The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems*

Kevin Jeffay
University of North Carolina at Chapel Hill

Abstract: A concurrent programming system for constructing hard-real-time applications is described. The system is based on a novel semantics of inter-process communication called the *real-time producer/consumer* (*RTP/C*) *paradigm*. Process interactions are modeled as producer/ consumer systems with a timing constraint on the rate at which the consumer must service the producer. The RTP/C paradigm provides a framework both for expressing processor-time-dependent computations and for reasoning about the real-time behavior of programs. A formal model of processor and resource allocation is used to determine necessary and sufficient conditions for realizing the RTP/C semantics of a program. The design of an interactive graphics system illustrates the use of the system.

Introduction

Real-time computer systems are loosely defined as the class of computer systems that must perform computations and I/O operations in a time-frame defined by processes in the environment external to the computer. Real-time systems differ from more traditional multiprogrammed systems in that real-time systems have a dual notion of correctness. In addition to being logically correct, *i.e.*, producing the correct outputs, real-time systems must also be temporally correct, *i.e.*, produce the correct output at the correct time. This paper describes a programming system for the construction and analysis of real-time systems. The initial focus is on the problem of designing and constructing *hard-real-time* systems. Hard-real-time systems are real-time systems that require deterministic guarantees of temporal correctness. These are systems in which the cost of failing to interact with the external environment in real-time is high. Cost can be measured in monetary terms (*e.g.*, inefficient use of raw materials in a process control system), aesthetic terms (*e.g.*, unrealistic output from a computer music or computer animation system), or possibly in human or environmental terms (*e.g.*, an accident due to untimely control in a nuclear power plant or fly-by-wire avionics system).

* Supported by a grant from the National Science Foundation (number CCR-9110938).

A programming language for hard-real-time systems must provide a means for specifying the real-time response requirements of processes as well as a means for predicting the real-time behavior of a program. We describe a concurrent programming system with a novel semantics of inter-process communication that captures the real-time response requirements of processes. The semantics, called *the real-time producer/consumer* (*RTP/C*) *paradigm*, models process interactions as producer/consumer systems with a timing constraint on the rate at which the consumer must service the producer. The RTP/C paradigm provides a framework both for expressing processor-time-dependent computations and for reasoning about the real-time behavior of programs.

Introducing response times into the semantics of a programming language, makes it possible to write programs that cannot be implemented. For a given implementation of the programming language there may not exist sufficient processing resources to ensure that the response time requirements of a program will be met in all cases. Whether or not the implementation can realize the real-time semantics of a program will depend on factors such as the rate(s) at which events are generated in the external environment, the resources required to respond to an event (*e.g.*, buffers), and the cost (in terms of required execution time) of algorithms employed in a program. A RTP/C program is modeled formally as a set of sporadic tasks that share a set of serially reusable single unit resources. The conditions under which it will be possible to correctly implement an arbitrary RTP/C program have been reported in [8]. This allows a programmer to easily test if the desired real-time behavior of a program will be realized at run-time.

The RPT/C paradigm is not a panacea for the problems of real-time computing. However, our thesis is that the RTP/C paradigm applies to a wide variety of interesting and important real-time applications where real-time data-flow is paramount — applications wherein *all* data arriving from sensors and devices must be input *and processed* in real-time. We provide evidence by way of example that the RTP/C paradigm yields a flexible methodology for constructing, and predicting the performance of, data-flow applications and hence represents an advance over existing methods such as cyclic executives [3]. To date, the RTP/C paradigm has been used as the basis for the design and construction of three real-time applications: an interactive 3-D graphics system, a HiPPI data-link interface, and a distributed workstation-based conferencing system using digital audio and video [7]. In each case the real-time processing requirements of the application were concisely represented using the RTP/C paradigm. The design of the

In: Proc. of the 1993 ACM/SIGAPP Symposium on Applied Computing, Indianapolis, IN, February, 1993, pp.796-804.

graphics system will be presented in the following section as an example of the use of the RTP/C paradigm.

The following section describes the RTP/C paradigm and presents a message passing programming discipline for hard-real-time systems. Next, we briefly discuss the formal implementation model and describe the design of a run-time system that efficiently implements the RTP/C paradigm. The following section discusses how the real-time behavior of RTP/C programs may be analyzed. Lastly, we discuss the use of the RTP/C paradigm and some of the implications of adding real-time communications primitives to a programming language. We conclude with a discussion of some related work.

The RTP/C Paradigm

A Semantics of Interprocess Communication

In traditional multiprogramming systems, a standard paradigm of process interaction is the *producer/consumer* paradigm. The crux of a producer/consumer systems is to synchronize a producer and consumer so that no data is lost, *i.e.*, so that all data objects produced by the producer are ultimately consumed by the consumer. For example, a canonical solution is to impose a circularly linked list of buffers between the two processes and synchronize access to the buffers by a monitor and pair of condition variables as outlined (in an abbreviated form) in Figure 2.1 [1]. In this scenario the producer will wait for the consumer when all the buffers are full. Similarly, the consumer will wait for the producer if the buffers are all empty. In this manner, all data objects deposited by the producer are eventually removed and consumed by the consumer. The semantics of the synchronization primitives, and a description of the algorithms used in the deposit and remove routines, are sufficient for reaching this conclusion. In particular, the correctness of the monitor-based implementation will not be dependent on the number of data objects that the producer produces, the number of buffers present in the monitor, or on the relative speeds at which the processes make progress.

```
process Producer
  loop
    < produce data >
    Buffer.Deposit( data)
  end loop
end Producer


process Consumer
  loop
    data := Buffer.Remove
    < consume data >
  end loop
end Consumer
```

```
monitor Buffer
  var full, empty : boolean
  var notEmpty,
      notFull : condition

  procedure Deposit( d : int)
    if (full) then
      wait( notFull)
      endif
  < fill a buffer >
    signal( notEmpty)
  end Deposit

  function Remove : int
    if (empty) then
      wait( notEmpty)
    endif
      < empty a buffer >
  signal( notFull)
  end Remove
end Buffer
```

Figure 2.1

In real-time systems software processes interact with processes in the external world. Abstractly, we can model these interactions as in the producer/consumer scenario with one fundamental modification. In Figure 2.1, the producer was forced to wait for the consumer if the consumer ever lagged too far behind. If the producer is a process external to the computer then, as the external process may not be under the control of the computer, it may not be possible to force the process to wait for the consumer. In this case we must ensure that all data produced by the producer is consumed by the consumer in the absence of the *wait* statement in the Deposit routine in the monitor in Figure 2.1 [14]. Now, if the consumer is not "fast enough," data from this external producer will be lost. In this case, knowledge of the semantics of synchronization, and the buffering algorithms in the monitor, are no longer sufficient to determine if all data deposited will be consumed. In order to determine the correctness of this system, one will need information on the *implementation* of the consumer and on the *real-time behavior* of the producer process in the external environment. At a minimum we will need to know the *rate* at which the producer produces data objects and how much *time*, in the worst case, it takes the consumer to process a data object. The correctness of this producer/consumer system now has a temporal component. We call such a producer/consumer system a *real-time producer/consumer* (*RTP/C*) *system*.

If we ignore the specifics of the monitor-based implementation of a producer/consumer system, we identify the following salient features of a RTP/C system. Abstractly, there are two processes, a producer and a consumer that are connected via a unidirectional communication channel. The channel is unidirectional, from the producer to the consumer, since the consumer cannot communicate with, or control the producer. We will assume that this communication channel imposes a bounded delay on the flow of units of information from the producer to the consumer. When the producer produces a data object, the producer sends the data object to the consumer on the channel. Let r be the maximum rate at which data objects travel on the channel from the producer to the consumer. If the rate r is realizable over an *arbitrarily long* interval of time and if the consumer does not consume data objects at rate r, then there is *no* amount of buffering that can be imposed between the producer and the consumer that will ensure every data object produced by the producer is consumed by the consumer. To guarantee the correctness of the system under *all* conditions, it is necessary for the consumer to consume data objects *at the rate at which they are produced*. This is the essence of a RTP/C system.

For hard-real-time systems it is appropriate to entertain worst case assumptions about the operation of the system and the behavior of the external environment. We therefore, define rate as the reciprocal of the minimum inter-arrival time of data objects at the consumer. For this definition of rate, the *real-time producer/consumer paradigm* is defined as a paradigm of process interaction wherein *the i^{th} output of the producer must be consumed by*

the consumer before the $(i+1)^{st}$ *output is produced.*

The RTP/C paradigm stipulates that a consumer must process information in a time frame defined by its producer. Abstractly, a producer defines a discrete time domain for a channel. The emission of messages on this channel corresponds to the "ticks" of a discrete time clock. If a pair of interconnected processes adheres to the RTP/C paradigm, then, relative to the channel's discrete time clock, a producer cannot tell the difference between a consumer that is infinitely fast and one that simply obeys the RTP/C paradigm. In this manner the RTP/C paradigm allows one to specify one form of real-time constraints: data-flow constraints wherein *all* data arriving from external sensors must be processed in real-time. An example of a system with such constraints is given below.

A Message Passing Programming System

The RTP/C paradigm forms the semantic basis of a message passing programming discipline. In brief, a program is expressed as a directed graph (called a *process graph*) where vertices represent processes and edges represent unidirectional communication channels. Processes exchange messages along communication channels. Processes may be either sequential programs that execute on a single processor or physical processes in the environment external to the processor that communicate with internal processes via interrupts. In the latter case, processes are simply stubs (*i.e.*, non-executable code) that specify the real-time characteristics of the external processes they represent. A message is typed collection of data. A simplified conceptual schema for an internal process is

```
process P
  loop
    Accept( in_mesg)
    <compute>
    Emit(out_mesg, channel1)
    <compute>
    Emit(out_mesg, channel2)
      :
      :
  end loop
end P1
```

Communication and synchronization in our system are based on the client/server paradigm of message passing [1]. A process has a single input port and a set of output ports. A process repeatedly accepts a message on its input port, processes the message — possibly emitting messages to other processes — and then attempts to accept another input message. Message passing is asynchronous. The Emit statement is always non-blocking while the Accept statement is potentially blocking. If a message is not available for a process that executes an Accept statement then the process is blocked until a message is available.

Each channel in a process graph defines a producer/consumer relationship between two processes. The novel abstraction in our programming system is that all interconnected pairs of processes adhere to the RTP/C paradigm. When a message is sent on a channel it will be received *and processed* before the next message is sent on that channel. For example, if a process *P* acts as a server for another process *P′*, then *P* will consume each message from *P′* before *P′* sends its next message. More precisely, for each output port of a process we define a *transmission rate function*. This function relates the maximum rate at which messages can be produced on an output channel to the rate at which messages arrive at the process. This rate is defined in terms of the worst case minimum interarrival time of messages. For this message passing system it can be shown that the transmission rate functions have the form $f(r) = (1/x)r$, where r is the rate at which messages arrive at the process and x is a positive, non-zero integer. For a given transmission rate function the value of x will depend on the logic within a process. Conceptually, a process is modeled as a finite state machine in which state transitions occur upon the receipt of a message. The parameter x in the transmission rate function is simply the minimum number of state transitions that separate two states in which a message is emitted on the output channel in question. Currently the value of x is specified by a programmer as they develop the code for each process.

Source nodes in the process graph (those with no input edges) represent processes in the external environment (*i.e.*, devices). The output channels of these nodes are labeled with constants (symbolic or numeric) that indicate the maximum message transmission rate on the channel. For example, a rate might indicate the maximum rate with which a particular type of interrupt is expected to arrive. The remaining edges in a process graph can be labeled with a transmission rate in the obvious manner. Starting from the source nodes, the transmission rate functions can be evaluated in topological order and each edge in the process graph can be labeled with an actual transmission rate. This rate will be either a numeric value or an expression containing symbolic constants. The precise semantics of message passing are defined in terms of this rate. If a process accepts messages from a channel with a transmission rate r, then the message must be consumed within $1/r$ time units of its arrival. Since in general a consumer has no knowledge as to when its producer will send its next message, these semantics are required to ensure the RTP/C paradigm is adhered to under all circumstances.

Processes that receive messages from multiple producers are treated as multiple instances of processes that receive messages from a single producer. For example, if a process *P* acts as a server for multiple processes, then *P* will consume a message from each client *P′* before *P′* sends its next message. Logically, if process *P* receives messages from n producers then *P* is logically treated as a set of n identical copies of process *P*. Each copy of *P* receives messages from a single producer and defines a RTP/C relationship with its message producer. The key point is that the RTP/C paradigm applies separately to each communication channel in a process graph. The rate at which a process *P* consumes messages from a producer *A* depends only on the rate at which *A* produces messages; independent of whether or not process *P* receives messages from producers other than *A*. From the point of view of a message producer, the rate at which a process gets serviced depends only on the rate at which *it* emits messages.

Processes that receive messages from multiple producers do, however, introduce additional constraints on the implementation of the RTP/C paradigm. If a process *P* receives messages from multiple producers, then to ensure the consistency of the function that process *P* performs, the consumption of a message by *P* from a producer may not be interrupted by processing of any other message at *P*.

For many applications, the temporal coupling of message senders and receivers is unnecessary and often undesirable. (An example of such an application will be presented in the next section.) Our programming system provides a facility for *non-time constrained communication* based on shared memory. Shared memory is referred to syntactically as a *data repository*. A data repository encapsulates shared data and exports a set of entry routines for accessing and manipulating the data. Processes invoke these routines via a procedure call. Like a monitor, data repositories provide mutually exclusive access to the data they encapsulate. Data repositories are also useful for sending large messages efficiently. Rather than sending a large structure in a message, a programmer may create a data repository to store the contents of such messages. Sending and receiving processes then may (safely) operate on the same copy of the data. In this case the physical message that is sent from a sender to a receiver is simply a synchronization signal. This style of interaction can eliminate the need for copying large amounts of data from a sender to a receiver.

Using the RTP/C Paradigm

The following example illustrates the use of the RTP/C programming system. We have used a prototype of the programming system to re-implement an interactive graphics system used for research in *virtual worlds* [5]. The graphics system is a head-mounted display system consisting of a helmet with miniature television monitors embedded in it, and tracking hardware for the helmet and for a hand-held pointing device. A computer generated image of a 3-dimensional "virtual world" is displayed in the helmet. The goal of the system is to track the user's head and pointing device in real-time and to update the image displayed in the helmet so as to maintain the illusion that the user is immersed in an artificial world. There are two separate real-time concerns in this application. First, the system must update the display at a rate sufficient for ensuring that animate objects displayed in the helmet move in a smooth and realistic manner. An update rate of 30 updates per second is ideal. Second, as the user moves their head or pointing device, the displayed image must appear to move with the user's movements. For example, if the users turns their head to the left, the image must be shifted to the right in concert with the user's movement. A lag of 50ms. was desired.

The process graph for the head-mounted display system — augmented with data repositories — is shown in Figure 2.2. Circles represent processes and double circles represent data repositories. Bold circles represent external processes (devices). The process graph consists of two disconnected components (data repositories are not considered part of the process graph). One component (the processes labeled Tracking HW, Tracker Int. Handler, and Position Processor) reads data from the tracking hardware, the other disconnected component periodically computes and displays the virtual world. Both tracking and display subgraphs are organized as simple pipelines. The two disconnected components in the process graph communicate through data repositories that store the current position of the user's head and hand. The Position Processor process calls a "write" routine in the Head and Hand data repositories to update the current position of the user. The Transform Coordinates process calls a "read" routine in the data repositories to acquire the current position. The semantics of a data repository ensure that read/write operations on the same data repository do not interfere.

Figure 2.2 also shows the transmission rate functions for each channel. With the exception of the channel between the Tracker Interrupt Handler and Position Processor processes, all channels have the identity transmission rate function. This means that in the worst case, every time a process such as the Timer Interrupt Handler process receives a message, it will emit a message on its output channel. The Tracker Interrupt Handler process will always wait for the arrival of at least 10 messages before it will emit a message on its output channel. That is, at least ten messages are received from the tracking hardware before a new position report is generated.

The channels from the external world are labeled with rate constants. For this application, the numeric value of these constants have been determined empirically. Had this not been the case then these channels would have been annotated with a symbolic constant. Given these constants and the transmission rate functions, the maximum rates at which messages will be emitted on each channel can be easily calculated. The maximum rates at which messages

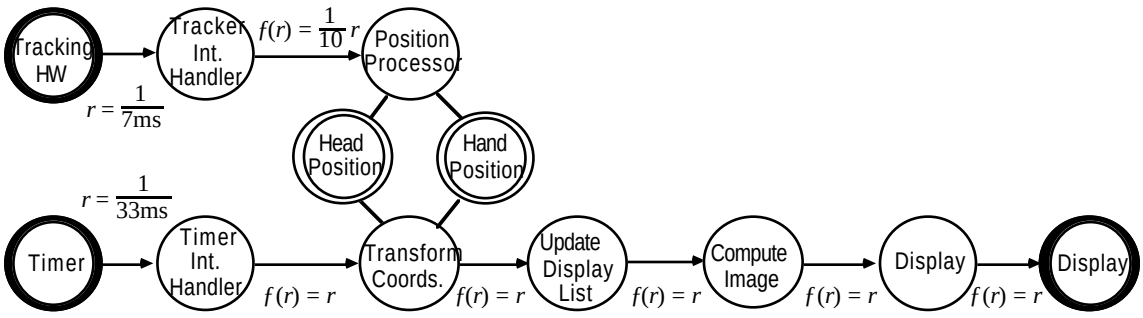

Figure 2.2

will be transmitted between display processes will be one message every 33 ms. Between the two tracking processes, messages will be produced every 70 ms.

The RTP/C paradigm provides a framework for both expressing processor-time-dependent computations and for reasoning about the real-time behavior of programs. For the head-mounted display system, the process graph in Figure 2.2 completely specifies the real-time behavior of the application. At a low-level, the RTP/C paradigm specifies an upper bound on the time to process each message. For example, messages from the Timer Interrupt Handler to the first Display process will be consumed (received and processed) within 33 ms. Messages from the Tracker Interrupt Handler to the Position Processor process will be consumed within 70 ms. At a higher-level, the RTP/C paradigm allows a designer to reason about the time required for a message, or more precisely a sequence of messages, to traverse a particular path through a graph. For example, for a particular implementation of our programming system (described next), we can demonstrate:

- the maximum time between the arrival of a timer interrupt and the generation of the commands to display an image is 33 ms,
- the maximum time between the arrival of a complete head and hand position report and the display of an image based on the new position information is 103 ms.

The derivation of these figures is discussed below. The latter result illustrated a flaw in our initial design. By using data repositories for communication between the tracking and display generation processes, there was no explicit temporal coupling between these processes. This had lead to an unacceptable worst case lag time between the arrival of a position report and the display of an image However, by eliminating the data repositories and restructuring the process graph as a simple pipeline as shown in Figure 2.3 and making minor modifications to the position processor process, we were able — in the space of about 15 minutes — to reduce the second performance guarantee to approximately 35 ms. Previously the tracker interrupt handler process emitted messages containing data for the current head and hand position. The data arrives serially at the tracker process; first the data for the head position arrives followed by the data for the head position. The primary insight was to have the Tracker Interrupt Handler emit messages whenever it received a complete head *or* hand position report. This halved the tracker process's output transmission rate and led to the optimization. While this is was a simple insight, we conjecture that it was made possible by the expressing the of program as a directed graph and the use of the RTP/C paradigm. For example, the original programmer of this system never saw this optimization.

Realizing the RTP/C

Theory

In order to correctly implement a process graph we must guarantee that all interconnected pairs of processes adhere to the RTP/C paradigm. For a given process graph, our ability to realize the RTP/C semantics of inter-process communication will be a function of: (1) the topology of the process graph, (2) the rates at which messages arrive at the system from the external world, (3) the transmission rate functions on communication channels, and (4) the cost (measured in execution time) of processing each message type. The topology of a process graph is a factor only if it contains a cycle. For simplicity, in the following we consider only acyclic process graphs.

Our approach to implementing process graphs is to treat each graph as an instance of a real-time processor scheduling and resource allocation problem. A process graph is modeled as a set of *sporadic* tasks. A sporadic task is a sequential program that is invoked repeatedly, with a lower bound on the inter-invocation time, and with a deadline for the completion of each invocation. A sporadic task is a generalization of the more commonly studied periodic task [11]. Informally, a task corresponds to a process. If a process receives messages from *n* producers then we associate *n* (identical) tasks with the process. Tasks are invoked whenever a message is produced on a channel connected to the process corresponding to the task. An execution of a task corresponds to the execution of the code required to consume a message sent on a channel. For example, the head-mounted display system shown in Figure 2.2 is implemented as 7 tasks; 2 for tracking and 5 for generating and displaying image. When a message is sent from the Tracker Interrupt Handler to the Position Processor process, the task corresponding to the channel is invoked. When the task is scheduled it will execute the code of the Position Processor process and consume the message.

Each task has a deadline for completion of each of its executions. The deadline is used to guarantee that communication adheres to the RTP/C paradigm in all cases. If messages are sent on a channel with (worst case) rate *r*, then the task that will process messages sent on that channel will be required to complete execution within 1/*r* time units of each invocation. For example, the task that is invoked to process messages sent from the Tracker Interrupt Handler to the Position Processor process will have a deadline of 70 ms. In addition to deadlines, tasks may have constraints on their execution due to the presence of critical sections. There are two types of critical sections to consider: operations on data repositories that are shared between processes in a process graph are critical sections, and the execution of tasks that derive from a common process (*i.e.*, tasks derived from a process that receives

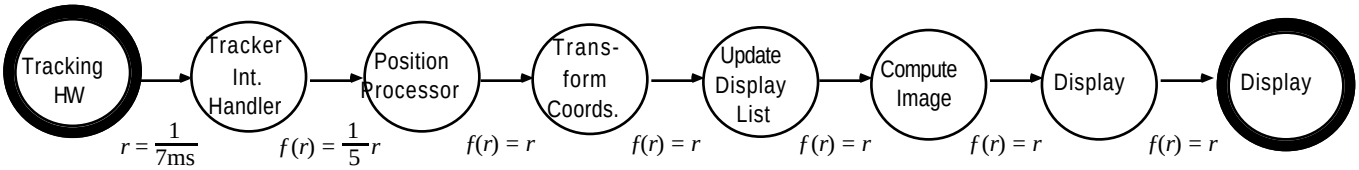

Figure 2.3

messages from multiple producers). For example, since the processing of a message from the Tracker process in Figure 2.2 requires an operation on a shared data repository (a write operation to update the current position), the task that is invoked to process messages sent from the Tracker Interrupt Handler will have critical sections.

We choose to model processes with sporadic tasks as there exists a lower bound on the time between the emission of messages on a channel (but no upper bound). With respect to the sporadic tasking model, we have developed an optimal algorithm for sequencing sporadic tasks on a single processor [8]. The algorithm is optimal in the sense that it can schedule a set of tasks in such a manner that (1) all invocations of all tasks will complete execution before their respective deadlines whenever it is possible to do so and (2) no two tasks ever execute in a critical section at the same time. The algorithm is a variation of the well-known *earliest deadline first* (*EDF*) scheduling algorithm; a preemptive priority driven scheduling algorithm with dynamic priority assignment [11].

Although the scheduling policy used is optimal, it is still quite possible to write a program which, when compiled into a set of tasks, cannot be scheduled (*i.e.*, the tasks cannot be guaranteed to meet their deadlines). We have developed an efficient decision procedure for deciding when a set of sporadic tasks can be scheduled. Let $r_1, r_2, ..., r_n$ be the rates at which message are emitted on channels, sorted in non-increasing order. Let $C_1, C_2, ..., C_n$ be the maximum execution times required to process a message on each channel. The processing of a message sent on channel i may require some number of operations on shared data repositories. Let n_i be this number for channel i and let c_{ij} be the maximum execution time required for the j^{th} data repository operation. Let c_{i0} be the maximum execution time required to execute the non-data repository code (sequential code in the process that consumes message on channel i). Hence $C_i = c_{i0} + c_{i1} + c_{i2} + ... + c_{in_i}$. A set of tasks can be scheduled on a single processor if:[1]

1) $\sum_{i=1}^{n} C_i r_i \leq 1,$

2) $\forall i, 1 < i \leq n; \forall k, 1 \leq k \leq n_i; \forall L, 1/r_1 < L < 1/r_i :$

$$L \geq c_{ik} + \sum_{j=1}^{i-1} \left\lfloor (L-1)\, r_j \right\rfloor C_j ,$$

The product $C_i r_i$ is the fraction of the processor that must be allocated to processing messages on channel i. The first condition stipulates that the processor not be overloaded. Condition (2) applies to tasks that require access to shared data repositories (tasks for which $n_i > 0$). It assesses the contention that may occur when accessing shared data repositories. Loosely speaking, the right hand side of condition (2) condition is an upper bound on the time that a task will be delayed while waiting to gain access to a critical section when scheduled according to an optimal discipline. Under all circumstances this bound must be less than or equal to the inter-arrival time (or a fraction thereof) of messages on channel i. A set of tasks can be tested against these conditions in time $O(n/r_n)$ (*i.e.*, in time proportional to the largest message inter-arrival time times the number of channels).

These results demonstrate that (1) the RTP/C paradigm can, in theory, be realized between sets of processes and (2) one can efficiently determine whether or not a process graph can be implemented on a uniprocessor. These results are applied as follows. A process graph is constructed and annotated with transmission rate functions following the methodology outlined in the previous section. Given a specification of the arrival rates of messages from the external world, the transmission rate functions are solved to yield a set of rate constants. Next the execution time of each process is measured. For a given process the measurement consists of the execution times of all operations on data repositories called by the process as well as the sequential code of the process itself. Measurements are currently done by hand although we anticipate that automated tools will be available to aid in this process (*e.g.*, [12]). The rates and execution times of processes are tested against the schedulability conditions listed above. If the parameters satisfy the conditions then we are guaranteed that (1) all messages can be consumed according to the RTP/C paradigm, and (2) all operations on shared data repositories can be executed in a mutually exclusive manner.

If any of the rate constants are symbolic constants then the schedulability conditions can be used to derive maximum input rates that can be sustained by a process graph (on a given processor). If it is believed that an external process may emit messages at a rate greater than the specified rate then the programmer will have to use a buffering scheme (as described below) to ensure that the ill-behaved device does not saturate the system.

Implementation

The programming system we have described is currently implemented as a set of extensions to the *C* programming language. A run-time system that implements both our process model and scheduling discipline has been constructed [6]. The run-time system is a bare machine operating system kernel (or "micro-kernel") that executes on IBM PS/2 computers. The kernel, contains routines to create processes, data repositories, and communication channels, bind ports of processes to channels, send and receive messages, and invoke operations on data repositories. Processes and data repositories are implemented as *C* functions. When a message is sent to a process the appropriate *C* function is placed on a run-queue to be logically forked (dispatched). Once forked, a process executes to completion. When a process accesses a data repository, the access is performed indirectly through the kernel (to ensure mutual exclusion).

The kernel is functionally similar to those for other message-passing systems. The major distinctions of our kernel are the processor and resource allocation policies used and the implementation of tasks. The scheduling algorithm we

[1] Necessary and sufficient conditions are proved in [11]. A simpler formulation is presented here for brevity.

have developed affords us an extremely efficient implementation of processes. Our variant of the EDF scheduling discipline has the following two useful properties. First, whenever a process *P* enters a critical section, the process has its priority elevated in such a manner that for the duration of the critical section, no other process that requires access to the same critical section will be able to preempt the resident process [8]. This is similar to the concept of a priority ceiling in priority inheritance protocols [13]. Because of this fact, the kernel need not provide any special locking facilities for critical sections. The second property is that if a process *P* is preempted, it is the case that any process that executes while *P* is preempted, is guaranteed to complete execution before *P* is resumed. Since processes execute to completion, we may execute all processes on a single runtime stack. This greatly improves memory utilization and reduces context switching overhead. This is similar to Baker's stack allocation policy [2].

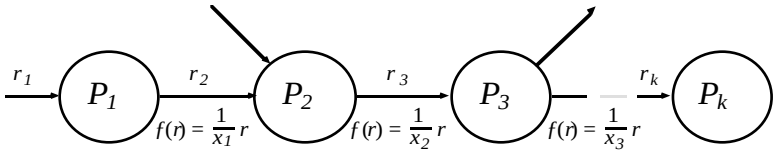

Figure 4.1

Analyzing RTP/C Programs

The RTP/C paradigm enables two types of analysis of real-time behaviors: assessments real-time latency, and real-time throughput. Real-time throughput is inherent in the RTP/C model. Latency refers to the time required for sequences of messages to propagate through a process graph. In order to assess the latency of message propagation, we must assume processes are well-behaved in the sense that they emit messages at some minimal rate. (This is easy to enforce by associating a timer and an "exception" process with each process. The timer and the exception process simply are another pair of processes that adhere to the RTP/C paradigm.) In what follows, we derive upper bounds on propagation latency for the (best) case where all processes emit messages at their maximum rate. The analysis for minimum emission rates is analogous.

We distinguish between two types of latency: direct message propagation delay and indirect message propagation delay. Direct message propagation delay refers to the time required for a message (more precisely a sequence of messages) to propagate from a source node to a sink node in a process graph. For example, in Figure 2.4, the time that may elapse between a timer interrupt and the generation of commands to update the display is a direct message propagation delay. Indirect message propagation delay refers to the time required for information (messages and the effect of data repository operations) to propagate from a source node in one disconnected component of a process graph, to a sink node in a different component in the graph. For example, in Figure 2.4, the time that may elapse between the arrival of the a message at the Tracker Interrupt Handler that completes a position report, and the generation of commands to update the display based on this position report is an indirect message propagation delay. For brevity we consider only the problem of assessing direct message propagation delay.

The determination of direct message propagation time through a sequence of processes depends on the rates at which messages are emitted and the slopes of the transmission rate functions for the channels the messages traverse. Let processes $P_1 ... P_k$ be processes on an acyclic path through a process graph as shown in Figure 4.1. For this path, let r_i be the rate at which the i^{th} process P_i receives messages on the i^{th} channel on the path and let x_i be the denominator of the coefficient of process P_i's transmission rate function for its output channel. Consider process P_1 in Figure 4.1. When a message arrives at P_1, the message must be consumed sometime within $1/r_1$ time units of its arrival. In addition, when process P_1 receives a message, it will wait for the arrival of at most $x_1 - 1$ additional messages before it must emit a message (assuming P_1 emits messages at its maximum rate). Therefore, when process P_1 receives a message, P_1 will delay for at most x_1/r_1 time units before emitting a message. For a sequence of k processes, the maximum delay between the arrival of a message at process P_1, and the emission of a message from process P_k can be at most $\sum_{i=1}^{k} x_i/r_i$ time units. This corresponds to a point in time when processes $P_1 ... P_k$ are "least synchronized." That is, while each process is emitting messages at its maximum rate, each process waits for the largest possible number of message arrivals before emitting a message. A more optimistic scenario occurs when processes $P_1 ... P_k$ are all synchronized. That is, each process emits a message when it receives a message. In this case, the arrival of a message at process P_1 will directly cause a message to be emitted by process P_k. Hence the maximum propagation delay for such a message will be at most $\sum_{i=1}^{k} 1/r_i$ time units. If all processes in the sequence have identity transmission rate functions then the previous two summations are identical. Note that if all processes emit messages at their maximum rate, then in any sequence of $x_1 \cdot x_2 \cdot ... \cdot x_k$ message arrivals at process P_1, there will always be messages whose propagation delay is bounded from above by these summations.

This result is not surprising as it corresponds well with our notion of propagation through a pipeline. Each stage of the pipeline introduces a delay of at least $1/r_i$ time units and at most x_i/r_i time units (again assuming a maximum message emission rate). However, with knowledge of the implementation of the RTP/C (*e.g.*, the fact the EDF scheduling is used) one can substantially improve these bounds. To see how this can be achieved requires a closer look at the scheduling model.

The decision procedure used for deciding whether or not it will be possible to faithfully execute a process graph is based on a formal model of the processing requirements implied by a process graph. While an affirmative output indeed guarantees correct execution, a negative output does not imply the opposite. There are several reasons for this.

The scheduling analysis uses no topological information. As such, the analysis of contention for shared resources assumes that all possible interleavings of message arrivals are possible. By incorporating topological information into the scheduling model it is likely that we may be able to perform a less pessimistic analysis. The primary reason for ignoring the structure of a process graph for analysis purposes is that it allows us to optimize the scheduling of messages transmitted along path in a graph to reduce the overall latency for a sequence of messages to traverse that path.

For example, assume that process *A* sends a message to process *B* and that both processes must consume messages within 1/*r* time units of their arrival (*i.e.*, the input channel to each process has the same transmission rate). One can easily show that processes *A* and *B* can never be consuming messages simultaneously when implemented on a single processor. If the decision procedure indicates that this process graph is schedulable, then processes *A* and *B* will be schedulable even if they were to consume messages simultaneously. At run-time, a scheduler can use the fact that *A* sends messages to *B* to schedule (*i.e.*, insert into the run-queue) *A* and *B* simultaneously whenever *A* receives a message. Process *B* is scheduled even though there is no message for it to consume. If process *A* is given priority over process *B*, process *B* will have a message to consume when it is eventually dispatched. This scheduling technique improves the response time that we can guarantee for a sequence of messages to propagate through processes *A* and *B*. It is this technique that allows us to claim, for example, that in the head mounted display system of Figure 2.2, the maximum time between the arrival of a timer interrupt and the display of an image is 33 ms (and similarly, that the corresponding bound for Figure 2.3 is 35 ms.). In Figure 2.4, when a timer interrupt occurs, *all* display tasks are simultaneously scheduled. If the set of tasks are schedulable then all display tasks are guaranteed to complete execution within 33 ms of each invocation.

This use of topological information at run-time effectively executes a pair (or sequence) of processes (with identical arrival rates) as a single process that contains the concatenation of the code of the processes. In this manner, a programmer may use freely use processes as a structuring mechanism in a program, *i.e.*, to decompose a process into a sequence of processes, without incurring a performance penalty due to lengthening a pipeline.

Discussion and Related Work

The RTP/C paradigm provides a framework both for expressing processor-time-dependent computations and for reasoning about the real-time behavior of programs. In modeling real-time computations as producer/consumer interactions, our emphasis has been on constraining the behavior of the consumer process. That is, we have considered only the problem of performing *input* operations in real-time. It is our thesis that this emphasis is sufficient for specifying time constrained output operations as well. Abstractly, an output constraint specifies that an output operation be performed during a particular interval of real-time. The endpoints of the interval may be specified relative to the occurrence of other events in the system or to events in the external environment. In order to ensure that an output constraint is adhered to, the system must be able to measure the passage of time in the units of time in which the constraint is specified. For example, if a computer music system must generate a note on the sixth beat of a measure, the system must be able to measure the passage of beats. The system need not measure beats directly but it must have available some reference stream of inputs from which it can accurately infer the passage of beats in real-time. A constraint on output can therefore be mapped into a constraint on the processing of the input reference stream.

The RTP/C paradigm requires that a consumer process messages at the rate at which they are produced. Our particular definition of rate in essence requires that producer/consumer systems work correctly with zero buffers. This is not meant to imply that buffers have no utility in a hard-real-time system. We assume that a specification of the minimum inter-arrival time of message from the external world is provided as part of a program. If we cannot control an external process then we surely cannot be guaranteed that the specified minimum inter-arrival time will be respected. If a minimum inter-arrival time is likely not to be respected then inputs can be buffered in a data repository and polled at the desired rate by a process driven by an external timer.

Numerous programming languages and systems have been proposed for the development of real-time systems. Most of these language, most notably languages such as Ada, do not deal with time in any fundamental manner. Notable exceptions include languages such as Real-Time Euclid [9], Concord/FLEX [10], and ESTEREL [4]. Real-Time Euclid is an extension of Concurrent Euclid that adds the ability to specify periodic and event driven timing constraints as well as exception handling mechanisms. FLEX is a language for specifying “imprecise” computations; a computation that is described by a monotonically increasing value function (of time). The more processing time that is allocated to a process, the more “precise” the result it produces. This is well-suited to softer real-time domains than those we consider as it assumes that applications are processing resource limited. FLEX provides a good framework for trading off processing accuracy for real-time response. Signal and ESTEREL are examples of languages from the “synchronous” programming school of thought. Our conceptual framework, specifically the notion that consumers process messages in “no time,” is borrowed from the *strong synchrony hypothesis* of ESTEREL. We view our work as applying the strong synchrony hypothesis to more realistic implementation environments. Our emphasis on scheduling allows us to deal with program artifacts such as critical sections in a more fundamental manner. Unlike languages such as Real-Time Euclid we have an efficient procedure for deciding when programs can be correctly implemented; albeit at expense of a less expressive programming discipline.

Summary

It is our thesis that real-time interactions can be effectively modeled as producer/consumer systems. We have developed the concept of a producer/consumer system in which the consumer is constrained to process information produced by a producer at the rate at which the information is produced. A programming system has been developed that support this paradigm of interaction.

Our definition of rate in terms of inter-arrival time has been motivated by the desire to provide minimal guarantees of response time for sequences of messages. To date, this has been appropriate for the systems we have studied. The preciseness of the RTP/C semantics follows from our desire to understand the cost, in terms of off-line analysis and run-time overhead, of hard-real-time computing. We believe we have been successful in this endeavor. We are currently investigating the impact on the programming and scheduling models of adopting a definition of rate based on aggregate process behavior. Such research is aimed at accommodating the "softer" real-time requirements of systems that cannot be implemented as hard-real-time systems (*e.g.*, because of an insufficiently powerful processor) but will function acceptably nonetheless without hard-real-time guarantees.

Our experience has been that the programming system is expressive enough to capture the desired real-time characteristics of actual systems. Moreover, it provides a framework for the analysis of interesting and important real-time program behaviors. The system has been applied to an interactive graphics system, a HiPPI data-link controller, and is being used in the development of a computer-based conferencing system using digital audio and video [7].

References

[1] Andrews, G.R., Schneider, F.B., *Concepts and Notations for Concurrent Programming*, Computing Surveys, 15, 1, (March 1983), pp. 3-43.

[2] Baker, T.P., *A Stack-Based Resource Allocation Policy for Real-Time Processes*, Proc. IEEE Real-Time Systems Symp., Orlando, FL, December 1990, pp 191-200.

[3] Baker, T.P., Shaw, A.C., *The Cyclic Executive Model and Ada*, Real-Time Systems, 1, 1, (June 1989), pp. 7-26.

[4] Berry, G., Cosserat, L., *The ESTEREL Synchronous Programming Language and its Mathematical Semantics*, Lecture Notes in Computer Science, 197, pp. 389-448.

[5] Chung, J.C., *et al.*, *Exploring Virtual Worlds with Head-Mounted Displays*, Non-Holographic True 3-Dimensional Display Technologies, SPIE Proceedings, Los Angeles, CA, January 1989.

[6] Jeffay, K. *et al.*, *YARTOS: Kernel support for efficient, predictable real-time systems*, in "Real-Time Programming," W. Halang and K. Ramamritham, eds., Pergamon Press, Oxford, UK, 1992.

[7] Jeffay, K., *et al.*, *Kernel Support for Live Digital Audio and Video*, Computer Communications, 15, 6, (July/August 1992) pp. 388-395. .

[8] Jeffay, K., *Scheduling Sporadic Tasks With Shared Resources in Real-Time Systems*, Proc. IEEE Real-Time Sys. Symp., Phoenix, AZ, December 1992, pp. 89-99.

[9] Kligerman, E., Stoyenko, A.D., *Real-Time Euclid: A Language for Reliable Real-Time Systems*, IEEE Trans on Soft. Eng., 12, 9, (September 1986), pp. 941-949.

[10] Lin, K.-J. *et al.*, *Concord: A System of Imprecise Computations*, Proc. of the IEEE COMPSAC '87, Tokyo, Japan, October 1987.

[11] Liu, C.L., Layland, J.W., *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, J. ACM, 20, 1, (January 1973), pp. 46-61.

[12] Park, C., Shaw, A.C., *Experiments With a Program Timing Tool Based On Source-Level Timing Schema*, IEEE Computer, 24, 5, (May 1991), pp. 48-57.

[13] Sha, L. *et al.*, *Priority Inheritance Protocols: An approach to real-time synchronization*, IEEE Trans. on Computers, 39, 9, (September 1990), pp. 1175-1185.

[14] Wirth, N., *Toward a discipline of real-time programming*, Comm. of the ACM, 20, 8 (August 1977), 577-583.