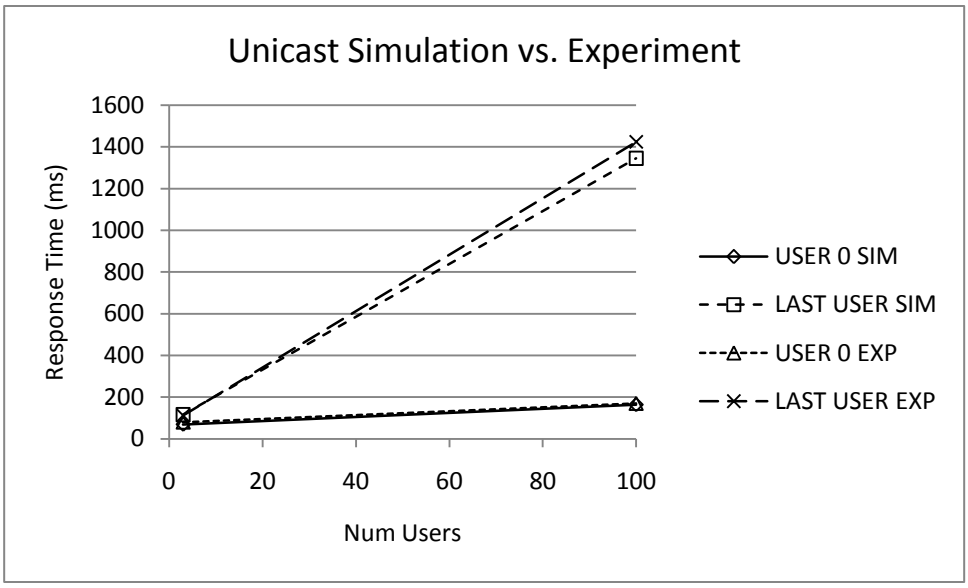

Figure 3-10. Simulation and experiment unicast response time comparison.

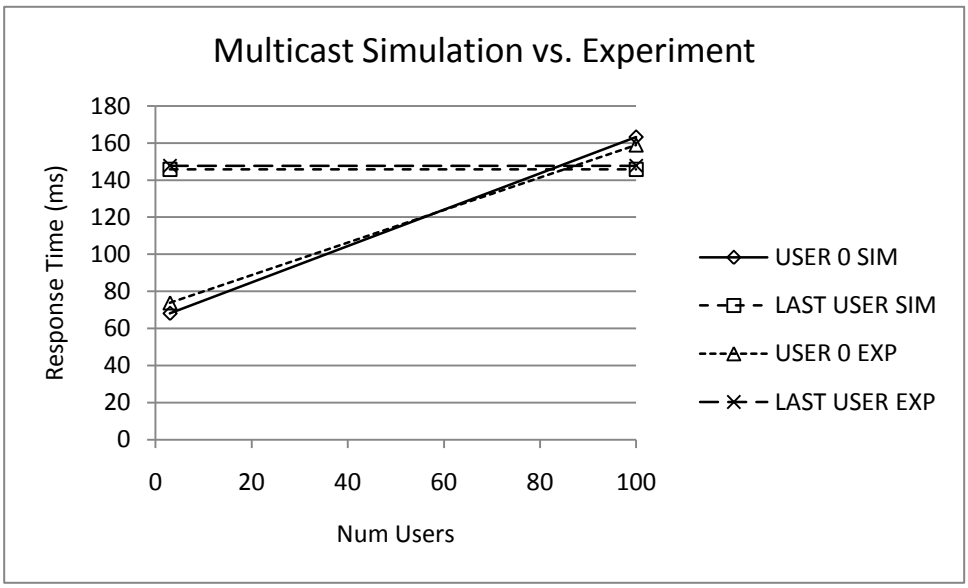

Figure 3-11. Simulation and experiment multicast response time comparison.

simulation. Moreover, the network latencies among the computers were assigned random, real values. Therefore, we ran forty simulations and reported the average.

We do not expect the simulated and measured response times to be exactly the same for two reasons. First, the values of parameters used in our simulations are average values as explained in Appendix A. Hence, these values can be slightly off from the values during a

particular experiment. Second, inherent to any experiment with distributed computers are fluctuations in network traffic and non-collaborative processes, such as those of the operating system and anti-virus software. These fluctuations can impact performance experiment to experiment. In our experience, reporting the average of ten experiments helps reduce but does not completely eliminate the fluctuations.

As Figures 3-10 and 3-11 show, the simulated response times are close to the experiment response times. The largest difference was for the unicast remote response time of $user_{100}$. The simulated value is was 79.077ms less than the measured value. The next highest difference between any other simulated and experiment value is 10.667ms. Therefore, even though most values are close, our simulations can be noticeably wrong. Nevertheless, even with the differences in simulated and measured values, the simulation shows that the self-optimizing system would have selected multicast instead of unicast which would have improved remote response times of $user_{100}$ by 1276.31ms according to the experiments.

3.5.4 Cost of Switching Communication Architecture

The above comparisons do not consider the impact of our system on response times as it gathers parameter values and switches communication architectures. Since we do not pause the inputting of commands during a communication architecture change, the cost of changing the communication is less important than the cost of a processing architecture change. Nevertheless, the change does require some CPU time and network communication from each user's computer, and hence it can affect response times. While we did not measure these costs directly, we also did not observe any differences in response times during the change because the change always happened during think time. In the experiment that

validated our simulations, the communication architecture change took an average of 2074ms in the ten experiment runs with a standard deviation of 309ms. The maximum time was 2741 and the minimum time was 1859ms. The think times from the log of PowerPoint actions we replayed during the experiment were higher than 2741ms 80% of time. Therefore, it is possible that the architecture change can occur completely during think time. When it does not, it will occur concurrently with user's commands. To minimize the impact on response times, the threads used to execute them are assigned a low priority as mentioned above. Moreover, the messages used during the constructions are small. Therefore, we expect the impact on response times to be low. However, future studies are needed to determine the exact impact, and more importantly, if it is noticeable.

A related issue is the amount of time required to calculate the overlay. The time required to calculate the overlay does not directly impact response times. The reason is that our system runs on a machine different from the user's machines, such as a server that hosts the session manager. Nevertheless, when the time taken to calculate the overlay is high, response times are impacted indirectly because many commands may be entered during that time and the response times for commands would not benefit from the optimized communication architecture that is returned by the calculation. Even worse, the parameters used to calculate the overlay may change during the long calculation. In the extreme case, if the calculation takes really long, the collaboration session may finish before the calculation does!

The amount of time required to calculate the overlay depends on two factors: the number of users and the number of multicast trees created. The number of users is a factor because the HMDM multicast algorithm requires $O((\text{number of users})^3)$ time to calculate a

single multicast tree for these users. On a Core2 desktop, calculating a multicast tree for 100 users requires only 12.275ms on average. However, had there been 500 users, the time required to calculate a multicast tree would have been 1.616s.

The second factor that determines the amount of time required to create the overlay is the number of multicast trees that form the overlay. As described earlier, the number depends on the total order function. If the function returns N trees, it takes N times as longer to calculate them than it does a single tree.

3.5.5 Limitations

While the discussion so far has focused on our system, we now address a computer architecture issue independent of our work that is nevertheless important. We found that when non-blocking communication is used and a command is transmitted to multiple destinations, there is no guarantee that the destination order that the CPU uses is the same as the destination order that the network card uses. This is an issue since both HMDM and our work assume that the CPU and network card transmission orders are the same. The issue arises because some network cards in use today have multiple transmission queues and the scheduling algorithm used by these cards is controlled by the network card driver. We are currently working on a way to configure the network card to either use a single transmission queue or a scheduling algorithm of our choosing, which should allow us to solve the problem. Meanwhile, we keep the default network card settings and validate our simulations as follows. A computer transmitting a command to multiple destinations pauses for 1) one second after transmitting to the first destination and 2) twenty seconds after transmitting to the second last destination. The first pause ensures that the destination to which the CPU transmits first is also the destination to which the network card transmits first. The second

pause is to ensure the same for the last destination. In all our experiments, the network card completed the transmission to all destinations regardless of the number of destinations in less than twenty seconds. Hence, a twenty second pause is sufficient to ensure that the last destination to which the CPU transmits to is the last destination to which the network card transmit. Clearly, the local response time on the sending computer and the remote response time on the final destination computer are off by twenty-one seconds. Therefore, we adjusted the reported values to cancel out the pauses.

3.6 Summary

To summarize, we have considered response times in any-scale collaboration scenarios with and without multicast. Traditional models of collaborative systems do not support multicast. Thus, we have introduced the bi-architecture model that encapsulates the traditional models and adds support for multicast. We have presented an analytical model of response times with and without multicast. The analytical model extends the one presented in the previous chapter: the new model is an N-user version of the old one; it does not assume negligible transmission costs; and it does not assume that there is no type-ahead. We have updated our self-optimizing system to use the bi-architecture and the new analytical model. As a result, the system is able to automate not only the processing but also the communication architecture. We have validated our model through simulations, which show that multicast can noticeably improve response times. We have also validated our simulations through experiments with our system. Finally, we have identified new implementation and policy issues that must be addressed by any system that automatically maintains the communication architecture. These results serve as proof of sub-thesis II and III and partial

proof of sub-thesis VI, which we re-state below. As a concluding remark, the results presented in this chapter show that the benefits of dynamic communication architecture changes exceed their costs.

SUB-THESIS II

The communication architecture impacts response times.

SUB-THESIS III

It is possible to develop a system that automatically switches to the communication architecture that satisfies any user-specified response time criteria better than existing approaches.

SUB-THESIS VI

It is possible to develop a model that analytically evaluates the impact on response times of different processing architectures, communication architectures, and scheduling policies to the degree necessary to automate their maintenance.

3.7 Relevant Publications

1. Junuzovic, S. and Dewan, P. Response time in N-user replicated, centralized, a proximity-based hybrid architectures. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 2006. pp: 129-138.
2. Junuzovic, S. and Dewan, P. Multicasting in groupware? *IEEE Conference on Collaborative Computing: Networking, Applications, and Worksharing*. 2007. pp: 168-177.
3. Junuzovic, S., and Dewan, P. Lazy scheduling of processing and transmission tasks collaborative systems. *ACM Conference on Supporting Group Work (GROUP)*. 2009. pp: 159-168.

CHAPTER 4

AUTOMATING SCHEDULING POLICY MAINTENANCE

4.1 Overview

In the previous two chapters, we have focused on improving response times by automating the maintenance of the processing and communication architectures. More specifically, in the second chapter, we presented (a) an analytical response time model for centralized and replicated small-scale collaborations, (b) a self-optimizing system that uses the model to automatically choose the processing architecture that best meets response times requirements, and (c) experimental results that show the system can improve response times in practical scenarios. In the following chapter, we (a) extended the analytical model from chapter two to large-scale collaboration scenarios, (b) used the extended model to show theoretical benefits of a multicast communication architecture for response times, (c) extended the self-optimizing system to automatically select the communication architecture that best meets response time requirements, and (d) presented simulations and experiments that show the extended system can improve response times in practical scenarios.

Both the processing and the communication architecture mandate specific tasks that the users' devices must perform. The processing architecture determines which computers process input commands in addition to processing outputs, while the communication architecture dictates the destinations to which a computer transmits commands. One issue

with our results so far is that they assume a fixed scheduling policy, in which the tasks were carried out on a single-core and precedence was given to the transmission task. In this chapter, we analyze the impact on response times of this and the other scheduling policies.

The implementation and evaluation of scheduling schemes for these tasks depend on how many cores are available for scheduling. For example, if two or more cores are available for scheduling, it is possible to carry out processing and transmission tasks in parallel. In this chapter, we will consider both single-core and multi-core scheduling policies. Regardless of the number of cores available for performing processing and transmission tasks, how these tasks are scheduled can influence response times.

When a single-core is available for scheduling, one approach is to create separate threads for these tasks and schedule the threads using a round-robin policy. An alternative to this concurrent policy is to schedule the tasks in a serial order, either processing or transmitting first. Intuitively, processing first tends to give the best local response times, transmitting first tends to give the best remote response times, and concurrent execution tends to give response times that are in between those supported by the other two policies.

One issue with the three existing policies is that there is no way to control the tradeoff between local and remote response times. Controlling the tradeoff is particularly attractive when an unnoticeable increase in one metric can result in a noticeable decrease in the other metric. As mentioned in the introduction chapter, human-perception studies have shown that certain increases are indeed unnoticeable – users cannot distinguish between local response times below 50ms [80] and visual and haptic remote response times below 50ms and 25ms, respectively [54]. On the other hand, users can notice 50ms changes in local [95] and remote response times [54]. Based on these observations and the principles in real-time scheduling,

we have devised a new lazy process-first scheduling policy that, like the process-first policy, gives precedence to the processing task, but delays its execution if the resulting increase in local response times cannot be noticed by humans. We can imagine a dual of this policy, lazy transmit-first scheduling, that like the transmit-first policy, gives precedence to the transmission task, but delays its execution if the resulting increases in remote response times are not noticeable. In this first cut at the idea of lazy scheduling of collaboration tasks, we have only considered lazy process-first scheduling, which we shall refer to as simply lazy scheduling.

The general idea of scheduling tasks so that they complete "just in time" is not new. In real-time systems, for example, there are both real-time tasks, which need to complete within some absolute deadline, and non-real-time tasks, which need to complete as soon as possible. Since in these systems a) there is no benefit to completing a real-time task before its deadline and b) real-time tasks typically have processing times that are shorter than their deadlines, a real-time task can be delayed by the difference between its deadline and processing time while still being able to complete in time. The amount of time the task can be delayed is called slack time. To improve the performance of non-real-time tasks, some slack-stealing scheduling algorithms schedule the non-real-time tasks during the slack times of the real-time tasks. By doing so, the non-real-time tasks can complete earlier. The idea of lazy scheduling, however, raises two questions that have not been answered by slack-stealing algorithms. First, how is it implemented in various kinds of distributed collaborative systems that exist today? Previous slack-stealing algorithms cannot be used because neither the processing nor the transmission tasks have so far been modeled as tasks with deadlines, which leads to the second question. How are deadlines incorporated into these tasks?

When more than one core is available for scheduling, one approach is to perform the processing and transmission tasks on different cores in parallel. This is the multi-core equivalent of the concurrent policy on a single-core machine. The remaining single-core policies also have multi-core equivalents. The lazy policy can be adapted to multi-core scenarios by delaying the processing task on all of the cores on which it executes. If the processing task is not parallelizable, then it executes on only one core regardless of the number of cores. Moreover, the sequential schemes would simply use all cores for one and then for the other task.

Scheduling tasks on multiple cores raises three additional issues. One issue is whether a task is statically assigned to a core at start time or dynamically mapped to a core at runtime based on availability. Dynamic scheduling is more flexible because it can make decisions at runtime based on current state. As a result, it can offer better performance. An orthogonal issue is whether a core can begin performing tasks of a command while other cores are still performing tasks for previous commands. It makes sense to allow this because it may improve performance. For instance, when think times are low, performing the transmission task of a command on one core even though another core is still performing the processing task of previous commands may improve the remote response time of the command. However, parallel execution can also hurt remote response times. For example, if one core begins to transmit a command while another core is transmitting a previous command, the remote response times of the previous command may increase. The reason is that when computer has a single network card, the latter command may be transmitted to some destinations before the earlier command is transmitted to all destinations. The final issue is whether or not there is any benefit to using multiple cores to perform transmission tasks.

After all, as mentioned in the previous chapter, the CPUs on the netbook, the P3 desktop, and the P4 desktop, which are all single-core, can transmit commands faster than the network cards on these computers can. Thus, using multiple cores may not make a difference.

In this chapter, we address these issues and present a system that better meets response time requirements than existing systems by automating the scheduling policy selection for any processing and communication architecture pair. In the process of creating the system, we extend our previous results in two ways. First, we present a new analytical model that can predict the impact of single-core and multi-core scheduling on response times. Second, we extend our self-optimizing framework to apply the new model. Since the basic question of this chapter is what scheduling policy should be used to best meet response time requirements, we analyze the impact of scheduling policy on response times in isolation of the impact of processing and communication architectures.

Chapter Scope:

We analyze the impact of scheduling policies on response times in isolation of the impact of processing and communication architectures. We consider any-scale collaboration scenarios involving both small and large numbers of users.

Chapter Goals:

We show that the scheduling policy impacts response times. We present an extension of our self-optimizing system that better meets response time requirements than existing systems by automating the selection of scheduling policies.

The system that automates the selection of the scheduling policy raises several issues. One issue is how the client-side component of our system measure CPU transmission and processing times of a command when non-sequential scheduling policies are used, such as concurrent and lazy. With these policies, the tasks interfere with each other and hence, measuring their duration from start to end does not give their actual costs as it does when sequential policies are used. A related issue is how the server-side component processes reported values when non-sequential policies are used.

Once the system gathers parameters values, it can apply the analytical model. The next question is how the system switches scheduling policies. Ideally, as in the case when switching communication architectures, it should not pause the execution of tasks of new input commands because such an approach degrades the response times of those commands.

We address these issues as we describe the self-optimizing system below. The rest of this chapter is organized as follows. We first define the single-core scheduling policies. Then, we present an analytical model for evaluating the impact on response times of the single-core polices. Following this, we extend both the single-core policy definitions and the analytical model to the multi-core case. We then extend our self-optimizing system to automatically choose the scheduling policy for any combination of the processing and communication architectures. Then, we present simulations and experiments conducted with the extended version of our self-optimization system in practical scenarios. Finally, we end with discussions and a brief summary.

4.2 Single-Core Scheduling Policies

As mentioned above, we consider four single-core scheduling policies: process-first, transmit-first, concurrent, and lazy. The definition of the first three policies is straightforward. With the transmit-first and process-first policies, a single thread carries out both the processing and transmission task one after the other starting with the processing and transmission tasks, respectively. With the concurrent policy, a separate thread is used for each task and the two threads are interleaved by the underlying thread management system. While the concurrent and the two sequential policies are the de facto approaches to scheduling tasks on a single core, the lazy policy is new. Thus, we define it precisely next.

4.2.1 Lazy Policy Approach

The idea behind the lazy policy is to delay the processing task for as long as possible without the user noticing the delay. This idea can lead to several different lazy policy implementations. One approach is to delay the processing task of a command by no more than the noticeable threshold from the moment the command was entered. Thus, before delaying the processing task, each computer would calculate the amount of time that has elapsed from the moment the command was entered to the moment it reached the computer. If the elapsed time is greater than or equal to the threshold, processing is not delayed; otherwise, it is delayed by the difference between the threshold and the elapsed time. One issue with this approach is that when the network latencies and transmission costs are high, most computers would not delay processing since both of these times are counted toward the delay. Therefore, this version of the lazy policy may not be very useful. An alternative approach, which does not have this problem, is to always delay the processing of a command by no more than the noticeable threshold from the moment the command was received. The

amount of time that processing is delayed is independent of the network latencies and transmission times so it does not matter if they are high. One issue with this approach, however, is that it favors downstream computers instead of upstream computers even though it is the response times of the users on the latter that are sacrificed to improve the response times of the users on the former. This was not an issue with the previous approach because eventually, computers would not delay processing at all. Therefore, we use a third approach that is a hybrid of the two and combines their good properties. Before delaying the processing task, each computer first calculates the amount of time that has elapsed from the moment a command is entered to the moment it is received by the computer. Then it subtracts from the calculated value the sum of the network latencies the command experienced on the way from the source to the computer. Processing is delayed only if the final value is less than the noticeable threshold. Like the first approach, it is fairer than the second approach, because eventually computers stop delaying processing completely. And like the second approach, it is more useful than the first approach, because high network latencies do not affect the decision whether or not to delay the processing task.

4.2.2 Lazy Policy Implementation

Our lazy scheduling algorithm works for both the centralized and replicated processing architectures and unicast and multicast communication architectures. It delays the execution of the processing task on a computer without allowing the local user to notice the delay. The pseudo-code for the algorithm is shown in Figure 4-1. As the figure shows, the algorithm accepts two parameters, namely, the local and remote response time degradation thresholds (Line 0). The algorithm supports different values of these thresholds because, as

```
0: INPUT: LOCAL_THRESH and REMOTE_THRESH response time
          degradation thresholds
DESTS     // this computer's destinations
Main()
1: loop ( forever )
2:     wait for command C

3:     if( centralized && C.isInput )
4:         CtoTrans = Process( C )
5:     else CtoTrans = C
6:     startTime = now
7:     for( each dest in DESTS ) dest.sentTo = false
8:     if( ( C.isInput && C.isFromLocalUser ) ||
                ( C.isOutput && C.isOutputToCmdByLocalUser ) )
9:         maxTransTime = LOCAL_THRESH - C.prevDelay
10:    else maxTransTime = REMOTE_THRESH - C.prevDelay
11:    Transmit( CtoTrans, maxTransTime )
12:    if(centralized) Process( CtoTrans )
13:    else Process( Process( C ) )
14:    Transmit( CtoTrans, INIFINITY )

Transmit( CtoTrans, maxTransTime )
15: for( each dest in DESTS )
16:     estTimeFromStart = now - startTime + EstTransTime(CtoTrans)
17:     if( estTimeFromStart >= maxTransTime )
            return
18:     if( dest.sentTo == false )
19:         CtoTrans.prevDelay += estTimeFromStart
20:         dest.send( CtoTrans )
21:         dest.sentTo = true
```

Figure 4-1. Lazy scheduling algorithm for single-core machines.

mentioned above, previous work has shown that noticeable local and remote response time degradations can be different.

The starting point of the policy, the `Main` function, is a loop which waits for the next command, `C`, which may be received from the local user or a remote computer. The first task is to compute from `C` the command `CtoTrans` to be actually transmitted to other computers (Lines 3-5). In all scheduling policies, this processing subtask is never delayed as it is necessary to define the transmission task. If the centralized architecture is being used and `C`

is an input command, then the transmitted command is the output command corresponding to the received command; otherwise it is simply the received command.

The next step is to compute the amount of time, `maxTransTime`, by which the (remaining) processing can be delayed, which depends on whether `C` is entered by or is a response to a command entered by the local or remote user (Lines 8-10). When `C` is an input command from the local user, the command has not experienced any delays and thus, `C.prevDelay` is zero. Therefore, the processing of the command can be delayed by as much as the time specified in the local response time degradation threshold (Line 9). Hence, `maxTransTime` is set to this threshold. When `C` is an output to a command entered by the local user, however, the command can be delayed for some time at the master for reasons given later. The delay is stored in `prevDelay` property of the command. Thus, `maxTransTime` is set to the difference between the local response time threshold and `prevDelay`. In all other cases, processing can be delayed by as much as the remote response time degradation threshold. Since the computers on the path from the source to the current computer contribute to the remote response times of the computer, the processing can be delayed only if the total previous delay for the command, which is stored in `prevDelay`, is less than the remote response time threshold. Thus, `maxTransTime` is set to the difference between this threshold and the total previous delay of the command.

Once `maxTransTime` has been calculated, the algorithm calls the `Transmit` function (Line 11). The `Transmit` function returns if it estimates that it will execute for longer than `maxTransTime` if it transmits to another destination (Line 17). To approximate the cost of the next transmission, the `Transmit` calls the `EstTransTime` function. To provide the estimate, the `EstTransTime` function could use data from previous

collaborations or can dynamically determine transmission costs based on transmission times of previous commands. If the `Transmit` function does not return, it transmits the command to the next destination. Because it can be called twice for the same command, once before and once after the processing task is performed, the function keeps track of destinations to which it has already sent the command and does not send to those destinations again when it is called the second time (Lines 18-21). For each transmitted command, it stores the delays the command has experienced so far in `prevDelay` (Line 19), which the next computer on the path reads (Lines 8-10).

After the `Transmit` method is called the first time, the processing of the command completes at the local computer (Lines 12-13), which depends on the processing architecture. In the centralized case, only the output command is processed. In the replicated case, the input and its computed output are processed. Then, the `Transmit` method is called again, this time with `maxTransTime` set to `INIFINITY`, which allows the transmission to complete.

4.3 Single-Core Response Time Analysis

We evaluate our lazy algorithm by comparing it with the existing sequential and concurrent policies. We present and illustrate an analytical model for the response times for all four policies and use the model to predict their relative performances. As in the previous chapter, we consider only non-blocking communication.

Analysis Scope:

As in the previous chapter, we focus on the non-blocking communication case.

In the analysis that follows, we recap the earlier analysis as the same style of reasoning can be used to analyze the response times of the other policies. As before, we first present an analysis of a command entered when the system is in a quiescent state. Following this, we present the analysis for consecutive commands entered by the user when the system is not in a quiescent state. Finally, we present the analysis for simultaneous commands entered by multiple users. Also, as in the previous chapter, we first develop the equations for replicated architecture remote response times for input commands entered by a master user.

4.3.1 Replicated Remote Response Time

As mentioned in the previous chapter, to reach a particular destination computer, a command must travel from the source computer to the destination computer along some path. The path may consist of additional computers. We refer to the source computer and these additional computers as the intermediate computers. The terms destination and intermediate are relative to a particular path. An intermediate computer on one path is a destination computer on a different path as all users see the output of an input command. Let π denote the path from the source to the destination, m denote the number of computers on the path including the source and destination computers, and $\pi_k, 1 \leq k \leq m$, denote the k^{th}computer on the path π, where π_1 is the source and π_m the destination computer.

Therefore, as in the previous chapter, the replicated remote response time of command i to computer j along path π is given by

$$remote_{REP,i,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{REP}(i, \pi_k) + dest_{REP}(i, \pi_m)$$

where $d(\pi_k, \pi_{k+1})$ is the network latency between the k^{th} and $k + 1^{st}$ computers on path π, $int_{REP}(i, \pi_k)$ is the delay of command i on the k^{th} intermediate computer on the path, and $dest_{REP}(i, \pi_m)$ is the delay of command i on the destination computer.

The first component of the remote response time equation is independent of the scheduling policy as it is a sum of the network latencies on the path from the source to the destination. The other two terms are scheduling policy dependent.

Transmit-First Policy Delays

Consider first the delay of command i on computer π_k, the k^{th} intermediate computer on path π. The delay is equal to the time that the computer requires to transmit the command to the next computer along the path, π_{k+1}. In general, computer π_k may have to transmit to more than one destination. Therefore, its delay depends on the number of other computers to which it transmits before transmitting to computer π_{k+1}. As described in the previous chapter, the transmit time is a function of the 1) the CPU transmit time, $x_{CPU}{}_{i,\pi_k}^{IN}$, which is the amount of time the CPU requires to copy the message data buffers from application space to the memory location from which the network card reads data for transmission, and 2) the network card transmit time, $x_{NIC}{}_{i,\pi_k}^{IN}$, which is the amount of time the network card requires to read the message data buffers from memory and send it on the physical wire. Either the

Analysis Scope:

Although in theory both the CPU and the network card can be transmission bottlenecks, we found that the network card was always the bottleneck in all of the collaboration logs we recorded. Therefore, as in the previous chapter, we focus on the case when the network card is the bottleneck.

CPU or the network card can be the bottleneck during transmission. As mentioned in the previous chapter, we focus on the case when the network card is the bottleneck because in all of our experiments, it always was.

Hence, the delay of computer π_k is equal to

$$int_{REP}^{TF}(i, \pi_k) = x_{CPU}{}_{i,\pi_k}^{IN} + pos(\pi_k, \pi_{k+1}) * x_{NIC}{}_{i,\pi_k}^{IN}$$

where $pos(\pi_k, \pi_{k+1})$ the position of computer π_{k+1}'s in computer π_k's list of destinations. The first term in the equation, $x_{CPU}{}_{i,\pi_k}^{IN}$, accounts for the fact that before the network card can begin transmitting a command to the first destination, the CPU must first queue the command for transmission to the destination.

The delay of command i on the destination computer π_m also depends on the number of computers to which computer π_m forwards commands because computer π_m must first transmit the command to all of them before processing the input command and its output. Unlike for intermediate computers, where the delay depended on the network card, in this case, the delay depends only on the CPU. The reason is that once the CPU queues messages in the network card's transmission queue, the work done by the network card does not take any CPU time. As in the previous chapter, let p_{i,π_m}^{IN} (p_{i,π_m}^{OUT}) denote the time computer π_m requires to process input (output) command i. Let fan_{π_m} denote the number of destinations to which computer π_m forwards commands, that is, the fanout degree from π_m. Thus, the delay of the destination computer equals

$$dest_{REP}^{TF}(i, \pi_m) = fan_{\pi_m} * x_{CPU}{}_{i,\pi_m}^{IN} + p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}$$

To illustrate the remote response time equations for this policy, consider the replicated-multicast architecture shown in Figure 4-2. The figure shows the transmission of a command entered by user$_1$. User$_1$'s computer transmits only to computers belonging to

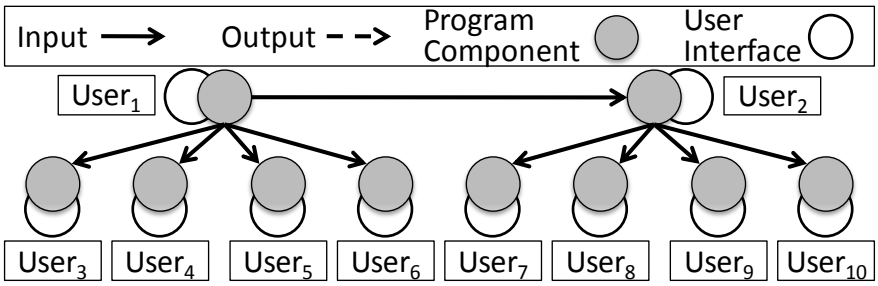

Figure 4-2. Replicated-multicast architecture with ten users.

$user_2$, $user_3$, $user_4$, $user_5$, and $user_6$, while $user_2$'s computer transmits to computers belonging to $user_7$, $user_8$, $user_9$, and $user_{10}$. Suppose the architecture has the following additional properties: 1) $user_1$ enters all of the commands; 2) $user_1$'s computer's transmission order is $user_2$, $user_3$, $user_4$, $user_5$, and $user_6$, and $user_2$'s transmission order is $user_7$, $user_8$, $user_9$, and $user_{10}$; 3) all of the users have the same computers; and 4) the local and remote response time thresholds are the same. We will use this *theoretical example* as a running example for illustrating our response time equations. Whenever we use the example, we will omit the user index terms from the transmission and processing parameters because we assume that all of the computers are the same.

Consider the remote response time of $user_{10}$. The path π from $user_1$'s to $user_{10}$'s computer is of length m=3 and π_1, π_2, and π_3 are $user_1$'s, $user_2$'s, and $user_{10}$'s computers, respectively. According to our equations, $user_1$'s delay is equal to $x_{CPU_i}^{IN} + pos(1,2) * x_{NIC_i}^{IN}$. Since $user_1$'s computer transmits to $user_2$ first, $user_1$'s delay is equal to $x_{CPU_i}^{IN} + x_{NIC_i}^{IN}$. Similarly, since $user_2$'s computer transmits to $user_{10}$ only after transmitting to three other destinations first, that is, $pos(1,2) = 4$, $user_2$'s delay is equal to $x_{CPU_i}^{IN} + 4 * x_{NIC_i}^{IN}$. $User_{10}$'s computer's delay is equal to the time the computer requires to process the input and the corresponding output command, $p_i^{IN} + p_i^{OUT}$. But if $user_{10}$'s computer had to also

forward the command to other computers, then the delay would include the time the computer requires to transmit the input command to them. Therefore, $user_{10}$'s remote response time is equal to $d(1,2) + d(2,10) + 2 * {x_{CPU}}_i^{IN} + 5 * {x_{NIC}}_i^{IN} + p_i^{IN} + p_i^{OUT}$.

Process-First Policy Delays

The equations for the process-first policy delays of the computers on the path π from the source to the destination are similar. Recall that in this policy, the computer starts the transmission task after it completes the processing task. Therefore, unlike the transmit-first delay, the process-first delay on an intermediate computer includes the time the computer requires to process the input and the corresponding output command. The process-first delay of an intermediate computer π_k is given by

$$int_{REP}^{PF}(i, \pi_k) = p_{i,\pi_k}^{IN} + p_{i,\pi_k}^{OUT} + {x_{CPU}}_{i,\pi_k}^{IN} + pos(\pi_k, \pi_{k+1}) * {x_{NIC}}_{i,\pi_k}^{IN}$$

In our example, $user_1$'s and $user_2$'s delays are equal to the time their computers require to process the input and output command, $p_i^{IN} + p_i^{OUT}$, plus the time they require to transmit the input to one and four destinations, ${x_{CPU}}_i^{IN} + {x_{NIC}}_i^{IN}$ and ${x_{CPU}}_i^{IN} + 4 * {x_{NIC}}_i^{IN}$, respectively.

The process-first delay of the destination π_m computer is simply the time the computer requires to process the input and the corresponding output command.

$$dest_{REP}^{PF}(i, \pi_m) = p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}$$

Concurrent Policy Delays

The delay equations for the concurrent policy are more complicated. The reason is that when the processing and transmission tasks execute concurrently, they interfere with each other's execution times. More precisely, the CPU transmission task interferes with the

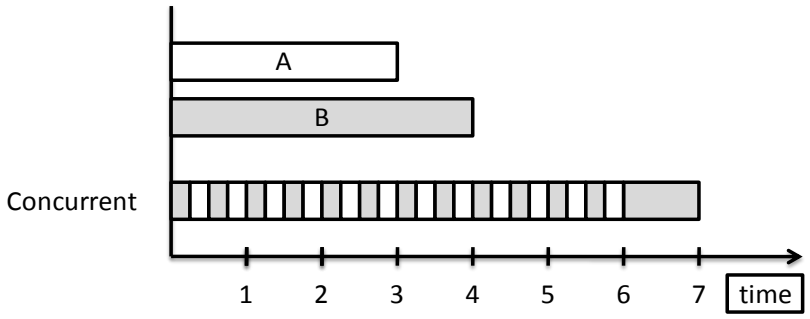

Figure 4-3. Concurrent execution time on a single processor.

processing task. We assume that the processing task does not block because it is difficult to predict their behavior, otherwise. The non-blocking task assumption is consistent with assumptions made in real-time systems when tight performance bounds are required. While results exist for blocking tasks, the upper-bounds for the performance in this case are loose. In addition, we consider context switch times negligible as we have found that they are no more than a few microseconds on modern operating systems running Pentium 4 desktops, which is several orders of magnitude lower than processing and transmission costs we have observed in real collaboration scenarios. Given these assumptions and our earlier assumption that a single core is available for scheduling, then the time required to complete 1) the shorter of the transmission and processing tasks is equal to twice the time required to complete it standalone and 2) the longer of the two tasks is equal to the time required to complete the two tasks sequentially. Both of these results are illustrated in Figure 4-3, which shows two tasks A and B executing concurrently. Task A is shorter than task B. As Figure 4-3 shows, the time required to complete the shorter task A is equal to twice the time required to complete it when it runs standalone. On the other hand, the longer task B completes in the amount of time required to complete the two tasks separately. This result is captured by the function

$$conc(a,b) = \begin{cases} 2a \; if \; a \leq b \\ a + b \; if \; a > b \end{cases}$$

where a and b are execution times of the two tasks.

The above function captures when the CPU completes transmitting. To analyze the time at which the network card completes the transmission, we must determine whether or not the network card is ready to transmit a command when the CPU begins a transmission quantum. Since we are considering the case in which the network card transmission cost is higher than that of the CPU, it cannot keep up with the CPU during a transmission quantum. Hence, during the upcoming processing quantum, it will be transmitting to destinations to which the CPU transmitted during the transmission quanta. If the network card completes transmitting to these destinations during the processing quantum, then it is ready to transmit to the destinations to which the CPU transmits in the next transmission quantum. If, on the other hand, it does not catch up to the CPU, then the response times of the destinations to which the CPU transmits to in the next transmission quantum include the time the network card requires to complete transmitting to the destinations to which the CPU transmitted in

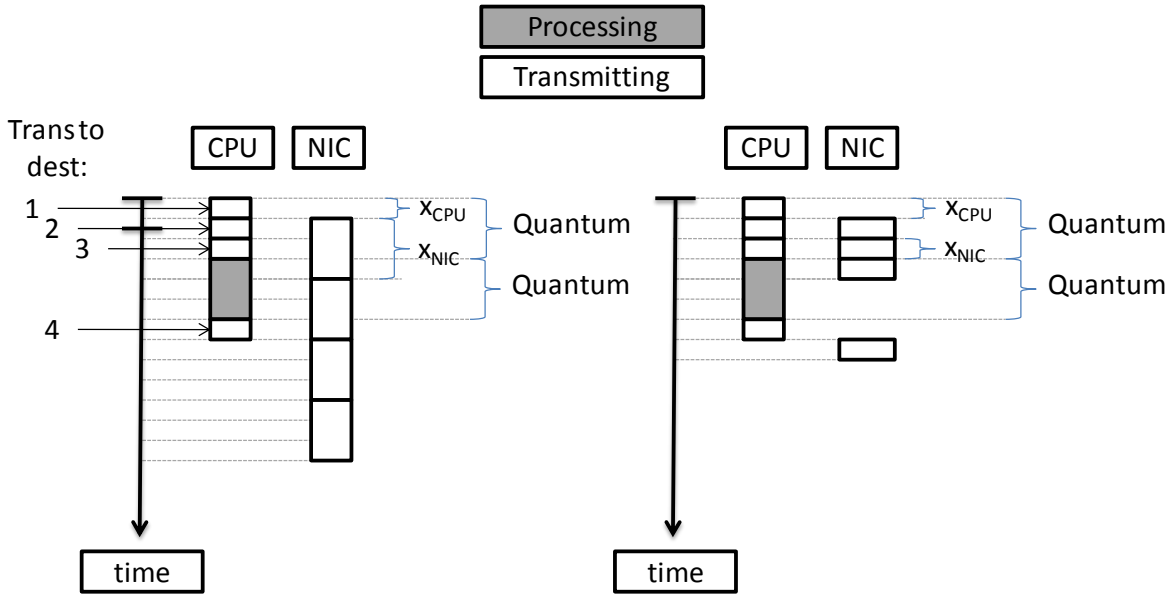

Figure 4-4. Concurrent transmission time on a single processor.

prior quanta. Both cases are illustrated in Figure 4-4. If the CPU transmit time is less than or equal to one half of the network transmit time, that is, if $2 * x_{CPU} \leq x_{NIC}$, then the network card will catch up; otherwise, it will not as shown in Figure 4-4 (right). As mentioned in the previous chapter, the CPU transmission cost was in fact less than half of the network card transmission cost. Therefore, we consider only the case when the network card does not catch up. In this case, once it starts transmitting a command, it does so continuously until it transmits the command to all destinations. Therefore, we can state the concurrent delay of an intermediate computer π_k as

$$int_{REP}^{CONC}(i, \pi_k) = {x_{CPU}}_{i,\pi_k}^{IN} + pos(\pi_k, \pi_{k+1}) * {x_{NIC}}_{i,\pi_k}^{IN}$$

In our example, user$_2$'s delay is equal to ${x_{CPU}}_i^{IN} + pos(2,10) * {x_{NIC}}_i^{IN}$. Since $pos(2,10) = 4$, user$_2$'s delay is equal to ${x_{CPU}}_i^{IN} + 4 * {x_{NIC}}_i^{IN}$.

The delay of the destination computer π_m is similar except that it captures how long the CPU takes to complete the processing task rather than how long the CPU and the network card take to complete the transmission task. Thus, the concurrent delay of the destination computer is given by the concurrent execution time function defined above

$$dest_{REP}^{CONC}(i, \pi_m) = conc\left(p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}, fan_{\pi_m} * {x_{CPU}}_{i,\pi_m}^{IN}\right)$$

Lazy Policy Delays

The equations for the lazy policy delays of the computers along the path π from the source to the destination must account for the local and remote response time degradation thresholds.

Consider the delay of an intermediate computer π_k. This delay depends on two factors: 1) whether the CPU transmits to the next computer on the path, π_{k+1}, before or after the processing task and 2) whether or not the network card catches up to the CPU

transmission while the CPU performs the processing task. The first factor depends on the difference between the sum of the delays the command has experienced so far and the amount of time by which the computer can delay the processing task without the local user noticing the delay, which is the local (remote) response time degradation threshold if the computer is (not) the source. Let

$$thresh_{\pi_k} = \begin{cases} local\ response\ time\ threshold\ if\ \pi_k\ is\ the\ source \\ remote\ response\ time\ threshold\ otherwise \end{cases}$$

Therefore, the maximum amount of time by which π_k can delay the processing of a command is the difference between π_k's threshold value and the total delays the command has experienced before reaching π_k. The difference is given by

$$delay_{REP}^{MAX}(i, \pi_k) = max\left(0, thresh_{\pi_k} - \sum_{l=1}^{k-1} int_{REP}(i, \pi_l)\right)$$

Thus, the intermediate computer π_k delays processing by a maximum of $delay_{REP}^{MAX}(i, \pi_k)$. The second factor depends on whether π_k transmits to the next downstream computer π_{k+1} before or after processing. Consider first the case when it transmits before processing. The delay of π_k is equal to the time its CPU requires to transmit to a single destination, $x_{CPU}{}_{i,\pi_k}^{IN}$, plus the amount of time its network card requires to transmit to the next downstream computer, $pos(\pi_k, \pi_{k+1}) * x_{NIC}{}_{i,\pi_k}^{IN}$. When the π_k transmits to the next downstream computer after processing, the delay depends on whether or not the network card catches up with the CPU during the time the CPU is processing the command. When the network card does not catch up to the CPU, then the delay is the same as when π_k transmits to the π_{k+1} before processing, as shown in Figure 4-5 (left). The reason is that once the network card starts to transmit, it does so continually until it transmits to π_{k+1}. Hence, the delay is equal to

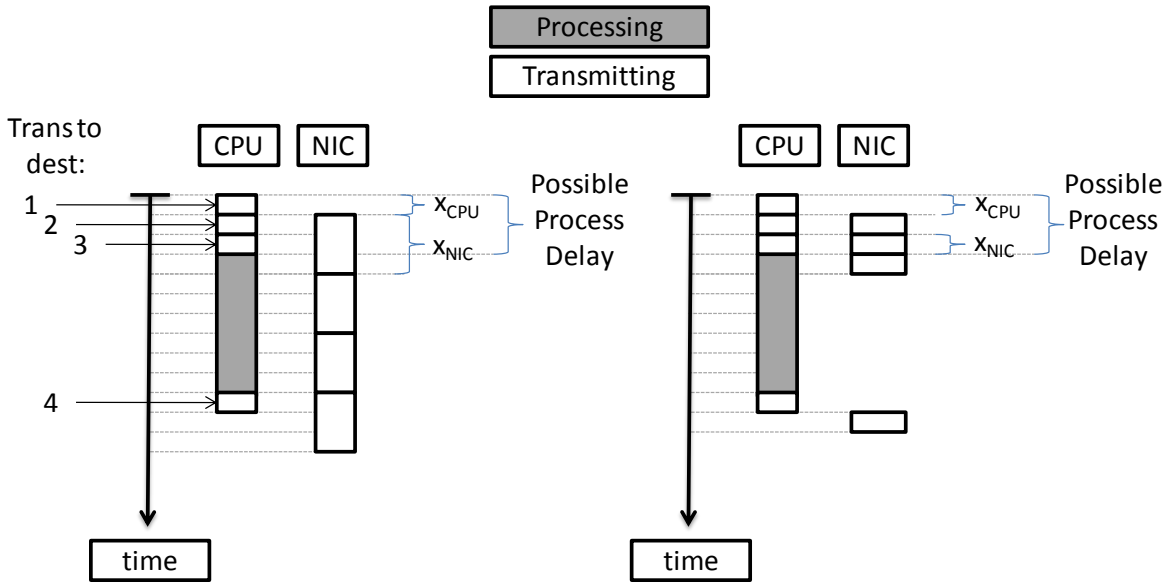

Figure 4-5. Lazy delay on a single processor.

$x_{CPU}{}_{i,\pi_k}^{IN} + pos(\pi_k, \pi_{k+1}) * x_{NIC}{}_{i,\pi_k}^{IN}$. When the network card catches up with the CPU during the time the CPU processes the command, then the network card is idle for some amount of time while the CPU processes the command, as shown in Figure 4-5 (right). If the CPU does not delay processing at all, then the network card is idle the entire time the CPU processes the command. If on the other hand, the CPU transmits to at least one destination before processing, then the network card idle time is equal to the amount of time processing was delayed, $delay_{REP}^{MAX}(i, \pi_k)$, plus the amount of time it required to process the command, $p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}$, minus the amount of time that elapsed from the moment π_k began performing tasks for the command to the moment the network card became idle. This final term depends on the number of destinations to which the CPU transmitted to before processing, which is given by $delay_{REP}^{MAX}(i, \pi_k)/x_{CPU}{}_{i,\pi_k}^{IN}$. Thus, the final term is equal to $x_{CPU}{}_{i,\pi_k}^{IN} + (delay_{REP}^{MAX}(i, \pi_k)/x_{CPU}{}_{i,\pi_k}^{IN}) * x_{NIC}{}_{i,\pi_k}^{IN}$. Therefore, the lazy delay of an intermediate computer π_k is given by

$$int_{REP}^{LAZY}(i,\pi_k)$$

$$= \begin{cases} x_{CPU}{}_{i,\pi_k}^{IN} + pos(\pi_k,\pi_{k+1}) * x_{NIC}{}_{i,\pi_k}^{IN} & if\ pos(\pi_k,\pi_{k+1}) * x_{CPU}{}_{i,\pi_k}^{IN} \leq delay_{REP}^{MAX}(i,\pi_k) \\ x_{CPU}{}_{i,\pi_k}^{IN} + pos(\pi_k,\pi_{k+1}) * x_{NIC}{}_{i,\pi_k}^{IN} + & \\ \quad max\left\{ \begin{matrix} 0 \\ delay_{REP}^{MAX}(i,\pi_k) + p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT} - \left(\frac{delay_{REP}^{MAX}(i,\pi_k)}{x_{CPU}{}_{i,\pi_k}^{IN}}\right) * x_{NIC}{}_{i,\pi_k}^{IN} \end{matrix} \right\} & otherwise \end{cases}$$

Consider the delay on user$_1$'s computer in our theoretical example. Instead of analyzing all possible cases, we will analyze only the case when (a) user$_1$'s computer delays processing before transmitting to user$_2$ and (b) user$_2$'s computer does not delay processing at all. If $x_{CPU}{}_{i}^{IN} \leq thresh$, then according to the lazy policy algorithm, user$_1$'s computer will transmit to user$_2$ before performing the processing task since it transmits to user$_2$ first and then to the other users. Therefore, user$_1$'s delay is equal to $x_{CPU}{}_{i,\pi_k}^{IN} + x_{NIC}{}_{i,\pi_k}^{IN}$. Suppose that $x_{CPU}{}_{i,\pi_k}^{IN} + x_{NIC}{}_{i,\pi_k}^{IN} > thresh$. Then, user$_2$'s computer will immediately begin performing the processing task in order to meet the remote response time threshold because $delay_{REP}^{MAX}(i,2) = 0$. Therefore, user$_2$'s delay is equal to the time user$_2$'s computer requires to process the input and output command, $p_i^{IN} + p_i^{OUT}$, plus the time it requires to transmit the command to four destinations, $x_{CPU}{}_{i}^{IN} + 4 * x_{NIC}{}_{i}^{IN}$.

The delay on the destination computer is slightly different as it tries to forward the command to as many other computers as possible while satisfying the remote response time threshold. Hence, if the time the computer requires to complete the transmission task plus the amount of time the command has already been delayed is less than the amount of time by which the computer can delay the processing time without the local user noticing the delay, then it will complete the transmission task before performing the processing task. Otherwise,

the computer will transmit for only as long as it can delay the processing task, $delay_{REP}^{MAX}(i, \pi_m)$. Hence, the delay of the destination computer is

$$dest_{REP}^{LAZY}(i, \pi_m)$$
$$= \begin{cases} fan_{\pi_m} * x_{CPU}{}_{i,\pi_m}^{IN} + p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT} \; if \; fan_{\pi_m} * x_{CPU}{}_{i,\pi_m}^{IN} \leq delay_{REP}^{MAX}(i, \pi_m) \\ delay_{REP}^{MAX}(i, \pi_m) + p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT} \; otherwise \end{cases}$$

In our example, since $user_{10}$'s computer does not forward the command to other computers, its delay is the processing time, $p_i^{IN} + p_i^{OUT}$.

4.3.2 Replicated Local Response Time

So far, we have presented only the equations for the replicated remote response times. We next present the equations for replicated local response times. Recall that the local response time is the time that elapses from the moment a user enters an input command to the moment the user sees the output for the command, which is equivalent to the time that elapses from the moment the inputting user's computer receives the command to the moment the computer completes processing the output of the command. Therefore, the local response time is exactly the delay of the destination computer defined above. This makes sense because the inputting user's computer is both the source and the destination. Thus, the transmit-first, process-first, concurrent, and lazy local response time equations for command i entered by $user_j$ are given by

$$local_{REP,i,j}^{TF} = dest_{REP}^{TF}(i, j) = fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT}$$

$$local_{REP,i,j}^{PF} = dest_{REP}^{PF}(i, j) = p_{i,j}^{IN} + p_{i,j}^{OUT}$$

$$local_{REP,i,j}^{CONC} = dest_{REP}^{CONC}(i, j) = conc\left(p_{i,j}^{IN} + p_{i,j}^{OUT}, fan_j * x_{CPU}{}_{i,j}^{IN}\right)$$

$$local_{REP,i,j}^{LAZY} = dest_{REP}^{LAZY}(i,j)\begin{cases} fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT} \ if \ fan_j * x_{CPU}{}_{i,j}^{IN} \leq thresh_j \\ thresh_j + p_{i,j}^{IN} + p_{i,j}^{OUT} \ otherwise \end{cases}$$

4.3.3 Centralized Architecture

The equations we have presented so far have considered the case in which the processing architecture is replicated and the input command is entered by a master user. Let us next consider the centralized architecture and slave commands.

As in the previous chapter, we can obtain the centralized architecture equations for commands entered by master users from the above replicated architecture equations by adjusting them for the two main differences in the two architectures. First, in the centralized architecture, only the master computer processes input commands, while all computers process output commands. Therefore, when calculating the delays of the computers on the path from the source to the destination, the processing times in the delays are equal to the time needed to process only output commands. Second, instead of transmitting input commands, the computers transmit output commands. Based on these two differences, the centralized architecture general remote response time equation is given by

$$remote_{CENT,i,j} = p_{i,\pi_1}^{IN} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{CENT}(i,\pi_k) + dest_{CENT}(i,\pi_m)$$

The new term, p_{i,π_1}^{IN}, accounts for the fact that the master computer must still process the input command. This is the equation that applies to all scheduling policies.

Based on the two differences between the centralized and replicated architectures, we can also derive the policy-specific delays created by the computers on the path from a source to a destination. The intermediate delays are given by

$$int_{CENT}^{TF}(i,\pi_k) = \ x_{CPU}{}_{i,\pi_k}^{OUT} + pos(\pi_k,\pi_{k+1}) * x_{NIC}{}_{i,\pi_k}^{OUT}$$

$$int_{CENT}^{PF}(i,\pi_k) = p_{i,\pi_k}^{OUT} + x\_CPU_{i,\pi_k}^{OUT} + pos(\pi_k,\pi_{k+1}) * x\_NIC_{i,\pi_k}^{OUT}$$

$$int_{CENT}^{CONC}(i,\pi_k) = x\_CPU_{i,\pi_k}^{OUT} + pos(\pi_k,\pi_{k+1}) * x\_NIC_{i,\pi_k}^{OUT}$$

$$int_{CENT}^{LAZY}(i,\pi_k)$$

$$= \begin{cases} x\_CPU_{i,\pi_k}^{OUT} + pos(\pi_k,\pi_{k+1}) * x\_NIC_{i,\pi_k}^{OUT} & if\ pos(\pi_k,\pi_{k+1}) * x\_CPU_{i,\pi_k}^{OUT} \le delay_{CENT}^{MAX}(i,\pi_k) \\ x\_CPU_{i,\pi_k}^{OUT} + pos(\pi_k,\pi_{k+1}) * x\_NIC_{i,\pi_k}^{OUT} + & \\ \quad max\left\{ delay_{CENT}^{MAX}(i,\pi_k) + p_{i,\pi_m}^{OUT} - \left(\frac{delay_{CENT}^{MAX}(i,\pi_k)}{x\_CPU_{i,\pi_k}^{OUT}} \right) * x\_NIC_{i,\pi_k}^{OUT} \right\} & otherwise \end{cases}$$

and the destination delays are given by

$$dest_{CENT}^{TF}(i,\pi_m) = fan_{\pi_m} * x\_CPU_{i,\pi_m}^{OUT} + p_{i,\pi_m}^{OUT}$$

$$dest_{CENT}^{PF}(i,\pi_k) = p_{i,\pi_k}^{OUT}$$

$$dest_{CENT}^{CONC}(i,\pi_m) = conc\left(p_{i,\pi_m}^{OUT}, fan_{\pi_m} * x\_CPU_{i,\pi_m}^{OUT}\right)$$

$$dest_{CENT}^{LAZY}(i,\pi_m)$$

$$= \begin{cases} fan_{\pi_m} * x\_CPU_{i,\pi_m}^{OUT} + p_{i,\pi_m}^{OUT}\ if\ fan_{\pi_m} * x\_CPU_{i,\pi_m}^{OUT} \le delay_{CENT}^{MAX}(i,\pi_m) \\ delay_{CENT}^{MAX}(i,\pi_m) + p_{i,\pi_m}^{OUT}\ otherwise \end{cases}$$

We can also derive the local response time equation in the same manner as the remote response time one to get

$$local_{CENT,i,j} = p_{i,j}^{IN} + dest_{CENT}(i,j)$$

The new term, p_{i,π_1}^{IN}, again accounts for the fact that the master computer must still process the input command.

As in the previous chapter, we can also obtain the equations for input commands entered by slave users by morphing the above equations for input commands entered by

master users. The only difference between the two kinds of input commands is that a command entered by a slave must first reach the master computer. Once the command reaches the master, the problem reduces to that of calculating the remote response time from the master to the slave, which we have already done above. The time the command takes to reach the master computer is equal to the time the slave computer requires to transmit the command to a single destination (i.e. the master) plus the time the command takes to traverse the network between the slave and master computers. Therefore, we can obtain the equations for the local and remote response time of command i entered by slave user_a whose master is user_b by adding the term ${x_{CPU}}_{i,a}^{IN} + {x_{NIC}}_{i,a}^{IN} + d(a,b)$ to the response time equations.

4.3.4 Implications

The analysis above helps us better understand the nature of the four single-core scheduling policies and how they differ from each other. It also helps us formally confirm intuitive expectations and, more interesting, derive some unintuitive results about the policies, especially about the lazy policy.

The lazy policy takes slack time in processing to transmit commands to other destinations. Thus, it gives processing less priority than process-first and more priority than concurrent and transmit-first policies. Intuitively, processing early (late) favors local (remote) response times, so, this seems to imply that, in comparison to (a) process-first, the local should be worse and remote response time better and (b) concurrent and transmit-first, the local should be better and remote response time worse. Our equations show that the differences are more subtle because the algorithm is run on each computer. We show the differences only for the replicated architecture and master commands. The results, however, apply to other cases also and can be derived similarly.

Lazy vs. Process-First

The difference in the lazy and process-first local response times is

$$local_{REP,i,j}^{LAZY} - local_{REP,i,j}^{PF} = min\left(fan_j * {x_{CPU}}_{i,j}^{IN}, delay_{REP}^{MAX}(i,j)\right)$$

By definition, the difference is never more than the local response time degradation threshold since $delay_{REP}^{MAX}(i,j) = tresh_j$ at the source. Thus, the equations predict that the lazy local response times are never noticeably worse than the process-first local response times.

To illustrate, consider user$_1$'s local response time in our example. When the lazy policy is used, user$_1$'s computer delays the processing task by the minimum of the local response time threshold, $thresh$, and the amount of time its CPU requires to transmit a command to five destinations, ${x_{CPU}}_i^{IN} + 5 * {x_{NIC}}_i^{IN}$. Since the computer requires $p_i^{IN} + p_i^{OUT}$ time for the processing task, user$_1$'s local response time is equal to $min\left({x_{CPU}}_i^{IN} + 5 * {x_{NIC}}_i^{IN}, thresh\right) + p_i^{IN} + p_i^{OUT}$. With the process-first policy, the computer immediately processes the command, resulting in a local response time of $p_i^{IN} + p_i^{OUT}$ to user$_1$. Thus, the lazy local response time is worse than that of the process-policy, but not by more than the local response time degradation threshold.

To derive the differences in the remote response times, we must compare both the intermediate and destination delays for the lazy and process-first policies. Consider first an intermediate computer. If the accumulated delays are such that this computer processes the command before transmitting to the downstream computer, then the difference in the delays is given by

$$int_{REP}^{LAZY}(i,\pi_k) - int_{REP}^{PF}(i,\pi_k) = 0$$

The reason is that in this case the lazy policy behaves like the process-first policy. In the other case, when the computer transmits to the downstream computer before processing, the difference in the delays is given by

$$int_{REP}^{LAZY}(i,\pi_k) - int_{REP}^{PF}(i,\pi_k) = -\left(p_{i,\pi_k}^{IN} + p_{i,\pi_k}^{OUT}\right)$$

Let us now consider the destination computer. If the destination computer does not delay processing, the lazy policy reduces to process-first. Hence, in this case

$$dest_{REP}^{LAZY}(i,\pi_m) - dest_{REP}^{PF}(i,\pi_m) = 0$$

Otherwise, the difference in the delays is

$$dest_{REP}^{LAZY}(i,\pi_m) - dest_{REP}^{PF}(i,\pi_m) = min\left(fan_{\pi_m} * x_{CPU}{}_{i,\pi_m}^{IN}, delay_{REP}^{MAX}(i,\pi_m)\right)$$

The difference is greater than equal to zero; however, it can never be more than the remote response time threshold by definition. Therefore, the destination delays with the lazy policy are never noticeably worse than with the process-first policy. On the other hand, processing costs can be significant. Therefore, the lazy delays on intermediate computers can be significantly better than the process-first delays. Therefore, overall, the lazy policy remote response times can be significantly better than the process-first remote response times.

To illustrate, consider $user_{10}$'s remote response time in our example. In this case, $user_1$'s and $user_2$'s computers are the intermediate computers. Since the lazy policy has multiple cases for intermediate and destination delays, let us as before consider one combination of these cases, in which $user_1$'s computer delays processing long enough to transmit to $user_2$ and $user_2$'s computer does not delay processing at all. $User_1$ and $user_2$ delays are equal to $x_{CPU}{}_i^{IN} + x_{NIC}{}_i^{IN}$ and $p_i^{IN} + p_i^{OUT} + x_{CPU}{}_i^{IN} + 4 * x_{NIC}{}_i^{IN}$, respectively, with the lazy policy, and $p_i^{IN} + p_i^{OUT} + x_{CPU}{}_i^{IN} + x_{NIC}{}_i^{IN}$ and $p_i^{IN} + p_i^{OUT} + x_{CPU}{}_i^{IN} + 4 * x_{NIC}{}_i^{IN}$,

respectively, with the process-first policy. User$_{10}$'s delays are $p_i^{IN} + p_i^{OUT}$ because user$_{10}$ does not forward commands. Thus, user$_{10}$'s remote response time is $p_i^{IN} + p_i^{OUT}$ less with the lazy than with the process-first policy. If processing costs are high, the difference is significant.

These results show some fundamental differences between lazy and process-first scheduling. The intermediate computers on the path that delay processing have an additive effect on the improvement in the remote response time. On the other hand, the destination computer does not degrade the remote response time by more than the remote response time threshold. Thus, like local response times, lazy remote response times can never be noticeably worse than those of process-first. More important, if processing costs are high, an unnoticeable increase in the remote response time of each intermediate computer that delays processing can result in a noticeable decrease in the remote response time of the destination computer.

Lazy vs. Transmit-First

The differences in the local response times of the lazy and transmit-first policies is given by

$$local_{REP,i,j}^{LAZY} - local_{REP,i,j}^{TF} = min\left(fan_j * {x_{CPU}}_{i,j}^{IN}, delay_{REP}^{MAX}(i,j)\right) - fan_j * {x_{CPU}}_{i,j}^{IN}$$

The difference is always less than or equal to zero. When the total CPU transmission time at the source is high, then the difference is also large. Hence, the lazy local response times can be significantly better than the transmit-first local response times, as one would expect intuitively.

To illustrate, in our example, user$_1$'s local response time is equal to $min\left(5 * x_{CPU}{}_i^{IN}, delay_{REP}^{MAX}(i,j)\right) + p_i^{IN} + p_i^{OUT}$ with the lazy policy. With the transmit-first policy, the local response time is equal to $5 * x_{CPU}{}_i^{IN} + p_i^{IN} + p_i^{OUT}$ since user$_1$'s computer transmits to five destinations. Thus, the lazy local response time is less than or equal to the transmit-first local response time. If the local response time threshold is equal to the amount of time the CPU requires to transmit the command to a single destination, that is $x_{CPU}{}_i^{IN} = thresh$, then the lazy local response time is $4 * x_{CPU}{}_i^{IN}$ less.

A comparison of the delay added to the remote response time by an intermediate node is somewhat counterintuitive. If the node processes before transmitting to the downstream computer, the difference in the delay is

$$int_{REP}^{LAZY}(i,\pi_k) - int_{REP}^{TF}(i,\pi_k)$$

$$= max\left\{delay_{REP}^{MAX}(i,\pi_k) + p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT} - \left(\frac{delay_{REP}^{MAX}(i,\pi_k)}{x_{CPU}{}_{i,\pi_k}^{IN}}\right) * x_{NIC}{}_{i,\pi_k}^{IN}, 0\right\}$$

If the computer does not delay processing at all, then the difference is equal to $p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}$. If the computer transmits the message to some destinations before processing but not to the downstream computer, then there are two cases to consider. First, if the processing time of the computer is high, then the difference is equal to $delay_{REP}^{MAX}(i,\pi_k) + p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT} - \left(delay_{REP}^{MAX}(i,\pi_k)/x_{CPU}{}_{i,\pi_k}^{IN}\right) * x_{NIC}{}_{i,\pi_k}^{IN}$. However, if the processing time of the computer is low and the network card transmission costs are high, then the difference is equal to zero. Thus, in this case, the lazy delay of the computer is the same for the lazy and transmit-first policies, even though the computer processed the message before queuing the message for transmission to the next downstream computer.

Thus, in this case, transmit-first performs better than or the same as the lazy policy. When the transmit-first performs better, the difference can be noticeable. Of course, if the intermediate node delays processing, the delays for the two policies are identical

$$int_{REP}^{LAZY}(i,\pi_k) - int_{REP}^{TF}(i,\pi_k) = 0$$

The results are also interesting when we consider the destination. If it does not delay processing, then the delay difference is

$$dest_{REP}^{LAZY}(i,\pi_m) - dest_{REP}^{TF}(i,\pi_m) = -\left(fan_{\pi_m} * {x_{CPU}}_{i,\pi_m}^{IN}\right)$$

Thus, if the sum of the processing times on all of the intermediate computers on the path from the source to the destination is greater than the total transmission time of the destination computer, then the transmit-first remote response times are better than those of the lazy policy. If the destination delays processing, on the other hand, then the difference in the delay is given by

$$dest_{REP}^{LAZY}(i,\pi_m) - dest_{REP}^{TF}(i,\pi_m) = min\left(fan_{\pi_m} * {x_{CPU}}_{i,\pi_m}^{IN}, delay_{REP}^{MAX}(i,\pi_m)\right) - fan_{\pi_m} * {x_{CPU}}_{i,\pi_m}^{IN}$$

The difference is always less than or equal to 0. When it is less than 0 and the intermediate delay difference is equal to 0, then the lazy remote response time will be better than the transmit-first remote response time. Hence, remote response times to some users can be better with the lazy than with the transmit-first policy, even through transmit-first policy gives higher priority to the transmission task.

To illustrate, consider user$_2$'s remote response time in our example. Consider the case, as above, when user$_1$'s computer transmits to user$_2$ before processing. Hence, user$_1$'s transmit-first and lazy delays are both equal to ${x_{CPU}}_i^{IN} + {x_{NIC}}_i^{IN}$ because in both cases, user$_1$'s computer transmits to user$_2$'s computer before processing. User$_2$'s lazy delay is equal to

$min\left(4 * {x_{CPU}}_i^{IN}, delay_{REP}^{MAX}(i,j)\right) + p_i^{IN} + p_i^{OUT}$ and user$_2$'s transmit-first delay is equal to $4 * {x_{CPU}}_i^{IN} + p_i^{IN} + p_i^{OUT}$ since user$_2$'s computer transmits to four other computers. Thus, user$_2$'s lazy remote response time can be ${x_{CPU}}_i^{IN} + 4 * {x_{NIC}}_i^{IN}$ less than the transmit-first remote response time. If the remote response time threshold, $thresh$, is less than the CPU transmission time, $4 * {x_{CPU}}_i^{IN}$, the lazy remote response time is significantly less.

Lazy vs. Concurrent

Finally, let us compare the response times of the lazy and concurrent policies. The differences in the local response times of these two policies is given by

$$local_{REP,i,j}^{LAZY} - local_{REP,i,j}^{CONC}$$
$$= min\left(fan_j * {x_{CPU}}_{i,j}^{IN}, delay_{REP}^{MAX}(i,j)\right) + p_{i,j}^{IN} + p_{i,j}^{OUT}$$
$$- conc\left(p_{i,j}^{IN} + p_{i,j}^{OUT}, fan_j * {x_{CPU}}_{i,j}^{IN}\right)$$

When the processing time is greater than the total CPU transmission time, then $conc\left(p_{i,j}^{IN} + p_{i,j}^{OUT}, fan_j * {x_{CPU}}_{i,j}^{IN}\right) = fan_j * {x_{CPU}}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT}$. Thus, the difference is this case is equal to $min\left(fan_j * {x_{CPU}}_{i,j}^{IN}, delay_{REP}^{MAX}(i,j)\right) - fan_j * {x_{CPU}}_{i,j}^{IN}$, which is less than or equal to zero. If the total CPU transmission time is high, then the lazy local response time is significantly better than the concurrent one. When the total CPU transmission time is greater than the processing time, then $conc\left(p_{i,j}^{IN} + p_{i,j}^{OUT}, fan_j * {x_{CPU}}_{i,j}^{IN}\right) = 2 * (p_{i,j}^{IN} + p_{i,j}^{OUT})$. In this case, the difference is equal to $min\left(fan_j * {x_{CPU}}_{i,j}^{IN}, delay_{REP}^{MAX}(i,j)\right) - p_{i,j}^{IN} + p_{i,j}^{OUT}$. If processing times are low, then the difference is greater than zero. However, it is never more than the response time threshold. Therefore, the lazy local response time is never

noticeably worse than the concurrent local response time. If, on the other hand, the processing time is high, then the lazy local response time is significantly better.

A comparison of the delay added to the remote response time by an intermediate node is the same as when comparing the lazy and transmit-first policies. The reason is that $int_{REP}^{CONC}(i,\pi_k) = int_{REP}^{TF}(i,\pi_k)$.

Comparing the destination delays is more interesting. If the destination computer does not delay processing, the difference is given by

$$dest_{REP}^{LAZY}(i,\pi_m) - dest_{REP}^{CONC}(i,\pi_m)$$
$$= p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT} - conc\left(p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}, fan_{\pi_m} * x_{CPU}{}_{i,\pi_m}^{IN}\right)$$

The difference is always less than or equal to zero. When, the destination computer's processing time is less than the total CPU transmission time, the difference is equal to $-(p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT})$; otherwise, it is equal to $-(fan_{\pi_m} * x_{CPU}{}_{i,\pi_m}^{IN})$. Hence, as was the case when we compared lazy and transmit-first policies, the lazy delay can be noticeably better than the concurrent delay if the processing time and the total CPU transmission time are high in the former and the latter cases, respectively. If the destination computer delays processing, the difference is given by

$$dest_{REP}^{LAZY}(i,\pi_m) - dest_{REP}^{CONC}(i,\pi_m)$$
$$= min\left(fan_{\pi_m} * x_{CPU}{}_{i,\pi_m}^{IN}, delay_{REP}^{MAX}(i,\pi_m)\right) + p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}$$
$$- conc\left(p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}, fan_{\pi_m} * x_{CPU}{}_{i,\pi_m}^{IN}\right)$$

Compared to the previous response time equation, the only new term is $min\left(fan_{\pi_m} * x_{CPU}{}_{i,\pi_m}^{IN}, delay_{REP}^{MAX}(i,\pi_m)\right)$ which is less than or equal to the response time threshold. Therefore, the lazy destination delay can never be noticeably worse than the

concurrent destination delay. Hence, remote response times to some users can be better with the lazy than with the concurrent policy.

In summary, the lazy 1) local response times can be significantly better and never noticeably worse and 2) remote response times are sometimes better and sometimes worse than those of the concurrent policy. Thus, as was the case when we compared the lazy and transmit-first polices, even though the lazy policy gives higher priority to the transmission task compared to the concurrent policy, the lazy remote response times can be better.

4.3.5 Consecutive Commands by the Same User

So far, we have considered response times of commands entered when the system is in a quiescent state. In this section, we consider the case when a user enters consecutive commands, by which we mean commands entered when the system is not in a quiescent state. The issue that arises in this situation, as mentioned in the previous chapter, is that the non-first commands may arrive at a computer before the computer has completed transmitting and processing the previous commands. As a result, the response time of these commands include an additional delay that is equal to the amount of time the computer requires to complete the tasks for the previous commands. Next, we derive the response times for consecutive commands for each single-core scheduling policy. We start with the response times of commands entered by a user in the replicated architecture.

Replicated Local Response Times

Consider first the local response times of master user_j . Suppose the user enters command $i + 1$ t_{i+1} time after entering command i. Suppose also that t_{i+1} is less than the time the CPU on user_j's computer requires to transmit and process command i. In this case, computer j delays tasks for command $i + 1$ until its CPU completes the tasks for command i.

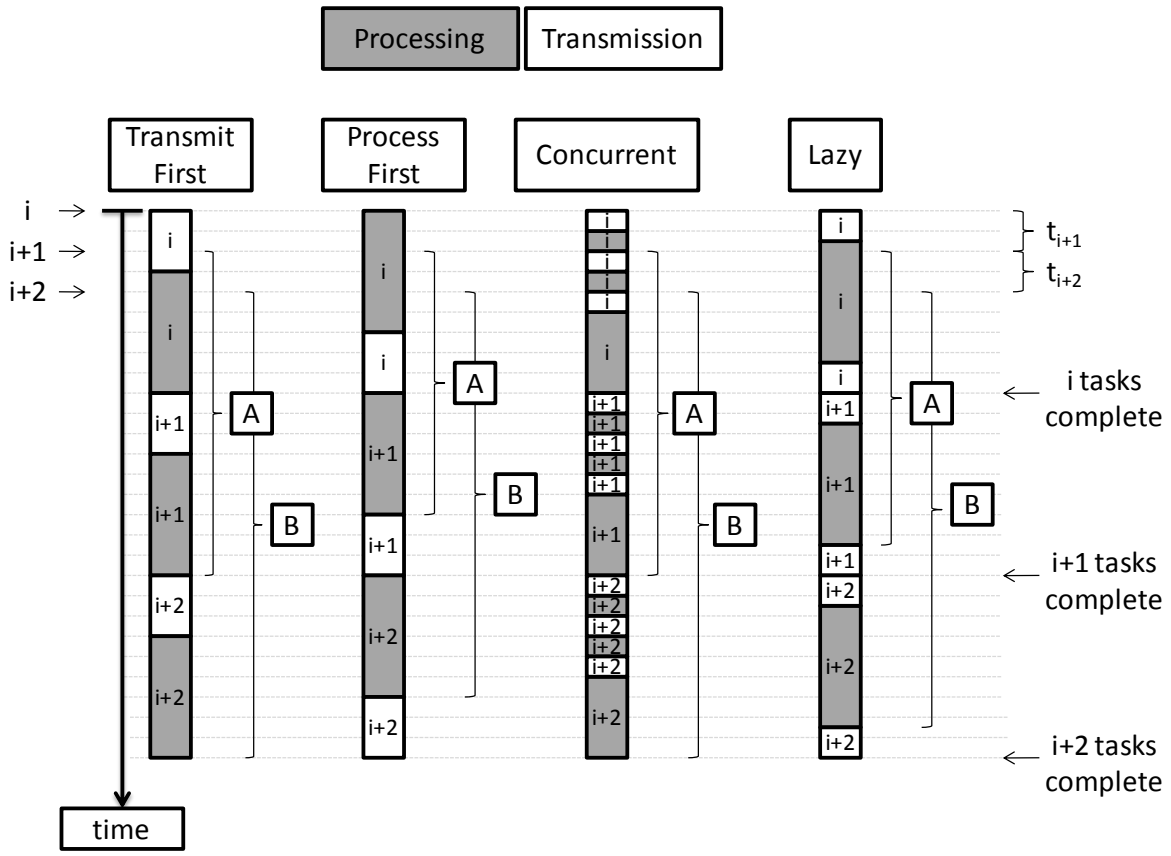

Figure 4-6. Local response times of consecutive commands under each scheduling policy.

The amount of time the CPU on computer j requires to complete tasks for command i is the same for all scheduling policies. The reason is that regardless of the order in the CPU carries out parts of the processing (transmission) task, the total amount of time required to carry out all parts of the task is equal to the time it requires to carry out the processing (transmission) task all at once, as illustrated in Figure 4-6. As Figure 4-6 shows, regardless of the scheduling policy, the CPU completes the tasks for command i at the same time. Therefore, regardless of scheduling policy, the time the CPU requires to perform the tasks for command i is given by $fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT}$. Thus, regardless of scheduling policy, computer j delays tasks for command i by $fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT} - t_{i+1}$. The amount of time that elapses from the moment the computer begins to perform tasks for command $i+1$ to the

moment the user sees the output to $i+1$ is equal to the user's response time of $i+1$ had it been entered in a quiescent state. Thus, the local response time of command $i+1$ is equal to

$$local_{REP,i+1,j}^{TF} = fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT} - t_{i+1} + dest_{REP}^{TF}(i,j)$$

$$local_{REP,i,j}^{PF} = fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT} - t_{i+1} + dest_{REP}^{PF}(i,j)$$

$$local_{REP,i,j}^{CONC} = fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT} - t_{i+1} + dest_{REP}^{CONC}(i,j)$$

$$local_{REP,i,j}^{LAZY} = fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT} - t_{i+1} + dest_{REP}^{LAZY}(i,j)$$

Suppose that user j enters command $i+2$ t_{i+2} time after entering command i. If the command is entered after user$_j$'s computer completes all tasks for command i, then this is the same case as for the local response time of command $i+1$. If, on the other hand, user$_j$'s computer is still carrying out tasks for command i, then the local response time of command $i+2$ is equal to (a) the amount of time the CPU requires to complete tasks for command i, $fan_j * x_{CPU}{}_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT} - t_{i+2}$, (b) plus the amount of time the CPU requires to complete tasks for command $i+1$, which is equal to $fan_j * x_{CPU}{}_{i+1,j}^{IN} + p_{i+1,j}^{IN} + p_{i+1,j}^{OUT}$ regardless of scheduling policy, (c) plus the local response time of command $i+2$ as if it had been entered in a quiescent state. Thus, the local response time of command $i+2$ is higher than the response time of command $i+1$ by the amount of time the CPU requires to perform tasks for command $i+1$ minus difference in their think times, $t_{i+2} - t_{i+1}$, as illustrated in Figure 4-6. In Figure 4-6, the local response time of command $i+1$ is labeled as A while that of command $i+2$ is labeled as B. As the figure shows, for all four scheduling policies, $A - B$ is equal to the amount of time the CPU needs to perform the processing and transmission tasks for command $i+1$.

More generally, the local response time of command $i+z$ entered t_{i+z} time after the CPU on user$_j$'s computer begins performing tasks for command i but before it completes them is given by

$$local^{TF}_{REP,i+z,j} = \left(\sum_{c=i}^{i+z-1} fan_j * x_{CPU}{}^{IN}_{c,j} + p^{IN}_{c,j} + p^{OUT}_{c,j} \right) - t_{i+z} + dest^{TF}_{REP}(i+z,j)$$

$$local^{PF}_{REP,i+z,j} = \left(\sum_{c=i}^{i+z-1} fan_j * x_{CPU}{}^{IN}_{c,j} + p^{IN}_{c,j} + p^{OUT}_{c,j} \right) - t_{i+z} + dest^{PF}_{REP}(i+z,j)$$

$$local^{CONC}_{REP,i+z,j} = \left(\sum_{c=i}^{i+z-1} fan_j * x_{CPU}{}^{IN}_{c,j} + p^{IN}_{c,j} + p^{OUT}_{c,j} \right) - t_{i+z} + dest^{CONC}_{REP}(i+z,j)$$

$$local^{LAZY}_{REP,i+z,j} = \left(\sum_{c=i}^{i+z-1} fan_j * x_{CPU}{}^{IN}_{c,j} + p^{IN}_{c,j} + p^{OUT}_{c,j} \right) - t_{i+z} + dest^{LAZY}_{REP}(i+z,j)$$

The first term in all four equations is scheduling policy independent. It is the total amount of time the CPU on user$_j$'s computer requires to transmit and process commands i through $i+z-1$. The last term is the time the amount of time that elapses from the moment the computer begins performing tasks for command $i+z$ to the moment it completes processing command $i+z$. Finally, we must subtract the amount of time, t_{i+z}, the computer had been performing tasks for command i when command $i+z$, is entered as this time does not contribute to the local response time of command $i+z$. As this equation shows, the response time is a function of the number of consecutive commands. Moreover, the response time is scheduling policy dependent. However, somewhat counter intuitively, the difference in response times among the four scheduling policies is independent of the number of consecutive commands. The reason is that, as mentioned above, the amount of time the CPU

requires to perform the tasks for commands i through $i + z - 1$ is the same for all scheduling policies.

The transmit-first case above gives the same result as the local response time equation given in the previous chapter, which is not surprising because in the previous chapter, we assumed that transmit-first scheduling is used. In particular, the amount of time the CPU requires to transmit then process a command is equal to the destination delay for the command if it had been entered in the quiescent state

$$dest_{REP}^{TF}(c,j)=fan_j * x_{CPU}{}_{c,j}^{IN} + p_{c,j}^{IN} + p_{c,j}^{OUT}$$

Substituting this equation into the above general equation gives

$$local_{REP,i+z,j}^{TF} = \left(\sum_{c=i}^{i+z-1} fan_c * x_{CPU}{}_{c,j}^{IN} + p_{c,j}^{IN} + p_{c,j}^{OUT} \right) - t_{i+z} + dest_{REP}^{TF}(i+z,j)$$

$$= \left(\sum_{c=i}^{i+z-1} dest_{REP}^{TF}(c,j) \right) - t_{i+z} + dest_{REP}^{TF}(i+z,j)$$

$$= \left(\sum_{c=i}^{i+z} dest_{REP}^{TF}(c,j) \right) - t_{i+z}$$

which is the equation we derived for the transmit-first local response time for command $i + z$ in the previous chapter.

Replicated Remote Response Times

As in the previous chapter, the remote response times of consecutive commands by the same user are more complicated. The reason is that as a command traverses a path from a source to a destination in the communication overlay, it may suffer additional delays on any computer on the path.

Recall that in the previous chapter, we have already shown that the unicast and multicast remote response time degradation experienced by the destination computer for consecutive messages depends on the computer on the path that requires the most time to process and transmit commands, where by transmit we mean the time at which the network card completes transmitting commands. We called this computer the critical computer. More formally, the critical computer is defined with respect to $user_b$'s response times of consecutive commands i through $i + z$ entered by $user_j$. A critical computer is the slowest computer on the path from $user_j$ to $user_b$ such that (a) command i is the last command it received in an idle state, (b) it has since then been continuously performing tasks for commands i though $i + z - 1$, and (c) it is still performing them when command $i + z$ arrives.

Using the notion of a critical computer, we compared the differences in the remote response times of consecutive commands with unicast and multicast. First, we derived the remote response times of command $i + z$ as follows. Let computer π_s, the s^{th} computer on the path from j to b, be the critical computer. The critical computer may be an intermediate computer or the destination computer. When it is an intermediate (destination) computer, then $user_b$'s remote response time of command $i + z$ is equal to 1) the amount of time that elapses from the moment command i is entered to the moment it reaches the critical computer, 2) plus the amount of time that elapses from the moment the critical computer receives command i to the moment i transmits command $i + z$ to the next computer on the path (completes processing of command $i + z$), 3) plus the amount of time that elapses from the moment the critical computer transmits command $i + z$ to the next downstream computer to the moment $user_b$ sees its output, 4) minus the amount of time that elapses from the

moment command i is entered to the moment command $i + z$ is entered. Note that the first (third) term is equal to 0 if the critical computer is the source (destination). As explained in the previous chapter, the definition of the critical computer implies that all computers must be idle when command i reaches them because π_s is the critical computer, and it is idle when command i reaches it. Similarly, the definition of the critical computer also implies that all of the computers downstream from π_s must be idle when command $i + z$ reaches them.

Of the terms contributing to the remote response time of command $i + z$, only the second term, which we refer to as the critical computer time, depends on the number of consecutive commands. In particular, the first term accounts for parts of the response time during which the critical computer and computers upstream from it are idle when they receive command i, the third term accounts for parts of the response times during which computers downstream from the critical computer are idle when they receive command $i + z$, and the fourth term is a constant. Therefore, when we analyzed the difference in the equations for unicast and multicast response times, the difference boiled down to the difference in the second term. As we will show, comparing the response times for different scheduling policies will amount to the same.

In the previous chapter, we had derived the first, third, and fourth terms when the transmit-first policy is used. The first term is equal to $\sum_{k=1}^{s-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int_{REP}^{TF}(i, \pi_k)$, the third term is equal to $\sum_{k=s}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=s+1}^{m-1} int_{REP}^{TF}(i + z, \pi_k) + dest_{REP}^{TF}(i + z, \pi_m)$, and the fourth term is simply equal to t_{i+z}. Therefore, the same derivation holds for other scheduling policies as long as we replace the intermediate and destination delay terms with those of each policy.

Let us now consider the critical computer time term. When the critical computer is the destination computer, the critical computer time depends on how long the CPU requires to process and transmit commands i through $i+z-1$ plus the amount of time the CPU requires for displaying the result of command $i+z$ to the local user once it starts to perform tasks for it. In particular, the network card transmission times on the destination computer do not contribute the response times because the CPU never waits on the network card. Therefore, the response time equations are given by

$$remote^{TF}_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int^{TF}_{REP}(i, \pi_k) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_m} * x_{CPU}{}^{IN}_{c,\pi_m} + p^{IN}_{c,\pi_m} + p^{OUT}_{c,\pi_m} \right) + dest^{TF}_{REP}(i+z, \pi_m) - t_{i+z}$$

$$remote^{PF}_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int^{PF}_{REP}(i, \pi_k) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_m} * x_{CPU}{}^{IN}_{c,\pi_m} + p^{IN}_{c,\pi_m} + p^{OUT}_{c,\pi_m} \right) + dest^{PF}_{REP}(i+z, \pi_m) - t_{i+z}$$

$$remote^{CONC}_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int^{CONC}_{REP}(i, \pi_k) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_m} * x_{CPU}{}^{IN}_{c,\pi_m} + p^{IN}_{c,\pi_m} + p^{OUT}_{c,\pi_m} \right) + dest^{CONC}_{REP}(i+z, \pi_m) - t_{i+z}$$

$$remote^{LAZY}_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int^{LAZY}_{REP}(i, \pi_k) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_m} * x_{CPU}{}^{IN}_{c,\pi_m} + p^{IN}_{c,\pi_m} + p^{OUT}_{c,\pi_m} \right) + dest^{LAZY}_{REP}(i+z, \pi_m) - t_{i+z}$$

The transmit-first equation presented in this chapter is equivalent to the transmit-first equation presented in the previous chapter. In particular, $dest_{REP}^{TF}(i+z,j) = fan_{\pi_m} * x_{CPU}{}_{i+z,\pi_m}^{IN} + p_{i+z,\pi_m}^{IN} + p_{i+z,\pi_m}^{OUT}$. Substituting this into the above transmit-first equation gives

$$remote_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{REP}(i, \pi_k) + \sum_{c=i}^{i+z} \left(fan_{\pi_m} * x_{CPU,c,\pi_m}^{IN} + p_{c,\pi_m}^{IN} + p_{c,\pi_m}^{OUT} \right) - t_{i+z}$$

When the critical computer is an intermediate computer, then the time it requires to complete the tasks of consecutive commands is a function of both the network card and CPU costs. It is a function of CPU costs because until the CPU completes tasks for a command, the network card cannot begin transmitting the next command even if it is idle because the CPU must first transmit it. The time is a also function of transmission times because until the network card completes transmitting a command to all destinations, it cannot begin transmitting the next command even if the CPU has completed the processing and transmission tasks for it. As in the previous chapter, we consider two cases. In one case, the total network card transmission time is higher than the total amount of time the CPU spends on each command. In this case, the network card falls further behind the CPU with each command, as shown in Figure 4-7. In the other case, the amount of time the CPU spends on each command is higher than the total network card transmission time of the command. In this case, the network card keeps up with the CPU, as shown in Figure 4-8. All other cases reduce to these two as explained in the previous chapter.

Let us first consider the extreme case when the network card time spent on each command dominates the CPU processing time. As in the previous chapter, the critical computer time is equal to (1) the amount of time that elapses from the moment the critical

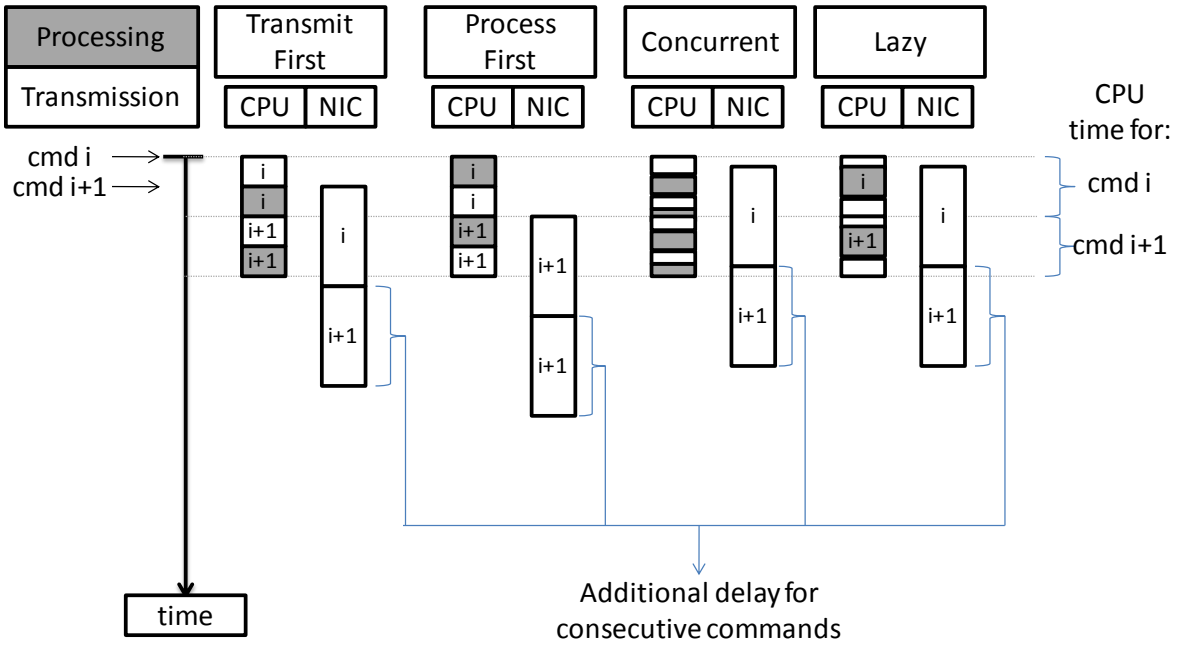

Figure 4-7. Critical computer time when the total network card transmission time of each command dominates the processing time for the command.

computer receives command i to the moment the CPU transmits i to the first destination, plus (2) the amount of time that elapses from the moment the network card begins to transmit i to the first destination to the moment it transmits $i + z - 1$ to the last destination, plus (3) the amount of time that elapses from the moment the network card begins transmitting $i + z$ to the moment it transmits it to the next downstream computer. The first term is scheduling policy dependent. We return to it momentarily. The second term is equal to $\sum_{c=i}^{i+z-1}\left(fan_{\pi_s} * x_{NIC}{}_{c,\pi_s}^{IN}\right)$ because once the network card begins transmitting i, it does not stop until it finishes transmitting $i + z$. The third term is equal to $pos(\pi_s, \pi_{s+1}) * x_{NIC}{}_{i+z,\pi_s}^{IN}$. Therefore, only the first term is scheduling policy dependent. When the process-first policy is used, the first term is equal to $p_{i,\pi_s}^{IN} + p_{i,\pi_s}^{OUT} + x_{CPU}{}_{i,\pi_s}^{IN}$ as the computer must first process command i before it begins transmitting. When the transmit-first policy is used, the first term is equal to

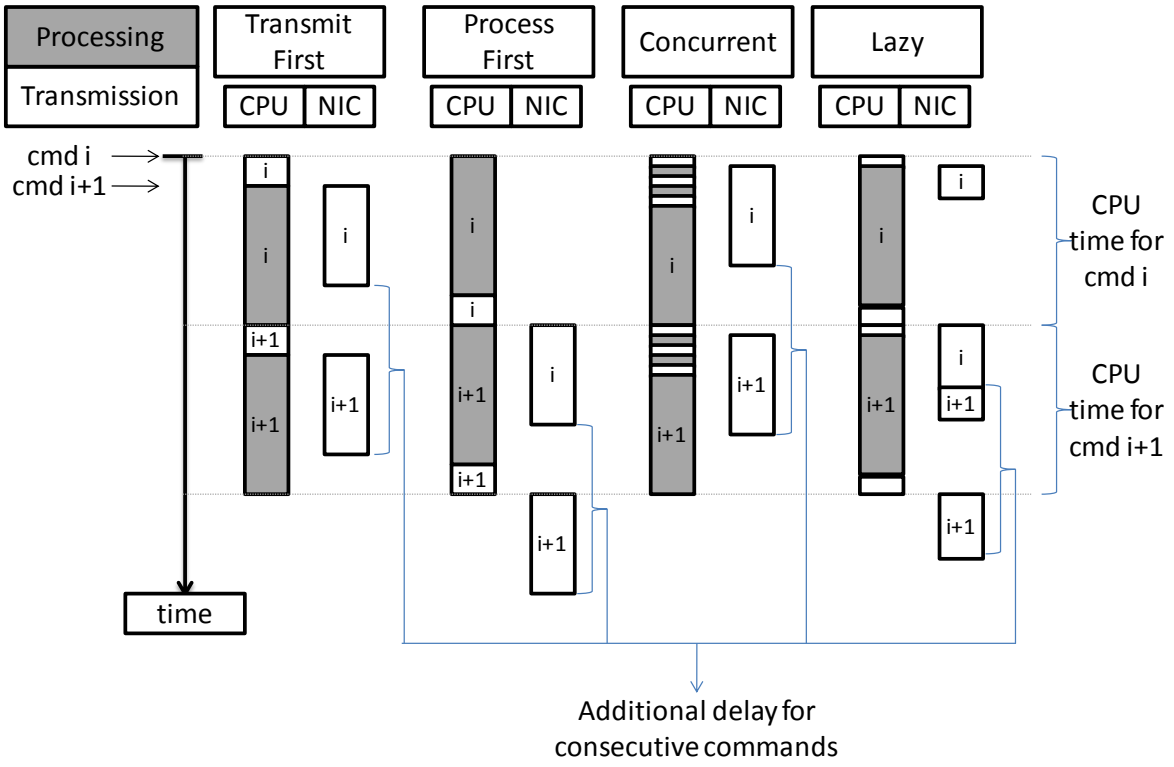

Figure 4-8. Critical computer time when the processing time of each command dominates the total network card transmission time for the command.

simply $x_{CPU}{}_{i,\pi_s}^{IN}$. When the concurrent policy is used, the first term is equal to $x_{CPU}{}_{i,\pi_s}^{IN}$. The lazy policy is slightly more complicated because one part of the CPU transmission happens before and the other after processing. In the worst case, the CPU does not delay processing. Thus, the first-term is equal to that of the process-first policy, $p_{i,\pi_s}^{IN} + p_{i,\pi_s}^{OUT} + x_{CPU}{}_{i,\pi_s}^{IN}$. In the best case, the CPU completes transmitting before processing. Thus, the first-term is equal to that of the transmit-first policy, $x_{CPU}{}_{i,\pi_s}^{IN}$. Therefore, we can state the response time equations when the critical computer is not the destination computer and the total network card transmission times dominate processing times as

$$remote^{TF}_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{TF}_{REP}(i, \pi_k)$$
$$+ x_{CPU}{}^{IN}_{i,\pi_s} + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{NIC}{}^{IN}_{c,\pi_s} \right) + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{IN}_{i+z,\pi_s}$$
$$+ \sum_{k=s+1}^{m-1} int^{TF}_{REP}(i+z, \pi_k) + dest^{TF}_{REP}(i+z, \pi_m) - t_{i+z}$$

$$remote^{PF}_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{PF}_{REP}(i, \pi_k)$$
$$+ p^{IN}_{i,\pi_s} + p^{OUT}_{i,\pi_s} + x_{CPU}{}^{IN}_{i,\pi_s} + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{NIC}{}^{IN}_{c,\pi_s} \right) + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{IN}_{i+z,\pi_s}$$
$$+ \sum_{k=s+1}^{m-1} int^{PF}_{REP}(i+z, \pi_k) + dest^{PF}_{REP}(i+z, \pi_m) - t_{i+z}$$

$$remote^{CONC}_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{CONC}_{REP}(i, \pi_k)$$
$$+ x_{CPU}{}^{IN}_{i,\pi_s} + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{NIC}{}^{IN}_{c,\pi_s} \right) + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{IN}_{i+z,\pi_s}$$
$$+ \sum_{k=s+1}^{m-1} int^{CONC}_{REP}(i+z, \pi_k) + dest^{CONC}_{REP}(i+z, \pi_m) - t_{i+z}$$

$$remote^{LAZY}_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{LAZY}_{REP}(i, \pi_k)$$
$$+ \begin{Bmatrix} x_{CPU}{}^{IN}_{i,\pi_s} \; if \; processing \; delayed \\ p^{IN}_{i,\pi_s} + p^{OUT}_{i,\pi_s} + x_{CPU}{}^{IN}_{i,\pi_s} \; otherwise \end{Bmatrix}$$
$$+ \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{NIC}{}^{IN}_{c,\pi_s} \right) + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{IN}_{i+z,\pi_s}$$
$$+ \sum_{k=s+1}^{m-1} int^{LAZY}_{REP}(i+z, \pi_k) + dest^{LAZY}_{REP}(i+z, \pi_m) - t_{i+z}$$

The transmit-first equation is the same as the equation given in the previous chapter when the critical computer is not the destination and the total network card transmission times dominate processing times.

The other extreme case is when the time the CPU requires to process a command dominates the total network card transmission time of the command. In this case, the critical computer time is equal to (1) the amount of time that elapses from the moment the CPU begins to perform tasks for command i to the moment the CPU completes that task for command $i + z - 1$, plus (2) the amount of time that elapses from the moment the CPU begins to perform tasks for command $i + z$ to the moment the network transmits it to the next downstream computer. The first term is scheduling policy independent because as explained above, the total amount of time the CPU spends on a command is independent of the scheduling policy. The second term depends on the scheduling policy. When the transmit-first and concurrent scheduling policies are used, the network card is idle when the CPU beings to perform tasks for command $i + z$. The reason is that the network card completes transmitting command $i + z - 1$ before the CPU finishes processing command $i + z - 1$. Therefore, for these two policies, the second term is equal to ${x_{CPU}}_{i+z,\pi_s}^{IN} + pos(\pi_s, \pi_{s+1}) * {x_{NIC}}_{i+z,\pi_s}^{IN}$. When the process-first policy is used, the network card is not idle when the CPU begins to process command $i + z$. However, it is idle by the time the CPU begins to transmit command $i + z$ because the network card catches up to the CPU while the CPU processes command $i + z$. Therefore, for this policy, the second term is equal to $p_{i+z,\pi_s}^{IN} + p_{i+z,\pi_s}^{OUT} + {x_{CPU}}_{i+z,\pi_s}^{IN} + pos(\pi_s, \pi_{s+1}) * {x_{NIC}}_{i+z,\pi_s}^{IN}$.

The lazy policy is more complicated because the network card may or may not be idle when the CPU begins to transmit command $i + z$ is the lazy policy. Consider first the case

when the network card catches up with the CPU. One way this happens is that the CPU does not delay the processing of either command $i+z-1$ or $i+z$. In this case, the lazy policy performs the same as the process-first policy. Another way this happens is when the CPU completes transmitting command $i+z-1$ and $i+z$ before processing them. Here, the lazy policy performs the same as the transmit-first policy. The only way the network card does not catch up to the CPU is when the CPU performs most of the transmission task for command $i+z-1$ after processing command $i+z-1$ and it performs a part of the transmission task for command $i+z$ before processing command $i+z$, then the network card does not have a chance to catch up to the CPU. In the worst case, the CPU transmits command $i+z-1$ to all but one destination after processing and transmits command $i+z$ to the next downstream computer before processing. In this case, command $i+z$ suffers an additional delay equal to $\left(fan_{\pi_s}-1\right)*{x_{NIC}}_{i+z-1,\pi_s}^{IN}-\left(fan_{\pi_s}-2\right)*{x_{CPU}}_{i+z-1,\pi_s}^{IN}-{x_{CPU}}_{i+z,\pi_s}^{IN}$. We subtract the CPU transmission time terms because during these times, the network card was catching up with the CPU.

Hence, we can finally state the response time equations when the critical computer is not the destination computer and processing times dominate the total network card transmission times.

$$\begin{aligned} remote_{REP,i+z,j}^{TF} = & \sum_{k=1}^{m-1} d(\pi_k,\pi_{k+1}) + \sum_{k=1}^{s-1} int_{REP}^{TF}(i,\pi_k) \\ & + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * {x_{CPU}}_{c,\pi_s}^{IN} + p_{c,\pi_s}^{IN} + p_{c,\pi_s}^{OUT}\right) \\ & + {x_{CPU}}_{i+z,\pi_s}^{IN} + pos(\pi_s,\pi_{s+1}) * {x_{NIC}}_{i+z,\pi_s}^{IN} \\ & + \sum_{k=s+1}^{m-1} int_{REP}^{TF}(i+z,\pi_k) + dest_{REP}^{TF}(i+z,\pi_m) - t_{i+z} \end{aligned}$$

$$remote^{PF}_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{PF}_{REP}(i, \pi_k)$$
$$+ \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}^{IN}_{c,\pi_s} + p^{IN}_{c,\pi_s} + p^{OUT}_{c,\pi_s} \right)$$
$$+ p^{IN}_{i+z,\pi_s} + p^{OUT}_{i+z,\pi_s} + x_{CPU}{}^{IN}_{i+z,\pi_s} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{IN}_{i+z,\pi_s}$$
$$+ \sum_{k=s+1}^{m-1} int^{PF}_{REP}(i+z, \pi_k) + dest^{PF}_{REP}(i+z, \pi_m) - t_{i+z}$$

$$remote^{CONC}_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{CONC}_{REP}(i, \pi_k)$$
$$+ \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}^{IN}_{c,\pi_s} + p^{IN}_{c,\pi_s} + p^{OUT}_{c,\pi_s} \right)$$
$$+ x_{CPU}{}^{IN}_{i+z,\pi_s} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{IN}_{i+z,\pi_s}$$
$$+ \sum_{k=s+1}^{m-1} int^{CONC}_{REP}(i+z, \pi_k) + dest^{CONC}_{REP}(i+z, \pi_m) - t_{i+z}$$

And finally the more complicated lazy equation is

$$remote^{LAZY}_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{LAZY}_{REP}(i, \pi_k)$$
$$+ \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}^{IN}_{c,\pi_s} + p^{IN}_{c,\pi_s} + p^{OUT}_{c,\pi_s} \right) + x_{CPU}{}^{IN}_{i+z,\pi_s}$$
$$+ \begin{Bmatrix} pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{IN}_{i+z,\pi_s} \; if \; processing \; is \; delayed \\ p^{IN}_{i+z,\pi_s} + p^{OUT}_{i+z,\pi_s} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{IN}_{i+z,\pi_s} \; otherwise \end{Bmatrix}$$
$$+ \begin{Bmatrix} 0 \; if \; network \; card \; caught \; up \; with \; CPU; \; otherwise \; it \; is \\ \left(fan_{\pi_s} - 1\right) * x_{NIC}{}^{IN}_{i+z-1,\pi_s} - \left(fan_{\pi_s} - 2\right) * x_{CPU}{}^{IN}_{i+z-1,\pi_s} - x_{CPU}{}^{IN}_{i+z,\pi_s} \end{Bmatrix}$$
$$+ \sum_{k=s+1}^{m-1} int^{LAZY}_{REP}(i+z, \pi_k) + dest^{LAZY}_{REP}(i+z, \pi_m) - t_{i+z}$$

The transmit-first equation is exactly the same as the equation given in the previous chapter when the critical computer is not the destination and processing times dominate the total network card transmission times.

Centralized Response Times

We can derive the centralized equations from the replicated equations using the same approach as in all of the other instances we had to do so. We start with the replicated equations and modify them to account for the two main differences in the centralized and replicated architectures. One, in the centralized architecture only the master computer processes input commands and all computers process outputs, while in the replicated architecture all computers process both inputs and outputs. Two, in the centralized architecture computers distribute outputs, while in the replicated architecture they distribute inputs. Since we have performed this modification procedure for 1) both quiescent state and continuous commands in the previous chapter and 2) quiescent state commands in this chapter, here we state the centralized equations for consecutive commands without derivation or explanation. The local response times of master user commands are given by

$$local^{TF}_{CENT,i+z,j} = \sum_{c=i}^{i+z} p^{IN}_{i,\pi_1} + \left(\sum_{c=i}^{i+z} dest^{TF}_{CENT}(c,j) \right) - t_{i+z}$$

$$local^{PF}_{CENT,i+z,j} = \sum_{c=i}^{i+z} p^{IN}_{i,\pi_1} + \left(\sum_{c=i}^{i+z} dest^{PF}_{CENT}(c,j) \right) - t_{i+z}$$

$$local^{CONC}_{CENT,i+z,j} = \sum_{c=i}^{i+z} p^{IN}_{i,\pi_1} + \left(\sum_{c=i}^{i+z} dest^{CONC}_{CENT}(c,j) \right) - t_{i+z}$$

$$local^{LAZY}_{CENT,i+z,j} = \sum_{c=i}^{i+z} p^{IN}_{i,\pi_1} + \left(\sum_{c=i}^{i+z} dest^{LAZY}_{CENT}(c,j) \right) - t_{i+z}$$

The equations for the remote response times of master user commands when the critical computer is the destination are given by

$$remote^{TF}_{CENT,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int^{TF}_{CENT}(i, \pi_k) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_m} * x_{CPU}{}^{OUT}_{c,\pi_m} + p^{OUT}_{c,\pi_m} \right) + dest^{TF}_{CENT}(i+z, \pi_m) - t_{i+z}$$

$$remote^{PF}_{CENT,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int^{PF}_{CENT}(i, \pi_k) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_m} * x_{CPU}{}^{OUT}_{c,\pi_m} + p^{OUT}_{c,\pi_m} \right) + dest^{PF}_{CENT}(i+z, \pi_m) - t_{i+z}$$

$$remote^{CONC}_{CENT,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int^{CONC}_{CENT}(i, \pi_k) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_m} * x_{CPU}{}^{OUT}_{c,\pi_m} + p^{OUT}_{c,\pi_m} \right) + dest^{CONC}_{CENT}(i+z, \pi_m) - t_{i+z}$$

$$remote^{LAZY}_{CENT,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int^{LAZY}_{CENT}(i, \pi_k) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_m} * x_{CPU}{}^{OUT}_{c,\pi_m} + p^{OUT}_{c,\pi_m} \right) + dest^{LAZY}_{CENT}(i+z, \pi_m) - t_{i+z}$$

The equations for the remote response times of master user commands when the critical computer is not the destination and the total network cart transmission times dominate the processing times are given by

$$remote^{TF}_{CENT,i+z,j} = p^{IN}_{i,\pi_1} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{TF}_{CENT}(i, \pi_k)$$

$$+ x_{CPU}{}^{OUT}_{i,\pi_s} + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{NIC}{}^{OUT}_{c,\pi_s} \right) + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{OUT}_{i+z,\pi_s}$$

$$+ \sum_{k=s+1}^{m-1} int^{TF}_{CENT}(i+z, \pi_k) + dest^{TF}_{CE\ NT}(i+z, \pi_m) - t_{i+z}$$

$$remote^{PF}_{CENT,i+z,j} = p^{IN}_{i,\pi_1} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{PF}_{CENT}(i, \pi_k)$$

$$+ p^{OUT}_{i,\pi_s} + x_{CPU}{}^{OUT}_{i,\pi_s} + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{NIC}{}^{OUT}_{c,\pi_s} \right) + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{OUT}_{i+z,\pi_s}$$

$$+ \sum_{k=s+1}^{m-1} int^{PF}_{CENT}(i+z, \pi_k) + dest^{PF}_{CENT}(i+z, \pi_m) - t_{i+z}$$

$$remote^{CONC}_{CENT,i+z,j} = p^{IN}_{i,\pi_1} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{CONC}_{CENT}(i, \pi_k)$$

$$+ x_{CPU}{}^{OUT}_{i,\pi_s} + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{NIC}{}^{OUT}_{c,\pi_s} \right) + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{OUT}_{i+z,\pi_s}$$

$$+ \sum_{k=s+1}^{m-1} int^{CONC}_{CENT}(i+z, \pi_k) + dest^{CONC}_{CENT}(i+z, \pi_m) - t_{i+z}$$

$$remote^{LAZY}_{CENT,i+z,j} = p^{IN}_{i,\pi_1} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{LAZY}_{CENT}(i, \pi_k)$$

$$+ \begin{Bmatrix} x_{CPU}{}^{OUT}_{i,\pi_s}\ if\ processing\ delayed \\ p^{OUT}_{i,\pi_s} + x_{CPU}{}^{OUT}_{i,\pi_s}\ otherwise \end{Bmatrix}$$

$$+ \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{NIC}{}^{OUT}_{c,\pi_s} \right) + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{OUT}_{i+z,\pi_s}$$

$$+ \sum_{k=s+1}^{m-1} int^{LAZY}_{CENT}(i+z, \pi_k) + dest^{LAZY}_{CENT}(i+z, \pi_m) - t_{i+z}$$

The equations for the remote response times of master user commands when the critical computer is neither the source nor the destination and the processing times dominate the total network cart transmission times are given by

$$remote^{TF}_{CENT,i+z,j} = p^{IN}_{i,\pi_1} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{TF}_{CENT}(i, \pi_k)$$
$$+ \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}^{OUT}_{c,\pi_s} + p^{OUT}_{c,\pi_s} \right)$$
$$+ x_{CPU}{}^{OUT}_{i+z,\pi_s} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{OUT}_{i+z,\pi_s}$$
$$+ \sum_{k=s+1}^{m-1} int^{TF}_{CENT}(i+z, \pi_k) + dest^{TF}_{CENT}(i+z, \pi_m) - t_{i+z}$$

$$remote^{PF}_{CENT,i+z,j} = p^{IN}_{i,\pi_1} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{PF}_{CENT}(i, \pi_k)$$
$$+ \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}^{OUT}_{c,\pi_s} + p^{OUT}_{c,\pi_s} \right) + p^{OUT}_{i+z,\pi_s}$$
$$+ x_{CPU}{}^{OUT}_{i+z,\pi_s} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{OUT}_{i+z,\pi_s}$$
$$+ \sum_{k=s+1}^{m-1} int^{PF}_{CENT}(i+z, \pi_k) + dest^{PF}_{CENT}(i+z, \pi_m) - t_{i+z}$$

$$remote^{CONC}_{CENT,i+z,j} = p^{IN}_{i,\pi_1} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{CONC}_{CENT}(i, \pi_k)$$
$$+ \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}^{OUT}_{c,\pi_s} + p^{OUT}_{c,\pi_s} \right)$$
$$+ x_{CPU}{}^{OUT}_{i+z,\pi_s} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{OUT}_{i+z,\pi_s}$$
$$+ \sum_{k=s+1}^{m-1} int^{CONC}_{CENT}(i+z, \pi_k) + dest^{CONC}_{CENT}(i+z, \pi_m) - t_{i+z}$$

$$remote_{CENT,i+z,j}^{LAZY} = p_{i,\pi_1}^{IN} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int_{CENT}^{LAZY}(i, \pi_k)$$

$$+ \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}_{c,\pi_s}^{IN} + p_{c,\pi_s}^{OUT} \right) + x_{CPU}{}_{i+z,\pi_s}^{OUT}$$

$$+ \begin{Bmatrix} pos(\pi_s, \pi_{s+1}) * x_{NIC}{}_{i+z,\pi_s}^{OUT} \; if \; processing \; is \; delayed \\ p_{i+z,\pi_s}^{OUT} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}_{i+z,\pi_s}^{OUT} \; otherwise \end{Bmatrix}$$

$$+ \begin{Bmatrix} 0 \; if \; network \; card \; caught \; up \; with \; CPU; otherwise, it \; is \\ \left(fan_{\pi_s} - 1 \right) * x_{NIC}{}_{i+z-1,\pi_s}^{OUT} - \left(fan_{\pi_s} - 2 \right) * x_{CPU}{}_{i+z-1,\pi_s}^{OUT} - x_{CPU}{}_{i+z,\pi_s}^{OUT} \end{Bmatrix}$$

$$+ \sum_{k=s+1}^{m-1} int_{CENT}^{LAZY}(i+z, \pi_k) + dest_{CENT}^{LAZY}(i+z, \pi_m) - t_{i+z}$$

Finally, the equations for the remote response times of master user commands when the critical computer is the source and the processing times dominate the total network cart transmission times are given by

$$remote_{CENT,i+z,j}^{TF} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}_{c,\pi_s}^{OUT} + p_{c,\pi_s}^{IN} + p_{c,\pi_s}^{OUT} \right)$$

$$+ x_{CPU}{}_{i+z,\pi_s}^{OUT} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}_{i+z,\pi_s}^{OUT}$$

$$+ \sum_{k=s+1}^{m-1} int_{CENT}^{TF}(i+z, \pi_k) + dest_{CENT}^{TF}(i+z, \pi_m) - t_{i+z}$$

$$remote_{CENT,i+z,j}^{PF} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}_{c,\pi_s}^{OUT} + p_{c,\pi_s}^{IN} + p_{c,\pi_s}^{OUT} \right)$$

$$+ p_{i+z,\pi_s}^{IN} + p_{i+z,\pi_s}^{OUT} + x_{CPU}{}_{i+z,\pi_s}^{OUT} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}_{i+z,\pi_s}^{OUT}$$

$$+ \sum_{k=s+1}^{m-1} int_{CENT}^{PF}(i+z, \pi_k) + dest_{CENT}^{PF}(i+z, \pi_m) - t_{i+z}$$

$$remote^{CONC}_{CENT,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}^{OUT}_{c,\pi_s} + p^{IN}_{c,\pi_s} + p^{OUT}_{c,\pi_s} \right)$$

$$+ x_{CPU}{}^{OUT}_{i+z,\pi_s} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{OUT}_{i+z,\pi_s}$$

$$+ \sum_{k=s+1}^{m-1} int^{CONC}_{CENT}(i+z, \pi_k) + dest^{CONC}_{CENT}(i+z, \pi_m) - t_{i+z}$$

$$remote^{LAZY}_{CENT,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{CPU}{}^{IN}_{c,\pi_s} + p^{IN}_{c,\pi_s} + p^{OUT}_{c,\pi_s} \right)$$

$$+ p^{IN}_{i+z,\pi_s} + x_{CPU}{}^{OUT}_{i+z,\pi_s} + \begin{Bmatrix} pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{OUT}_{i+z,\pi_s} \; if \; processing \; is \; delayed \\ p^{OUT}_{i+z,\pi_s} + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{OUT}_{i+z,\pi_s} \; otherwise \end{Bmatrix}$$

$$+ \begin{Bmatrix} 0 \; if \; network \; card \; caught \; up \; with \; CPU; otherwise, it \; is \\ \left(fan_{\pi_s} - 1\right) * x_{NIC}{}^{OUT}_{i+z-1,\pi_s} - \left(fan_{\pi_s} - 2\right) * x_{CPU}{}^{OUT}_{i+z-1,\pi_s} - x_{CPU}{}^{OUT}_{i+z,\pi_s} \end{Bmatrix}$$

$$+ \sum_{k=s+1}^{m-1} int^{LAZY}_{CENT}(i+z, \pi_k) + dest^{LAZY}_{CENT}(i+z, \pi_m) - t_{i+z}$$

4.3.6 Implications for Consecutive Commands by a Single User

We can compare the response times of consecutive commands with different scheduling policies by analyzing the difference in the respective equations. As in the previous chapter, when we compared unicast and multicast equations, we can make two comparisons. We can compare the absolute response times of consecutive commands with each scheduling policy. We can also compare how much worse do response times of consecutive commands become with each scheduling policies. Both comparisons will give the same answers. The reason is that we can obtain the former by adding the latter to the response times of commands entered in a quiescent state, and we can obtain the latter by subtracting the response times of commands entered in a quiescent state from the former. Unlike in the previous chapter, in which we did the former comparisons because the

difference equations were simpler, in this chapter, we do the latter because it gives cleaner difference equations. Moreover, we present the simplified difference equations, in which only the terms that matter are shown.

Local Response Times

Let us start with the local response times. The difference in the increase in response times of consecutive commands for any two policies A and B in the replicated and centralized architectures is given by

$$local^{A}_{REP,i+z,j} - local^{B}_{REP,i+z,j} = dest^{A}_{REP}(i+z,j) - dest^{B}_{REP}(i+z,j)$$

$$local^{A}_{CENT,i+z,j} - local^{B}_{CENT,i+z,j} = dest^{A}_{CENT}(i+z,j) - dest^{B}_{CENT}(i+z,j)$$

As these difference equations show, the difference in the response times of consecutive commands for any two scheduling policies is equal to the local response time difference provided by these two policies for command $i+z$ when it is entered in a quiescent state. This does not mean that the absolute response times is at most this difference. All of the individual local response time equations have terms which are a function of the amount of time needed to perform the tasks for all of the commands. Hence, the absolute response times of consecutive commands can be high. However, the extent to which one scheduling policy can provide better response times over another is given by the above difference equations. Therefore, when choosing the scheduling policy, we need to analyze only the local response times of quiescent state commands and the results apply to consecutive commands also.

Remote Response Times

The remote response time comparisons are more complicated as we have several cases to consider. Consider the case when the destination computer is the critical computer.

In this case, the difference in the increase in response times of consecutive commands for any two policies A and B in the replicated and centralized architectures is given by

$$remote^{A}_{REP,i+z,j} - remote^{B}_{REP,i+z,j} = \sum_{k=1}^{m-1} int^{A}_{REP}(i,\pi_k) + dest^{A}_{REP}(i+z,\pi_m) - \sum_{k=1}^{m-1} int^{B}_{REP}(i,\pi_k) - dest^{B}_{REP}(i+z,\pi_m)$$

$$remote^{A}_{CENT,i+z,j} - remote^{B}_{CENT,i+z,j} = \sum_{k=1}^{m-1} int^{A}_{CENT}(i,\pi_k) + dest^{A}_{CENT}(i+z,\pi_m) - \sum_{k=1}^{m-1} int^{B}_{CENT}(i,\pi_k) - dest^{B}_{CENT}(i+z,\pi_m)$$

As these difference equations show, the absolute difference in the response times of consecutive commands between any two scheduling policies is independent of the number of consecutive commands. Also, the difference equations contain only the terms which we have already compared above for the quiescent state commands. The analysis for quiescent state commands may not apply directly, however. The reason is that the intermediate delays are for command i while the destination delay is for command $i + z$. If the two commands have different processing and transmission costs, then the quiescent state analysis does not apply directly. We can still make some inferences, however, based on the quiescent state analysis. For instance, if the two commands have similar processing and transmission costs, then we can use the quiescent state analysis for command i directly. Otherwise, we can separately compare the destination delays and the sum of the intermediate delays. If both are larger (smaller) for one policy than for another, then the differences illustrated for the quiescent state commands are magnified; otherwise, the differences are reduced.

Consider now the case when the critical computer is not the destination and the total network transmission times dominate processing times. In this case, the difference equations for the transmit-first and process-first response times are given by

$$remote^{TF}_{REP,i+z,j} - remote^{PF}_{REP,i+z,j}$$

$$= \left(\sum_{k=1}^{s-1} int^{TF}_{REP}(i,\pi_k) + \sum_{k=s+1}^{m-1} int^{TF}_{REP}(i+z,\pi_k) + dest^{TF}_{REP}(i+z,\pi_m) \right)$$

$$- \left(\sum_{k=1}^{s-1} int^{PF}_{REP}(i,\pi_k) + p^{IN}_{i,\pi_s} + p^{OUT}_{i,\pi_s} + \sum_{k=s+1}^{m-1} int^{PF}_{REP}(i+z,\pi_k) + dest^{PF}_{REP}(i+z,\pi_m) \right)$$

$$remote^{TF}_{CENT,i+z,j} - remote^{PF}_{CENT,i+z,j}$$

$$= \left(\sum_{k=1}^{s-1} int^{TF}_{CENT}(i,\pi_k) + \sum_{k=s+1}^{m-1} int^{TF}_{CENT}(i+z,\pi_k) + dest^{TF}_{CENT}(i+z,\pi_m) \right)$$

$$- \left(\sum_{k=1}^{s-1} int^{PF}_{CENT}(i,\pi_k) + p^{IN}_{i,\pi_s} + p^{OUT}_{i,\pi_s} + \sum_{k=s+1}^{m-1} int^{PF}_{CENT}(i+z,\pi_k) + dest^{PF}_{CENT}(i+z,\pi_m) \right)$$

As in the previous case, these difference equations show that the absolute difference in the response times of consecutive commands between any two scheduling policies is independent of the number of consecutive commands. The difference equations essentially capture the transmit-first and process-first response time differences of quiescent state commands. The delay terms of all of the computers except the critical computer do so by definition. Moreover, the difference between transmit-first and process-first of an intermediate computer π_s is exactly $p^{IN}_{i,\pi_s} + p^{OUT}_{i,\pi_s}$, which is equal to the difference in the transmit-first and process-first delays of computer π_s when command i is entered in a quiescent state. Therefore, as in the case when the destination was the critical computer, we can use the quiescent state analysis to deduce which of these two policies provides better

response times for consecutive commands. We have to apply the analysis separately for the delays of upstream from the critical computer as they are for command i and of the critical computer and the computers downstream from it as they are for command $i + z$.

The difference equations between the transmit-first and other policies give similar results. The only new twist is the difference in the critical computer delay. More specifically, $remote^{TF}_{REP,i+z,j} - remote^{CONC}_{REP,i+z,j} = 0$, and $0 \leq remote^{TF}_{REP,i+z,j} - remote^{LAZY}_{REP,i+z,j} \leq p^{IN}_{i,\pi_s} + p^{OUT}_{i,\pi_s}$. The differences in the critical computer delay when it is an intermediate computer for any two scheduling policies is at most $p^{IN}_{i,\pi_s} + p^{OUT}_{i,\pi_s}$. That result is encapsulated in the analysis of response times for quiescent state commands. Thus, again, we can reuse the analysis of quiescent state commands suffices to deduce differences in response times of consecutive commands.

Consider now the case when the critical computer is not the destination and the processing times dominate the total network transmission times. When we derived the response time equations, we had to separately consider the cases when the critical computer is and is not the source in the centralized architecture. However, the differences in the response time equations are the same in both cases. The difference equation for the transmit-first and process-first response times are almost identical to the difference equation from the previous case.

$$
\begin{aligned}
& remote^{TF}_{REP,i+z,j} - remote^{PF}_{REP,i+z,j} \\
&= \left(\sum_{k=1}^{s-1} int^{TF}_{REP}(i, \pi_k) + \sum_{k=s+1}^{m-1} int^{TF}_{REP}(i+z, \pi_k) + dest^{TF}_{REP}(i+z, \pi_m) \right) \\
&- \left(\sum_{k=1}^{s-1} int^{PF}_{REP}(i, \pi_k) + p^{IN}_{i+z,\pi_s} + p^{OUT}_{i+z,\pi_s} + \sum_{k=s+1}^{m-1} int^{PF}_{REP}(i+z, \pi_k) + dest^{PF}_{REP}(i+z, \pi_m) \right)
\end{aligned}
$$

$$remote^{TF}_{CENT,i+z,j} - remote^{PF}_{CENT,i+z,j}$$

$$= \left(\sum_{k=1}^{s-1} int^{TF}_{CENT}(i,\pi_k) + \sum_{k=s+1}^{m-1} int^{TF}_{CENT}(i+z,\pi_k) + dest^{TF}_{CENT}(i+z,\pi_m) \right)$$

$$- \left(\sum_{k=1}^{s-1} int^{PF}_{CENT}(i,\pi_k) + p^{IN}_{i+z,\pi_s} + p^{OUT}_{i+z,\pi_s} + \sum_{k=s+1}^{m-1} int^{PF}_{CENT}(i+z,\pi_k) + dest^{PF}_{CENT}(i+z,\pi_m) \right)$$

The only difference between these difference equations and those from the previous case is that the intermediate computer delay difference is equal to $p^{IN}_{i+z,\pi_s} + p^{OUT}_{i+z,\pi_s}$ instead of $p^{IN}_{i,\pi_s} + p^{OUT}_{i,\pi_s}$. But $p^{IN}_{i+z,\pi_s} + p^{OUT}_{i+z,\pi_s}$ is exactly the difference the transmit-first and process-first response times of command $i+z$ when it is entered in a quiescent state. Therefore, we can use the quiescent state analysis to deduce which of these two policies provides better response times for consecutive commands.

The difference equations between the transmit-first and other policies give similar results. As in the previous case, the only new twist is the difference in the critical computer delay. More specifically, $remote^{TF}_{REP,i+z,j} - remote^{CONC}_{REP,i+z,j} = 0$, and $0 \leq remote^{TF}_{REP,i+z,j} - remote^{LAZY}_{REP,i+z,j} \leq p^{IN}_{i,\pi_s} + p^{OUT}_{i,\pi_s} + \left(fan_{\pi_s} - 1\right) * x_{NIC}{}^{IN}_{i+z-1,\pi_s} - \left(fan_{\pi_s} - 2\right) * x_{CPU}{}^{IN}_{i+z-1,\pi_s} - x_{CPU}{}^{IN}_{i+z,\pi_s}$. Thus, the differences in the critical computer delay when it is an intermediate computer for any two of the transmit-first, process-first, and concurrent scheduling policies is at most $p^{IN}_{i,\pi_s} + p^{OUT}_{i,\pi_s}$. That result is encapsulated in the analysis of response times for quiescent state commands. Thus, the analysis of quiescent state commands suffices to deduce differences in response times of consecutive commands for these three policies. However, there is a new result that is not captured by the quiescent state analysis when we consider lazy response times. In particular, the quiescent state analysis shows that compared to transmit-first response times, lazy response times can never be more than $p^{IN}_{i,\pi_s} + p^{OUT}_{i,\pi_s}$;

however, as the above difference equations show, the difference can be as high as $p_{i,\pi_s}^{IN} + p_{i,\pi_s}^{OUT} + (fan_{\pi_s} - 1) * x_{NIC}{}_{i+z-1,\pi_s}^{IN} - (fan_{\pi_s} - 2) * x_{CPU}{}_{i+z-1,\pi_s}^{IN} - x_{CPU}{}_{i+z,\pi_s}^{IN}$ for consecutive commands.

4.3.7 Simultaneous Commands

We have considered the remote response times of commands entered by the same user in a quiescent or a non-quiescent state. The final case to consider is the remote response times of simultaneous commands entered by different users.

As in the previous chapter, when simultaneous commands are entered by different users, the worst possible remote response times for i command entered by user_j occurs when (a) it arrives at the critical computer at the same time as commands entered by other users, and (b) the critical computer performs the tasks for command i last. Therefore analysis of the remote response times of command i is almost the same as the analysis of the commands entered consecutively by the same user. The only difference is that the critical computer term includes the time required to perform tasks of commands entered by all of the other users.

4.4 Multi-Core Scheduling Policies

The single-core scheduling policies we have analyzed also have multi-core equivalents. The multi-core equivalent of the concurrent policy schedules the transmission and processing tasks on different cores. The lazy policy can be adapted to multi-core scenarios by delaying the processing task on all of the (one of the) cores if the processing task is (is not) parallelizable. Moreover, the sequential schemes would simply use all cores for one and then for the other task.

From a general framework perspective, the processing task is opaque – hence, the policies cannot parallelize it explicitly. As a result, our analysis assumes that the processing task is always scheduled on one core. The CPU transmission task, on the other hand, is parallelizable by nature. For example, two cores can transmit to two different destinations in parallel. More importantly, as our framework defines the transmission tasks on each computer, it can explicitly parallelize it.

Static vs. Dynamic Scheduling

There are two ways of selecting cores for performing the transmission and processing tasks. One approach is to statically assign these tasks to cores (this technique is also known as static or partitioned scheduling in real-time systems). When a command is received, the processing and transmission tasks are performed only by the cores assigned to them. One issue with this approach is that cores may be idle even if there are tasks ready for execution. To illustrate, consider a dual-core system in which the processing and transmission tasks are mapped to different cores. In this case, if the processing core finishes before the transmission core, it will idle even though the transmission task can be completed quicker overall if the remainder of the transmission task is evenly divided among the two cores. An alternative approach that does not have this problem is dynamically choosing the cores on which the tasks execute as cores become available. This technique is also known as dynamic or global scheduling in real-time systems. In our example, a dynamic scheduler would divide the transmission task among the two cores once the processing core became available. Therefore, the transmission task would complete sooner.

One issue with dynamic scheduling is that there is a cost associated with task migration between cores. However, if these costs are low, then, overall, a dynamic scheduler would offer better performance in collaborative systems.

In our system, we use static scheduling, even though dynamic scheduling may offer better performance. One reason we do this is because with dynamic scheduling it becomes difficult to model and predict response times. To illustrate, consider the transmit-first delay of the intermediate computer π_k which has $c = 2$ cores that can be used for performing the transmission and processing tasks. Suppose that the computer forwards command to two destinations and that it transmits to computer π_{k+1} second, that is, $pos(\pi_k, \pi_{k+1}) = 2$. Suppose the dynamic scheduler uses both cores to perform the transmission task. An important issue that arises when multiple cores perform the transmission task in parallel is that the order in which the network card transmits a message to the destinations is unpredictable. The reason is that when cores run in parallel, they create a race-condition for sharing the bus between the memory and the cores, and either core can win. This unpredictability is problematic because it makes it impossible to accurately predict the delay of π_k. If the two cores in parallel copy the message for transmission to the first and second destination, then the core that transmits to the second destination, that is, computer π_{k+1} may or may not win the race condition for the bus. As a result, we cannot rely on the destination order to accurately predict when π_k's network card actually transmits the copy of the message to π_{k+1}. One potential solution is to apply some sort of probabilistic model for the order in which the cores will queue the messages for transmission. Unfortunately, we are not aware of a model that provides this information.

Since the multi-core versions of the transmit-first, process-first, and lazy policies may all dynamically assign multiple cores to perform the transmission task, we cannot predict the performance for any of them. The only multi-core policy for which we can accurately calculate remote response times is the concurrent policy. Moreover, we can only do so for when a single core is used for transmission. When the concurrent policy is used on a dual-core computer, one of the cores is used for processing and the other for transmission. On computers with more than two cores, once again more than one core can be used for transmission even with the concurrent policy, and hence accurately calculating remote response times is not possible. Therefore, we only consider the multi-core scheduling policy when the processing and transmission tasks are performed by two different cores. We refer to this policy as the parallel policy.

Another other reason we use static instead of dynamic scheduling is that when non-blocking communication is used and the network card is the bottleneck, increasing the number of cores that perform the transmission task in parallel does not improve remote response times. In all of the collaboration logs we recorded, regardless of message size or the number of destinations to which a message is transmitted to, the CPU transmission cost was lower than the network card transmission cost. This was true on all the single-core devices we tested, including the P3 desktop, P4 desktop, and the netbook. Moreover, it was true when a single-core was used for the transmission task on the dual-core Core2 desktop.

One performance drawback of not using dynamic scheduling is that we do not support parallel processing of an output for command i with the processing of input command $i + 1$. This requires a dynamic scheduler as it would parallelize them when the transmission task completes. Such a migration is useful, for example, for a master computer

when both input and output processing are computationally expensive because it could perform the input processing task of command $i + 1$ in parallel while performing the output processing task of command i. This would in turn reduce the response time to the local user of command $i + 1$. However, in the applications we consider, the input and output processing costs are never both high. In fact, if one is high, usually the other is negligible. Thus, we expect the benefit of performing input and output processing of different commands in parallel to be low for the applications we consider.

Parallel Scheduling Policy:

We only consider the parallel scheduling policy in which the processing and transmission tasks are performed by two different cores. We do not consider other policies, such as those in which task migration is supported, because it is not possible to predict remote response times for them. Moreover, based on the collaboration logs we used to collect processing and transmission times, it is not clear that these policies offer any performance benefit over the parallel policy.

Work-Conserving Scheduling

An important issue with the parallel policy is whether or not it is work-conserving [37]. A work-conserving scheduler does not idle a processor or a core if it can be used to perform some task. All of our single-core analysis has assumed work-conserving scheduling. Such scheduling is consistent with our assumption that whenever a command arrives at an idle computer, the computer immediately begins performing tasks for the command. When multi-core devices are considered, a work-conserving algorithm would require that we support task migration and use multiple cores to perform the transmission task whenever it is

possible to do so. Since we do not support either of these, we cannot support work-conserving scheduling. We do, however, have work-conserving scheduling on a per core basis. In our scheduling policy, one core performs the transmission task while another core performs the processing task. When a core completes the processing (transmission) tasks of command i, it does not wait for the other core to complete the transmission (processing) tasks of command i when command $i + 1$ has already been received. In particular, the transmission core can perform the transmission task of command $i + 1$ while the processing core is performing the processing tasks of command i, and vice versa. Hence, we still satisfy our assumption of not delaying tasks of a command once the command has been received.

4.5 Multi-Core Performance Analysis

We can derive the response times for the parallel multi-core scheduling policy using the same approach as for the response times with single-core policies. In particular, we need to derive the delays of the intermediate and destination computers on the path from the source to the destination. As before, we will derive the delays for the replicated architecture first. The delays for the centralized architecture follow directly from those of the replicated architecture.

4.5.1 Replicated Remote Response Time

Recall from above that the replicated remote response time of command i to computer j along path π is given by

$$remote_{REP,i,j}^{PARA} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{REP}^{PARA}(i, \pi_k) + dest_{REP}^{PARA}(i, \pi_m)$$

where $d(\pi_k, \pi_{k+1})$ is the network latency between the k^{th} and $k + 1^{st}$ computers on path π, $int_{REP}^{PARA}(i, \pi_k)$ is the delay of the k^{th} intermediate computer on the path, and $dest_{REP}^{PARA}(i, \pi_m)$ is the delay of the destination computer. Since the first term is a sum of network latencies on the path, it is independent of scheduling policy.

The derivation of the delay terms is easier with the multi-core than the single-core policies because the transmission and processing tasks do not interfere with each other. Consider the parallel delay of the intermediate computer π_k which has $c = 2$ cores that can be used for performing the transmission and processing tasks. Its delay is equal to

$$int_{REP}^{PARA}(i, \pi_k) = x_{NIC}{}_{i,\pi_k}^{IN} + pos(\pi_k, \pi_{k+1}) * x_{NIC}{}_{i,\pi_k}^{IN}$$

Consider the parallel delay of the destination computer π_m. The destination delay is equal to

$$dest_{REP}^{PARA}(i, \pi_m) = p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}$$

4.5.2 Replicated Local Response Time

Recall from above that the replicated local response time of command i entered by computer j is equal to computer j's destination delay. Thus, the local response time is given by

$$local_{REP}^{PARA}(i, j) = p_{i,j}^{IN} + p_{i,j}^{OUT}$$

4.5.3 Centralized Response Times

We can obtain the centralized architecture equations from the replicated equations using the same approach we have used before: adjust the equations for the two differences in the two architectures. As we have done this several times already for other equations in this

and the previous chapter, we just state the equations. The local response time of commands entered by a master user is given by

$$local_{CENT}^{PARA}(i,j) = p_{i,j}^{OUT}$$

The remote response time for commands entered by the master user is given by

$$remote_{CENT,i,j}^{PARA} = p_{i,j}^{IN} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{CENT}^{PARA}(i,\pi_k) + dest_{CENT}^{PARA}(i,\pi_m)$$

$$int_{CENT}^{PARA}(i,\pi_k) = x_{NIC}{}_{i,\pi_k}^{OUT} + pos(\pi_k, \pi_{k+1}) * x_{NIC}{}_{i,\pi_k}^{OUT}$$

$$dest_{REP}^{PARA}(i,\pi_m) = p_{i,\pi_m}^{OUT}$$

Finally, we can obtain the equations for the local and remote response time of command i entered by slave user$_a$ whose master is user$_b$ by adding the term $x_{CPU}{}_{i,a}^{IN} + x_{NIC}{}_{i,a}^{IN} + d(a,b)$ to the response time equations.

4.5.4 Consecutive and Simultaneous Commands

The benefit of the per-core work-conserving aspect of the parallel policy increases when we consider the response times of commands entered in a non-quiescent state.

Replicated Local Response Times

Based on the above properties of the parallel policy, we can derive the replicated local response time of command $i + z$ entered by user$_j$ t_{i+z} time after entering command i, where t_{i+z} is less than the time user$_j$'s computer requires to process command i. The response time of command $i + z$ is equal to the time user$_j$'s computer requires to process commands i through $i + z$ minus the amount of time that elapses from the moment command i is entered to the moment command $i + z$ is entered, t_{i+z}. The response time is given by

$$local_{REP,i+z,j}^{PARA} = \sum_{k=i}^{i+z-1} \left(p_{i,j}^{IN} + p_{i,j}^{OUT}\right) - t_{i+z} + dest_{REP}^{PARA}(i+z,j)$$

$$= \sum_{k=i}^{i+z} \left(p_{i,j}^{IN} + p_{i,j}^{OUT}\right) - t_{i+z}$$

Remote Response Times

Consider now replicated remote response times of command entered in a non-quiescent state. As mentioned above, deriving the remote response time equations is more complicated than deriving the local ones because the commands may suffer additional delays on any computer from the source to the destination. As in the single-core case, we can derive the general remote response time equation $user_b$'s response time of command $i + z$ entered by $user_j$ is using the notion of a critical computer, which we defined earlier as the computer that takes the longest to complete the tasks for the consecutive commands. When the critical computer is an intermediate (destination) computer, then $user_b$'s remote response time of command $i + z$ is equal to 1) the amount of time that elapses from the moment command i is entered to the moment it reaches the critical computer, 2) plus the amount of time that elapses from the moment the critical computer receives command i to the moment i transmits command $i + z$ to the next computer on the path (completes processing of command $i + z$), 3) plus the amount of time that elapses from the moment the critical computer transmits command $i + z$ to the next downstream computer to the moment $user_b$ sees its output, 4) minus the amount of time that elapses from the moment command i is entered to the moment command $i + z$ is entered. The first, third, and fourth terms derived for the single-core policies can be adapted to the parallel multi-core policy in a straightforward fashion. The first term is equal to $\sum_{k=1}^{s-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int_{REP}^{PARA}(i, \pi_k)$, the third term is equal to

$\sum_{k=s}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=s+1}^{m-1} int_{REP}^{PARA}(i+z, \pi_k) + dest_{REP}^{PARA}(i+z, \pi_m)$, and the fourth term is simply equal to t_{i+z}.

As above, deriving the second term, the critical computer time, is more complicated. Unlike for the single-core scheduling policies, where the critical computer time was a function of both CPU processing and CPU transmission times, with the parallel multi-core policy, it is function of only the CPU transmission time. The reason is that even if the transmission core is falling behind the processing core, the critical computer time is not impacted by the CPU transmission times. In particular, the critical computer time depends on how long the CPU requires to process commands i through $i+z-1$ plus the amount of time the CPU requires for displaying the result of command $i+z$ to the local user once it starts to perform tasks for it. Therefore, the response time equation is given by

$$remote_{REP,i+z,j}^{PARA} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int_{REP}^{PARA}(i, \pi_k) + \sum_{c=i}^{i+z-1} \left(p_{c,\pi_m}^{IN} + p_{c,\pi_m}^{OUT}\right) + dest_{REP}^{PARA}(i+z, \pi_m) - t_{i+z}$$

When the critical computer is not the destination computer, the critical computer term depends on how long the network card requires to transmit commands i through $i+z-1$ plus the amount of time it requires to transmit command $i+z$ to the next downstream computer once to perform tasks for it. In particular, it does not depend on the CPU processing times. Thus, unlike for the single-core scheduling case, it does not matter whether the total network card transmission time and the CPU processing times dominate each other. As above, the critical computer time is equal to (1) the amount of time that elapses from the moment the critical computer receives command i to the moment the CPU transmits i to the first destination, plus (2) the amount of time that elapses from the moment the network card

begins to transmit i to the first destination to the moment it transmits $i + z - 1$ to the last destination, plus (3) the amount of time that elapses from the moment the network card begins transmitting $i + z$ to the moment it transmits it to the next downstream computer. The first term is equal to ${x_{CPU}}_{i,\pi_s}^{IN}$, the second term is equal to $\sum_{c=i}^{i+z-1}\left(fan_{\pi_s} * {x_{NIC}}_{c,\pi_s}^{IN}\right)$, and the third term is equal to $pos(\pi_s, \pi_{s+1}) * {x_{NIC}}_{i+z,\pi_s}^{IN}$. Therefore, the response time equation is given by

$$remote_{REP,i+z,j}^{PARA} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int_{REP}^{PARA}(i, \pi_k)$$
$$+ {x_{CPU}}_{i,\pi_s}^{IN} + \sum_{c=i}^{i+z-1}\left(fan_{\pi_s} * {x_{NIC}}_{c,\pi_s}^{IN}\right) + pos(\pi_s, \pi_{s+1}) * {x_{NIC}}_{i+z,\pi_s}^{IN}$$
$$+ \sum_{k=s+1}^{m-1} int_{REP}^{PARA}(i+z, \pi_k) + dest_{REP}^{PARA}(i+z, \pi_m) - t_{i+z}$$

Centralized Response Times

We can derive the centralized equations from the replicated equations by adjusting the replicated equations for the two differences between the two architectures. We do not derive the centralized equations; we simply state them. The local response time equation for commands entered by a master user is given by

$$local_{CONC,i+z,j}^{PARA} = \sum_{k=i}^{i+z-1} p_{i,j}^{IN} - t_{i+z} + dest_{CENT}^{PARA}(i+z, \pi_m)$$
$$= \sum_{k=i}^{i+z}\left(p_{i,j}^{IN} + p_{i,j}^{OUT}\right) - t_{i+z}$$

The remote response time for commands entered by a master user when the critical computer is the destination is given by

$$remote^{PARA}_{CENT,i+z,j} = p^{IN}_{i,\pi_1} + \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int^{PARA}_{CENT}(i, \pi_k)$$

$$+ \sum_{c=i}^{i+z-1} p^{OUT}_{c,\pi_m} + dest^{PARA}_{CENT}(i+z, \pi_m) - t_{i+z}$$

Unlike in the single-core case, we do not need separate centralized equations for when the critical computer is and is not the source. Separating the cases was important for the single-core policies since the source computer performs input processing in addition to output processing. The additional processing time impacted the critical computer time when it was the source. However, for the parallel policy, the processing time of any kind does not impact the critical computer time. Therefore, the response time when the critical computer is not the destination is given by

$$remote^{PARA}_{REP,i+z,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{s-1} int^{PARA}_{CENT}(i, \pi_k)$$

$$+ x_{CPU}{}^{OUT}_{i,\pi_s} + \sum_{c=i}^{i+z-1} \left(fan_{\pi_s} * x_{NIC}{}^{IN}_{c,\pi_s} \right) + pos(\pi_s, \pi_{s+1}) * x_{NIC}{}^{OUT}_{i+z,\pi_s}$$

$$+ \sum_{k=s+1}^{m-1} int^{PARA}_{CENT}(i+z, \pi_k) + dest^{PARA}_{CENT}(i+z, \pi_m) - t_{i+z}$$

4.6 Single-Core vs. Multi-Core

We can now compare the response time and response times of single-core and multi-core scheduling.

4.6.1 Quiescent State Commands

The difference between the local response times of the single-core process-first policy and the parallel multi-core scheduling policy is given by

$$local_{REP}^{PF}(i,j) - local_{REP}^{PARA}(i,j) = \left(p_{i,j}^{IN} + p_{i,j}^{OUT}\right) - \left(p_{i,j}^{IN} + p_{i,j}^{OUT}\right) = 0$$

Therefore, the local response times are the same with the two policies. Since the process-first local response times are as good or better than those of any other single-core scheduling policy, multi-core scheduling response times are as good as or better than those of single-core scheduling.

Moreover, the difference between delays on the intermediate computers with the transmit-first and parallel policies is given by

$$int_{REP}^{TF}(i,\pi_k) - int_{REP}^{PARA}(i,\pi_k) = \left(x_{CPU}{}_{i,\pi_k}^{IN} + pos(\pi_k,\pi_{k+1}) * x_{NIC}{}_{i,\pi_k}^{IN}\right) - \left(x_{CPU}{}_{i,\pi_k}^{IN} + pos(\pi_k,\pi_{k+1}) * x_{NIC}{}_{i,\pi_k}^{IN}\right) = 0$$

Therefore, the intermediate delays with these two policies are the same. Since the transmit-first intermediate delays are as good as or better than those of any other single-core policy, multi-core scheduling intermediate delays are as good as or better than those of single-core scheduling.

Similarly, the difference between delays on the destination computers with the process-first and parallel policies is given by

$$dest_{REP}^{PF}(i,\pi_m) - dest_{REP}^{PARA}(i,\pi_m) = \left(p_{i,j}^{IN} + p_{i,j}^{OUT}\right) - \left(p_{i,j}^{IN} + p_{i,j}^{OUT}\right) = 0$$

Therefore, the destination delays with these two policies are the same. Since the process-first destination delays are as good as or better than those of any other single-core policy, multi-core scheduling destination delays are as good as or better than those of single-core scheduling.

Based on these three difference results, we conclude that if a computer has multiple cores available for performing the processing and transmission tasks, then the parallel multi-

core scheduling policy should always be used. In particular, the local response times are as good as the best local response times provided by any of the single core scheduling policies. Moreover, the remote response times are as good as or better than the remote response times provided by any of the single-core scheduling policies. The reason they can be better is that the delays on intermediate computers consists of the transmission time only and the delays on the destination computer consist of the processing time only. The delays of all single-core scheduling policies have additional times contributing to either the intermediate or destination delays or even both of them.

4.6.2 Consecutive and Simultaneous Commands

We compare the single-core and multi-core response times of consecutive commands by analyzing the difference in the respective equations. We compare the differences in the increase in response times of such commands. If we compared the absolute response times, like we did for the response times of consecutive commands for the four single-core policies, we would get difference equations with intermediate and destination delays of the multi-core and single-core policies. We already know based on the above discussion that the parallel policy delays are as good as or better than the best single-core policy delays. Therefore, the comparison of absolute response times would not provide us with any new insights. But as we will see, a comparison of the increases in the response times of consecutive commands does give us new insights, so that is the comparison we make next. Moreover, we present the simplified difference equations, in which only the terms that matter are shown.

Local Response Times

Let us start with the comparison of the parallel and any single-core policy A local response times of commands. The difference is given by

$$local^{PARA}_{REP,i+z,j} - local^{A}_{REP,i+z,j} = 0$$

$$local^{PARA}_{CENT,i+z,j} - local^{A}_{CENT,i+z,j} = 0$$

Thus, the local response time increases are the same for the parallel and any single-core policy. Next, we compare the increases in the remote response times of the parallel policy and the single-core policies.

Remote Response Times

Consider first the case when the critical computer is the destination. The response time difference in the increases in remote response times of consecutive commands of the parallel policy and any single-core policy A is given by

$$remote^{PARA}_{REP,i+z,j} - remote^{TF}_{REP,i+z,j} = -\sum_{c=i}^{i+z-1} \left(fan_{\pi_m} * x_{CPU}{}^{IN}_{c,\pi_m} \right)$$

$$remote^{PARA}_{CENT,i+z,j} - remote^{TF}_{CENT,i+z,j} = -\sum_{c=i}^{i+z-1} \left(fan_{\pi_m} * x_{CPU}{}^{OUT}_{c,\pi_m} \right)$$

Thus, if the critical computer is the destination, the parallel policy gives lower increases remote response times of consecutive commands than any single-core policy. Moreover, the benefit of the parallel policy over the transmit-first policy is additive: as the number of consecutive commands increases, the difference between the response times of the two policies also increases.

When the critical computer is not the destination, there is only one case for the parallel policy and three cases for the single-core policy, one of which matches the parallel case. When the network transmission times dominate the processing times, the difference in the increase in remote response times of the parallel policy and any single-core policy A is given by

$$remote^{PARA}_{REP,i+z,j} - remote^{A}_{REP,i+z,j} = 0$$

$$remote^{PARA}_{CENT,i+z,j} - remote^{A}_{CENT,i+z,j} = 0$$

Hence, in this case, the parallel and single-core policies behave the same. The reason is that the single-core policies are not hurt by processing times because the CPU can both process and transmit all commands before the network card can transmit all the commands. When the when the critical computer is neither the source nor the destination and processing times dominate the total network card transmission times, the response time difference is given by

$$remote^{PARA}_{REP,i+z,j} - remote^{A}_{REP,i+z,j} = -\sum_{c=i}^{i+z-1}\left(p^{IN}_{c,\pi_s} + p^{OUT}_{c,\pi_s}\right)$$

$$remote^{PARA}_{CENT,i+z,j} - remote^{A}_{CENT,i+z,j} = -p^{IN}_{i,\pi_1} - \sum_{c=i}^{i+z-1}\left(p^{IN}_{c,\pi_s} + p^{OUT}_{c,\pi_s}\right)$$

As the difference equations show, the increases in remote response times given by the parallel policy are lower than those of any single-core policy. And once again, the benefits of the parallel policy are additive. Finally, when the when the critical computer is the source and processing times dominate the total network card transmission times, the response time difference is given by

$$remote^{PARA}_{REP,i+z,j} - remote^{A}_{REP,i+z,j} = -\sum_{c=i}^{i+z-1}\left(p^{IN}_{c,\pi_s} + p^{OUT}_{c,\pi_s}\right)$$

$$remote^{PARA}_{CENT,i+z,j} - remote^{A}_{CENT,i+z,j} = -\sum_{c=i}^{i+z-1}\left(p^{IN}_{c,\pi_s} + p^{OUT}_{c,\pi_s}\right)$$

As above, the increases in remote response times given by the parallel policy are lower than those of any single-core policy. In addition, the benefits of the parallel policy are additive.

As the difference equation show, the parallel policy should always be used on a multi-core computer. There are two reasons why the increases in the local and remote response times of the parallel policy are as good as or better than those of the single-core

policies.. First, when the critical computer is an intermediate computer, then with single-core policies, this time is a function of both the processing and network card transmission times, while with the multi-core policy, on the other hand, it is a function of only the network card transmission times. Second, when the critical computer is the destination computer, then with single-core policies, this time is a function of both the CPU processing and CPU transmission times, while with the multi-core policy, it is a function of only the processing time.

4.7 Self-Optimizing System Implementation

In the previous two chapters, we have developed a system that can select at start time or dynamically switch at runtime to the processing and communication architectures that best meet user-provided response time requirements. We have extended that system to do the same for the scheduling policy. The extended system applies the analytical model presented in this chapter to decide which scheduling policy to use. The model itself is an extension of the model presented in the previous chapter that assumed a fixed scheduling policy. The new model extends the old one by considering scheduling policies and their impact on response times.

4.7.1 Gathering Parameter Values

To make decisions regarding the scheduling policy, the self-optimizing system applies the new analytical model presented in this chapter. In order to apply the model, the system must collect values of all of the response time parameters identified by the model. It may seem that the system presented in the previous chapter already gathers all of the necessary parameters because the new model does not introduce any new parameters. Indeed,

the new system does not need to gather any additional parameters; however, it has to gather the previous parameters in a slightly different fashion.

CPU Transmission and Processing Costs

The method used in the earlier versions of the system to collect CPU processing and transmission costs does not work when the concurrent or lazy policies are considered. The client-side components in those systems measured the CPU processing (CPU transmission) time by measuring the amount of time that elapses from the moment processing (transmission) starts to the moment it completes. This method assumes that once a task begins, it runs without interruption until it completes. Since the previous chapters assumed transmit-first scheduling, the assumption was true. It is also true when process-first or parallel scheduling is used. However, it is not true when concurrent or lazy scheduling is used because the tasks do not run sequentially.

When concurrent scheduling is used, the operating system interleaves the processing and transmission task threads. One way of measuring the durations of each task is to record the total amount of CPU time for each task. It is difficult to do so accurately since the operating system may schedule and un-schedule the threads at any time without notice. An alternative approach, which is the one we use, is to measure the task durations using the same method as before, then apply the concurrent scheduling analytical model to estimate what the actual durations were, and finally report the estimated durations to the server-side component. The client-side component can use the model analytical models follows. The model states that when concurrent scheduling is used, (a) the shorter of the two tasks executes in twice the time it requires to execute when it runs sequentially and (b) the longer task executes in time it requires to execute both tasks sequentially. Therefore, the client-side

component can estimate that the amount of CPU time required for the task whose measured duration was shorter is actually only half of its measured duration. Similarly, the amount of CPU time needed for the task whose measured duration was longer is equal to its measured duration minus the estimated CPU time of the shorter task.

When the lazy policy is used, the processing task runs sequentially but it may divide the transmission task into two parts, each of which runs sequentially. Therefore, the client-side component measures how long the processing task and each part of the transmission task requires to complete and reports those values. Since the server-side component needs to know how many destinations the CPU transmits to in a given amount of time to estimate the transmission cost per destination, the client-side also reports the number of destinations to which the CPU transmitted during each part of the transmission task.

Network Card Transmission Costs

The method used to collect the network card transmission costs in previous versions of the self-optimizing system also needs to be modified in order to work with the lazy policy. In these earlier systems, the client-side component on the source computer reported the time at which the transmission of a command began and the client-side component on the final destination computer reported when the command is received. The difference in these two times divided by the number of destinations to which the source computer transmits gives the network card transmission time per destination. This method assumes that once the network card begins to transmit a command, it does so continuously without idle periods until it transmits to the last destination. This assumption holds for transmit-first and process-first policies because once the CPU begins the transmission task, it completes it without interruption. In fact, it also holds for the concurrent policy even though the CPU transmission

task is interleaved with the processing task. The reason it holds is that the CPU transmission costs are less than one half of the network card transmission costs as mentioned above. Therefore, the network card does not catch up to the CPU during the quanta the CPU is performing the processing task. Unfortunately, the assumption does not hold for the lazy policy. In particular, if a computer performs a part of the transmission task before processing, the network card may catch-up to the CPU while the CPU performs the processing task. In this case, the time reported by the final destination will include the network card idle time and hence, the network card costs will be overestimated. One way of solving the problem is the use the analytical model to estimate the idle time on the network card and adjust the network card cost calculations for it. A better, more accurate approach, which is the one we use, is to apply the technique from the previous systems, but only for the first part of the transmission task. The source computer still reports when the transmission begins. However, instead of the final destination reporting the reception time, the destination to which the CPU transmits last during the first part of the transmission task reports it. Moreover, the source computer reports the number of destinations to which it transmits in the first part of the transmission task. Then, as before, the server-side component estimates the network card transmission time by calculating the difference between the transmission start and reception times and then dividing the difference by the reported number of destinations.

4.7.2 Applying the Analytical Model

Using the collected values of the analytical model parameters, our system applies the model as follows. First, for each computer type, it calculates the average processing and transmission times. Second, it uses the calculated values and the network latencies in the model equations to calculate the estimated response times of commands by each inputting

user for each scheduling policy. Third, it invokes the user-defined total order function using the estimated response times for each system and the user identities as the parameters of the function. Fourth, it switches to the scheduling policy that is ranked as the best by the total order function.

To illustrate the procedure, consider a three user scenario in which each user inputs commands. After estimating the values of the model parameters, our system crates the 4 x 3 x 3 response time matrix. The [x, y, z] entries in the matrix gives $user_y$'s response for a command entered by $user_z$ when a single-core scheduling policy x is used. The reason that the first dimension is not of size five to accommodate the parallel multi-core policy is that the parallel multi-core policy should always be used on a multi-core computer, regardless of what scheduling policy is used on single-core computers. Therefore, when calculating the response time matrix, our system assumes that the parallel multi-core scheduling policy is used on multi-core computers, regardless of the single-core policy used on other computers. Since there are three users in the session and each one inputs commands, the second and third dimensions are of size three. It then invokes the total order function, passing the response time matrix, the list of inputting user indices sorted from lowest to highest, and a list of the identities of the users as parameters. When the total order function returns, it simply returns an index of the single-core policy that best satisfied the users' response time requirements. When our system changes the scheduling policy, it makes all of the single-core computers use the policy returned by the total order function and all the multi-core computers use the parallel policy.

As in the previous chapters, the values of some parameters may not be reported for a computer type. In those cases, the self-optimizing system estimated the missing parameter

values using the parameter values that have been reported. Since the model presented in this chapter did not introduce any new parameters, there are no new estimates that are required in addition to those already implemented in the previous versions of the self-optimizing system.

4.7.3 Scheduling Policy Switch Mechanism

Once the total order function returns the scheduling policy that should be used, our system must deploy it. In general, it takes time to switch to a different scheduling policy on all of the computers. The same was true when switching processing architectures. Recall that because of this, the processing architecture switch had to be done atomically with respect to user commands; otherwise, some commands may be shared in a manner that is inconsistent with the notion of centralized and replicated architectures. However, this is not an issue when switching scheduling policies because the scheduling policies do not define the semantics of the architecture. In particular, the scheduling policy does not determine the destinations to which a computer forwards inputs and outputs. Therefore, if during the switch, some computers use the old and some the new scheduling policy, no inconsistency issues arise. Thus, we do not pause the execution of tasks for new user commands during the scheduling policy switch. Instead, the server-side component simply instructs each user's computer to switch to the new scheduling policy.

While there are no inconsistency issues when different computers use different scheduling policies during a switch, some performance issues may arise. In particular, the response time calculations were predicted for the old and the new scheduling policy. They were not predicted for the case when the computers use a mix of these policies. Thus, it may happen that during the switch, the users experience worse response times than those they experienced before the switch. Fortunately, the response time degradation is temporary. As

soon as all of the computers switch to the new policy, the new response times will be better than those before the switch.

4.7.4 Lazy Policy Implementation

So far, we have shown how our self-optimizing system can decide when to use the lazy policy and also continue to receive performance information when the policy is used. One issue left unaddressed is how exactly a user's computer estimates the amount of time by which it can delay the processing task when the lazy policy is being used. As described above, the delay is equal to the noticeable threshold, minus the total amount of time that has elapsed from the moment the source computer begins transmitting a command to the moment the user's computer receives it, plus the network latencies on the path from the source to the user's computer. The reason network latencies are added back is because they are included in the second term. For reasons given earlier, we do not determine the processing task delay based on network latencies.

Our approach for calculating the possible delay on each user's computer is as follows. When sending a command, the source computer includes the time at which the command was entered. Then, a receiving computer 1) subtracts that time from the time at which it begins to perform the tasks for the command, 2) adds the network latencies the command experienced as it travelled from the source, and 3) finally subtracts the value from the response time threshold to determine how long it can delay the processing task. For this approach to work, the clocks on the source and the user's computers must be synchronized, and the user's computer must know the latencies the command has experienced as it travelled from the source. As explained in the previous chapter, each user's computer synchronizes its clock with the computer on which the server-side component is running. The algorithm used for

this clock synchronization can be applied between each pair of users' computers. A less resource and time intensive approach, which is the one we use, is for the server-side component to calculate the clock synchronization between each pair of computers and send that information to them. The server-side component calculates the clock difference between two computers by adding up the clock differences between each computer and the computer on which it is running on. The component is also aware of the path from the source to the user's computer because it controls the communication architecture. Moreover, it is aware of the network latencies on each link along the path because it asks each computer on the path to measure the network latency to the upstream and downstream computers. Therefore, the component calculates the total network latency that a command experiences from the source to the user's computer and sends it to the computer. As network latencies change, it periodically re-measures them as explained in the previous two chapters. Thus, whenever the measured network latencies change, it sends to each user's computer the updated network latencies a message experiences from a source to the user's computer.

4.8 Evaluation

We have presented an analytical model of response times for large-scale collaborations. Using the model, we have shown theoretical results about the benefit of each scheduling policy in collaborative systems. It helped us formally confirm intuitive expectations and, more interesting, derive some unintuitive results about the lazy policy and multi-core scheduling. In particular, the analysis derived the unintuitive multi-core result that even when more than two cores are available for scheduling these tasks, using only two – one for the transmission and one for the processing task – is sufficient to get the maximum

response time benefit. In addition, we have created a self-optimizing system that uses the model to automatically select the scheduling policy that best meets response-time requirements. While these results are a contribution on their own, it is important to see whether or not (a) the theoretical differences given by the model can be significant in practical scenarios and (b) the self-optimizing system can better meet response time requirements than existing systems in these scenarios. Since the self-optimizing system applies the analytical model, we can accomplish both evaluation goals by checking whether or not the system significantly improves response times in practical scenarios.

As in the previous chapter, we verify only the part of the model that predicts absolute response times for commands entered in a quiescent state. Similarly, we verify only the part of our system that automatically improves the response times of such commands. The reason we do not evaluate the other parts of the model or the system – specifically the parts regarding the response time predictions and improvements of consecutive and simultaneous commands – is because in our logs, commands were always entered in a quiescent state. As described in the prior chapter, even telepointer actions are entered in a quiescent state in our logs. Of course, in some collaborations, users may actually enter consecutive and simultaneous commands. Therefore, an important future work direction for us is to verify our model and system for such application.

Ideally, we would perform experiments that show the significance of the predictions. However, as described in the previous chapter, our resources limit experiments we can perform. As a result, we use the same approach as in the previous chapter. We first show simulation results for response times given by each scheduling policy in a practical scenario.

We then validate the simulations through "fixed" experiments which we can actually perform using our limited set of resources.

4.8.1 Single-Core Simulations

Our theoretical results predict that in theory, the choice of scheduling policy can improve the response times. To check if these improvements can be significant in practical circumstances, we consider a scenario in which a PowerPoint presentation is being given to 600 audience members around the world.

The self optimizing system decides which scheduling policy to use for this scenario in five steps. First, it gathers the parameters of the analytical model presented in this chapter. These parameters include (a) the PowerPoint processing and transmission costs contained in performance reports sent by the users' computers and (b) the network latency measurements performed by each computer. Second, the system applies the model. In particular, it plugs the measured parameter values into the model to calculate the response times to commands by each inputting user with each scheduling policy. Third, it passes these response times to the total order function which then informs it which scheduling policy to use. Fifth, it deploys the scheduling policy.

As in the previous chapter, it is not possible to simulate the first step because in a simulation there are no computers that send these reports. Thus, the simulation must be provided these values, along with the response time thresholds. As before, based on the published network latency data between 1740 computers [67], we set the network latencies between all users equal to those between a random subset of 600 of the 1740 computers. One issue with randomly selecting the subset is whether the subset preserves properties, such as triangle inequality and latency distributions, of the entire set. Zhang et al. [96] analyzed

random subsets taken from latencies measured between 3997 computers and found that they were representative of the overall measurements. Moreover, we have measured realistic PowerPoint processing and transmission costs for the netbook, P3 desktop, P4 desktop, Core2 desktop as described in Appendix A. In our simulated scenario, the lecturer is using a netbook and all of the other users are using either netbooks, a P4 desktop, or a P3 desktop. Also, as mentioned before, users can notice 50ms degradations in local [80] and remote [54] response times. Thus, we set both response time degradation thresholds to 50ms.

Once the values are provided, the next two steps are executed in the same manner regardless of how the analytical model parameter values provided. Therefore, these steps are the same whether the system is being simulated or used in an experiment. The fourth and final step, which deploys the optimal scheduling policy, like the first step, needs actual computers. The reason is that the policy is deployed by sending messages to the users' computers informing them how to establish communication channels with each other. For now, however, we do not need the fourth step; our goal is to show that in theory, choosing the scheduling policy can better meet response time requirements, which can be done in the first three steps.

As mentioned in the introduction of the chapter, we are optimizing the scheduling policy for a given processing and communication architecture. Therefore, in the simulations, we consider the centralized architecture in which the lecturer's computer is the master. Also, we consider the communication architecture in which the presenter's computer and five more of the 600 computers forward commands to all of the other computers. The source multicasts commands to these six computers, and each of these six computers unicasts commands to 99

of the remaining 594 computers. We fixed the latencies among the six forwarding computers to be low (i.e., 0ms).

The architecture we have outlined is a Webex-like architecture. In Webex, a user joining a session connects to one of several infrastructure computers. These computers, which are connected by high-bandwidth low-latency connections, serve as forwarders of commands. In our scenario, the six users, including the presenter, which forward commands to all of the other computers are like the dedicated Webex forwarders. However, unlike the Webex forwarders which only forward commands, the computers' belonging to these six users also process commands.

Results

In this scenario, the differences between the remote response times are shown in Figure 4-9. As Figure 4-9 shows, the lazy remote response times are either significantly better than (604ms) or equal to the process-first transmission times, as predicted by the theoretical analysis. More importantly, the lazy remote response times can be as much as 158ms better for some users than both the transmit-first and concurrent remote response times. On the other hand, the lazy remote response times are worse by as much as 240ms than those of the transmit-first and concurrent policies. These results show that for some users, the lazy policy provides significantly better remote response times than other policies.

Overall, the lazy policy remote response times are better than those of process-first for 598 of the 599 audience members. However, they are only noticeably better for 5 audience members than those of the transmit-first and concurrent policies. For 407 other users, they are the same as those of transmit-first and concurrent policies. And for the remaining 187, they are significantly worse. These results raise two issues: 1) is it worth it to

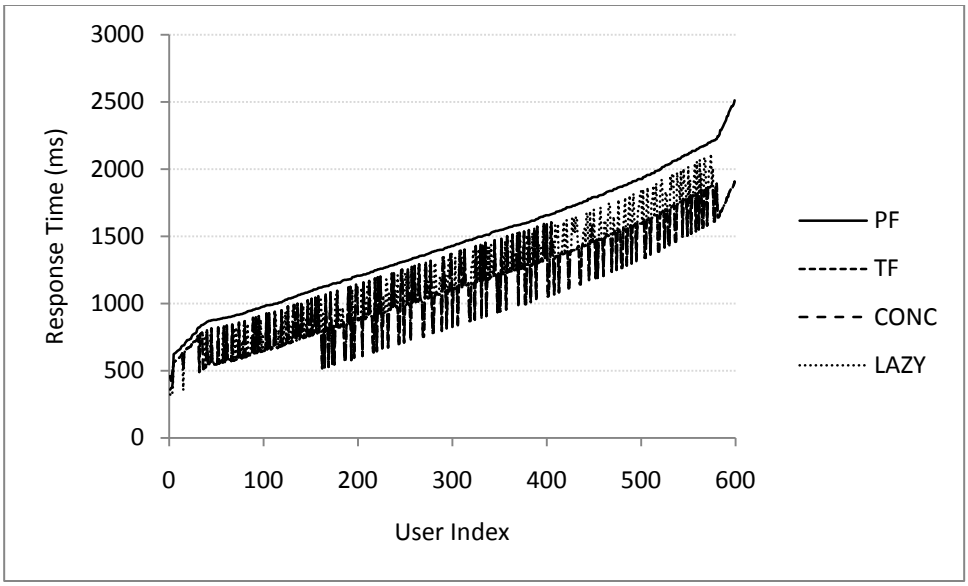

Figure 4-9. Remote response times of all audience members.

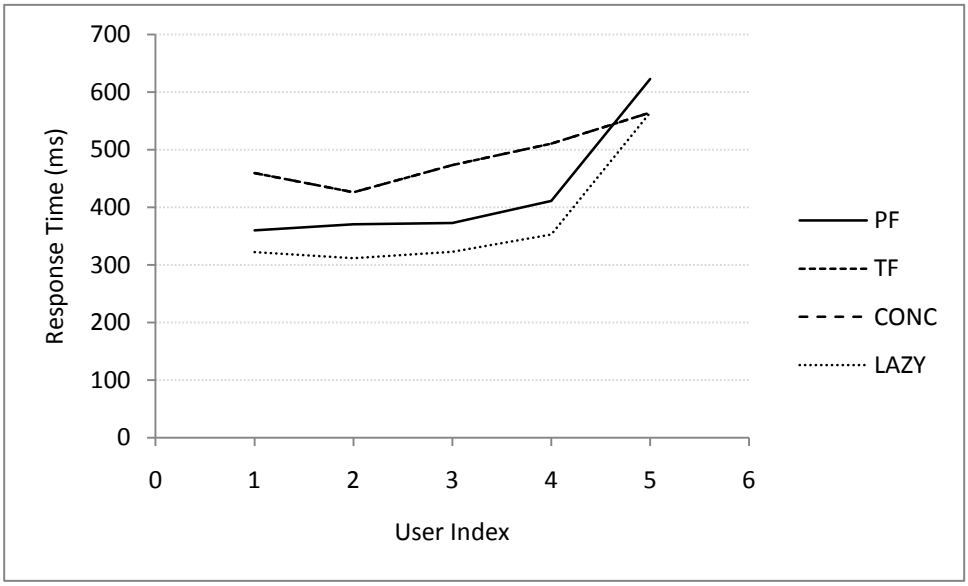

Figure 4-10. Remote response times of forwarding audience members.

sacrifice the performance of 187 users for improving performance of only 5 other users and 2) why are the performances under the three policies the same for more than two thirds of the users? The answer to the first question is revealed by a closer look at the remote response times of the users whose computers forward the presenter's commands. The lazy policy

provides significantly better remote response times than the transmit-first or concurrent policies to 4 of the 5 users whose computers are forwarding the presenter's commands (158,151,137, and 114ms) as shown in Figure 4-10. Therefore, compared to the transmit-first and concurrent policies, the lazy policy is more fair to users who help improve the performance of other users. The answer to the second question is given by the fact that non-blocking communication is being simulated. In our experience, when non-blocking communication is used, a computer's CPU can transmit commands much faster than the network card can. Thus, with the lazy policy, during the time the CPU is processing a command, the network may not finish transmitting the command to the destinations to which the CPU queue the command for transmission before the CPU began processing the command. For the same reason, when the concurrent policy is used, the network card may not finish transmitting commands queued in one quantum by the CPU before the CPU begins queuing messages again in the next transmission quantum. When this is the case, the computers downstream receive messages at the same time with all three of these policies, as shown by the simulation results.

As a final note regarding remote response times, Figure 4-10 shows that for 3 of these 5 forwarding users, the process-first remote response times are noticeably better than those of the transmit-first and concurrent policies (101,99, and 56ms) but noticeably worse than those of the lazy policy (58, 58, and 50ms). These results also agree with the somewhat counterintuitive prediction made by the equations that process-first remote response times can be noticeably better than those of the transmit-first and concurrent policies.

The lazy policy local response time, 107ms, is 49ms worse than the process-first local response times, 58ms, which is within the local response time degradation threshold. The

transmit-first local response times are 177ms, which is significantly worse than those of the lazy policy. Finally, the concurrent policy local response times, 118ms, are not noticeably worse than the lazy response times. On the other hand, they are noticeably worse than the process-first response times, 58ms, since the difference is above the local response time degradation threshold.

The simulation shows that our purely theoretical results that scheduling policies impact response times apply to practical scenarios. Moreover, they show that depending on the total order function, our system can choose the scheduling policy that improves response times. For instance, if a total order function that chooses the system that provides the best absolute local response times is specified, it would return the process-first scheduling policy. If on the other hand, the function chooses the system that provides the best remote response times, it would return the transmit-first or the concurrent scheduling policy because the performance of these policies was identical in this particular scenario. Finally, if the function chooses a system that provides the best remote response times without increasing the local response times more than what is noticeable, it would return the lazy policy. Of course, the function would need to be aware of what the response time thresholds are. Since these are anyway provided by the users as a parameter to our system, the users can incorporate these thresholds in the function when they create it. Whatever policy the function returns, our system switches to that policy.

4.8.2 Multi-Core Simulations

Our theoretical analysis also predicts that when multiple cores are available for scheduling, the parallel scheduling policy can provide significantly better response times than the single-core policies. To evaluate whether these differences can be noticeable in practical

scenarios, we performed simulations which considered the scenario used in the single-core simulations with the following differences: 1) there are 1200 instead of 600 users including the presenter, 2) each of the six forwarding computers forwards to 199 of the remaining 1194 computers, 3) all of the users use the Core2 desktop, 4) instead of the single-core concurrent scheduling policy, the computers use the multi-core parallel policy.

Results

For this scenario, the remote response times are shown in Table 4-1. We give tables instead of graphs for these results as the graphs have too many points and as a result are difficult to read. As Table 4-1 shows, the parallel policy provides noticeably better remote response times than the process-first policy to 999 of the 1199 audience members, while the remote response time differences to the remaining 200 users are not noticeable. Moreover, compared to the remote response times of the transmit-first policy, the parallel policy provides noticeably better response times to only 5 users and provides unnoticeably different response times to the remaining 1194 users. The five users whose response times are improved are those who forward the presenter's commands to other computers. This makes sense as the processing on these computers is delayed until transmission completes when the

Table 4-1. Parallel vs. Single-core Policy Remote Response Time Statistics

	PARA - TF	PARA – PF	PARA – LAZY
MAX	-1.792	-9.876	0
MIN	-69.328	-311.224	-28
0	0	0	1198
>= 50	0	0	0
<= 50	5	999	0
REST	1194	200	1

Table 4-2. Parallel vs. Single-core Local Response Times

TF	PF	LAZY	PARA
78.868	9.876	59.716	9.876

transmit-first policy is used, while the processing is not delayed when the parallel policy is used. Finally, the parallel policy remote response times are the same as those of the lazy policy for 1198 users. The reason is because, as explained above, when non-blocking communication is used, the network card does not catch up to the CPU in terms of transmission during the time the CPU performs the processing task. The remote response times for the remaining user are not noticeably different for the two policies.

The local response times for the simulation are given in Table 4-2. As Table 4-2 shows, the parallel policy gives the same local response times as the process-first policy. This makes sense as in both cases, some core immediately begins the processing task. Moreover, the transmit-first policy gives significantly worse local response times than these two policies. Finally, the lazy policy gives worse, but not noticeably worse local response times than the parallel and process-first policies.

Therefore, the combination of the local and remote response time results suggests that in practical scenarios involving multi-core computers, it can be actually better to use the lazy or even the transmit-first policy on a single core than to use a parallel policy on multiple cores. The reason is that the performance is comparable with all three policies, but using just a single core for the collaborative application implies that the computer can use the remaining cores to perform other tasks and improve their performance, such as OS tasks or tasks of other applications the user is using at the same time while collaborating.

4.8.3 Experiments

While the simulation results are a contribution on their own, it is also important to validate if they reflect what happens in reality. As explained in the previous chapter, simulations can be validated through experiments. However, we used simulations because we

did not have sufficient resources to run all experiments. Nevertheless, we can run some limited set of experiments with our resources to validate the simulations.

Single-Core Results

To validate our simulations for the single-core scheduling policies, we used a scenario that is one "part" of the scenario used in the single-core simulations. As when we validated the multicast simulations in the previous chapter, we focused on a single path in the communication overlay used in the simulations and measured the response times of the computers on that path. The path we chose to evaluate was the path from the source computer to one of the five other computers that forward messages. Moreover, as we wanted to evaluate how the lazy policy behaves in an actual experiment as a command traverses a long path, we chose to evaluate a path in which each of the remaining four computers that forwards commands is an intermediate node. To remove any issues with the order in which the network card actually transmits commands, the computers only forwarded commands to the next computer down the path. Therefore, there were a total of six users in our experiment.

As explained in the previous chapter, to ensure the timing data we measure for these users is accurate, each user's computer on the path must run on a dedicated physical machine. The presenter was running on the netbook, and each of the other five users was running on a Core2 desktop on which a single core was used to perform both the processing and transmission tasks. We then provided our self-optimizing system the following performance parameters at start time. To ensure that our system creates a path from the source to one of the five other users on which the remaining four users were intermediate nodes, we provided our system with the following simulated network latencies: the latency between $user_i$ and $user_j$, where $1 \leq i \leq 5$ and $2 \leq i \leq 6$, is that of a LAN plus 0ms, while

the remaining latencies were all the LAN latencies plus 100000ms. The values of other parameters are several orders of magnitude less than these high latencies and therefore the latencies would drive the creation of the communication overlay. The remaining parameters provided to our system were 1) the input and output processing costs of master and slave computers and 2) the CPU and network card transmission times of output commands. We used the same values as those in the above single-core simulations. The parameters were provided for both the netbook and the Core2 desktop type of computer.

To compare the simulation and experimental results for this scenario, we ran our system four times, each time configuring it to use a different single-core scheduling policy. In addition, we performed a simulation of response times for each of these policies. The remote response times measured during the experiment and those predicted by the simulations are shown in Figures 4-11 and 4-12, respectively. As the experimental results in Figure 4-11 show, the remote response time of the computers significantly degrades with each hop when the process-first policy is used. The reason is that on average, the Core2 computers require 75ms to process the output, and hence each computer on the path contributes at least 75ms of time to the remote response time of the downstream computers. Moreover, the remaining three policies give the same remote response times for all but one computer. In particular, with the lazy policy, the destination computer's remote response time is significantly worse than with the transmit-first and concurrent policies. The reason is that by the time the second last computer on the path receives the command, the sum of the transmission times on the upstream computers grows above 50ms, which is the remote response time degradation threshold. Thus, the second last computer on the path does not delay processing before transmitting even to a single destination. As a result, the destination

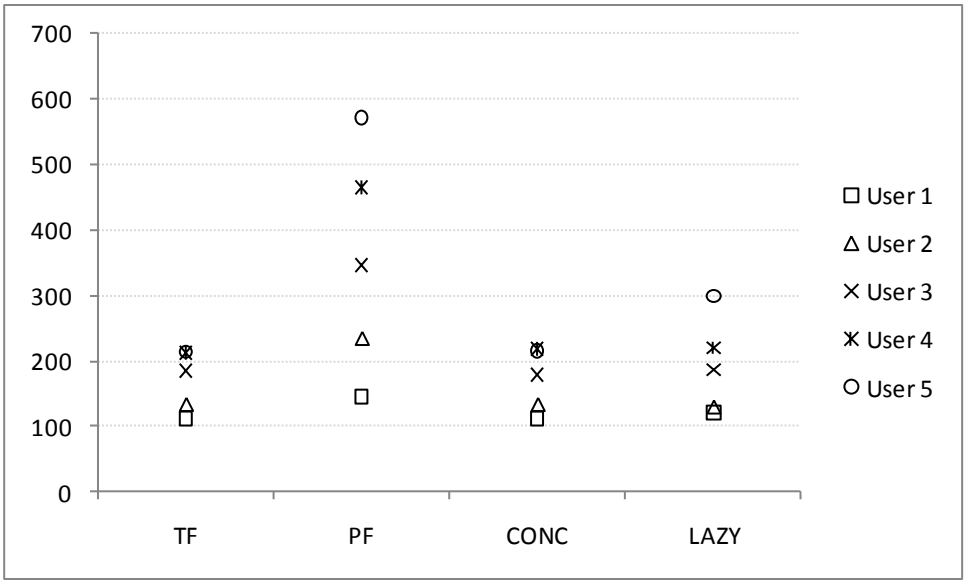

Figure 4-11. Measured remote response times.

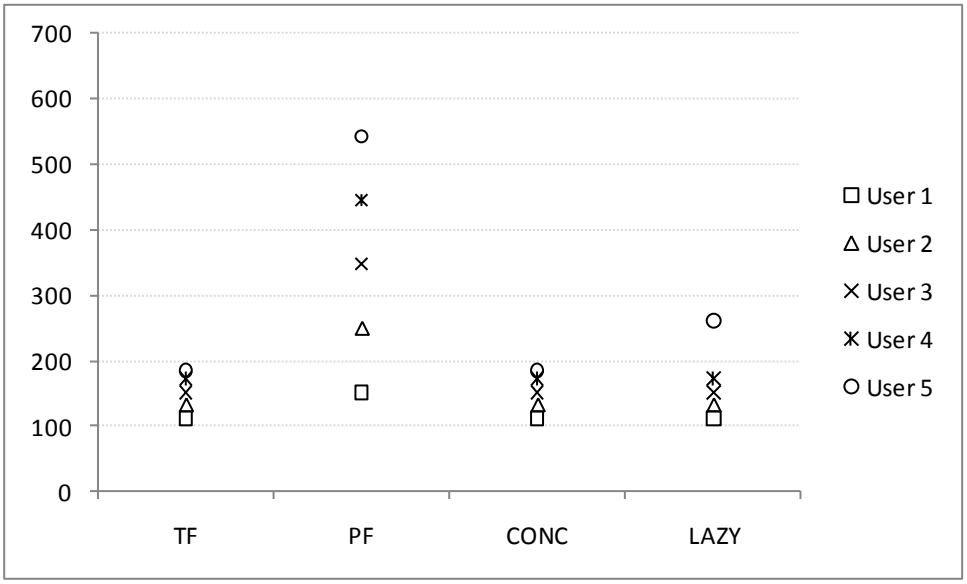

Figure 4-12. Simulated remote response times.

computer's remote response time includes the processing time on the second-last computer. Also, note that some of the Core2 computers require slightly more than others to process the output command. This is the reason why with the transmit-first, concurrent, and lazy policies, the jump in response times from the second to the second-last computer on the path is not even.

The remote response times predicted by the simulations are given in Figure 4-12. While they are not identical to the experimental values, they exhibit all of the same properties. In particular, the simulated process-first remote response times grow significantly with each hop, while the lazy policy remote response times grow significantly only on the last hop. Note that the uneven increase in remote response times under the non process-first policies is not displayed by the simulations. The reason is that the simulations are given the average parameter values of the Core2 desktops which do hide the small differences in performance across these machines. The absolute difference between the predicted and measured values is never noticeable. The highest value we observed was 48ms, which was for the second last user under the lazy scheduling policy. One reason for the differences is that, as mentioned above, the simulations are running based on average values for the type of computer rather than using individually measured values for each computer. Another reason may be that either our system has overheads or there are some parameters which are not being considered in the simulations. Regardless, however, of what we have missed, our simulations are accurate enough to predict trends in using each policy. Moreover, while the simulations are always predicting response times to be lower than they actually are, they are doing so consistently across all policies. Moreover, in this set of simulation and experiments, they are never noticeably incorrect.

The local response times predicated by the experiments and simulations are shown in Figures 4-13 and 4-14, respectively. As these figures show, our simulations predicted almost identical response times as those that were measured in all cases except when the concurrent policy is used. Since with the concurrent policy, we have the least control over

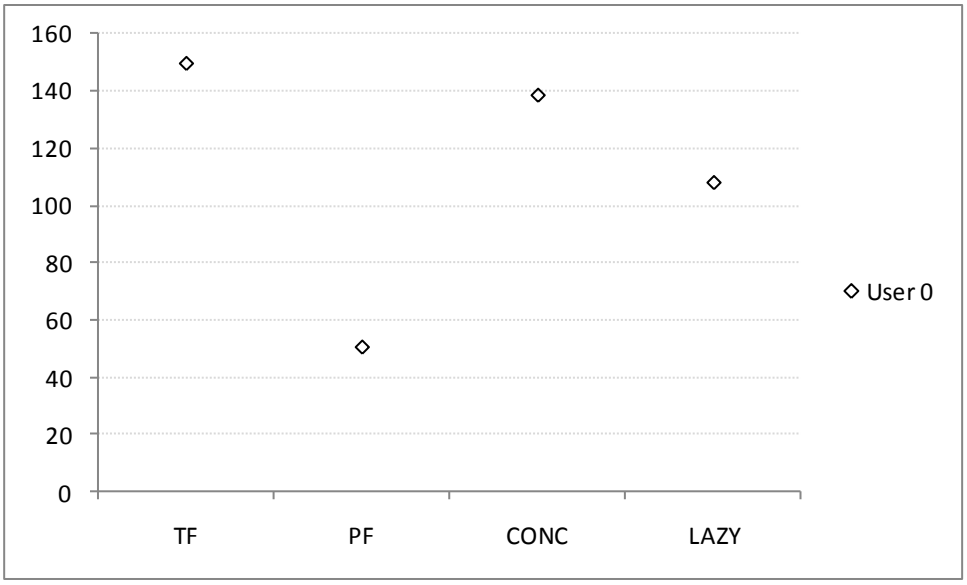

Figure 4-13. Measured local response times.

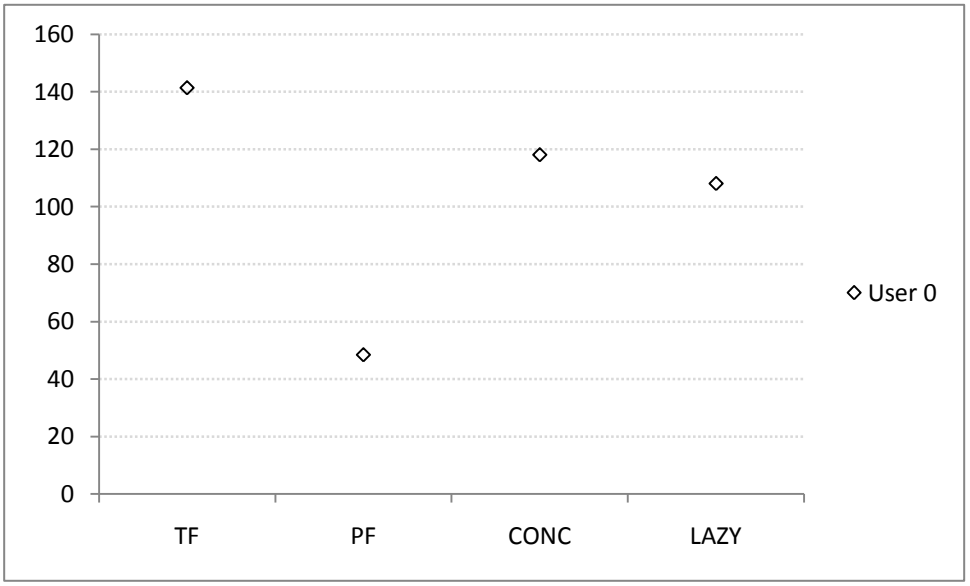

Figure 4-14. Simulated local response times.

how the operating system actually schedules threads, the difference in simulation and measured values is somewhat expected. Note, however, that the difference is not significant.

Therefore, as the simulations and experimental results match, our system is able to correctly predict the response times given by single-core policies. Thus, when it provides

these response times to the total order function, the scheduling policy returned by the function will best satisfy the response time requirements.

Multi-Core Results

To validate our simulations for the multi-core parallel scheduling policy, we used a Checkers scenario in which a single player is playing against the computer and a large number of users are observing. Ideally, we would like to evaluate both local and remote response times of the parallel policy. Unfortunately, in our multi-core experiments, we are not able to measure remote response times properly. The reason is that, as mentioned in the previous chapter, the network card does not transmit commands in the order in which the CPU transmits them. We cannot add delays during the transmission, which is the approach we used before to ensure some order in the transmission. Adding a one second break after the first transmission means that the processing and transmission tasks will not actually run in parallel because the processing task does not take one second. Therefore, we focus only on the local response times. Overall, there were 601 users in our experiment: the user who is playing against the computer and the 600 users who are observing. We had to use a large number of users because the Checkers transmission costs to a single destination are low – transmitting to 600 destinations requires only about 12ms of CPU time on the Core2 machines.

As explained in the previous chapter, to ensure the timing data we measure is accurate, the presenter's computer must run on dedicated physical machine. The remaining 600 users were mapped to only two physical machines. All of the machines we used were the Core2 desktops. We then provided our self-optimizing system the following performance parameters at start time. The replicated-unicast architecture should be used. Since we are

measuring only the local response times, the network latencies do not matter. Hence, we assumed that all of the users are on the same LAN. Moreover, since we are using a unicast architecture, we do not need to force our self-optimizing system to deploy a particular multicast architecture. Therefore, unlike in the above experiment, we do not have to override some of the network latency values.

To compare the simulation and experimental results for this scenario, we ran our system with four different scheduling policies: the transmit-first, process-first, lazy, and parallel. When we used the single-core policies, a single core was used for both of the tasks. For each policy, we ran ten experiments. We report the average measured response times. In addition to the experimental results, we performed a simulation of response times for each of these three policies. The remote response times for the forwarding user predicted by the simulations and measured during the experiments are shown in Table 4-3. As Table 4-3 shows, the simulation and experimental results match for all four policies. More interestingly, the experimental results show, and the simulation agrees, that the parallel policy can provide as good local response times as any of the single-core policies. These results agree with our theoretical predictions.

Table 4-3. Parallel vs. Single-core Policy Local Response Times

	PF	TF	LAZY	PARA
Simulated	17.58	29.58	29.58	17.58
Measured	18.13	28.83	29.86	18.48

While we were not able to show the benefit of the parallel policy for remote response times, we make the following observation. The parallel policy can at most improve the response time of a command for a user by the amount of time the user's computer's CPU requires to transmit the command to downstream computers. Based on the CPU transmission

costs we collected, we know that the Core2 desktop requires only about 12ms to transmit a Checkers command to 600 destinations, and only about 21ms to transmit a PowerPoint command to 100 destinations. Therefore, whether the transmission task runs in parallel with the processing task, as with the parallel policy, or whether it runs before the processing task, as in the transmit-first policy, the difference in response times may not be noticeable in fairly large scale scenarios. Hence, it may pay to use the single-core policies instead of the multi-core policy on a multi-core machine because the single-core policies free all cores but one for use by applications other than the application being shared.

4.8.4 Cost of Switching Scheduling Policies

The above comparisons do not consider the impact of our system on response times as it gathers parameter values and switches scheduling policies. Since the analytical model presented in this chapter did not introduce any new parameters, the impact of gathering parameter values should be the same as it was in the previous two chapters – unnoticeable in our experiments. In addition, since we do not pause the inputting of commands during a scheduling policy change, the cost of changing the scheduling policy is less important than the cost of a processing architecture change. Moreover, carrying it out requires both less transmitted information and less CPU time than a communication architecture switch. In particular, to change the scheduling policy, the server-side invokes on each computer a method with a single parameter, the scheduling policy to be used, and that method simply changes the variable on the computer whose value indicates the policy that should be used for future commands. To change the communication architecture, on the other hand, the server-side component has to send to each computer a list of destinations to which the computer forwards inputs and outputs, and that computer must update all of its transmission

data structures, establish new connections, and eventually close old connections. As mentioned in the previous chapter, the cost of switching communication architectures did not impact response times. Thus, the same should be true for switching scheduling policies. Indeed it was. The average time required to change the scheduling policy in the above experiments was 104.3ms and the maximum was 110.4ms.

4.9 Summary

To summarize, we have considered response times with the (a) transmit-first, process-first, concurrent, and lazy single-core and (b) parallel multi-core scheduling policies. The lazy and parallel policies have not been considered before. We have presented not only their definitions but also issues that must be resolved in order to implement them. We have presented an analytical model of response times that extends the model presented in the previous chapter by considering the impact of scheduling policies on response times. We have also updated our self-optimizing system to automatically select the scheduling policy that best meets response time requirements for any pair of processing and communication architectures. We have validated our model and system through simulations and experiments for both single-core and multi-core policies. Finally, we have identified several new implementation issues that must be addressed by any system that automatically maintains the scheduling policy. These results serve as proof of sub-thesis IV and V and partial proof of sub-thesis VI, which we re-state below. The partial proof of sub-thesis VI in this chapter, combined with partial proofs for it in the previous two chapters, completes the sub-thesis VI proof.

SUB-THESIS IV

The scheduling policy used for executing the processing and transmission tasks impacts response times.

SUB-THESIS V

It is possible to develop a system that automatically switches to the scheduling policy that satisfies any user-specified response time criteria better than existing systems.

SUB-THESIS VI

It is possible to develop a model that analytically evaluates the impact on response times of different processing architectures, communication architectures, and scheduling policies to the degree necessary to automate their maintenance.

4.10 Relevant Publications

1. Junuzovic, S. and Dewan, P. Serial vs. concurrent scheduling of transmission and processing tasks in collaborative systems. *Conference on Collaborative Computing: Networking, Applications, and Worksharing*. 2008. pp: 746-759.
2. Junuzovic, S., and Dewan, P. Lazy scheduling of processing and transmission tasks collaborative systems. *ACM Conference on Supporting Group Work (GROUP)*. 2009. pp: 159-168.

CHAPTER 5

RELATED WORK

5.1 Overview

In the previous three chapters, we have presented an analytical model that can predict response times for any combination of (a) centralized and replicated processing architectures, (b) unicast and multicast communication architectures, and (c) transmit-first, process-first, concurrent, lazy, and parallel scheduling policies. The model is an important contribution as it can provide guidance to users regarding which combination of these factors best meets the users' response time requirements. However, as it may be difficult and tedious to manually apply the model, we have also presented a system that can better meet response time requirements than existing systems by automatically maintaining these three factors of performance based on predictions made by the model.

We are not the first to study the performance of collaboration systems. In fact, as mentioned throughout the previous three chapters, the thesis was motivated by Chung's [21] work on performance. In this chapter, we give an overview of Chung's and other prior work that serves as either a basis for our work or provides an avenue for extending it. While we focus mostly on work from the distributed collaboration area, we also review several relevant research efforts from other computer science fields, such as human-computer interaction, networking, distributed systems, and real-time systems.

The rest of this chapter is organized as follows. We first present prior work that, like our analytical model, provides guidance to users regarding how to improve response times. Then, we present several prior techniques that may be used to further improve response times. Following this, we present the results of user studies that can be used to decide when to apply the techniques and guidelines. Then, we present related work on how to automatically apply them. We end with a brief summary.

5.2 Guidelines for Improving Response Times

As mentioned above, the analytical model presented in the previous three chapters is a powerful contribution as it provides guidance to users regarding which combination of processing architecture, communication architecture, and scheduling policies best meets the users' response time requirements. While it is the first to provide such guidance regarding multicast and scheduling policy, prior work has addressed the choice of processing architecture and its impact on response times. Such guidelines have been provided through a comparison of abstract architectures, concrete architectures used in actual systems, and empirical results. We next describe the guidelines emerging from these three areas of previous work, beginning with a discussion on abstract architectures.

5.2.1 Abstract Architectures

As mentioned in the introduction chapter, a *collaboration architecture* defines the logical system components, their physical distribution, and the interaction among them. Since collaboration architectures differ along these dimensions, it is difficult to reason about the relative performance merits of each architecture without a model that covers the entire design

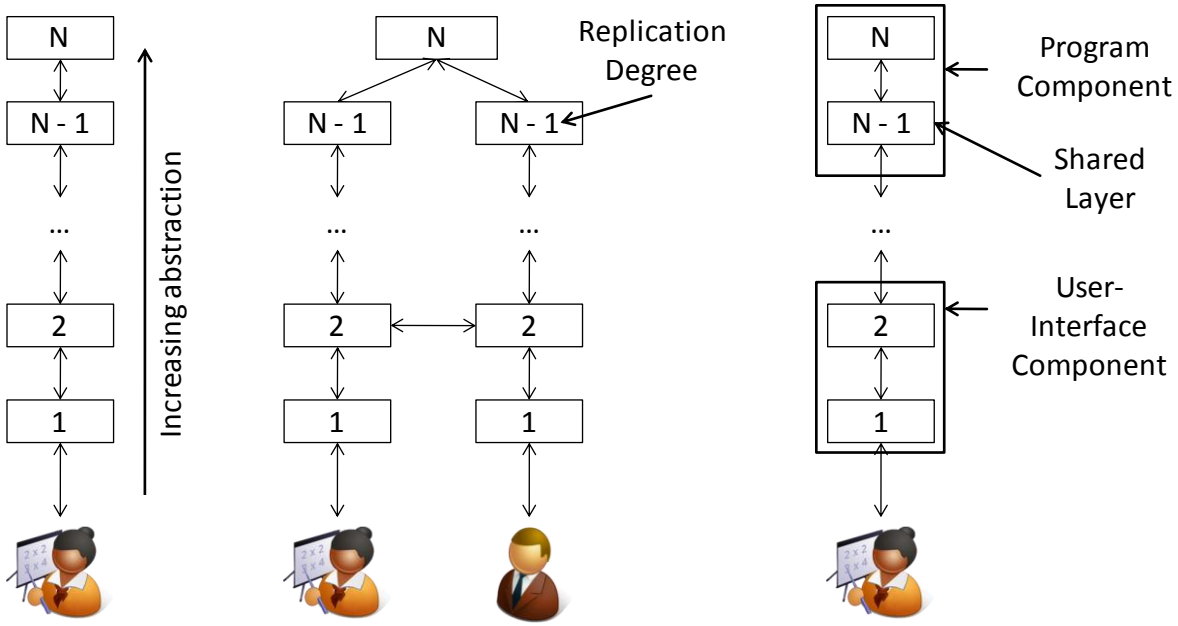

Figure 5-1. (left) Dewan's general model of collaborative applications; (center) the general model with a replication degree of N-1; (right) user-interface and program components as defined by the general model.

space defined by the dimensions. A well-known model of collaboration architectures is Dewan's general model [27], which represents an application as a stack of N layered components as shown in Figure 5-1 (left). Each component in the stack provides an abstraction to and services requests from the component at the next lower layer. Conversely, each component in the stack renders the information from the component at the next higher layer. To share an application, a layer is selected to be logically shared. The layers above the shared layer are also shared as they are abstractions of it. On the other hand, the layers below the shared layer are replicated for each user and can diverge as they may render the abstractions below them differently.

The general model exposes several parameters through which it is possible to reason about the performance (and other) properties of architectures. One of these parameters is the replication degree, which allows us to reason about the choice of processing architecture. The

replication degree is equal to the layer of the highest component in the stack that is replicated. An architecture with a replication degree of N-1 is shown in Figure 5-1 (center). When processing powers are equal, a high replication degree favors local response times. The reason is that a higher replication degree allows more of the application stack to be distributed on each user's machine. By executing more of the application on each user's machine, a higher number of user commands can be processed locally without incurring the costs of remote communication. Since in a replicated processing architecture, all processing is done locally, this result implies that equal processing powers favor a replicated architecture.

Chung [21] confirmed predictions made by Dewan analytically through a two-user analytical response time model for centralized and replicated architectures. The model assumed 1) constant costs of processing user's input commands, 2) zero costs of processing outputs, 3) zero costs of transmitting input and output commands, 4) constant think times, and 5) no type-ahead. In addition to confirming predictions made by Dewan, Chung's model also predicts that that asymmetric processing powers and low network latencies favor lower replication degrees. The reason is that it pays for the slower computer to incur the costs of remote communication in order to use the faster computer as a high-speed computation-server.

As mentioned above, Chung's model served as a basis for our own analytical model. We extended his work by creating a new analytical model that relaxed all of the assumptions made by his model. In addition, we added support for multicast and scheduling policies.

5.2.2 Concrete Architectures

One issue with the above predictions is that they may not accurately predict behavior of actual systems because the concrete architecture implemented in the systems may not fully reflect the properties of abstract architectures. In particular, while designers may use Dewan's model for a high-level organization of an application's components, they may also use one of many other collaborative architectures that decompose the components in a more fine-grained manner [68]. Examples of these models include PAC* [15], Clover [59], model-view-controller (MVC) [14], abstraction-link-view (ALV) [52], Clock [40], and Chiron-2 [87]. They are not motivated by performance. Their goal is to help the designers of collaborative applications bridge the gap between an abstract architecture and the actual code needed to implement the architecture. Thus, in addition to reasoning about the relative performances of abstract architectures, it is also useful to compare the performances of popular concrete architectures. The experiences from using architectures in actual systems may help to qualify the high-level intuition and, ultimately, make it more useful.

Graham et al. [42] qualitatively compared the properties, including response times, of five architectures each of which is used in at least three popular commercial collaborative systems. Of these five architectures, three used a centralized and two used the replicated architecture. Their analysis agrees with the general model prediction that the replicated architecture favors local response times. However, they qualified the conclusion as follows. When (a) input commands are smaller than output commands, (b) each input command generates one output command, and (c) bandwidth is limited, as is the case in some popular multiplayer games such as Half-Life [88] and Halo 2 [61], then the replicated architectures may offer worse remote response times than the centralized architectures. The reason is that

synchronizing the users' computers in the replicated case requires that each computer sends its local user's input commands to all of the other computers, while the centralized case, the inputs are sent only to the server, which then sends outputs to all users. However, as Ahuja et al. [4] show, in some applications, input and output commands can be of similar size. Moreover, there can be multiple outputs generated for each input command. They performed experiments to compare the network load imposed by the centralized and replicated implementations of a shared drawing program. They found the following three results: 1) when output was not buffered, there were 6 times as many output events as input events; 2) when output was buffered, there were 3.6 times as many output events as input events; and 3) an output event was about the same size as an input event – about 25 bytes (presumably not including network headers). Therefore, the replicated response times of the drawing application used in their experiments are better than those of centralized when bandwidth is limited for this application.

The results by Graham et al. and Ahuja et al. show that the number and size of input and output commands is application dependent. Therefore, the amount of bandwidth these commands require is non-zero and application dependent. Since the network card transmission costs are a function of bandwidth, it is important to consider the impact on response times of input and output transmission costs when there is a large number of users or when the commands are large. These results confirm the need to account for the transmission costs in our analytical model.

Concurrency Control

Graham et al. [42] also offered another qualification for the general model prediction that the replicated architecture favors local response times. In particular, they state that the

replicated mapping may actually offer worse local response times because of concurrency control mechanisms. The role of concurrency control mechanisms is to prevent users from entering conflicting commands. These mechanisms are pessimistic or optimistic by nature. Pessimistic schemes prevent the execution of a user's command until they verify that the command does not conflict with other users' commands. Optimistic schemes do not prevent an action from executing, and they recover from any conflicts that result using state rollback or undo/redo mechanisms [43]. To illustrate the impact of these schemes on response times, consider a pessimistic concurrency control mechanism. In the replicated case, the mechanism has to coordinate the command with each user's computer. In the centralized case, the mechanism has to coordinate the command only with the server. Therefore, the coordination delay is longer in replicated than in centralized architectures. As a result, local response times may be worse in replicated than in centralized architectures. The extra concurrency control coordination penalty in the replicated architecture can be avoided by using an optimistic mechanism. However, the scheme must recover from any conflicts that occur by rolling back state or undoing conflicting commands [43]. Since these recovery mechanisms require CPU time, they can degrade response times by delaying the execution of incoming user commands. Therefore, even an optimistic scheme can degrade response times. But compared to the pessimistic scheme, it only degrades response times when there are conflicts.

Regardless of whether a concurrency control mechanism is optimistic or pessimistic, an issue arises when there continuous contention for the same resource. To illustrate, consider a lock concurrency control mechanisms. Suppose several computers are repeatedly requesting the lock for the shared object. Regardless of whether the lock is pessimistic or optimistic, some computer may never get the lock if the timing of the requests is such that the

other computers always get it. This is known as starvation. Starvation is an issue because it can lead to unbounded response times in collaborative systems. It is also an issue in real-time systems because tasks may miss their deadlines if they starve. Anderson et al. [6] have shown that in real-time systems, starvation can be prevented by using lock-free concurrency control mechanisms. As their name implies, these mechanisms provide concurrency control without locks. Such mechanisms are usually implemented using "retry loops." A failed attempt to gain access to the object is automatically retried; with locks, the computer requesting the lock must request the lock again when the response to the original local request was denied. Anderson et al. show that the number of retries to get the lock is bounded when using a lock-free approach. A similar approach can be used in concurrency control mechanisms in collaborative systems to ensure that response times are not unbounded when there is continuous contention for a resource.

As the comparison of the concurrency mechanisms shows, it is important to consider the impact on response times of concurrency control mechanisms in applications that use them. Our thesis considered only applications that do not use such mechanisms. Since many popular and important applications need these mechanisms in order to function correctly, an important future direction is to extend our work to applications that do.

5.2.3 Empirical Results

Another source of response time guidelines can be obtained by comparing various architectures through experiments. The empirical data obtained in such experiments is useful for two reasons. First, the data can offer performance insights in addition to those provided by the qualitative analysis. Second, unlike the qualitative comparisons, empirical data can inform designers about the relative importance of the factors that impact response times.

There have been relatively few studies that have directly targeted collaboration. One can, however, make some collaboration implications indirectly from studies of distributed window systems.

Nieh, Yang, Novik et al. [64] conducted experiments that measured the relative performances of two distributed window systems, the Linux implementation of VNC [71] and Microsoft's Windows 2000 RDP implementation. The architecture used was essentially a two-user centralized architecture with the user at the hosting site inactive. Such a setup gives an idea of the performance experienced by a remote user interacting with a centralized program, assuming the host site does not become a bottleneck. They found that these systems perform well in LAN environments, but that their performance degrades significantly in broadband environments. Moreover, in LAN settings, the VNC implementation gave noticeably better response times than the RDP implementation for typing and scrolling commands while it gave noticeably worse response times to bitmap load operations. These studies compared two different implementations of the centralized architecture and do not addresses the relative performances of different architecture configurations.

Wong and Seltzer [94] measured the network load for various remote user operations. They found that typing at a rate of 75 words per minute, moving the mouse, menu navigation, and scrolling require a bandwidth as high as 11.67KBps, 1.95KBps, 40KBps, and 200KBps, respectively. Danskin and Hanrahan [23] measured the frequencies of these operations. They found that they account for 75% of all communication in a 2D drawing program and a text editor. Together, these two results give an idea of the actual bandwidth requirements for a variety of remote desktop tasks. Two other studies, one involving Microsoft's Terminal Services [63] and the other by Droms and Dyksen [30] showed that

average and maximum network bandwidth requirements of remote desktop operations can vary greatly. Again, these studies do not address the relative performances of different architecture configurations. This limitation was addressed by the following two works.

Ahuja et al. [4] performed experiments to compare the network load imposed by the centralized and replicated implementations of a shared drawing program. As mentioned above, they found that input and output commands were of the same size but there were 6 (3.6) times more output events when outputs were (not) buffered.

Like Ahuja, Ensor, Lucco et al. [4], Chung and Dewan [22] compared the centralized and replicated architectures, addressing response and task completion times instead of network load. Their experiments confirm the prediction made by Chung's two-user analytical model that (a) low network latency favors the centralized architecture and (b) asymmetric processing powers favor the centralized architecture. As these conditions can change dynamically, they developed a system that supports architecture changes at runtime. They also performed experiments showing that when a user with a powerful computer joins the collaboration, it is useful to dynamically centralize the shared program to the new user's computer.

Abdelkhalek et al. [1] also evaluated a collaborative system, but instead of comparing the performances of different architectures, they compared the impact of the processing and communication costs. They investigated the bottlenecks in the performance of the game server for Quake [53], a multi-player first-person shooter game. They found that the amount of data transmitted between the server and a user's computer was small (on the order of a few kilobytes per second). Moreover, they found that the server spends an equal amount of time on game-related and network-related processing. According to these results, when

considering the cost of remote communication, the available bandwidth is not the limiting factor. The limiting factor instead is the time the processor requires for transmitting a command to a single destination. The transmission time is a function both of the processing power and the available bandwidth. As Brosh and Shavitt observe [13], the reason is that when transmitting a command, the operating system requires time to traverse the network stack and copy message data buffers along the way before the message transmission can begin.

As results by Nieh, Yang, Novik et al. [64] show, it is important to consider different processing costs for different commands. Therefore, it is important to relax the assumption made by Chung's model that the costs of processing user commands are constant. The analytical model presented in the thesis relaxes this assumption. Moreover, results by Abdelkhalek et al. [1] show that both the CPU and the network card speeds are transmission cost factors. This result agrees with separating the time the CPU and the time the network card on each user's computer require to transmit a command to a single destination, which is what our model does.

5.2.4 Summary

In summary, we have presented several guidelines that users can use to decide whether to user the replicated or centralized architecture to improve response times. However, it is difficult to apply them in all scenarios because they require both the knowledge of architectural details of systems and empirical data regarding numbers and sizes of commands. Chung's two-user analytical model makes this easier as it requires only the latter. However, it made several assumptions that conflict with empirical findings. The analytical model presented in the thesis relaxes all of these assumptions.

5.3 Techniques for Improving Response Times

So far, we have presented general guidelines for making architectural and other choices (in a well-known design space) to improve response times of collaborative applications. In this section, we describe a number of techniques for improving response times for all points in the design space. These techniques focus on reducing the time required to perform mandatory collaborative application tasks, the processing and distribution of user commands, which are also the tasks we consider in our analysis. Any reduction in processing and transmission task times helps improve response times. To illustrate, consider a centralized architecture with multiple slave users. Suppose that all of the slave users simultaneously enter a command and that all of the commands reach the server at the same time. As the server will process the commands in some order, reducing the processing times helps improve local response times of commands which are processed later. As the server must also transmit an output for each command to all of the users, reducing the transmission times helps improve remote response times of the users to server transmits to last. The response times of replicated architectures can also improve when transmission and processing times are improved. The reasoning is similar.

We first present techniques for reducing processing times, then those for transmission times, and finally techniques that consider both of these factors.

5.3.1 Reducing Processing Times

As mentioned above, Chung [21] developed a system that supports architecture changes at runtime. He also performed experiments showing that when a user with a powerful computer joins the collaboration, it is useful to dynamically centralize the shared

program to the new user's computer. However, he did not provide an algorithm for deciding when to perform architecture changes.

Load-balancing

While no such algorithm exists for collaborative-systems, one could be obtained by modifying load-balancing techniques used in distributed systems. One such algorithm is presented by Seshasayee et al. [77], which load-balances a computationally heavy task across a network of mobile devices. The system attempts to optimize two metrics: the computation time and the application lifetime. The computation time is similar to the response time metrics as it measures how much time the devices require to complete some task. The application lifetime metric, which is defined only for mobile systems, measures how many computations the devices can carry out before running out of power. When a metric must be sacrificed, the algorithm sacrifices computation time, which makes sense because slow progress towards completing the task is better than no progress. The scheme continuously monitors the remaining battery charge of each device as the computation proceeds. When a device begins to run out of power, the scheme moves computation from this device onto devices with more power, even if computation time increases. As a result, the system can increase application lifetime by 10% compared to load-balancing algorithms that do not consider power. Our self-optimizing system could use a modified version of this scheme to improve response times. Roughly, the modified algorithm would 1) give preference to computation time instead of application lifetime, 2) monitor available processing powers of the devices instead of their remaining battery charges, and 3) centralize the program component on the most powerful computer.

Interest Management

One issue with centralizing the program on the most powerful computer is that the computer may become overloaded as the number of collaborators grows. For example, in massively multi-player online games, such as EverQuest [57], a single server cannot handle all of the users. Instead, a cluster of servers is used, and each server is assigned a geographic region of the game world [60]. Typically, these regions are divided by barriers that prevent any inter-region player interactions. Thus, a server needs to handle only the interactions of the players in its region. If too many players enter a region, which is an issue known as "crowding," a mini-cluster of servers is allocated to serve the region. The players in the region are evenly divided among the servers. Each server in the cluster processes only 1) actions of its users and 2) actions of users assigned to other servers that affect its users. A server is aware which of its users impact users on other servers through a technique called "interest management" [62], and thus forwards only their actions to other servers. Interest management techniques are based on a spatial model of interaction [10], which has been used

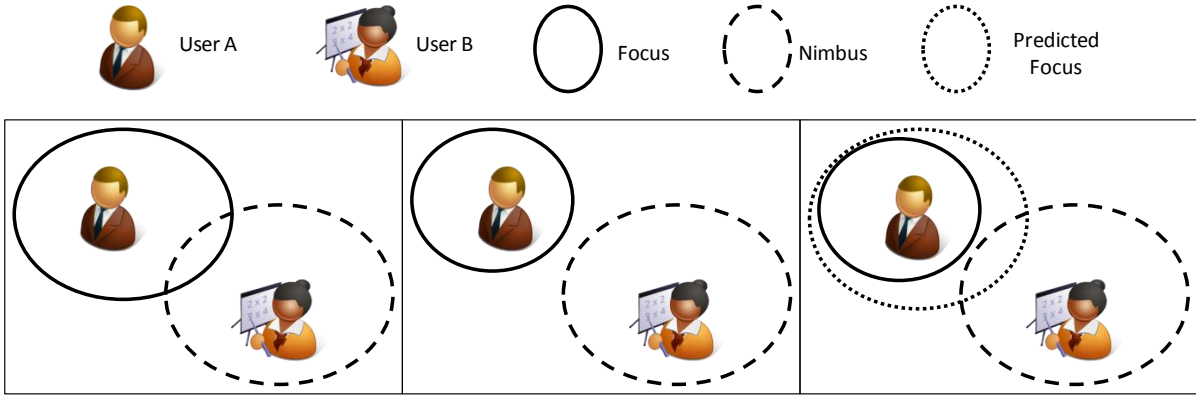

Figure 5-2. (left) User A is interested in user B's actions because user A's focus intersects with user B's nimbus, (center) user A is not interested in user B's actions because user A's focus does not intersect with user B's nimbus, (right) prefetching of user B's data onto user A's computer begins because user A's predicted focus intersects with user B's nimbus.

to create several systems, including the MASSIVE [46] and DIVE [16] virtual reality systems. This model determines which objects can interact with each other based on their proximity. Each object has an aura. If the auras of two objects intersect, then it is possible for the objects to interact. The actual way in which the objects can interact is determined by the notion of a focus and a nimbus. If a user's focus (some area around the user) intersects with another user's nimbus (an area around the other user) (Figure 5-2 (left)), then the former is interested in actions of the latter; otherwise (Figure 5-2 (center)), the former is not interested in the actions of the latter.

Interest management techniques can be used to improve response times by reducing the number of commands processed on each computer. To illustrate, consider a master computer in the replicated architecture. Recall that to keep the program replicas running on different masters in sync in the replicated architecture, each master forwards input commands from its local user to all of the other masters. Therefore, each master computer must process all input and output commands. However, some user actions, such as telepointing, do not affect the shared state. Moreover, when people collaborate, not every person may be interested in telepointer motions of every other user. For instance, users may turn off telepointer display on their machines if they do not wish to be disturbed. In addition, if the shared application enables users to work on different parts of the document simultaneously (i.e., if the application supports non-WYSIWIS interaction), then users cannot see the telepointer of a user working on different part of the document. In either case, there is no reason to send telepointer actions to these users' computers. As a result, there is no reason for a master to forward telepointer input commands to another master computer unless the user on that computer is interested in seeing the telepointer. By employing interest management in

our system, we could reduce the number of telepointer commands that are processed by each master. This, in turn, helps improve response times because it frees up resources on each user's computer to process commands that impact shared state or in which the user is interested. Applying a similar technique in centralized architectures can also improve response times by reducing the number of output commands processed by each slave.

5.3.2 Reducing Transmission Times

Interest management schemes, in addition to reducing the number of commands each computer has to process, also reduce the amount of communication among a master and 1) a slave in the centralized architecture 2) another master in the replicated architecture. In this section, we describe several other techniques which can further reduce transmission costs.

Intelligent Prefetching and Interest Management

A technique that can be combined with interest management to reduce the impact of transmission times on response times is prefetching. Prefetching does not actually reduce the amount of data that is transmitted; nevertheless, it is useful for two reasons. First, it amortizes the transmission costs over a longer period of time. Amortization reduces bottlenecks by spreading the cost of an expensive operation over many operations. Second, by the time the data is needed at a client, the client does not have to wait for the server to transmit it.

Scholtes et al. [76] developed a prefetching algorithm for massively multi-player online games. Their algorithm predicts future positions of a user in the game world based on the user's history of movement in the world. Any data that is predicted to be needed at the client in the near future, such as the data for rendering other players, is gradually prefetched onto the client. The set of predicted future positions can be thought of as a union of the user's

future foci. When the predicted foci intersect with a nimbus, the data of the player who owns the nimbus is prefetched (Figure 5-2 (right)).

Prefetching is especially useful when players go through a portal, which is special entity in the game world that can teleport a player not only to a distant area in the player's current region, but also to a different region. In this case, the player's focus may in an instant intersect with a large new set of nimbi. Since this set of nimbi was not "close" to the player's position before entering the portal, they have not been prefetched. Thus, the player needs to wait until the data for region on the other side of the portal is downloaded. To reduce the waiting time, Glinka et al. [38] create a nimbus around each portal. When the player's focus intersects with a portal's nimbus, the prefetching of data required to render the world on the other side of the portal begins. Since the player may use the portal to move to a different region, their algorithm also begins the process of proactively moving the player's data onto the new server to reduce the delay required to hand-off the player's data between the servers (or mini-clusters) responsible for the players' interactions in the current region and the region on the other side of the portal.

Prefetching techniques could be employed by our self-optimizing system to improve response times of data-centric applications such as PowerPoint. To illustrate, consider a PowerPoint presentation in which a presenter is giving a talk to a large number of audience members. Suppose that the computers belonging to the users are organized in a centralized architecture in which the presenter's computer is the master. Prefetching could improve response times as follows. While the presenter is discussing a slide, our system could use the time to transmit the next slide or the next several slides to all of the users. Therefore, when the presenter eventually advances to the next slide, the slide will already be present on the

remote user's machines. This would reduce local response times as in the transmit-first policy case as the CPU on the presenter's computer would not have to transmit the slide first. Moreover, it would reduce the remote response times as the remote computers would not have to wait for the presenter's network card to transmit the slide. However, prefetching can also hurt response times when non-blocking communication is used and think times are low. To illustrate, suppose that the presenter enters a "next animation" command after the CPU but before the network card has completed the transmission task for prefetching the next slide. In this case, the network card will begin the transmission of the next animation command only once it completes the transmission of the prefetched data. Therefore, the remote response times of the next animation command increase. More generally, in data-centric applications, when non-blocking communication is used and think times are low, prefetching data that has not yet been distributed to all users can degrade the remote response times of user commands that interact with data that has been distributed. Of course, when blocking communication is used, the thread that is transmitting the prefetched data could be paused, and hence the network card would be ready to transmit the user command immediately. It would be interesting to consider a communication protocol in which the prefetched data is transmitted using blocking communication while user commands are transmitted using non-blocking communication. In this case, it would be possible to benefit from prefetching without hurting remote response times.

General Message Compression

The simplest approach to reducing the transmission times is to compress each message before it is transmitted. However, there are two issues with this approach. First, such algorithms require processor time to run both on the sender and the receiver, and can

therefore actually increase the time needed to transmit and process messages. Second, if a message is not large, traditional compression algorithms may not actually compress it significantly. For example, Gutwin et al. [51] showed that the popular Lempel-ziv algorithm compressed an 88 byte telepointer message as little as two percent.

The poor compression stems from the fact that there is not enough redundant repetition within a single message for general compression schemes to work well. However, Gutwin et al. [51] also found that a typical application transmits only certain message types during a session and that the data in two messages of the same type from the same user has a significant amount of overlap. For example, they observed that two consecutive telepointer messages originating on the same computer differ only in the X and Y coordinates of the telepointer. As a result, they developed a compression technique that capitalizes on the inter-message rather than intra-message redundancy. After several messages of the same type are transmitted by the computer, their redundancy algorithm is able to compress messages as much as 92%. Moreover, it can perform this compression with low memory, processing, and communication overheads. Thus, their scheme can significantly reduce the amount of transmitted data, and thus the transmission costs.

Compression techniques can be implemented in our self-optimization system. For instance, in Checkers, messages are small. Moreover, input commands are similar in nature to telepointer commands – instead of containing X and Y mouse positions, they contain the X and Y board position of the piece that moved. Therefore, the scheme presented by Gutwin et al. [51] can be used to reduce the transmission costs of these commands. On the other hand, in PowerPoint, message content can vary significantly since two slides can contain significantly different data. Moreover, the commands that contain the slides carry large

amounts of data. Therefore, it may be better to use traditional compression algorithms, such as Lempel-Ziv, instead of Gutwin et al.'s algorithm to compress these commands and reduce their transmission costs. In fact, as part of the self-optimizing calculations, our system could dynamically decide which compression technique to use in order to improve response times the most.

Multicast

The message compression and prefetching techniques can improve response times by reducing and amortizing transmission times. Another approach is to distribute and parallelize the task of transmitting a message to a set of destinations across several computers, which is the approach taken by multicast algorithms.

Previous work in operating systems and networking has shown that multicast can reduce the worst case end-to-end delays in content-streaming systems [18][56]. The end-to-end delay metric is related to the remote response time metric as both contain the total network latencies a message experiences from the source to the destination. They are not exactly the same, however, as the remote response times include the time required to process the message at the destination computer. Nevertheless, since it can reduce end-to-end delays, multicast schemes are still useful for improving remote response times.

The idea of multicast requires the creation of multicast communication overlays, which consists of paths from the source to all of the receivers. The end-to-end delays are improved by using an overlay which 1) minimizes the sum of transmission delays and network latencies on a path from the source to each receiver and 2) respects the bandwidth capabilities of the devices. Therefore, traditional multicast schemes focus on minimizing the diameter of the multicast overlay while satisfying the host constraints. The implicit

assumption in this approach is that the diameter of the overlay determines the largest end-to-end cost. However, as mentioned above, transmission times[1] and not the bandwidth is the limiting factor at the application-layer, and therefore, in collaborative systems. As described in chapter three, the transmission times are a factor of both the CPU and network card transmission times. In particular, they are a function of (a) the amount of time the CPU requires for duplicating a message and queuing it for transmission by the network card and (b) the amount of time the network card requires for reading a message from memory and sending it on the physical wire. Bandwidth is still a factor. It is accounted for in the time the network card requires for sending a message on the physical wire. Hence, the network latency and bandwidth oriented (i.e., traditional) multicast algorithms may not distribute and parallelize the transmission task in an optimal fashion. Recognizing that the transmission times can be important in some scenarios, Brosh and Shavitt developed a multicast algorithm called HMDM [13] that explicitly considers transmission times on the total end-to-end delays. This is the main reason why we use HMDM to create multicast trees in our self-optimizing system instead of a different multicast scheme. By doing so, their algorithm is able to significantly reduce maximum end-to-end delays. As our results in chapter three show, HMDM can be used to improve the maximum remote response times in collaborative applications. In addition to HMDM, other efficient application-layer multicast schemes have been proposed. One of these schemes is the NICE multicast protocol described by Banerjee et al. [8]. The NICE protocol organizes end-user devices into a hierarchy of L layers. At each

[1] As explained in chapter 3, networking researchers use the term processing time to denote the amount of time the CPU requires for duplicating a message and queuing it for transmission by the network card. What they call processing times, we call CPU transmission times. We use a more specific term as there are other kinds of processing tasks in our domain, such as those of processing input and output commands.

layer, devices are organized in clusters. Devices can communicate directly only with other devices in their cluster. Unicast is used for intra-cluster communication. At a non-top-most layer, a leader device is selected in each cluster. The leaders are grouped into clusters at the next higher layer. Therefore, leaders can be members of multiple clusters but each of the clusters are at different layers. The leaders distribute messages among all of the devices in their clusters. However, the NICE algorithm does not explicitly consider application-layer transmission times, that is, both CPU and network card transmission costs, when forming the clusters and the layers. In fact, no other multicast scheme of which we are aware accounts for them. Since in collaborative systems, they are a factor, we used HMDM. In the future, it would be interesting to study the impact on response times of NICE and other application-layer multicast schemes. For example, it would be useful to investigate the formation of NICE clusters and layers based on the processing and transmission powers of the devices and evaluating NICE under various combinations of processing architectures and scheduling policies.

One issue with the HMDM multicast scheme is that it can result in a large end-to-end delay variations. Therefore, using HMDM in collaborative systems could result in high remote response time variations, which is an issue in some scenarios. For example, in a multiplayer game, if a unique item becomes available for sale, ideally, all of the players should become aware of the item availability at the same time to give all of them an equal chance of buying it. The Chains scheme presented by Banik et al. [9] can be used to solve the problem. Their algorithm explicitly creates overlays with minimal variations in end-to-end delays. It does this by calculating for each destination k shortest paths from the source to the destination. The path lengths are calculated based on the bandwidth of each node on the path

and the network latencies along the path. Then, it selects for each destination one of the k paths, such that the variation in the lengths of the picked paths is minimized. If there are several combinations of paths with equal delay variations, then Chains picks the combination that gives a multicast tree with a minimum diameter. It would be interesting to combine Chains and HMDM. In particular, when finding the shortest paths from the source to a destination, the Chains algorithm would, like HMDM, measure the path length using transmission costs and network latencies. Such a scheme could provide bounded remote response time variations in collaborative systems while improving remote response times.

Reducing the Quality and Rate of Message Streams

The multicast schemes described so far focus on the paths from the source to each destination. A different approach is to focus on paths from the source to a group of destinations with similar capabilities. Grouping devices according to their capabilities would allow the multicast scheme to customize data that is forwarded to members of a group based on the group's capabilities and as a result reduce the cost of transmitting data to the group. To illustrate, consider a scenario in which the video of a user is being transmitted to other users. Suppose that some of these users are using smart phones and others LAN-connected desktop computers. When transmitting the video frames, the multicast scheme could automatically adjust the rate at which it transmits frames to a group to match the maximum frame rate supported by the devices in the group. Thus, it would transmit lower frame-rate video to the smart phones than the desktops. As a result, it would reduce the costs of transmitting data to the smart phones destinations without those users noticing a difference in quality. Chawathe et al. [19] present RMX, which is multicast scheme that works precisely in this manner.

RMX adaptive transmission rate is an instance of a more general idea of controlling flow of information in push-based systems. With push-based systems, operations are pushed out by some computer to one or more remote computers. If some computers are slower than others, then a push rate that fast computers can handle may actually overload the slower computers. One approach to prevent overload on slow computers is to monitor how quickly remote computers can process the pushed commands and then adapt a rate that does not overload any of the computers. This is approach taken by T.120 [24], which monitors the receive queue sizes at the remote sites. The push-rate is selected based on the queue sizes at the slowest computer. If the queue size is below some low threshold, then the push-rate is increased. If the queue size is above some high threshold, then the push-rate is decreased. An issue with decreasing the push-rate to accommodate a user with slow computers is that performance is degraded for all other users. As a result, T.120 forces a user to leave the collaborative session if the queue size on a computer exceeds some even higher threshold.

The dual of pull-based communication is push-based communication. In pull-based communication, each computer pulls data from the server. An instance of pull-based communication is the intelligent prefetching idea described above. An important advantage of pull-based communication is that pulling computers can request data at their own pace. As a result, slow computers can make sure that they pull at a rate that does not overload them. More importantly, the slow computer will not affect faster computers as the fast computers can pull at a faster rate. Because of this advantage, VNC [71] allows remote computers to pull data at their own pace. The benefit to transmission times is that data is not transmitted to a computer unless the computer is ready to process it.

Reducing the quality and rate of message streams is useful for more than just preventing slow machines from being overloaded. It is also useful to adapt the quality and rate based on the importance of the stream. Greenhalgh et al. [46] designed a collaborative virtual environment that also customizes the frame rates of videos a user sees of other users, using interest management instead of the capabilities of the devices to determine the frame rate. The basics of their scheme are as follows. The quality of the video stream exchanged between two users is based on the proximity of the users' avatars in the virtual world. The smaller the distance between their avatars, the higher the quality of videos they receive of each other. If a user can see several avatars at once, high quality video is shown for the remote user whose avatar appears in the center of the user's view while low quality video is shown for the other users. The reason is that in a collaborative virtual environment, standing directly in front of an avatar usually indicates interest.

Video lends itself well to the scheme used by Greenhalgh et al. [46] because a low quality video can still convey a significant amount of information. The same is true of pictures – a low quality version of a picture can provide much of the information that is found in a full quality version. It would be interesting to incorporate the idea of trading-off the quality of data sent to users for response times. To illustrate, consider a distributed PowerPoint presentation. In many presentations, slides contain pictures. If these pictures are of high quality, then they are also large (in terms of bytes). Therefore, the cost of transmitting the slide containing such pictures will be high. Our self-optimizing system could calculate the best tradeoff between picture quality and transmission costs before transmitting the slide. As a result, the lower transmission costs would help improve response times while the remote users could still understand the information on the slide. However, for this to work,

the framework would have to be aware of some application semantics because it would need to figure out which of the data in the commands is the image data in order to compress it. Alternatively, it could invoke a command in the application that reduces the quality of data that is in the shared commands.

Gutwin et al. [50] extended the reduced video and image quality idea to telepointers. They discovered that when a dead-reckoning algorithm is used, telepointer motions can be understood by a remote user even if telepointer commands are late or lost, assuming unreliable communication. In general, when commands from a remote user are late (or lost), dead-reckoning algorithms estimate what these commands would have been had they arrived on time. The algorithms extrapolate the late commands based on the most recent commands received from the remote users. To illustrate, consider a scenario in which a remote user is using a telepointer to underline some object. Telepointing commands are sent at a regular pace of thirty per second, or one every 33ms. Suppose that at after some command, more than 33ms elapses and no new command is received, that is, the next command is late. The dead-reckoning algorithm analyzes the previous commands and sees that the remote user has been making a line motion. Based on the direction and the velocity of the motion, the algorithm estimates what the late command should be and displays its output to the local user. A perfectly accurate dead-reckoning algorithm can therefore mask any missing telepointing commands. Gutwin et al. suggest that when a dead-reckoning algorithm is used for telepointer motions, it would be useful to selectively drop telepointer messages in order to reduce network loads (and thus, reduce transmission times). However, they did not provide an algorithm that would selectively drop the messages. In general, dead-reckoning algorithms work well as long as the number of consecutive messages that are lost or delayed is limited.

The reason is that even if the algorithm incorrectly predicts a command, it soon receives an actual command from the remote user and can therefore correct what is being displayed to the local user. What is needed then is a message transmission scheme that ensures that the number of dropped or delayed messages in a group of consecutive messages is below the maximum percentage of dropped or delayed messages that dead-reckoning algorithms can handle effectively. The algorithm presented by West et al. [89] makes exactly such a guarantee by ensuring than at least X out of Y consecutive messages will be delivered. Moreover, it guarantees that the arrival rate of these messages will not be worse than the rate at which they were originally sent. Therefore, this algorithm could be used to strategically drop some number of telepointer messages while delivering a sufficient stream of messages for the dead-reckoning algorithms to work correctly. The end result is a reduction in total transmission times, which helps to improve response times. Of course, the dead-reckoning algorithms running on the destination computers have to adjust for the missing commands, which in turn increase processing costs of commands, and therefore, the response times. Thus, the ideal number of messages that should be dropped needs to be carefully calculated on a per scenario basis in order to improve the response times the most. Gutwin et al. [50] did not present any such equations, but if they exist, then they can be incorporated into our self-optimizing system. Based on these equations, our system could selectively drop the transmission a command to one or more destinations.

Scheduling Issues

In some cases, however, messages cannot be dropped. In this case, Abdelkhalek et al. [2] demonstrated that it is useful to use multiple threads to perform the transmission task. They performed experiments in which a special parallel version of the Quake game server

[53] mentioned above ran on an eight-processor machine. One, two, four, and eight threads were used to perform the transmission task, each thread running on a different processor. Their results show that as the number of threads increased, the amount of time that elapsed form the moment the CPU began to the moment it finished transmitting the messages to all of the users' computers decreased. In particular, the policy states that only one core should be used for the transmission task because otherwise, it is difficult to predict remote response times. As stated in chapter four, if predicting the response times is not important, then the CPU can complete the transmission sooner by using multiple threads for the transmission task, each running on a different core, which is in agreement with results shown by Abdelkhalek et al. [2].

When a single-core is available for scheduling, Ostrowski and Birman [56] showed that using a single transmission thread provides the best performance. The reason is that the garbage collector mechanisms can slow down the system because of high memory overheads associated with having many threads active at once. All of our single-core policies use a single thread for performing the transmission task, which follows the Ostrowski and Birman's guideline.

5.3.3 Reducing Both Processing and Transmission Times

So far, we've looked at scheduling algorithms only from an individual machine's point of view. In collaborative systems, however, scheduling of tasks across machines is also important. For example, the response time on a slave machine depends both on it and on its master computer. Moreover, if multicast is used, the response time for a machine depends on all of the machines on the path from the source to the machine. Thus, it is important to consider scheduling schemes that optimize the performance of the entire system. One

approach is to greedily schedule the processing task on the most powerful machines and make these machines responsible for most of the transmission task. Proceeding in a greedy fashion, the algorithm could schedule a smaller percentage of the transmission task on the next most powerful set of machines, and continue until every part of the transmission task is scheduled on some machine. For instance, in a collaborative system, such an algorithm could create a centralized architecture in which the most powerful machine is the master. However, the master would transmit commands to most but not all of its slaves, since the algorithm would create a multicast overlay in which the slave machines participate in the transmission of commands. An algorithm that uses such an approach for scheduling the stages of a complex task on the nodes of a high-performance computer is Streamline [3]. Streamline can schedule tasks that are either computation intensive, communication intensive, or both. As a result, it should be able to schedule the processing and transmission tasks in a wide range of collaborative applications.

Interestingly, it should be possible to unify the scheduling aspects of Streamline and the reallocation of computation aspects of the energy-aware load-balancing algorithm presented by Seshasayee et al. [77] to create an algorithm that both schedules and load-balances the processing and transmission tasks in a collaborative system.

5.3.4 Other Lazy Systems

In fact, the algorithm presented by Seshasayee et al. [77] is in one respect similar to our lazy policy – both of them rely on the notion of completing tasks "just in time."Their algorithm uses slack to complete tasks just in time while maximizing the battery life of each device. For instance, if a task completes early, they use dynamic voltage and frequency scaling to reduce the amount of power consumed by the mobile devices. As a result, the task

completes later, but on time. Unlike our lazy policy, which trades-off local and remote response times, their scheme trades-off application lifetime for response times.

The notion of "just in time" is also found in real-time systems. The idea of lazy scheduling when multicast is used is similar to the problem addressed by distributed real-time systems, which use end-to-end scheduling algorithms. In end-to-end scheduling, a task is divided into subtasks, and each subtask is allocated to a different processor. The sub-tasks are governed by precedence constraints – subtask k cannot start until subtask k-1 completes. Distributing subtasks in this fashion is similar to building multicast overlays. Multicast schemes divide the transmission task into subtasks and schedule each subtask on a different machine. Precedence constraints are implicit as a machine must wait for a command to arrive before forwarding it. However, there are several important differences between end-to-end scheduling and the lazy policy. First, in end-to-end scheduling, the system has freedom in mapping a task to any processor, while in a collaborative system, the processing architecture constrains the mapping. In particular, regardless of the architecture, input (output) processing must be done on all master (master and slave) machines. Thus, processing tasks are not distributable in end-to-end scheduling. Second, the two kinds of systems have fundamentally different goals. The main goal of end-to-end scheduling is to complete the final subtask by the overall task deadline, while the main goal of the lazy policy is to meet the processing task deadline on as many computers as possible. As a result, our scheme trades off local and remote response times, while end-to-end scheduling schemes only guarantee that a task completes on time.

5.3.5 Summary

In summary, this section presented several techniques for improving response times, including load-balancing, message compression, prefetching, multicast, quality and rate of data adjustments, and finally scheduling. All of these techniques are related to one another, which is indicative of the fact that all of these approaches can be combined in a number of ways. Moreover, we have outlined a number of ways that these techniques can be incorporated into the self-optimizing system.

5.4 User Studies

So far, we have seen guidelines and techniques for improving response times. Previous work has also defined when these guidelines and techniques should be applied. As mentioned in the introduction chapter, Sheiderman [80] has shown that users cannot notice local response times below 50ms. Jay et al [54] complement these results by showing remote response times of visual and haptic operations of 50ms and 25ms, respectively, are noticeable. Moreover, they have shown that 50ms increments in the remote response times of both types of operations are noticeable. While changes in local response time increments were not directly addressed, Youmans [95] has shown indirectly that a 50ms change is noticeable.

In addition to using noticeable response time thresholds for comparing response times of two systems, it is also useful to consider intolerable thresholds. Such thresholds are important because they inform designers what the maximum response can be without the users quitting the application. Steinmetz [82] studied what remote response time values the users feel are annoying for a variety of collaborative applications. The results revealed

different thresholds for different applications. For example, if a user is talking and telepointing at the same time, then the telepointer should not be displayed on a remote user's screen more than 750ms after the audio. Similarly, during a slide show, the slides should appear no later than 500ms after the presenters audio. Since, typically, audio is established through separate specialized channels, such as the phone system, its remote response time is low (i.e. 0ms). Thus, the above results show that if the remote response times of telepointers and presentation operations are more than 750ms and 500ms, then the users will be annoyed. However, these are the thresholds at which the remote response time becomes annoying. In every single experiment Steinmetz presents, users began to notice threshold times much earlier than the above thresholds suggest.

Our self-optimizing system could be configured to find the system configuration that ensures that the response times are below the annoying thresholds. In fact, the only change needed is in the total order function that specifies the users' response time requirements. Instead of comparing response times across systems, the function could simply check if any user's response time is above the annoying threshold and if so, reject the system. If all possible systems break the threshold, then the total order function could inform our system to stop the collaborative session. This is similar to the way the T.120 protocol [24] removes a user from the session if the capabilities of the user's device are causing a significant degradation in the performance of other users.

As mentioned above, it is not only absolute response time thresholds that are useful. In particular, the variation in the response times of all users is also a useful metric. A related metric is difference between the local and remote response times. Stuckel and Gutwin [83] have shown that in two-user tightly-coupled scenarios in which latencies are high, it is useful

if both collaborators see the output to a command at approximately the same time. They showed that delaying the command locally by as much as 200ms did not result in significant negative impact on the user experience. Interestingly, the lazy policy we propose also reduces the difference in remote and local response times. The difference between their policy and ours is that they minimize the difference by making them both high (i.e., noticeable), while the lazy policy tries to make them both low (i.e., unnoticeable).

5.5 Automatically Improving Response Times

While each of the above guidelines and techniques can improve response times, none of them is without issues. First, we have shown that the general guidelines have a number of qualifications that dictate whether or not to apply the guideline. As more research is carried out, the number of these qualifications is likely to increase. Second, even though we have presented several techniques that can help with response times, many of them, such as multicast, load-balancing, and scheduling, require a considerable amount of research to make them useable in collaborative applications.

One approach to resolving these problems is to develop toolkits and programming environments that help developers improve response times by automatically applying one or more of the response time guidelines and techniques. Several systems already exist. For example, RTF [38] and ClockWorks [41] enable a developer to use special programming statements to dictate whether a component is replicated or centralized. However, the user must still decide when and what to replicate or centralize. Several toolkits, such as GroupKit [74] and Suite [26] make it easier to develop collaborative applications but they do not support arbitrary architectures, which may be needed for to maximize the potential

improvement on response times by using load-balancing algorithms. Chung's system supports arbitrary architectures [21] but leaves the load-balancing up to the users who may not make optimal (or in fact any) load-balancing decisions. What is really needed is a system designed specifically for collaborative systems that can automatically improve response times [68][92]. This is a hole that the self-optimizing system we have created is designed to fill.

The only other system of which we are aware that is similar to the self-optimizing system in this thesis has been recently described by Wolfe et al [93]. Their work was done concurrently with ours. There are two important differences between the systems. First, the system in this thesis automates the maintenance of the processing architecture, communication architecture, and scheduling policy, while Fiia automates the maintenance of only the processing architecture. Second, the system in this thesis uses a predictive analytical model to decide which configuration to use. Fiia, on the other hand, relies on developer provided hints, which are given through custom annotations in the actual program code, and a set of rules of thumb, which are based on previously published performance results and guidelines. These rules of thumb are similar to the guidelines we have described above, which were imprecise. In particular, they did not account for nuances in collaborative scenarios, which can be important when optimizing performance. Hence, one drawback of Fiia is that it can potentially choose the wrong processing architecture. To illustrate, consider the following rule of thumb: when local response times are important, the replicated architecture should be used. As mentioned above, Chung [21] has shown that this rule of thumb is not correct in all situations. Hence, Fiia can degrade performance in some cases.

As mentioned above, unlike Fiia, the self-optimizing system in this thesis applies an analytical model to decide how to reconfigure the application. It estimates values of the parameters in the model based on the values observed for them earlier in the session or during previous sessions. The idea of using known values of a parameter to estimate its future values has been used in other works, not just by us. For example, Block [12] has presented a self-optimizing real-time scheduler that uses the same approach. Typically, real-time schedulers are provided with the amount of processor time required to complete a task. If a task is repetitive, the same provided value is used as the time for all of them. In general, the values can be wrong, which can lead to sub-optimal scheduling decisions. For this reason, Block's scheduler attempts to refine the provided values during the scheduling session. His scheduler estimates the amount of processor time required to complete a task based on the amount of processor time that was required to complete the task in the past. By making scheduling decisions based on the estimated values, the scheduler is better able to schedule real-time tasks than existing schedulers, which do not attempt to refine initially provided values.

5.6 Summary

A significant amount of previous research has investigated ways of improving the performance of collaborative systems. In this chapter, we have shown how the techniques and guidelines stemming from the prior work serve either as a basis for our work or provide an avenue for extending it.

CHAPTER 6

DISCUSSIONS

6.1 Overview

The results presented in the preceding chapters serve as proof of our thesis. They show that for certain classes of applications, it is possible to improve the response times through a new framework without requiring changes to the hardware, network, or the user-interface. The framework improves performance by automating the maintenance of the processing architecture, communication architecture, and the scheduling policy. The results show that automating the maintenance of one of these three factors of performance in isolation of the others can improve response times. However, there is nothing in the analytical model or the self-optimizing system that prevents changing more than one factor during a collaboration session. In this chapter, we will discuss how our self-optimizing system can change all three factors in a single optimization step.

The self-optimization system, however, cannot improve the performance of all applications. More specifically, the system is useable only with thin-client applications that neither use atomic or causal broadcast nor employ access, awareness, concurrency, or consistency mechanisms. Several important applications satisfy these assumptions, including those in the three problems that motivated the thesis: a distributed PowerPoint presentation, a collaborative Checkers game, and instant messaging. However, other important applications,

such as Second Life, do not, and hence our system cannot be used for it. Therefore, an important future direction for us is to relax these assumptions. For now, in this chapter, we will discuss the extent to which our current results apply to applications for which the assumptions do not hold.

Finally, the system presented in the thesis is a complex prototype that can under certain conditions have clear benefits over existing groupware systems. However, as mentioned above, several simplifying assumptions were made as the system and the analytical model it uses were developed. Therefore, an important question is whether the system should be recommended as a practical alternative to software developers. The simple answer is that the system is a realistic alternative to existing solutions, but it cannot be applied blindly as it involves a number of trade-offs and unknowns. Thus, a careful cost-benefit analysis is necessary before deciding whether or not to use the system in practice.

The rest of this chapter is organized as follows. First, we describe how our system can change the processing architecture, communication architecture, and the scheduling policy in a single optimization step. Following this, we present the extent to which our system applies when the application assumptions are relaxed. Finally, we address the cost-benefit tradeoff of the self-optimizing system.

6.2 Self-Optimizing All At Once

The self-optimizing system can change the processing architecture, communication architectures, and scheduling policy in a single optimization step. To illustrate how it does this, consider first the high-level steps the self-optimizing system takes to optimize each of them. Regardless of what factor our system is optimizing, the first step is always to gather

values of all parameters in the analytical model. Once it gathers the values, the second step is takes is to calculate the response time matrix, although in a slightly different manner for each performance factor. As described in the previous chapters, the response time matrix has three dimensions. An entry [x,y,z] in the matrix gives the response time of $user_z$ for an input command entered by $user_y$ when the factor being optimized has value x. The exact calculation of the matrix for each factor is as follows:

Processing Architecture: calculate response time matrix, that is, calculate all users' response times of a command entered by each inputting user in the replicated and all centralized architectures.

Communication architecture: calculate response time matrix, that is, for the current processing architecture, calculate all user's response times of a command entered by each inputting user with and without multicast.

Scheduling Policy: calculate response time matrix, that is, for the current processing and communication architecture, calculate all users' response times of a command entered by each inputting user with each scheduling policy.

Once it calculates the response time matrix, the third step the system takes is to invoke the total order function, passing it the response time matrix, list of inputting users, and identities of all users. Like the first step, the third step is the same regardless of what factor our system is optimizing. Finally, the system takes a fourth step that depends on the factor being optimized:

Processing architecture: switch to the processing architecture returned by the total order function.

Communication architecture: switch to the communication architecture returned by the total order function.

Scheduling policy: switch to the scheduling policy returned by the total order function.

Hence, to change all three factors at once, we have to reconcile the second and fourth steps.

Consider the second step which calculates the response time matrix. As we had mentioned the earlier chapters, both unicast and multicast can be used in both centralized and replicated architectures. Moreover, any of the scheduling policies can be used with any processing and communication architecture. Therefore, when calculating the response time matrix, our system calculates the response times of commands entered by each inputting user for all possible combinations of the processing architecture, communication architecture, and scheduling policy. Hence, it actually calculates a five dimension response time matrix instead of the three dimension matrix described previously. The first dimension of the matrix indexes the processing architecture, the second the communication architecture, and the third the scheduling policy. When the total order function decides which system best meets response time requirements, it then effectively decides which combination of the three does so. The value returned by the total order function gives the index of the processing architecture that should be used and the index of the scheduling policy that can be used since all users use a single processing architecture and scheduling policy at a time. Since there can be multiple multicast trees deployed in a communication architecture, as described in chapter three, the function also returns for each inputting user a pair of values. The first value indicates whether unicast or multicast should be used to distribute the inputting user's input commands (the corresponding output commands) in the replicated (centralized) processing architecture. If it

indicates multicast, then the second value indicates which user is the root of the multicast tree that should be used to distribute commands.

Finally, consider the fourth step which deploys the processing architecture, communication architecture, and scheduling policy returned by the total order function. One approach to changing all three is to change them in a single atomic step during which the acceptance of new input commands is paused. This is the approach we use to change the processing architecture in chapter two. As described in chapter two, one issue with the atomic change is that changing the processing architecture takes time and therefore, the response times of input commands entered during the change can be high. As our results in the previous three chapters show, while switching scheduling policies can be done relatively quickly, changing the communication architecture can take longer than changing the processing architecture. Thus, changing all three atomically can result in even higher response times of commands entered during the change. An alternative approach, which is the one we use, is to change the three sequentially. A sequential change is an attractive idea because, as described in the previous three chapters, only the processing architecture change must be done atomically. Therefore, the advantage of the sequential approach is that we need to pause input only for the duration the processing architecture change.

When sequentially deploying the processing architecture, communication architecture, and scheduling policy, we must select the order in which we do so. Since the processing architecture defines which computers process input commands, it must be deployed before the communication architecture as the communication architecture dictates how the input commands are distributed. Otherwise, a slave computer that will be a master in the new processing architecture may receive input commands before it actually becomes a

master, which is inconsistent with the notion of the collaboration architectures. Therefore, we first deploy the processing architecture and then the communication architecture. As described in chapter four, the scheduling policy is independent of the processing and communication architectures. Thus, we arbitrarily chose to deploy it last

One issue when deploying a new processing architecture is deciding what communication architecture and scheduling policy should be used until they are also changed. In particular, since the new processing architecture redefines which users process input commands, a previous multicast architecture may distribute input commands to slave computers, which is again inconsistent with the notion of collaboration architectures. To illustrate, consider a six-user scenario in which $user_1$ is the only inputting user. Suppose that

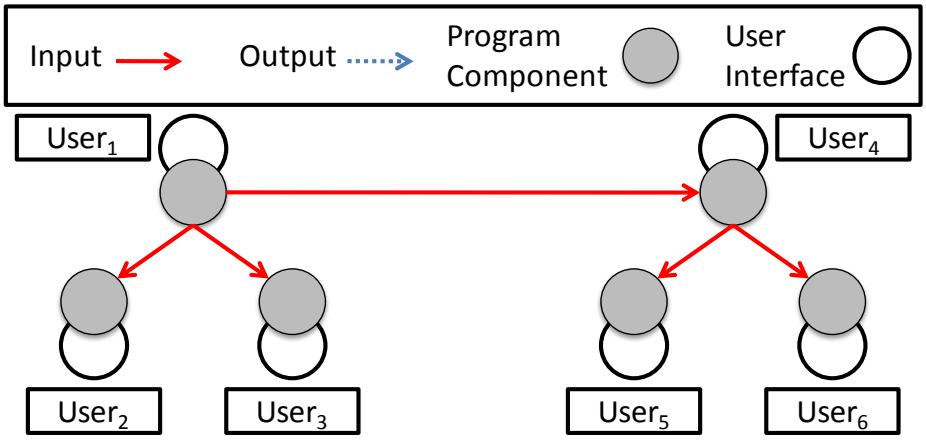

Figure 6-1. Replicated-multicast (consistent) architecture.

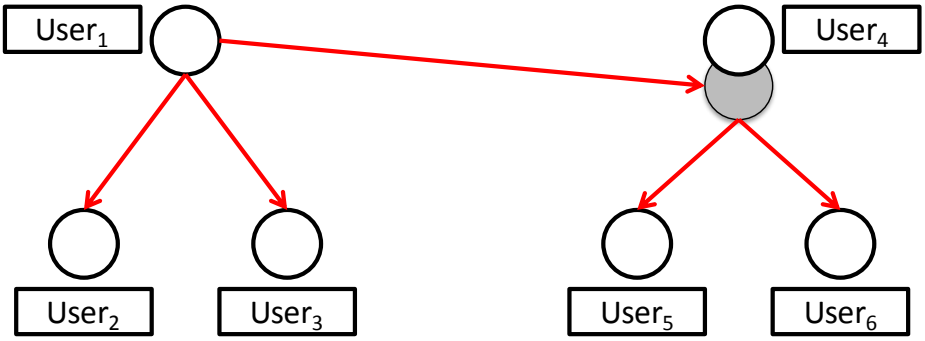

Figure 6-2. Centralized-multicast (inconsistent) architecture.

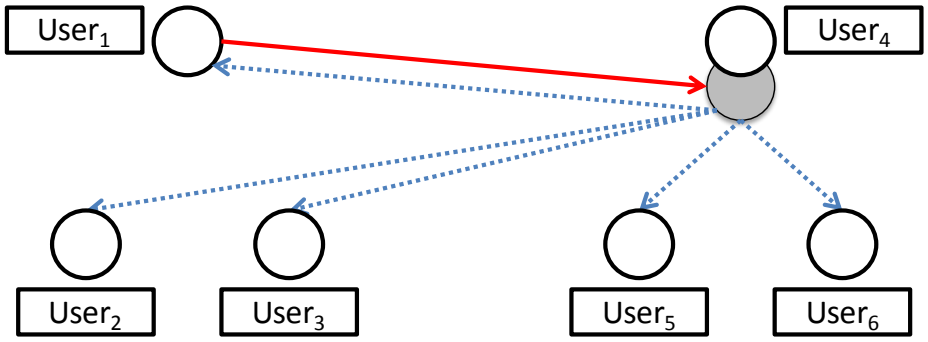

Figure 6-3. Centralized-unicast (consistent) architecture.

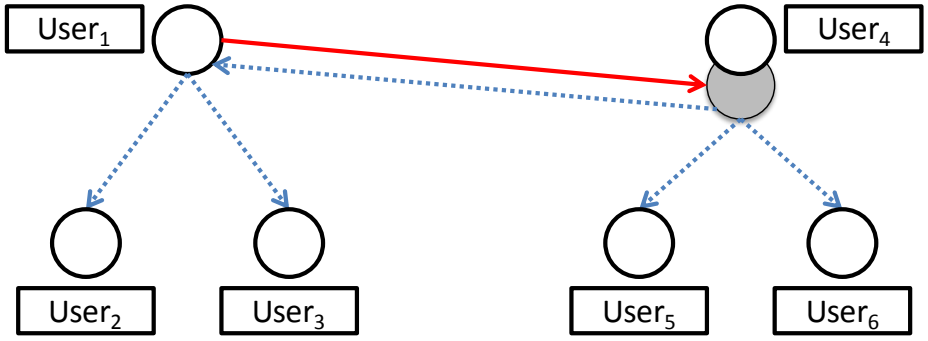

Figure 6-4. Centralized-multicast (consistent) architecture.

a change is made from a replicated-multicast architecture shown in Figure 6-1 to a centralized-multicast architecture in which $user_4$'s computer is the master shown in Figure 6-4. We first change the processing architecture to centralized. If we leave the old communication architecture, then slave users receive input commands, as illustrated in Figure 6-2. Ideally, we would like to compute a new multicast architecture that is consistent with the new processing architecture and deploy it immediately. However, it is quicker to compute a unicast architecture that is consistent. Therefore, when we change the processing architecture, we also change the communication architecture to unicast, to get a centralized-unicast architecture as seen in Figure 6-3. Eventually, we change to the centralized-multicast architecture shown in Figure 6-4. We must also choose a scheduling policy to use. Our system simply keeps the old scheduling policy since the scheduling policy does not bring out any inconsistency issues regardless of which one is used.

A potential problem with our procedure is that the framework may temporarily configure the application in a non-optimal fashion. In particular, since inputs are resumed once the processing architecture is changed, the combination of the new processing architecture and the old communication architecture and scheduling policy may give sub-optimal response times. For instance, in our transition example, when the centralized-unicast architecture is deployed, $user_4$ has to distribute outputs to all other users. However, in the final centralized-multicast architecture, $user_4$ transmits commands only to $user_5$ and $user_6$. Since the multicast communication architecture is used because it improves response times, using unicast communication may result in sub-optimal performance. Similar temporary problems can arise because of using the old instead of the new scheduling policy. Future work is needed to create an atomic approach through which these the durations of these temporary periods is reduced.

So far, we have discussed only how our system can improve performance by changing more than one factor of performance in a single optimization step. We next discuss how these changes can be extended to applications that do not satisfy our application assumptions.

6.3 Thick-Client Architectures

In our analysis, we have analyzed the response times of user commands that interact with the shared state, that is, those commands which the user-interface sends to the program component. In general, the user-interface may not forward all commands to the program component. The commands that the user-interface does not forward to the program component interact with only the local state contained in the user-interface. These commands

have been referred to as “syntactic sugar” commands [27]. Syntactic sugar commands may or may not be computationally heavy. When an application supports computationally heavy syntactic sugar commands, its user-interface has been described as “thick” [42]. On the other hand, when an application does not support computationally heavy syntactic sugar commands, its user-interface has been described as “thin” [42].

Our analysis has focused on applications with thin user-interfaces, also known as thin-client architectures [42]. As mentioned before, such applications are used frequently in practical collaboration scenarios. They include distributed PowerPoint, a collaborative checkers game, and instant messaging. On the other hand, many popular applications, such as massively multiplayer online games [42], have thick user-interface components. Therefore, it is important to consider thick-client applications, as well. After all, syntactic sugar commands should have good local response times.

The thin-client analysis does not directly apply to thick-client applications. The reason is that computationally-heavy syntactic-sugar commands require a lot of CPU time. As a result, they will either delay the time at which the CPU processes a shared command if the CPU performs them first or delay the time at which the CPU completes processing a shared command if the CPU processes it concurrently with them. Either way, the time at which the processing of a shared command is completed is increased, which increases the response times of the command. Moreover, the syntactic-sugar command will also increase the time the CPU requires to queue a command for transmission. The combination of these effects will lead to increases in response times of the shared commands. To accurately predict the resulting increase in response times requires, a model of syntactic-sugar commands is needed. Unfortunately, the model is application-dependent. Thus, a separate

model is needed for each application. Developing these models for all applications is beyond the scope of our work.

Our analysis can, actually, indirectly account for computationally-heavy syntactic sugar commands as the cost of executing them will be reflected in the cost of processing the shared commands. To illustrate, consider a thick-client application scenario. Suppose that the users have similar computers, which in turn have similar network connections. Suppose also, that one of these users is illustrating something to the other users, and that in the process of illustrating, the user is generating both shared and syntactic-sugar commands. Because the syntactic-sugar commands increase the time the CPU requires to process and transmit the shared commands, the processing and transmission costs of shared commands reported by the illustrating user are going to be higher than those reported by the observing users. Therefore, it will appear to our system that one of the user's computers is slower than all of the other users' computers. Therefore, indirectly, our analysis and system can account for thick-client architectures.

The issue arising from the differences between thin-client and thick-client architectures is an architectural issue. Next, we consider issues raised by collaborative functionality.

6.4 Collaborative Functionality

A collaborative application can offer a variety of collaborative functions to the users. Two functions are mandatory: the processing and distribution of user commands. Our analysis of response times of user commands has focused on these mandatory tasks. In general, an application can provide additional optional collaborative functions, such as

concurrency control, consistency maintenance, awareness, and access control that can impact the response times of the user commands.

6.4.1 Concurrency Control

When multiple users enter commands, it is possible that their commands are inconsistent, that is, their commands conflict. The main goal of concurrency control mechanisms is to prevent the users from entering inconsistent commands. As described in the related work chapter, concurrency control mechanisms are pessimistic or optimistic by nature. As described in the related work section, both pessimistic and optimistic concurrency control schemes require additional commands to be transmitted and processed in order to coordinate the prevention of inconsistent commands. Future work is needed to not only model these tasks but also study the scheduling policies for them.

Interestingly, Greenberg and Marwood [43] observe that the amount of conflicts that happen under optimistic concurrency control mechanisms is a function of remote response times. More specifically, the lower the remote response times, the sooner a user becomes aware of other users' actions, and thus makes fewer conflicting actions. Therefore, our analytical model can be used to choose system configuration that provides the lowest remote response times among inputting users (as opposed to observing users), and thus reduce potential conflicts. A reduction in conflicts would reduce the use of conflict recovery mechanisms. In turn, this enables the lazy scheduling policy to further delay processing commands, thus improving remote response times even more in a self-feeding cycle. This cycle calls for reducing remote response times as much as possible, even if the users cannot notice the reduction. Of course, if the users notice that there are fewer conflicts, then it means that they noticed the effect of the response time reduction!

Greenberg and Marwood [43] also conclude that if users cannot notice the degradation in local response times caused by pessimistic approaches, then these approaches should be used since their optimistic counter-parts are considerably more difficult to implement. This notion of using pessimistic mechanisms if users cannot notice the delay is similar to the notion of delaying the processing task in our lazy policy if the users cannot notice the delay. It would be interesting to combine the idea of lazy scheduling with concurrency control. Merging these ideas can create a new pessimistic-optimistic hybrid concurrency control scheme, which works as follows. When a user enters a command, the scheme behaves pessimistically at first. If the time required to check for conflicts is longer than the local response time degradation threshold, however, the scheme turns optimistic and uses conflict resolution mechanisms to handle any conflicts that occur. If, on the other hand, the time required for conflict checking is less than the local response time threshold, the scheme remains pessimistic. Such a scheme would be useful because the user would never notice the delays during the pessimistic phase, yet conflicts detected during the phase can be handled easily by rejecting the user's command.

6.4.2 Consistency Maintenance

So far, we have looked at the case when concurrent user commands conflict. In many cases, simultaneous commands do not conflict. Nevertheless, they may still lead to state inconsistencies if they are not executed in the same order by each user's computer. That is not to say they always do. For example, when commands commute, then state inconsistencies cannot arise. Regardless of the order in which the commands are executed, the final state is the same. For example, in a distributed PowerPoint presentation, if the presenter enters two commands to advance the presentation, the commands commute.

Moreover, when simultaneous commands lead to state inconsistencies, the users may not care [43]. For example, in an instant-messaging application, when a user enters the message, it is immediately displayed in the local message history. Therefore, if multiple users enter messages at the same time, then their histories may be different because, in general, it takes some time for a message to go from one user's computer to another. But even though the message histories are inconsistent, the users do not seem to care – perhaps because the history is not important to them or they can deduce the correct message order by "eye-balling" the history.

When the commands do not commute and the users care about having a consistent view of the workspaces, then consistency maintenance mechanisms need to be used. The mechanisms that are used in an application depend on the consistency requirements. The strongest consistency requirement is that all computers execute all commands in the same order. This is accomplished using atomic broadcast [25]. Atomic broadcast delays the execution of a received command until the broadcast ensures that the order in which the command is executed with respect to other commands on the receiving computer is the same as the order in which it is executed on all other computers. This delay increases response times. As we stated in the introduction, we assume that when an idle computer receives a command, it immediately begins performing the tasks for that command. Therefore, the assumption circumvents the delay in performing these tasks that are necessary to ensure atomic consistency.

A less stringent consistency requirement is that causality is preserved [58]. The causal relationship is defined through a "happens-before" function. It effectively states that when a command entered locally is executed, every other command that has been executed by the

computer happened-before the command. To preserve causality, whenever a different computer receives the command, it cannot execute the command until it executes all of the commands that "happened-before." When a "happened-before" relationship does not exist between two commands, the commands are said to be concurrent. Concurrent commands occur, for example, when two users enter a command at the same time. There is no delay in processing concurrent commands when they arrive. Nevertheless, the delay required to ensure that the order of execution of non-concurrent commands satisfies the happened-before relationship can increase response times.

When concurrent commands are executed in different orders on different computers, they can lead to state inconsistencies. One solution is to transform these operations in some special way so that even though they are executed in different orders on the users' machines, the resulting state is the same. Such transformations are possible only when some knowledge of the semantics of the commands is available. For example, in multi-user editor applications, a promising solution is operation transformations [85]. Another solution is to undo commands executed out of order and redo them in the same order on all devices [70]. Alternatively, state can be rolled-back to some consistent state on all devices and then commands can be repeated on all devices in the same order form that state. Regardless, of the mechanisms, the extra computation costs increase the processing costs of commands, and hence can increase response times.

All published implementations of the replicated architecture assume that the program component does not implement atomic broadcast to ensure good response times [27], requiring instead causal consistency and relying on some application-specific scheme to do consistent real-time merging of concurrent input. Therefore, it is important that we consider

causal consistency maintenance in the future. When we extend our work in this direction, we have to adjust the analysis only for the delays of non-concurrent commands. In particular, our analysis already accounts for the case when consistency is maintained using operation transformation mechanisms. The reason is that operation transformation algorithms simply increase the processing costs of commands and nothing else. We have not, however, measured these costs. A recent paper by Shao et al. [78] has measured them in asynchronous mobile scenarios. They provide an algorithm which is able to transform as many as 3000 text-editing operations in less than 1.5 seconds on a mobile tablet device with a 900Mhz processor. While no costs for transforming a single operation are reported, it appears that the average time required for on operation transformation is around 0.5ms. Since this is two orders of magnitude less than our noticeable threshold value of 50ms, it seems that all of our results should hold when operation transformations are used.

So far, we have only considered the impact on response times of consistency mechanisms. An interesting observation is that improving response times can improve the effectiveness of some of these mechanisms. For instance, Fletcher et al. [34] have shown that dead-reckoning algorithms can perform better when remote response times are low. There are no thresholds in this case. The response times simply need to be as low as possible. Thus, if dead-reckoning and other optimistic concurrency control mechanisms are used in a collaborative system, it is important to improve remote response times as much as possible.

6.4.3 Access Control

Access control mechanisms ensure that a user cannot execute unauthorized commands. In many respects, they are similar to concurrency control mechanisms. They can be optimistic or pessimistic. Optimistic mechanisms allow an action to proceed and later

revoke it if it turns out the action was not authorized. Pessimistic mechanisms do not allow an action to proceed until it is first verified that the action is authorized. Like locking concurrency control mechanisms, access control mechanisms must contact a manager, only instead of contacting a lock manager, they contact the access manager. Therefore, from the point of view of our analysis, they are not supported because they can delay actions and have their own processing and transmission tasks.

6.4.4 Awareness

Not all collaborative functions are correctness driven. One such functionality is provided a user with awareness of other users' actions. Awareness mechanisms can be roughly classified by whether or not they must be explicitly invoked by the user to convey information to other users.

Awareness commands explicitly invoked by the users are effectively shared commands. The reason is that the action of invoking the mechanism can be considered a shared action. Therefore, our analysis and results account for them. An example of an awareness mechanism that requires implicit action from a user to convey information about that user is a telepointer. A telepointer enables a user to point at objects on a remote user's screen. Since a user must explicitly invoke a telepointer to perform a telepointing operation on a remote user's screen, telepointing commands are effectively shared commands.

Awareness mechanisms that are not explicitly invoked by the user provide awareness implicitly based on shared commands. As a result, they create additional commands for updating awareness information on other users' screens. Examples of such awareness mechanisms are radar views [7][49][75][81], fish-eye views [44][75], and multi-user scrollbars [7][49]. Our analysis can account for these additional commands as long as the

awareness commands are piggy-backed on the shared commands. In this case, the transmission costs measured by our system would increase. Moreover, the processing of the commands on the remote machines would also increase because they would include the cost of processing both commands. However, our analysis does not account for all implementations of awareness mechanisms. For example, some mechanisms such as multi-user scrollbars transmit commands even when no shared commands are transmitted. A multi-user scrollbar uses a separate scrollbar elevator to indicate the position of each collaborator in the workspace. If one user is simply scrolling or reading the workspace without making modifications, awareness commands will be sent to update the multi-user scrollbar on other users' machines even though the user did not manipulate the shared state. The impact of these awareness commands on the local and remote response times is not captured by our analysis. Of course, if the scrollbar commands are considered shared commands, than our analysis applies. In this case, the shared scrollbar commands would have different processing costs on the local and remote computers.

6.5 Immediate Applicability of Results

This dissertation shows that under certain conditions, (a) the choice of processing architecture, communication architecture, and scheduling policy can noticeably impact performance and (b) the self-optimizing system that automates their maintenance can have clear performance benefits over traditional groupware systems. The thesis clearly shows how these conditions can occur in some practical scenarios, such as instant messaging, collaborative board games, and distributed presentations. However, it does not show whether

they occur in actual scenarios. Thus the question that remains is whether the proposed system should be recommended as an alternative to software engineers.

6.5.1 Self-Optimizing System Complexity

The simple answer is that the self-optimizing system is a realistic alternative to existing frameworks, but that it cannot be applied blindly as it involves a series of trade-offs and unknowns. At the very least, the system is more complex than traditional client-server systems. As with any complex program, the complexity can have serious implications in terms of software development. For instance, bugs and deployment issues are likely in initial iterations while maintenance is an issue in the long term. Ultimately, to recommend the system to software developers, the cost of this added complexity must be worth the benefits. Therefore, if the community finds system performance to be a significant interaction roadblock, then the risk is worth it. On the other hand, if the current systems offer sufficiently good performance, then the risk is not worth it.

6.5.2 Application Complexity

Another software development issue is that to fully utilize the system, applications need to support both centralized and replicated semantics. However, a large number of groupware applications are constructed as centralized systems, such as NetMeeting Application Sharing, Live Meeting, Webex Application Sharing, VNC, and Google Wave. Their preponderance is likely due to their simplicity: there are many known issues with replicated architectures in certain application settings. Thus, gaining performance benefits of dynamically switching between centralized and replicated architectures demands a greater sophistication in terms of how the program collaborative applications are constructed. As

was the case with the framework development issues, this implies a trade-off. In cases where performance benefits are warranted, this extra work is worth it. Indeed, many commercial systems, such as NetMeeting Whiteboard, Grove PowerPoint, Webex PowerPoint, and GroupKit [72] are constructed on a replicated architecture, and Google's Docs and Spreadsheet uses a semi-replicated architecture. Thus, in general, replicated semantics are not a barrier to the creation of collaborative applications. Hence, they should not be a barrier to using the self-optimizing system. Moreover, even if an application is locked into centralized or replicated semantics, the system proposed in this thesis can still improve its performance by automating the maintenance of the communication architecture and the scheduling policy.

6.5.3 Networking Issues

An important question that is not related to the cost-benefit analysis of software development complexities is whether the analysis presented in this thesis ignores certain important network related factors. The typical set of network factors in any distributed application consists of bandwidth, network latency, jitter, packet loss, and firewall traversal.

Network Latency and Bandwidth: In the analytical model presented in the thesis, network latency is a parameter. And while bandwidth is not an explicit parameter, it is captured by the network card transmission times. Therefore, network latency and bandwidth are accounted for.

Jitter: The model does not account for jitter because there predictive model has been presented that can predict frequency, duration, and magnitude of jitter. However, there are ways of reducing its on performance. To illustrate, consider a telepointer action. Stuckel and Gutwin [83] have shown that it is possible to handle jitter on the receiver side using a dead-reckoning algorithm. These algorithms typically introduce

more commands that the receiver must process. The model presented in this thesis does not directly account for the impact of these additional commands on processing costs. But as mentioned above, it can indirectly measure the impact.

Packet Loss: Packet loss is an issue when using non-reliable transmission protocols such as UDP. The self-optimizing system uses TCP communication, and therefore, packet loss is not an issue. Communication costs arising from TCP's guarantees are accounted for in the network latency.

Firewall Traversal: The model does not directly account for the impact of firewalls on the direct communication and communication costs. The costs of firewall traversal, however, are indirectly accounted for in the network latency. Moreover, if a firewall blocks two computers from communicating directly, the system assigns a value of "infinity" to the network latency between the computers. It passes this value to the model, which effectively means that the link is not going to be used.

6.5.4 Windows of Opportunity

Finally, the dissertation targets the window of opportunity. As mentioned in the introduction, this window includes the cases in which the network and processor resources are somewhat stressed, specifically the cases between high stress (in which performance is likely to always be poor) and low stress (in which performance is likely to always be good). The question that has not been answered by this thesis is whether a window of opportunity exists in all scenarios. As mentioned in the introduction, windows of opportunity exist in other fields, such as multimedia networking. Therefore, it can be expected that windows of opportunity also exists in collaborative systems. However, further analysis of network and

processor bottlenecks during real collaborative sessions is required to determine whether they occur in actual collaborative sessions.

6.6 Future Applicability of Results

While currently windows of opportunity can be expected, a related issue is whether they will exist in the future. In particular, processing powers, network speeds, and the number of cores on a typical computer are constantly increases. Therefore, it may seem that eventually, the choice of processing architecture, communication architecture, and scheduling policy may not matter. However, an analysis of the past and current application trends revels that this is not likely going to be the case.

6.6.1 Increasing Processing Powers

One argument against the existence of window of opportunity in the future is that because processing powers are constantly increasing, eventually, processing powers will be insignificant, and therefore, the choice of processing architecture will not matter. While this argument may be true, we actually believe that it is wrong. In particular, processing powers have constantly been increasing since the invention of the first computer. And yet today, with multi-core desktops, even text-editing operations in Microsoft Word have noticeable response times. Some even have multi-second response times. The fact is that as processing powers have increased, the consumers have demanded more complex applications. These complex applications have increasing processing costs which largely offset the increases in processing powers. Since the demand for more complex applications is not likely to stop, processing costs will remain an issue, and therefore, the processing architecture will remain a response time factor.

6.6.2 Increasing Network Speeds

Another way to debate the existence of windows of opportunity in the future is from a networking perspective. In particular, networking speeds are constantly increasing. Therefore, communication costs will eventually become insignificant, which means that the choice of communication architecture will not matter. However, like the processing power argument, this network speed argument does not consider the trends in applications and hardware. For instance, the demand for more complex applications implies not only increasing processing costs but also increasing communication costs. To illustrate, consider the latest generation of videoconferencing applications, commonly referred to as telepresence systems. The main selling point of these applications is that they stream high-definition video. These streams are sufficient to saturate many network links. Therefore, communication costs will remain an issue. Moreover, cellular phone network speeds have not been increasing at a breakneck pace. One important reason is that the hardware needed to utilize higher speed links uses too much power. Therefore, until battery life of mobile devices improves, higher cellular network speeds may not be used. Given these arguments, communication costs are likely to stay significant in the future, which means that the communication architecture will remain a response time factor.

6.6.3 Increasing Numbers of Cores

The existence of the window of opportunity in the future can also be argued against from the multi-core processor perspective. In particular, the thesis shows that when multiple-cores are available for scheduling, then the parallel multi-core policy should always be used. Since multi-core desktops are becoming pervasive, it appears that the choice of scheduling policy will not matter. However, there are at least two important trends predicting that a

single-core may be all that is available to carry out processing and transmission tasks. For instance, most, if not all, consumer mobile devices today are single-core. An important reason that they are not multi core is that multi-core processors are generally more power hungry than single-core processors. Given that battery life is an issue with these devices, it makes sense that they are still single core. Another important reason is that more complex applications may occupy most of the cores, perhaps leaving a single-core for the collaborative tasks. In particular, in state of the art telepresence systems, rendering high definition videos of just two remote participants results in 100% utilization of a high-end dual-core and a near 100% utilization of a high-end quad-core desktop. Moreover, as users typically multi-task, some of the cores may have to be dedicated to their private applications, again perhaps leaving only a single core for the collaborative application. Overall, these arguments imply that scheduling policy will remain a response time factor.

6.7 Summary

In this chapter, we have shown how our system changes the processing architecture, communication architecture, and scheduling policy in a single optimization step. Moreover, we have discussed the extent to which our analysis and results apply to scenarios outside of the scope we outlined in the introduction. Some extensions beyond our scope are straightforward, such as those for supporting awareness and operation transformations. On the other hand, supporting concurrency control, access control, and general consistency mechanisms require significant further analysis. Finally, we have presented an initial cost-benefit analysis of using the self-optimizing system as an alternative to existing groupware systems. The analysis shows that if the performance of current systems is an issue, then the

added complexity of the self-optimizing system is worth it. Moreover, it shows both that the benefits are possible in actual scenarios, both today and in the future.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This thesis makes contributions at a number of levels. At the highest level, it is the first work to identify and exploit the window of opportunity for improving performance in collaborative systems. The window of opportunity is an important concept as it relies on algorithmic ways of improving performance rather than brute-force monetary investments in hardware and network systems. The thesis proves that, when there is a window of opportunity,

THESIS

For certain classes of applications, it is possible to meet performance criteria better than existing systems through a new a collaborative framework without requiring hardware, network, or user-interface changes.

The next highest-level contribution of the thesis is identifying that multicast and scheduling policies are important response-time factors in collaborative applications. These factors compliment an earlier result by Chung [21] who found that the processing architecture also impacts response times. The importance of the new factors was shown 1) theoretically, through examples, 2) analytically, through mathematical equations, and 3) practically, through simulations and experiments. In particular, these results show that

SUB-THESIS II

The communication architecture impacts response times.

SUB-THESIS IV

The scheduling policy used for executing the processing and transmission tasks impacts response times.

The identification of multicast and scheduling policies as response time parameters resulted in three innovations. First, the identification of multicast as a response time parameter required an update to the traditional models of collaborative systems because they did not support multicast. The bi-architecture model of collaborative systems, which supports multicast, overcomes this problem. Second, the identification of scheduling as a response-time parameter led to discovery of two scheduling policies, one for single-core and another for multi-core systems. For single-core systems, a new policy, the lazy policy, is identified. The policy, which marries results from human psychology and computer science, makes the traditional process-first policy obsolete. Moreover, the thesis results have shown that on single-core systems, the lazy policy and the other two traditional policies, transmit-first and concurrent, do not dominate each other and hence all of them must be supported. The parallel policy is presented for multi-core systems. When the processing task is (is not) parallelizable, it schedules the transmission task on one core and the processing task on a different core (multiple different cores). The policy, somewhat counter-intuitively, never uses more than one core for the transmission task. It does so for two reasons, both of which are described in chapter four: when multiple cores perform the transmission task 1) the remote response times

become unpredictable, and 2) there is no remote response time benefit as the network card is the bottleneck which serializes the transmission. The benefits of the parallel policy were shown theoretically and experimentally.

The next highest-level contribution of the thesis is an analytical model that can predict the performance of a collaborative application for any combination of values for the processing architecture, communication architecture, and scheduling policy. Like analytical models in other computer science fields, it increases our understanding of the subject analyzed. In the case of collaboration architectures, it helps us better understand and compare the event flow and performance of the centralized and replicated architectures with and without multicast. Moreover, it provides guidance for users with varying degrees of choice regarding the combination of the processing architecture, communication architecture, and scheduling policy in the collaboration systems they use. It can be used by (a) users of adaptive systems to decide when to reconfigure the systems, (b) users who have a choice of systems with different configurations to choose the system most suited for a particular collaboration mode (defined by the values of the collaboration parameters), and (c) users locked into a specific configuration to decide how to change the hardware and other collaboration parameters to improve performance. When one or two of the processing architecture, communication architecture, and scheduling policy is fixed, the model can help guide the users to choose values that optimize response times for the performance parameters that are not fixed.

Relying on the users to make the configuration decisions, however, is problematic for several reasons. The main reasons are that the users can make errors as they apply the model, forget to apply the model, or choose not to apply it because is tedious to do so at the start and

during each collaborative session. Therefore, the next contribution of this thesis is the self-optimizing system that applies the model to automatically select and deploy the processing architecture, communication architecture, and scheduling policy combination that best meets the user-response time requirements. The system can select these values at start time or switch them dynamically at runtime. Moreover, as the number of sessions in which it is used increases, the accuracy of its response time predictions improves, and therefore, it can better meet the performance requirements. The thesis has shown through experiments that the system can indeed calculate the configuration that best meets the requirements and switch to that configuration both at start and at run time. In particular, the results show that

SUB-THESIS I

It is possible to develop a system that automatically switches to the processing architecture that satisfies any user-specified response time criteria better than existing approaches.

SUB-THESIS III

It is possible to develop a system that automatically switches to the communication architecture that satisfies any user-specified response time criteria better than existing approaches.

SUB-THESIS V

It is possible to develop a system that automatically switches to the scheduling policy that satisfies any user-specified response time criteria better than existing systems.

The fact that the self-optimizing system employs the analytical model to make optimization decisions proves that

SUB-THESIS VI

It is possible to develop a model that analytically evaluates the impact on response times of different processing architectures, communication architectures, and scheduling policies to the degree necessary to automate their maintenance.

The next contribution of this thesis is the response time simulator. The simulator can calculate the expected response times for any combination of the processing architectures, communication architectures, and scheduling policies. As such, it is the first NS-like system for collaborative systems. It is not exactly like the NS system because it does not provide users with a UI. Nevertheless, like the NS system, it gives textual output that users can use as input into their own analytical programs or represent visually through graphs, which leads to our final contribution.

The simulator and the self-optimizing system can both be used as teaching tools in collaborative systems courses. The simulator can be used to give students a quick idea of advantages and disadvantages of choose certain configurations. The system can be used to let the users experience the effect on response times of various configurations.

This work suggests several new directions for research. One important direction is to evaluate the benefit of the self-optimizing system through user studies. Previous work has shown that the response time benefits of using our system should be noticeable to users. However, our system raises several other questions. For example, our system allows users to specify the response time requirements through total order functions. While we can speculate

some useful requirements, no work has investigated what the users would actually specify. In particular, what kind of total order functions do users prefer? Also, in our experience, users stop using a collaborative system when performance is poor. Would they continue to suffer poor performance if they know that eventually our system may improve it? In addition, recall that our system can temporarily degrade performance when it changes two or more of the processing architecture, communication architecture, and scheduling policy in a single optimization step. Are users willing to experience temporarily degraded performance for better future performance?

In addition to motivating new user studies, this work motivates new algorithm research. For example, an interesting future direction is to create collaboration specific multicast algorithms. As mentioned in the thesis, we use the HMDM algorithm because it is the only one that considers application-level communication costs, which can be significant in collaborative applications. However, HMDM does not consider all collaboration parameters, such as processing costs and scheduling policies, when creating the multicast tree and hence can actually hurt performance. It will be useful to create the first multicast scheme that considers all of the parameters in our analytical model in order to take the next step in improving response times.

It would also be interesting to create a multicast scheme that not only improves response times but also minimizes the variation in response times. As mentioned in the related work chapter, this idea can be achieved by combining the aspects of the HMDM [13] multicast scheme that reduces response times and the aspects of the Chains [9] scheme which minimizes end-to-end delay variations.

Another attractive algorithm to investigate is a prefetching scheme that does not impact the response times of the shared commands. As mentioned in the related work chapter, such a scheme could use blocking communication for the prefetched data and non-blocking communication otherwise. This would allow the system to pause the prefetching of the data and free up the network card for immediate transmission of the shared commands.

It is also attractive to evaluate the benefit of using different scheduling policies on different users' devices. For example, it would be interesting to evaluate the response time when using the process-first policy on the inputting user's computer, to provide good local response times, and the transmit-first policy on all of the other users' computers, to provide good remote response times. Our system already deploys the parallel policy on multi-core computers regardless of what scheduling policy it deploys on single-core computers is used. Therefore, the system is capable of deploying different scheduling policies on different machines; what it needs is an updated analytical model that supports such scheduling schemes.

Moreover, it would be useful to combine aspects of the lazy policy with concurrency control mechanisms as described in the discussion chapter. The combination would allow the creation of a new hybrid pessimistic-optimistic scheme that reduces the response time degradations caused by concurrency control mechanisms.

Another important future work direction is to extend our work to other applications. More specifically, it would be useful to investigate analytical models and extensions to our self-optimizing system for thick-client applications and applications that use concurrency control, consistency maintenance, awareness, or access control mechanisms. One important thick-client application to consider is Second Life. Second Life has been used for various

kinds of collaborations, including classroom lectures, group meetings, and presentations. Unfortunately, it does not support large collaborations because of performance issues. While the processing architecture is fixed to centralized, the communication architecture and scheduling policies can be changed. Hence, it would be interesting to evaluate how the scalability of Second Life improves when the self-optimizing framework is improving its performance.

Also, it would be useful to investigate applications in which users enter consecutive and simultaneous commands. Recording and then replying logs from these applications would allow the verification of our model and system for such commands.

Yet another interesting direction is building a cluster of computers that can be used for large scale collaboration experiments, not only by us and our system, but by other researchers as well. Such a cluster would provide more fine grained control over what its resources than is provided by Planet Lab and Amazon's CE2.

It would also be useful to further investigate several aspects of our system. For example, it would be useful to incorporate Chung's solution for changing processing architectures without pausing inputs into our system. Moreover, it would be interesting to evaluate the optimal values of some of the system parameters. One of these parameters is the length of the period that should elapse before the system calculates if a new configuration would benefit response times. Another is the weights that are assigned to the historical and recent performance measurements in the cost estimation calculations. We used arbitrary values that seemed to work well in our experiments. However, more optimal values could exist. It would also be useful to investigate solutions that reduce the periods of sub-optimal

configurations that can occur when our system switches communication architectures and scheduling policies.

Finally, an important future work is to further evaluate the parallel multi-core policy and investigate how our theory differs from what happens during experiments. While many devices in use today, such as netbooks, smart-phones, and even many desktop computers and laptops still have processors with a single-core, we can expect all of them to migrate to multi-core processors. Desktops and laptops are already doing so. Thus, further investigation of multi-core scheduling policies is required.

APPENDIX A

COLLECTING DATA FOR SIMULATIONS AND EXPERIMENTS

1.1 Overview

The purpose of this appendix is to summarize our data collection approach for experiments and simulations. We present the results of our simulations and experiments as needed in chapters 3, 4, and 5. These chapters are carefully organized to make the proof of our thesis tractable. Each chapter builds on the results from the previous chapter. Similarly, when we collected the data for experiments and simulations for each chapter, the data set for one chapter subsumed the data from previous chapters. Hence, in this appendix, we describe our data collection for the set of experiments and simulations presented in chapter 5. As a result, parts of the discussion presented here is not relevant for chapters 3 and 4. For instance, transmission times, which are relevant in chapters 4 and 5, are not relevant for chapter 3. Nevertheless, we present all of our data collection at once to minimize repetition. As such, this appendix is meant to be used for lookup of information referenced in each chapter.

1.2 Gathering Performance Parameter Values

In general, to evaluate the performance of a system, one must first identify the parameters relevant to performance. We refer to these as performance parameters. Ideally, the parameters must be assigned values that reflect reality.

Our analytical model has provided us with the parameters relevant to the performance: the input and output processing and transmission times for a command on each user's computer; think times of users' commands; the network latencies between the

collaborators' computers; the number of collaborators; the type of computer that each collaborator uses. We must also consider a second set of parameters. These are the parameters needed by HMDM to construct the multicast overlay. We refer to these as overlay parameters. The overlay parameters must first be assigned values to construct the multicast tree, and then the performance parameters must be assigned values to evaluate the overlay under various conditions.

We next describe how we assigned values to these parameters. The first two parameters, the costs and think times, depend on the user's actions. As a result, we call these parameters user parameters. The remaining parameters are hardware and network related, so we call them system parameters. Following this, we present our approach to assigning values to the parameters specified by our model. Then, we present our approach for assigning values of the system parameters. Finally, we discuss how we assigned values to the overlay parameters.

1.3 Gathering User Parameters

To assign values to user parameters, we must first gather users' actions. Thus, we first present our approach for collecting user actions.

1.3.1 Logs of Users' Actions

Several approaches could be used to gather user actions.

- Live interaction: Under this approach, pairs of users would perform a collaborative task multiple times as the architecture and system parameters are varied in a controlled manner each time.

- Actual logs: Another approach is to use logs of actual collaborations and assume that these are independent of the system parameters such as architecture, machines used, and network delays. These logs can then be replayed under different values of system parameters.
- Synthetic logs: With this approach, the user logs can be created by varying the user parameters using some mathematical distribution such as Poisson's.

Since users cannot be relied upon to perform the same sequence of actions and have the same think times in different collaborative sessions, the live interaction approach is impractical. The other two approaches require a large number of logs to ensure that a wide range of values for user parameters are covered. This is not a problem for synthetic logs, but such logs do not address the practicality concern as it is not clear parameter values based on mathematical distributions represent reality. Logs of actual interaction are not provided in any public database and we were unsuccessful in obtaining them from researchers who we knew had logged their collaboration tasks. Thus to use the actual-log approach, we would have to gather a large number of actual logs ourselves, which is beyond the scope of our work: the analytical model is our primary contribution and the experiments are addressed mainly to validate the model. In other fields such as real-time systems where benchmarks are not widely available, it is customary to resort to the synthetic-log approach to validate new theoretical results. We did a little better by using a hybrid of the synthetic and actual log approaches. We recorded a small number of actual logs to obtain realistic values of some user parameters and then used these values to create a large number of synthetic logs that we then replayed in the actual experiments using different architectures and system parameters.

One issue with creating many synthetic logs from several actual logs approach is that the parameter values obtained from the recorded collaborations may not be representative of the values of these parameters in other collaborations. In general, applications used in synchronous collaborations involving shared access to a distributed object can be divided into four categories: 1) ***logic-centric***, which process computationally expensive input commands; 2) ***data-centric***, which distribute large amounts of data; 3) ***logic-and-data-centric***, which both process computationally expensive input commands and distribute large amounts of data; and 4) ***stateless***, which do neither.

We reduce the problem of log generality by analyzing collaborations involving applications belonging to three of these categories: a Checkers game, which is logic-centric; PowerPoint, which is data-centric; and a chat application, which is stateless. The checkers game fostered collaboration rather the competition: multiple users formed a team that played against the application, which used a computationally expensive algorithm to calculate its next move. The algorithm optimized the computer's move by analyzing scenarios five moves ahead. The users used an audio channel and a telepointer to determine their next move. Any of the users could then make the actual move. This application was created by extending an existing single-user checkers program.

We analyzed recordings of two PowerPoint presentations that were given by one presenter to thirty and sixty audience members, respectively. In addition, we recorded two chat-room sessions consisting of eighty participants of which as many as eight posted messages. Finally, we recorded a collaborative checkers game in which the team consisted of two users. Thus, in these applications, not only did the nature of the application engine vary (logic-centric, data-centric, stateless) but also the number of actors and observers. None of

the applications had concurrency control – the PowerPoint and chat applications did not require such control and users of the checker's program used social protocol to decide who made a move.

These recordings contain actual data and users' actions – PowerPoint commands and slides, checkers moves, and chat messages. The checkers engine used in the actual tasks was transformed into a collaborative program using an infrastructure that has facilities for logging and replaying commands. Therefore, extracting input and output commands from the generated checker logs was relatively simple. The chat programs we logged were the ones implemented by the chat rooms we observed. We ran them under Microsoft Live Meeting 2005 and used its screen-recording capabilities. As a result, we had to use a tedious manual process to extract the input command messages in the sessions – analyzing one ten-minute recording required two hours of work! In addition to being three qualitatively different applications, IM, Checkers, and PowerPoint turned out to be a good choice of applications for which to analyze actual logs for two reasons: 1) the parameters values we measured in these logs were fairly wide spread, and 2) they represent the kind of tasks users do on a daily basis.

1.3.2 Processing and Transmission Costs

To obtain the transmission time parameter values, we replayed these logs using a Java-based infrastructure that has facilities for logging and replaying commands. The checkers program used in the actual tasks was already transformed into a collaborative program using this infrastructure. Hence, the checker logs were replayed directly to the program used in the actual task. To replay the chat commands, we used the replay-supporting infrastructure to create our own version of the chat application. To replay the PowerPoint

commands, we had to bridge the gap between our Java-based replay-supporting infrastructure and the PowerPoint application. We used the J-Integra library to create this bridge and relay the replayed commands to the PowerPoint application. This library required us to assume that slave computers could directly access file systems because the information about a slide contained in an output command from the master had to be saved in a file before it could be displayed by the slave.

We measured the transmission times for a P3 500MHz laptop, a P4 1.7GHz desktop, a Core2 2.0 GHz desktop, and a 1.6 GHz Atom netbook. All the computers were running Windows XP except the netbook, which was running Windows 7. The P3 laptop and the P4 desktop are used to simulate next generation smart-phones and netbooks, respectively. We recorded the average amortized input and output command transmission times of each machine for checkers, PowerPoint, and chat applications. We removed any "outlier" entries from the average calculation, caused for instance, by operating system process scheduling issues. To reduce these issues, we removed as many active processes on each system as possible. Ideally, while we replay the recordings, we should run a set of applications users typically execute on their systems. However, the typical working set of applications is not publicly available so we would have to guess which applications to run. For fear of incorrectly affecting transmission times by running random applications, we used a working set of size zero, a common assumption in experiments comparing alternatives. In order to control network-related variability, we ran our experiments on our local 100Mbit LAN. In addition, we assumed that the data and users' actions in the logs are independent of the number of collaborators, the processing powers of the collaborators' computers, and network latencies.

While the process of obtaining transmission times was fairly complicated, it did have a nice side effect that it provided the values of the processing parameters. In particular, during the experiments, we measured not only the transmission times of the input and output commands for each computer, but also the processing times of these commands. As for transmission times, we recorded the average amortized input and output command processing times of each computer for checkers, PowerPoint, and chat applications.

1.4 System Parameters

As mentioned above, the system parameters are the network latencies between the users' machines, the types of users' machines, and the number of these machines. We next describe how we assign the values to these parameters.

1.4.1 Network Latencies

Based on pings done on two different LANs, we use 0ms to simulate half the round-trip time between two computers on the same LAN in both small-scale and large-scale experiments and situations. Moreover, for small-scale experiments, we used additional latencies based on Chung's and Dewan's experiments. We added 72, 162 and 370 ms to the LAN delays to estimate half the round-trip time from a U.S. East Coast LAN-connected computer to a German LAN-connected computer, German modem-connected computer, and Indian LAN-connected computer, respectively. In addition, for large-scale experiments, we use publicly available network latencies measured among 1740 computers distributed around the world [85] to simulate latencies between two computers on different LANs.

1.4.2 Number of Users

As mentioned above, in the collaboration recordings that we analyzed, the number of users ranged from as few as two in Checkers, between thirty and sixty in PowerPoint, and as many as eighty in the chat application. Unfortunately, this is not a wide enough range of values; in particular, the maximum value of the parameter needs to be much bigger to be representative of large collaborations, such as a company-wide PowerPoint presentation. Therefore, we chose synthetic but not unrealistic values for the number of observers. As observers do not input commands, they do not influence the logs. Moreover, the talks we observed had tight time constraints which did not allow questions. Thus, they were independent of the number of observers. We used the following as the number of collaborators in a session: 2, 25, 50, 100, 200, 300, 400, and 500.

1.4.3 Types of Users' Computers

We are not unaware of any public data about and our logs did not record the distribution of processing powers of the collaborators' computers during a collaborative session. Therefore, we randomly assigned the type of computer of each user to be a P3 desktop, P4 desktop, Core2 desktop, or a netbook.

1.5 Overlay Parameters

It turns out that Brosh and Shavitt use number of collaborators, network latencies, and the transmission times as parameters for constructing the HMDM overlay. Therefore, all of the HMDM-scheme overlay parameters are covered by the above parameters.

REFERENCES

1. Abdelkhalek, A., Bilas, A., and Moshovos, A. Behavior and performance of interactive multi-player game servers. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* 2001. pp: 137-146.

2. Abdelkhalek, A. and Bilas, A. Parallelization and performance of interactive multi-player game servers. *IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 2004. pp: 72-81.

3. Agarwalla, B., Ahmed, N., Hilley, D., and Ramachandran, U. Streamline: scheduling streaming applications in a wide area ntwork. *Transactions on Multimedia Systems 2007*. Vol. 13 (1). Sep 2007. pp: 69-85.

4. Ahuja, S., Ensor, J.R., Lucco, S.E. A Comparison of Application Sharing Mechanisms in Real-time Desktop Conferencing Systems. *ACM Conference on Office Information Systems (OIS)*. 1990. pp: 238-248.

5. Amazon Elastic Compute Cloud EC2. http://aws.amazon.com/ec2/. Jan 15, 2010.

6. Anderson, J.H., Ramamurthy, S., and Jeffay, K. Real-time computing with lock-free shared objects. *IEEE Real-Time Systems Symposium (RTSS)*. 1995. pp: 28-37.

7. Baecker, R.M, Nastos, D., Posner, I.R., and Mawby, K.L. The user-centered iterative design of collaborative writing software. *Conference on Human Factors in Computing Systems (INTERCHI).* 1993. pp: 399-405.

8. Banerjee, S., Bhattacharjee, B., and Kommareddy, C. Scalable application layer multicast. *ACM Conference on Application, Technologies, Architectures, and Protocols for Computer Communications*. 2002. pp: 205-217.

9. Banik, S.M., Radhakrishnan, S., and Sekharan, C.N. Multicast routing with delay and delay variation constraints for collaborative applications on overlay networks. *IEEE Transactions on Parallel and Distributed Systems*. Vol. 18 (3). March 2007. pp: 421-431.

10. Benford, S. A spatial model of interaction in large virtual environments. *European Conference on Computer Supported Cooperative Work (ECSCW).* 1993. pp: 109-124.

11. Bier, E.A. and Freeman, S. MMM: a user interface architecture for shared editors on a single screen. *ACM Symposium on User Interface Software and Technology (UIST).* 1991. pp: 79-86.

12. Block, A. Multiprocessor adaptive real-time systems. Ph.D. Dissertation. University of North Carolina at Chapel Hill. 2008.

13. Brosh, E. and Yuval, S. Approximation and heuristic algorithms for minimum-delay application-layer multicast trees. *IEEE Conference on Computer and Communication Societies (INFOCOM)*. 2004. pp: 2697-2707.

14. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons Ltd.. 1996.

15. Calvary, G., Coutaz, J., and Nigay, L. From single-user architectural design to PAC*: a generic software architecture model for CSCW. *ACM Conference on Human Factors in Computing Systems (CHI)*. 1997. pp: 242-249.

16. Carlsson, C., and Hagsand, O. DIVE a multi-user virtual reality system. *IEEE Virtual Reality Annual International Symposium (VRAIS)*. 1993. pp: 394-400.

17. Capin, T.K., Pandzic, I.S., Thalmann, N.M., and Thalmann, D. A dead-reckoning algorithm for virtual human figures. *IEEE Virtual Reality Annual International Symposium (VRAIS)*. 1997. pp: 161-169.

18. Castro, M., Druschel, P., Kermarrec, A., Nandi, A., Rowstron, A., and Singh, A. SplitStream: high-bandwidth multicast in cooperative environments. *ACM Symposium on Operating System Principles (SOSP)*. 2003. pp: 298-313.

19. Chawathe, Y., McCanne, S., and Brewer, E.A. RMX: reliable multicast for heterogeneous networks. *IEEE Conference on Computer and Communication Societies (INFOCOM)*. 2000. pp: 795-804.

20. Chung, G. and Dewan P. Flexible Support for Application-Sharing Architecture. *European Conference on Computer Supported Cooperative Work (ECSCW)*. 2001. pp: 99-118.

21. Chung, G. Log-based collaboration infrastructure. Ph.D. dissertation. University of North Carolina at Chapel Hill. 2002.

22. Chung, G. and Dewan, P. Towards dynamic collaboration architectures. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 2004. pp: 1-10.

23. Danskin, J. and Hanrahan, P. Profiling the X Protocol. *ACM Conference on Measurement and Modeling of Computer Systems*. 1994. pp: 272-273.

24. DataBeam. T.120 Framework. http://www.dtic.mil/ieb_cctwg/contrib-docs/T.120/T.120-WP.html. Nov 17, 2009.

25. Defago, X., Schiper, A., and Urban, P. Total order broadcast and multicast algorithms: taxonomy and survey. *ACM Computing Surveys (CSUR)*. Vol. 36 (4). Dec 2004.

26. Dewan, P. and Choudhary, R. A high-level and flexible framework for implementing multiuser user interfaces. *ACM Transactions on Information Systems (TOIS)*. Vol. 10 (4). Oct 1992.

27. Dewan, P. Architectures for collaborative applications. *Trends in Software: Computer Supported Co-operative Work* (edited by: M. Beaudouin-Lafon). 1999. pp: 165-194.

28. Diot, C., Levine, B., Lyles, J., Kassem, H., and Balensiefen, D. Deployment issues for the IP multicast service and architecture. *IEEE Network*. Vol. 14 (1). Jan-Feb 2000. pp: 78-88.

29. Dourish, P. and Bellotti, V. Awareness and coordination in shared workspaces. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 1992. pp: 107-114.

30. Droms, R. and Dyksen, W. Performance Measures of the X Window System Communication Protocol. *Software – Practice and Experience (SPE)*. Vol. 20 (S2). Oct 1990. pp: 119-136.

31. Dyck, J., Gutwin, C. Subramanian, S., Fedak, C. High-performance Telepointers. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 2004. pp: 172-181.

32. Ellis, C.A. and Gibbs, S.J. Concurrency control in groupware systems. *ACM SIGMOD Record*. Vol. 18 (2). Jun 1989. pp: 399-407.

33. Ellis, C.A, Gibbs, S.J., and Rein, G. Groupware: some issues and experiences. ACM *Communications of the ACM (CACM)*. Vol. 34 (1). Jan 1991. pp: 39-58.

34. Fletcher, R.D.S, Graham, T.C.N., and Wolfe, C. Plug-replaceable consistency maintenance for multiplayer games. *ACM Workshop on Network and System Support for Games (NetGames)*. 2006. Article 34.

35. Floyd, S., Jacobson, V., Liu, C. G., McCanne, S., and Zhang, L. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*. Vol. 5 (6). Dec 1997. pp: 784-803.

36. Foster, G. and Stefik, M. Cognoter: theory and practice of a collaborative tool. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 1986. pp: 7-15.

37. Funaoka, K., Kato, S., and Yamasaki, N. Work-conserving optimal real-time scheduling on multiprocessors. *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*. 2008. pp: 13-22.

38. Glinka, F., Plob, A., Muller-Iden, J., and Gorlatch, S. RTF: a real-time framework for developing scalable multiplayer online games. *ACM Workshop on Network and System Support for Games (NetGames)*. 2007. pp: 81-86.

39. Google Wave. http://wave.google.com. Dec 26, 2009.

40. Graham, T.C.N. and Urnes, T. Linguistic Support for the Evolutionary Design of Software Architectures. *IEEE Conference on Software Engineering (SE)*. 1996. pp: 418-427.

41. Graham, T.C.N., Morton, C.A., and Urnes, T. ClockWorks: visual programming of component-based software architectures. *Journal of Visual Languages and Computing*. Vol. 7 (2). Jun 1996. pp: 175-196.

42. Graham, T.C.N., Phillips, W.G., and Wolfe, C. Quality Analysis of Distribution Architectures for Synchronous Groupware. *Conference on Collaborative Computing: Networking, Applications, and Worksharing (CollaborateCom)*. 2006. pp: 1-9.

43. Greenberg, S. and Marwood, D. Real time groupware as a distributed system: concurrency control and its effect on the interface. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 1994. pp: 207-217.

44. Greenberg, S., Gutwin, C., and Cockburn, A. Awareness through fisheye views in relaxed-WYSIWIS groupware. *Conference on Graphics Interface (GI)*. 1996. pp: 28-38.

45. Greenhalgh, C. and Benford, S. MASSIVE: a virtual reality system for teleconferencing. *ACM Transactions on Computer-Human Interaction (TOCHI)*. Vol. 2 (3). Sep 1995. pp: 239-261.

46. Greenhalgh, C., Benford, S., and Reynard, G. A QoS architecture for collaborative virtual environments. *ACM Conference on Multimedia (MM)*. 1999. pp: 121-130.

47. Greif, I., Seliger, R., and Weihl, W. Atomic Data Abstractions in a Distributed Collaborative Editing System. *Symposium on Principles of Programming Languages*. 1986. pp: 160-172.

48. Gutwin, C., Roseman, M., and Greenberg, S. A Usability study of awareness widgets in a shared workspace groupware system. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 1996. pp: 258-267.

49. Gutwin, C., Greenberg, S., and Roseman, M. Workspace awareness support with radar views. *ACM Conference on Human Factors in Computing Systems (CHI)*. 1996. pp: 210-211.

50. Gutwin, C., Dyck, J., and Burkitt, J. Using Cursor Prediction to Smooth Telepointer Actions. *ACM Conference on Supporting Group Work (GROUP)*. 2003. pp: 294-301.

51. Gutwin, C., Fedak, C., Watson, M., Dyck, J., and Bell, T. Improving network efficiency in real-time groupware with general message compression. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 2006. pp: 119-128.

52. Hill, R.D. The abstraction-link-view paradigm: using constraints to connect user interfaces to applications. *ACM Conference on Human Factors in Computing Systems (CHI)*. 1992. pp: 335-342.

53. id Software. Quake Home Page. http://www.idsoftware.com/games/quake/quake/.

54. Jay, C., Glencross, M., and Hubbold, R. Modeling the Effects of Delayed Haptic and Visual Feedback in a Collaborative Virtual Environment. *ACM Transactions on Computer-Human Interaction (TOCHI)*. Vol. 14 (2). Aug 2007. Article 8.

55. Jeffay, K. Issues in Multimedia Delivery Over Today's Internet. *IEEE Conference on Multimedia Systems. Tutorial*. 1998.

56. Kostic, D., Rodriguez, A., Albrecht, J., and Vahdat, A. Bullet: High bandwidth data dissemination using an overlay mesh. *ACM Symposium on Operating System Principles (SOSP)*. 2003. pp: 282-297.

57. Kushner, D. Engineering EverQuest. *IEEE Spectrum Magazine*. July 2005.

58. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM (CACM)*. Vol. 21 (7). Jul 1978. pp: 558-565.

59. Laurillau, Y. and Nigay, L. Clover architecture for groupware. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 2002. pp: 236-245.

60. Lu, F., Parkin, S., and Morgan, G. Load balancing for massively multiplayer online games. *ACM Workshop on Network and System Support for Games (NetGames)*. 2006. Article 1.

61. Microsoft. http://www.microsoft.com/games/pc/halo2.aspx. Jan 13, 2010.

62. Morse, K.L., Bic, L., and Dillencourt, M. Interest management in large-scale virtual environments. *Presence: Teleoperators and Virtual Environments*. Vol. 9 (1). Feb 2000. pp: 52-68.

63. NEC 2000. Windows 2000 Terminal Services Capacity and Scaling. *NEC*. 2000.

64. Nieh, J., Yang, S. and Novik, N. A Comparison of Thin Client Computing Architectures. *Technical Report CUCS-022-00 Columbia University*. Nov 2000.

65. NTP. http://en.wikipedia.org/wiki/Network_Time_Protocol. Jan 2, 2010.

66. Ostrowski, K. and Birman, K. Implementing high performance multicast in a managed environment. *Technical Report QSM-TR2007-2087 Cornell University*. 2007.

67. p2pSim: a simulator for peer-to-peer protocols. http://pdos.csail.mit.edu/p2psim/kingdata. Mar 4, 2009.

68. Phillips, W.G. and Graham, T.C.N. Architectures for synchronous groupware. *Technical Report 1999-425 Queen's University*. 1999.

69. PlanetLab. http://www.planet-lab.org/. Jan 15, 2010.

70. Prakash, A. and Knister, M.J. Undoing actions in collaborative work. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 1992. pp: 273-280.

71. Richardson, T., Stafford-Fraser, Q., Wood, K., and Hopper, A. Virtual Network Computing. *IEEE Internet Computing*. Vol. 2 (1). Jan-Feb 1998. pp: 33-38.

72. Roseman, M. and Greenberg, S. GroupKit: a groupware toolkit for building real-time conferencing applications. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 1992. pp: 43-50.

73. Roseman, M. and Greenberg, S. Building flexible groupware through open protocols. *ACM Conference on Organizational Computing Systems*. 2003. pp: 279-288.

74. Roseman, M. and Greenberg, S. Building real-time groupware with GroupKit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction (TOCHI)*. Vol. 3 (1). Mar 1996. pp: 66-106.

75. Schafer, W.A. and Bowman, D.A. A comparison of traditional and fish-eye radar view techniques for spatial collaboration. *Conference on Graphics Interface (GI)*. 2003. pp: 23-46.

76. Scholtes, I., Botev, J., Esch, M., Schloss, H., and Sturm, P. Minimizing load delays in P2P distributed virtual environments using epidemic hoarding. *Conference on Collaborative Computing: Networking, Applications, and Worksharing (CollaborateCom)*. 2008. pp: 578-593.

77. Seshasayee, B., Nathuji, R., and Schwan, K. Energy-aware mobile service overlays: cooperative dynamic power management in distributed mobile systems. *IEEE Conference on Autonomic Computing (ICAC)*. 2007. pp: 6.

78. Shao, B., Li, D., and Gu, N. A sequence transformation algorithm for supporting cooperative work on mobile devices. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 2010. pp: 159-168.

79. Shneiderman, B. Response time and display rate in human performance with computers. *ACM Computing Surveys (CSUR)*. Vol. 16 (3). Sep 1984. pp: 265-285.

80. Shneiderman, B. *Designing the user interface: strategies for effective human-computer interaction*. 3rd ed. Addison Wesley. 1997.

81. Smith, R.B., O'Shea, T., O'Malley, C., Scanlon, E., and Taylor, J. Preliminary experiments with a distributed, multi-media, problem solving experiment. *European Conference on Computer Supported Cooperative Work (ECSCW)*. 1989. pp: 19-34.

82. Steinmetz, R. Human perception of jitter and media synchronization. *IEEE Journal on Selected Areas in Communications*. Vol. 14 (1). Jan 1996. pp: 61-72.

83. Stuckel, D., and Gutwin, C. The Effects of Local-lag on Tightly-Coupled Interaction in Distributed Groupware. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 2008. pp: 447-456.

84. Sun, J. and Liu, J. Synchronization protocols in distributed real-time systems. In *Proc. IEEE Conference on Distributed Computing Systems (ICDCS)*. 1996. pp: 38.

85. Sun, C. and Ellis, C. Operational transformation in real-time group editors: issues, algorithms, and achievements. *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 1998. pp: 59-68.

86. Tatar, D.G., Foster, G. and Bobrow, D.G. Design for conversation: Lessons from Cognoter. *Journal of Man-Machine Studies*. Vol. 34 (2). Feb 1991. pp: 185-209.

87. Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J., and Robbins, J.E. A component- and message-based architectural style for GUI software. *ACM Conference on Software Engineering (SE)*. 1995. pp: 295-304.

88. Valve. http://www.valvesoftware.com/games.html. Jan 13, 2010.

89. West, R., Schwan, K., and Poellabauer, C. Dynamic window-constrained scheduling of real-time streams in media servers. *IEEE Transactions on Computers*. Vol. 53 (6). Jun 2004. pp: 744-759.

90. Wikipedia.org. http://en.wikipedia.org/wiki/Bootstrapping_node. Dec 27, 2009.

91. WireShark. http://www.wireshark.org/. Jan 2, 2010.

92. Wolfe, C. Conceptual programming models of distributed systems. *Technical Report 2006-525 Queen's University*. 2006.

93. Wolfe, C., Graham, T.C.N., Phillips, W.G., and Roy, B. Fiia: user-centered development of adaptive groupware systems. *ACM Symposium on Interactive Computing Systems*. 2009. pp: 275-284.

94. Wong, A. and Seltzer, M. Evaluating Windows NT Terminal Server Performance. *USENIX Windows NT Symposium*. 1999. pp: 15-15.

95. Youmans, D.M. User requirements for future office workstations with emphasis on preferred response times. *IBM United Kingdom Laboratories*. Sep 1981.

96. Zhang, B., Ng, T.S.E, Nandi, A., Riedi, R., Druschel, P., and Wang, G. Measurement-based analysis, modeling, and synthesis of the internet delay space. *ACM Conference on Internet Measurement*. 2006. pp: 85-98.