CHAPTER 3

Abstract Source-level Modeling

> ***model**: (11a) a description or analogy used to help visualize something (as an atom) that cannot be directly observed.*
>
> — Merrian–Webster English Dictionary

> *Anything that has real and lasting value is always a gift from within.*
>
> — Franz Kafka (1883–1924)

Abstract source-level modeling provides a method to describe the workload of a TCP connection at the source level in a manner than is not tied to the specifics of individual applications. The starting point of this method is the observation that at the transport level, a TCP endpoint is doing nothing more than sending and receiving data. Each application (*i.e.*, web browsing, file sharing, *etc.*) employs its own set of data units for carrying application-level control messages, files, and other information. The actual meaning of the data is irrelevant to TCP, which is only responsible for delivering data in a reliable, ordered, and congestion-responsive manner. As a consequence, we can describe the workload of TCP in terms of the demands by upper layers of the protocol stack for sending and receiving *Application Data Units* (ADUs). This workload characterization captures only the sizes of the units of data that TCP is responsible for delivering, and abstracts away the details of each application (*e.g.*, the meaning of its ADUs, the size of the socket reads and writes, *etc.*). The approach makes it feasible to model the entire range of TCP workloads, and not just those that derive from a few well-understood applications as is the case today. This provides a way to overcome the inherent scalability problem of application-level modeling.

While the work of a TCP endpoint is to send and receive data units, its lifetime is not only dictated by the time these operations take, but also by *quiet times* in which the TCP connection remains idle, waiting for upper layers to make new demands. TCP is only affected by the duration of these periods of inactivity and not by the cause of these quiet times, which depends on the dynamics of each application

(*e.g.*, waiting for user input, processing a file, *etc.*). Longer lifetimes have an important impact, since the endpoint resources needed to handle TCP state must remain reserved for a longer period of time[1]. Furthermore, the window mechanism in TCP tends to aggregate the data of those ADUs that are sent within a short period of time, reducing the number of segments that have to travel from source to destination. This is only possible when TCP receives a number of back-to-back requests to send data. If these requests are separated by significant quiet times, no aggregation occurs and the data is sent using at least as many segments as ADUs.

We have formalized these ideas into the *a-b-t model*, which describes TCP connections as sets of ADU exchanges and quiet times. The term a-b-t is descriptive of the basic building blocks of this model: *a-type* ADUs (*a*'s), which are sent from the connection initiator to the connection acceptor, *b-type* ADUs (*b*'s), which flow in the opposite direction, and quiet times (*t*'s), during which no data segments are exchanged. We will make use of these terms to describe the source-level behavior of TCP connections throughout this dissertation. The a-b-t model has two different flavors depending on whether ADU interleaving is sequential or concurrent. The *sequential a-b-t model* is used for modeling connections in which only one ADU is being sent from one endpoint to the other at any given point in time. This means that the two endpoints engage in an orderly conversation in which one endpoint will not send a new ADU until it has completely received the previous ADU from the other endpoint. On the contrary, the *concurrent a-b-t model* is used for modeling connections in which both endpoints send and receive ADUs simultaneously.

The a-b-t model not only provides a reasonable description of the workload of TCP at the source-level, but it is also simple enough to be populated from measurement. Control data contained in TCP headers provide enough information to determine the number and sizes of the ADUs in a TCP connection and the durations of the quiet times between these ADUs. This makes it possible to convert an arbitrary trace of segment headers into a set of *a-b-t connection vectors*, in which each vector describes one of the TCP connections in the trace. As long as this process is accurate, this approach provides realistic characterizations of TCP workloads, in the sense that they can be empirically derived from measurements of real Internet links.

In this chapter, we describe the a-b-t model and its two flavors in detail. For each flavor, we first discuss a number of sample connections that illustrate the power of the a-b-t model to describe TCP connections driven by different applications, and point out some limitations of this approach. We then present a set of techniques for analyzing segment headers in order to construct a-b-t connection vectors and provide a validation of these techniques using traces from synthetic applications. We finally examine

[1]Similarly, if resources are allocated along the connection's path, they must be committed for a longer period.

the characteristics of a set of real traces from the point of view of the a-b-t model, providing a source-level view of the workload of TCP.

3.1 The Sequential a-b-t Model

3.1.1 Client/Server Applications

The a-b-t connection vector of a sequential TCP connection is a sequence of one or more *epochs*. Each epoch describes the properties of a pair of ADUs exchanged between the two endpoints. The concept of an epoch arises from the client/server structure of many distributed systems, in which one endpoint acts as a client and the other one as a server. The client sends a request for some service (*e.g.*, performing a computation, retrieving some data, *etc.*) that is followed by a response from the server (*e.g.*, the results of the requested action, a status code, *etc.*). An epoch represents our abstract characterization of a request/response exchange. An epoch is characterized by the size a of the request and the size b of the response.

The HTTP that underlines the World-Wide Web provides a good example of the kinds of TCP workloads created by client/server applications. Figure 1 shows a simple *a-b-t diagram* that represents a TCP connection between a web browser and a web server, which communicate using the HTTP 1.0 application-layer protocol [BLFF96]. In this example, the web browser (client side) initiates a TCP connection to a web server (server side) and sends a request for an object (*e.g.*, HTML source code, an image, *etc.*) specified using a Universal Resource Locator (URL). This request constitutes an ADU of size 341 bytes. The server then responds by sending the requested object in an ADU of size 2,555 bytes. The representation in the figure captures:

- the sequential order of the ADUs within the TCP connection (first the HTTP request then the HTTP response – in this case, order also implies "causality"),

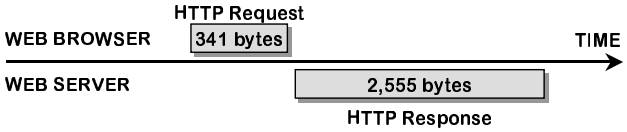

Figure 3.1: An a-b-t diagram representing a typical ADU exchange in HTTP version 1.0.

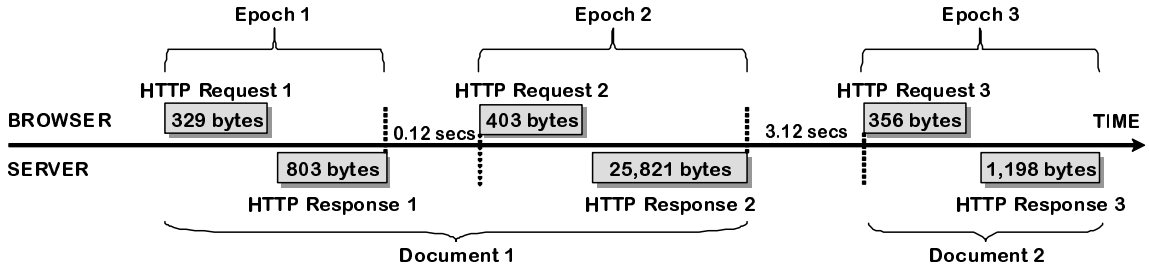

Figure 3.2: An a-b-t diagram illustrating a persistent HTTP connection.

- the direction in which the ADUs flow (above the time line for the ADU sent from the connection initiator to the connection acceptor; below the time line for the ADU sent from the connection acceptor to the connection initiator), and
- the sizes of the ADUs (using annotations and the lengths of the rectangles, which are proportional to the number of bytes).

The diagram provides a visualization in the spirit of abstract source-level modeling, since it does not incorporate any specific information about the actual contents of the ADUs. The bytes in the first ADU (HTTP request) represent an HTTP header that includes a URL, and the bytes in the second ADU (HTTP response) represent an HTTP header (with a success code of `200 OK`) followed by the requested object (*e.g.*, HTML source code). In this example, the purpose of this particular connection was well-understood, and that allowed us to assign labels to the ADUs (HTTP request and response) and to the TCP endpoints (web browser and server). In general, when we examine how the ADUs flow in an arbitrary TCP connection, we do not have this application-specific information (or we can only guess it). The same diagram (without the HTTP-specific labels) could be used to represent different connections with completely different payloads in ADUs of the same size. The diagram does not include any network-level information either, so this diagram could also represent connections with very different maximum segment sizes, round-trip times, and other network properties below the application level. Note that this example, and the following ones, came from real connections that were actually observed. In some cases, we had access to the actual segment payloads and used them to add annotations to the ADUs. In other cases, we used port numbers and our understanding of the protocols to add these annotations.

Some client/server applications use a new connection for each request/response exchange, while other applications reuse a connection for more than one exchange, creating connections with more than one epoch. As long as the application has enough data to send, multi-epoch connections can

improve performance substantially, by avoiding the connection establishment delay and TCP's slow start phase. For example, HTTP was revised to support more than one request/response exchange in the same "persistent" TCP connection [FGM+97]. Figure 3.2 illustrates this type of interaction. This is a connection between a web browser and a web server, in which the browser first requests the source code of an HTML page, and receives it from the web server, just like in Figure 3.1. However, the use of persistent HTTP makes it possible for the browser to send another request using the same connection. Unlike the example in Figure 3.1, this persistent connection remains open after the first object is downloaded, so the browser can send another request without first closing the connection and reopening a new one. In Figure 3.2 the web browser sends three ADUs that specify three different URLs, and the server responds with three ADUs. Each ADU contains an HTTP header that precedes the actual requested object. If the requested object is not available, the ADU may only contain the HTTP header with an error code. Note that the diagram has been annotated with extra application-level information showing that the first two epochs were the result of requesting objects from the same document (*i.e.*, same web page), and the last epoch was the result of requesting a different document.

The diagram in Figure 3.2 includes two time gaps between epochs (represented with dashed lines). In both cases, these are quiet times in the interaction between the two endpoints. We call the time between the end of one epoch and the beginning of the next, the *inter-epoch quiet time*. The first quiet time in the a-b-t diagram represents processing time in the web browser, which parsed the web page it received, retrieved some objects from the local cache, and then made another request for an object in the same document (that was not in the local cache). Because of its longer duration, the second quiet time is most likely due to the time taken by the user to read the web page, and click on one of the links, starting another page download from the same web server.

As will be discussed in Section 3.3, it is difficult to distinguish quiet times caused by application dynamics, which are relevant for a source-level model, and those due to network performance and characteristics, which should not be part of a source-level model (because they are not caused by the behavior of the application). The basic heuristic employed to distinguish between these two cases is the observation that the scale of network events is hardly ever above a few hundred milliseconds[2]. Going back to the example in Figure 3.2, the only quiet time that could be safely assumed to be due to the application (in this case, due to the user) is the one between the second and third epochs. The 120 milliseconds quiet time between the first and second epochs could easily be due to network effects

[2]Some infrequent events, such as routing changes due to link failures, can last several seconds. We generally model large numbers of TCP connections, so the few occasions in which we confuse application quiet times with long network quiet times have no measurable statistical impact when generating network traffic.

(such as having the sending of the second request delayed by Nagle's algorithm [Nag84]), and therefore should not be part of the source-level behavior. Similarly, the two a-b-t diagrams shown so far have not depicted any time between the request and the response inside the same epoch. In general, web servers process requests so quickly that there is no need to incorporate *intra-epoch quiet times* in a model of the workload of a TCP connection. While this is by far the most common case, some applications do have long intra-epoch quiet times, and the a-b-t model can include these.

Formally, a sequential a-b-t connection vector has the form $C_i = (e_1, e_2, \ldots, e_n)$ with $n \geq 1$ epoch tuples. An epoch tuple has the form $e_j = (a_j, ta_j, b_j, tb_j)$ where

- a_j is the size of the j^{th} ADU sent from the connection initiator to the connection acceptor. a_j will also be used to name the jth ADU sent from the initiator to the acceptor.
- b_j is the size of the j^{th} ADU sent in the opposite direction (and generally in response to the request made by a_j).
- ta_j is the duration of the quiet time between the arrival of the last segment of a_j and the departure of the first segment of b_j. ta_j is defined from the point of view of the acceptor (often the server), but ultimately our estimate of the duration is based on the arrival times of segments at some monitoring point.
- tb_j is either the duration of the quiet time between b_j and a_{j+1} (for connections with at least $j+1$ epochs), or the quiet time between the last data segment (*i.e.*, last segment with a payload) in the connection and the first control segment used to terminate the connection.

Note that ta_j is a quiet time as seen from the acceptor side, while tb_j is a quiet time as seen from the initiator side. The idea of these definitions is to capture the network-independent component of quiet times, without being concerned with the specific measurement method. In a persistent HTTP connection, a's would usually be associated to HTTP requests, b's to HTTP responses, ta's to processing times on the web server, and tb's to browser processing times and user think times. We can say that a quiet time ta_j is "caused" by an ADU a_j, and that a quiet time tb_j is caused by an ADU b_j. Both time components are defined as quiet times observed at one of the endpoints, and not at some point in the middle of the network where the packet header tracing takes place.

As mentioned in the introduction, the name of the model comes from the three variable names used in this model, which are used to capture the essential source-level properties: data in the "a" direction, data

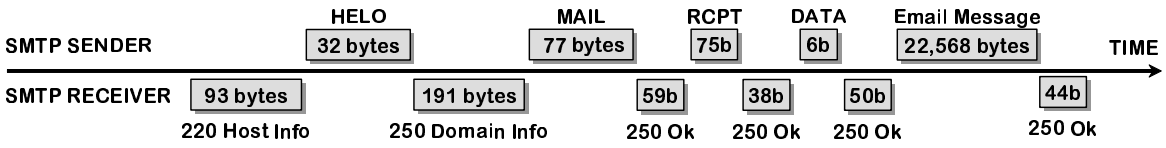

Figure 3.3: An a-b-t diagram illustrating an SMTP connection.

in the "b" direction, and time "t" (non-directional, but associated with the processing of the preceding ADU, as discussed in Section 3.1.1). Using the notation of the a-b-t model, we can succinctly describe the HTTP connection in Figure 3.1 as a single-epoch connection vector of the form

$$((341, 0, 2555, 0))$$

where the first ADU, a_1, has a size of 341 bytes, and the second ADU, b_1, has a size of 2,555 bytes. In this example the time between the transmission of the two data units and the time between the end of b_1 and connection termination are considered too small to be included in the source level representation, so they are set to 0. Similarly, we can represent the persistent HTTP connection shown in Figure 3.2 as

$$((329, 0, 403, 0.12), (403, 0, 25821, 3.12), (356, 0, 1198, 15.3))$$

where quiet times are given in seconds. Notice that tb_3 is not zero for this connection, but a large number of seconds (in fact, probably larger than the duration of the rest of the activity in the connection!). Persistent connections are often left open in case the client decides to send a new HTTP request reusing the same TCP connection[3]. As we will show in Section 3.5, this separation is frequent enough to justify incorporating it in the model. Gaps between connection establishment and the sending of a_1 are almost nonexistent.

As another example, the Simple Mail Transfer Protocol (SMTP) connection in Figure 3.3 illustrates a sample sequence of data units exchanged by two SMTP servers. The first server (labeled "sender") previously received an email from an email client, and uses the TCP connection in the diagram to contact the destination SMTP server (*i.e.*, the server for the domain of the destination email address). In this example, most data units are small and correspond to application-level (SMTP) control messages (*e.g.*, the host info message, the initial HELO message, *etc.*) rather than application objects. The actual email

[3]In general, persistent HTTP connections are closed by web servers after a maximum number of request/response exchanges (epochs) is reached or a maximum quiet time threshold is exceeded. By default, Apache, the most popular web server, limits the number of epochs to 5 and the maximum quiet time to 15 seconds.

message of 22,568 bytes was carried in ADU a_6. The a-b-t connection vector for this connection is

$$((0,0,93,0),(32,0,191,0),(77,0,59,0),(75,0,38,0),(6,0,50,0),(22568,0,44,0)).$$

Note that this TCP connection illustrates a variation of the client/server design in which the server sends a first ADU identifying itself without any prior request from the client. This pattern of exchange is specified by the SMTP protocol wherein servers identify themselves to clients right after connection establishment. Since b_1 is not preceded by any ADU sent from the connection initiator to the connection acceptor, the vector has $a_1 = 0$ (we sometimes refer to this phenomenon as a "half-epoch").

This last example illustrates an important characteristic of TCP workloads that is often ignored in traffic generation experiments. TCP connections do not simply carry files (and requests for files), but are often driven by more complicated interactions that impact TCP performance. An epoch where $a_j > 0$ and $b_j > 0$ requires at least one segment to carry a_j from the connection initiator to the acceptor, and at least another segment to carry b_j in the opposite direction. The minimum duration of an epoch is therefore one round-trip time (which is precisely defined as the time to send a segment from the initiator to the acceptor plus the time to send a segment from the acceptor back to the initiator). This means that the number of epochs imposes a minimum duration and a minimum number of segments for a TCP connection. The connection in Figure 3.3 needs 4 round-trip times to complete the "negotiation" that occurs during epochs 2 to 5, even if the ADUs involved are rather small. The actual email message in ADU b_6 is transferred in only 2 round-trip times. This is because b_6 fits in 16 segments[4], and it is sent during TCP's slow start. Thus the first round-trip time is used to send 6 segments, and the second round-trip time is used to send the remaining 10 segments. The duration of this connection is therefore dominated by the control messages, and not by the size of the email. In particular, this is true despite the fact that the email message is much larger than the combined size of the control messages. If the application protocol (*i.e.*, SMTP) were modified to somehow carry control messages and the email content in ADU a_2, then the entire connection would last only 4 round-trip times instead of 6, and would require fewer segments. In our experience, it is common to find connections in which the number of control messages is orders of magnitude larger than the number of ADUs from files or other dynamically-generated content. Clearly, epoch structure has an impact on the performance (more precisely, on the duration) of TCP connections and should therefore be modeled accurately.

Application protocols can be rather complicated, supporting a wide range of interactions between

[4]This assumes the standard maximum segment size, 1,460 bytes, and a maximum receiver window of at least 10 full size segments. A large fraction of TCP connections observed on real networks satisfy these assumptions.

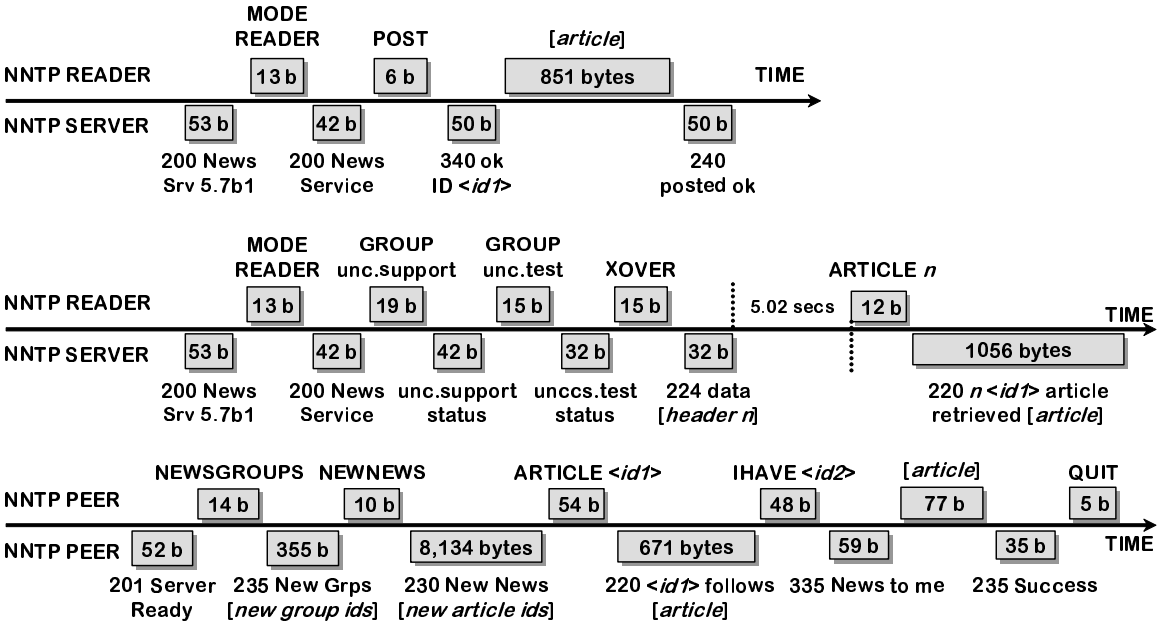

Figure 3.4: Three a-b-t diagrams representing three different types of NNTP interactions.

the two endpoints. Most of them assume a client/server model of interaction and hence can be cast into the sequential a-b-t model. For example, Figure 3.4 shows three types of interactions that are supported by the Network News Transfer Protocol (NNTP) [KL86, Bar00]. The first a-b-t diagram exhibits the straightforward behavior of an NNTP reader (*i.e.*, a client for reading newsgroup postings) posting a new article. The two endpoints exchange a few control messages in the first three epochs, and then the client uploads the content of the article in ADU a_4.

The second connection shows an NNTP reader using a TCP connection to first check whether the server knows about any new articles in two newsgroups (unc.support and unc.test). After that, the reader requests an overview of those messages (using XOVER). The server replies with the subjects of the new articles and some other information. Finally, after a 5.02 seconds of inactivity, the reader requests the content of one of the new articles. This relatively long time suggests that the user of the NNTP reader waited some time before actually requesting the reader to display the content of a new article.

The way NNTP servers interact is illustrated in the third connection. One of the peers will ask the other about new newsgroups and articles. This typically involves hundreds or even thousands of ADUs sent in each direction. The connection shown here has only a small subset of the ADUs observed in one of these connections between NNTP peers. Here the initiator peer asked for new groups first, and then

for new articles. One article was sent from the initiator to the acceptor, and another one in the opposite direction.

These examples provide a good illustration of the complexity of modeling applications one by one, and they provide further evidence supporting the claim that our abstract source-level model is widely applicable. In general, the use of a multi-epoch model is essential to accurately describe how applications drive TCP connections.

Incorporating Quiet Times into Source-Level Modeling

Unlike ADUs, which flow from the initiator to the acceptor or *vice versa*, quiet times are not associated with any particular direction of a TCP connection. However, we have chosen to use two types of quiet times in our sequential a-b-t model. This choice is motivated by the intended meaning of quiet time, and by the difference between the duration of the quiet times observed at different points in the connection's path. When we were developing the model, we initially considered quiet times independent of the endpoint causing them. They were simply "connection quiet times". In practice, quiet times in sequential connections are associated with source-level behavior in only one of the endpoints. For example, a "user think time" in an HTTP connection is associated with a quiet time on the initiator side (which is waiting for the user action), while a server processing delay in a Telnet connection is associated with the acceptor side (which is waiting for a result). In every case, one endpoint is quiet for some period before sending new data, and the other endpoint remains quiet, waiting for these new data to arrive. Having two types of quiet times, ta and tb, makes it possible to annotate the side of the connection that is the source of the quiet time.

The second reason for the use of two types of quiet times is that the duration of the quiet time depends on the point at which the quiet time is measured. The endpoint that is not the source of the quiet time will observe a quiet time that depends on the network and not only on the source-level behavior of the other endpoint. This is because the new ADU which defines the end of the quiet time needs some time to reach its destination. In the example in Figure 3.2, the quiet time between a_1 and b_1 observed by the server endpoint is very small (only the time needed to retrieve the requested URL). However, this quiet time is longer when observed by the client, since it is the time between the last socket write of a_1 and the first socket read of b_1. It includes the server processing time, and at least one full round-trip time. Ideally, we would like to measure this quiet time ta_1 on the server side, in order to characterize source-level behavior in a completely network-independent manner. Similarly, we

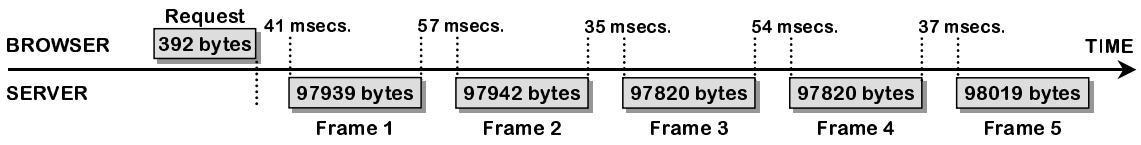

Figure 3.5: An a-b-t diagram illustrating a server push from a webcam using a persistent HTTP connection.

would like to measure tb_1 on the client side. In summary, source-level quiet times are non-directional, in the sense that they do not travel in one direction or the other, but they are associated with one of the endpoints, which is the source of the quiet time.

3.1.2 Beyond Client/Server Applications

Not all applications follow the strict pattern of requests and responses that characterizes traditional client/server applications. For example, HTTP is commonly used for server push operations[5], in which the server periodically refreshes the state of the client without any prior request. Figure 3.5 illustrates this behavior using a TCP connection where a web browser first requests a webcam URL (UNC's "Pitcam" in this example), and the web server responds with a sequence of image frames separated by small quiet times. The browser renders each frame as soon as it is received, creating a continuous movie. Each frame can be considered an individual ADU, so this connection does not follow the basic request/response sequence of previous examples. The notation provided by the sequential a-b-t model can still be used to represent this source-level behavior using the connection vector $(e_1, e_2, e_3, e_4, e_5)$ where $e_1 = (392, 0.041, 97939, 0)$, $e_2 = (0, 0.057, 97942, 0)$, $e_3 = (0, 0.035, 97820, 0)$, $e_4 = (0, 0.054, 97820, 0)$, and $e_5 = (0, 0.037, 98019, 0)$. While this connection has no natural epochs in the request/response sense, we can describe the connection by assigning each frame to a separate b_j, and each quiet time between frames to a ta_j (since the connection vector is intended to capture a quiet time on the server side).

The same type of server push behavior is found in streaming applications. A TCP connection carrying Icecast traffic (from `ibiblio.org`) is shown in Figure 3.6. Icecast is a popular audio streaming application that follows the same pattern of ADUs discussed in the previous paragraph, and can be

[5]HTTP server push is implemented using a special content type, `x-mixed-replace`, which makes the browser expect a response object that is composed of other objects (separated by a configurable boundary string). Since no limit is imposed on the number of objects in this composite, webcam movies are usually implemented as a simple sequence of JPEG images that the web browser reads and renders continuously until the user moves to another page. This type of web service should not be confused with HTML's automatic page refresh tag, which is commonly used for slow rate webcams (*e.g.*, one image every 30 seconds). In this case, the browser refreshes the current page by downloading again the current page and hence the interaction follows the regular request/response pattern.

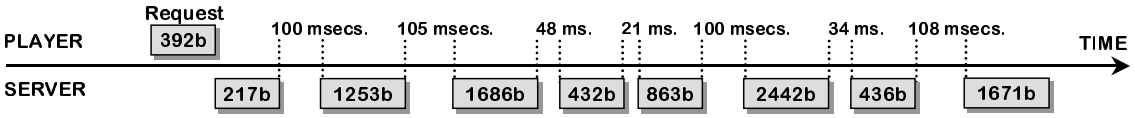

Figure 3.6: An a-b-t diagram illustrating Icecast audio streaming in a TCP connection.

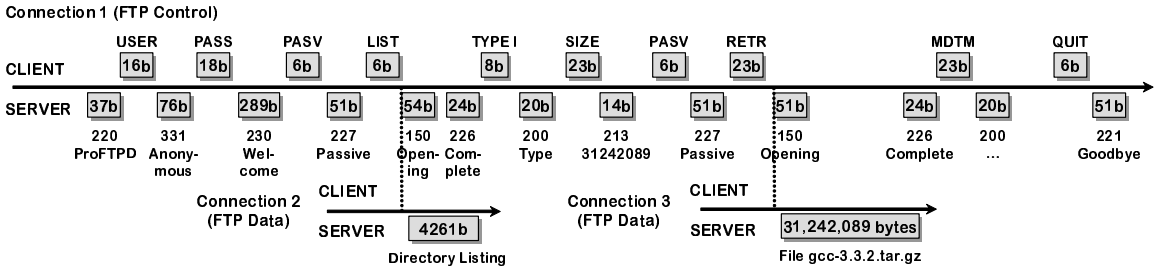

Figure 3.7: Three a-b-t diagrams of connections taking part in the interaction between an FTP client and an FTP server.

described using the same type of connection vector. Each b_j is associated to an MPEG audio frame. Note that the sizes of the ADUs and the durations of the quiet times between them are highly variable, unlike the example in Figure 3.5. Perhaps surprisingly, TCP is widely used for carrying streaming traffic today, despite its inability to perform the typical trade-off between loss recovery and delay in multimedia applications. Streaming over TCP has two significant benefits:

- Streaming traffic can use TCP port numbers associated with web traffic and therefore overcome firewalls that block other port numbers. This is important for web sites that deliver web pages and multimedia streams, since it guarantees that the user will be able to download the multimedia content.

- Most clients experience such low loss rates, that TCP's loss recovery mechanisms have an insignificant impact on the timing of the stream. The common use of stream buffering prior to the beginning of the playback further reduces the impact of loss recovery.

The interaction between the two endpoints of a client/server application does not generally require more than one TCP connection to be opened between the two endpoints. As we have seen, some applications use a new connection for each request/response exchange, while others make use of multi-epoch connections (*e.g.*, persistent connections in HTTP/1.1). Handling more than one TCP connection can have some performance benefits, but it does complicate the implementation of the applications (*e.g.*,

it may require using concurrent programming techniques). However, some applications do interact using several TCP connections and this creates interdependencies between ADUs. For example, Figure 3.7 illustrates an FTP session[6] between an FTP client program and FTP server in which three connections are used. The connection in the top row is the "FTP control" connection used by the client to first identify itself (with username and password), then list the contents of a directory, and then retrieve a large file. The actual directory listing and the file are received using separate "FTP data" connections (established by the client) with a single ADU b_1. The figure illustrates how the start of the data connections depends on the use of some ADUs in the control connection (*i.e.*, the directory listing LIST does not occur until after the RETR ADUs has been received), and how the control connection does not send the `226 Complete` ADU until the data connections have completed.

While the sequential a-b-t model can accurately describe the source-level properties of these three connections, the model cannot capture the interdependency between the connections. The FTP example in Figure 3.7 shows three connections with a strong dependency. The two FTP data connections necessarily followed a `150 Opening` operation in the FTP control connection. Our current model cannot express this kind of dependencies between connections or between the ADUs of more than one connection. It would be possible to develop a more sophisticated model capable of describing these types of dependencies, but it seems very difficult to populate such a model from traces in an accurate manner without knowledge of application semantics. As an alternative, the traffic generation approach proposed in this dissertation carefully reproduces relative differences in connection start times, which tend to preserve temporal dependencies between connections. Our experimental results also suggest that the impact of interconnection dependencies is negligible, at least for our collection Internet traces.

3.2 The Concurrent a-b-t Model

In the sequential model we have considered so far, application data is either flowing from the client to the server or from the server to the client. However, some TCP connections are not driven by this traditional style of client/server interaction. Some applications send data from both endpoints of the connection at the same time. Figure 3.8 shows an NNTP connection between two NNTP peers (servers) in which NNTP's "streaming mode" is used. As shown in the diagram, ADUs b_5 and b_6 are sent from the connection acceptor to the connection initiator while ADU a_6 is being sent in the opposite direction.

[6]This is an abbreviated version of the original session, in which there was some directory navigation and more directory listings. The control connection used port 21, while the data connections used dynamically selected port numbers. Note also that significant inter-ADU times due to user think time are not shown in the diagram.

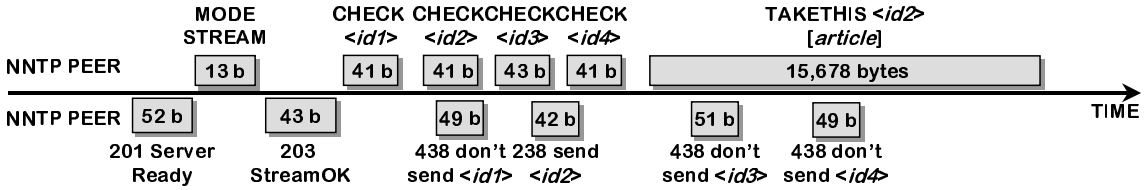

Figure 3.8: An a-b-t diagram illustrating an NNTP connection in "stream-mode", which exhibits data exchange concurrency.

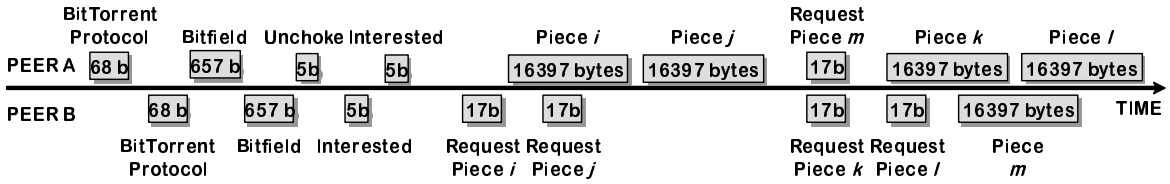

Figure 3.9: An a-b-t diagram illustrating the interaction between two BitTorrent peers.

ADUs b_5 and b_6 carry `438` messages, where the acceptor NNTP peer tells the initiator that it is not interested in articles `id3` and `id4`. ADU a_6 carried article `id2` in the opposite direction. There is no causal dependency between these ADUs, which make it possible for the two endpoints to send data independently. Therefore this connection is said to exhibit *data exchange concurrency* in the sense that one or more pairs of ADUs are exchanged simultaneously. In contrast, the connections illustrated in previous figures exchanged data units in a sequential fashion. A fundamental difference between these two types of communication patterns is that sequential request/response exchanges (*i.e.*, epochs) always take a minimum of one round-trip time. Data exchange concurrency makes it possible to send and receive more than one ADU per round-trip time, and this can increase throughput substantially. In the figure, the initiator NNTP peer is able to send check requests to the other party quicker because it can do so without waiting for the corresponding responses, each of which would take a minimum of one full round-trip time to arrive.

Another example of concurrent data exchange is shown in Figure 3.9. Here two BitTorrent peers [Coh03] exchange pieces of a large file that both peers are trying to download. The BitTorrent protocol supports the backlogging of requests (*i.e.*, pieces k and m of the file are requested before the download of the preceding piece is completed), and also the simultaneous exchange of file pieces (*i.e.*, the transmission of pieces k and l of the file coexist with the transmission of piece m). As discussed above, this type of behavior helps to avoid quiet times in BitTorrent connections, thereby increasing average throughput. Furthermore, this example illustrates a type of application in which both endpoints act as client and

server (both request and receive file pieces).

Application designers make use of data concurrency for two primary purposes:

- *Keeping the pipe full*, by making use of requests that overlap with uncompleted responses. Rather than waiting for the response of the last request to arrive, the client keeps sending new requests to the server, building up a backlog of pending requests. The server can therefore send responses back-to-back, and maximize its use of the path from the server to the client. Without concurrency, the server remains idle between the end of a response and the arrival of a new request, hence the path cannot be fully utilized.
- *Supporting "natural" concurrency*, in the sense that some applications do not need to follow the traditional request/response paradigm. In some cases, the endpoints are genuinely independent, and there is no natural concept of request/response.

Examples of protocols that attempt to keep the pipe full are the pipelining mode in HTTP, the streaming mode in NNTP, the Rsync protocol for file system synchronization, and the BitTorrent protocol for file-sharing. Examples of protocols/applications that support natural concurrency are instant messaging and Gnutella (in which the search messages are simply forwarded to other peers without any response message). Since BitTorrent supports client/server exchanges in both directions, and these exchanges are independent of each other, we can say that BitTorrent also supports a form of natural concurrency.

For data-concurrent connections, we use a different version of our a-b-t model in which the two directions of the connection are modeled independently by a pair (α, β) of connection vectors of the form

$$\alpha = ((a_1, ta_1), (a_2, ta_2), \ldots, (a_{n_a}, ta_{n_a}))$$

and

$$\beta = ((b_1, tb_1), (b_2, tb_2), \ldots, (b_{n_b}, tb_{n_b}))$$

Depending on the nature of the concurrent connection, this model may or may not be a simplification. If the sides of the connection are truly independent, the model is accurate. Otherwise, if some dependency exists, it is not reflected in our characterization (*e.g.*, the fact that request a_i necessarily preceded response b_j is lost). Our current data acquisition techniques cannot distinguish these two cases, and we doubt that a technique to accurately distinguish them exists. In any case, the two independent vectors in our concurrent a-b-t model provide enough detail to capture the two uses of concurrent data exchange

in a manner relevant for traffic generation. In the case of pipelined requests, one side of the connection mostly carries large ADUs with little or no quiet time between them (*i.e.*, backlogged responses). The exact timing at which the requests arrive in the opposite direction is irrelevant as long as there is always an ADU carrying a response to be sent. It is precisely the purpose of the concurrency to decouple the two directions to avoid the one round-trip time per request/response pair that sequential connections must incur in. There is, therefore, substantial independence in concurrent connections of this type, which supports the use of a model like the one we propose. In the case of connections that are "naturally" concurrent, the two sides are accurately described using two separate connection vectors.

3.3 Abstract Source-Level Measurement

The a-b-t model provides an intuitive way of describing source behavior in an application-neutral manner that is relevant for the performance of TCP. However, this would be of little use without a method for measuring real network traffic and casting TCP connections into the a-b-t model. We have developed an efficient algorithm that can convert an arbitrary trace of TCP/IP protocol headers into a set of connection vectors. The algorithm makes use of the wealth of information that segment headers provide to extract an accurate description of the abstract source-level behavior of the applications driving each TCP connection in the trace. It should be noted that this algorithm is a first solution to a complex inference problem in which we are trying to understand application behavior from the segment headers of a measured TCP connection without examining payloads, and hence without any knowledge of the identity of the application driving the connection. This implies "reversing" the effects of TCP and the network mechanisms that determine how ADUs are converted into the observed segments that carry the ADU. The presented algorithm is by no means the only one possible, or the most sophisticated one. However, we believe it is sufficiently accurate for our purpose, and we provide substantial experimental evidence in this and later chapters to support this claim.

3.3.1 From TCP Sequence Numbers to Application Data Units

The starting point of the algorithm is a trace of TCP segment headers, $\mathcal{T}_h$, measured on some network link. Our technique applies to TCP connections for which both directions are measured (known as a *bidirectional* trace), but we will also comment on the problem of extracting a-b-t connection vectors from a trace with only one measured direction (a *unidirectional* trace). While most public traces are bidirec-

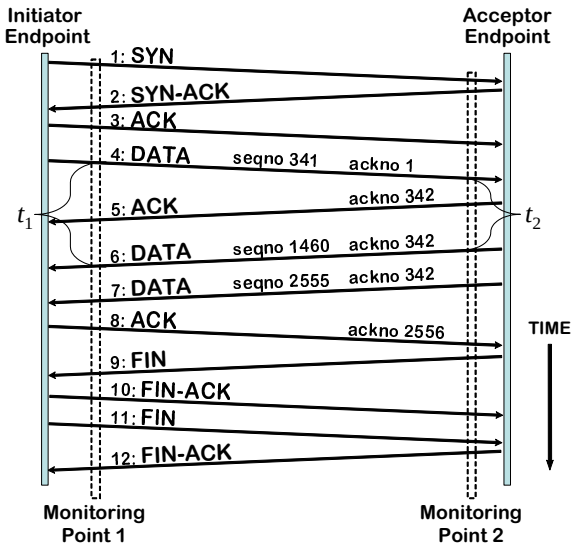

Figure 3.10: A first set of TCP segments for the connection vector in Figure 3.1: lossless example.

tional (*e.g.*, those in the NLANR repository [nlaa]), unidirectional traces are sometimes collected when resources (*e.g.*, disk space) are limited. Furthermore, routing asymmetries often result in connections that only traverse the measured link in one direction.

We will use Figure 3.10 to describe the basic technique for measuring ADU sizes and quiet time durations. The figure shows a set of TCP segments representing the exchange of data illustrated in the a-b-t diagram of Figure 3.1. After connection establishment (first three segments), a data segment is sent from the connection initiator to the connection acceptor. This data segment carries ADU a_1, and its size is given by the difference between the end sequence number and the beginning sequence number assigned to the data (bytes 1 to 341). In response to this data segment, the other endpoint first sends a pure acknowledgment segment (with cumulative acknowledgment number 342), followed by two data segments (with the same acknowledgment numbers). This change in the directionality of the data transmission makes it possible to establish a boundary between the first data unit a_1, which was transported using a single segment and had a size of 341 bytes, and the second data unit b_1, which was transported using two segments and had a size of 2,555 bytes.

The trace of TCP segments $\mathcal{T}_h$ must include a timestamp for each segment that reports the time at which the segment was observed at the monitoring device. Timestamps provide a way of estimating the duration of quiet times between ADUs. The duration of ta_1 is given by the difference between the timestamp of the 4th segment (the last and only segment of a_1), and the timestamp of the 6th segment (the first segment of b_1). The duration of tb_1 is given by the difference between the timestamp of the

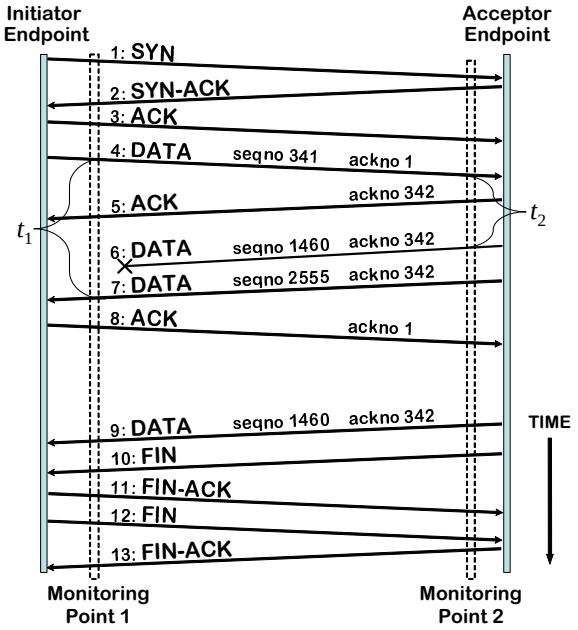

Figure 3.11: A second set of TCP segments for the connection vector in Figure 3.1: lossy example.

last data segment of b_1 (7th segment in the connection) and the timestamp of the first FIN segment (8th segment in the connection).

Note that the location of the monitoring point between the two endpoints affects the measured duration of ta_1 and tb_1 (but not the measured sizes of a_1 and b_1). Measuring the duration of ta_1 from the monitoring point 1 shown in Figure 3.10 results in an estimated time t_1 that is larger than the estimated time t_2 measured at monitoring point 2. Inferring application-layer quiet time durations is always complicated by this kind of measurement variability (among other causes), so short quiet times (with durations up to a few hundred milliseconds) should not be taken into account. Fortunately, the larger the quiet time duration, the less significant the measurement variability becomes, and the more important the effect of the quiet time is on the lifetime of the TCP connection. We can therefore choose to assign a value of zero to any measured quiet time whose duration is below some threshold, *e.g.*, 1 second, or simply use the measurement disregarding the minor impact of its inaccuracy.

If all connections were as "well-behaved" as the one illustrated in Figure 3.10, it would be trivial to create an algorithm to extract connection vectors from segment header traces. This could be done by simply examining the segments of each connection and counting the bytes sent between data directionality changes. In practice, segment reordering, loss, retransmission, duplication, and concurrency make the analysis much more complicated. Figure 3.11 shows a second set of segment exchanges that carry the

same a-b-t connection vector of Figure 3.1. The first data segment of the ADU sent from the connection acceptor, the 6th segment, is lost somewhere in the network, forcing this endpoint to retransmit this segment some time later as the 9th segment. Depending on the location of the monitor (before or after the point of loss), the collected segment header trace may or may not include the 6th segment. If this segment is present in the trace (like in the trace collected at monitoring point 2), the analysis program must detect that the 9th segment is a retransmission and ignore it. This ensures we compute the correct size of b_1, *i.e.*, 2,555 bytes rather than 4,015 bytes. If the lost segment is not present in the trace (like in the trace collected at monitoring point 1), the analysis must detect the reordering of segments using their sequence numbers and still output a size for b_1 of 2,555 bytes. Measuring the duration of ta_1 is more difficult in this case, since the monitor never saw the 6th segment. The best estimation is the time t_1 between the segment with sequence number 341 and the segment with sequence number 2555. Note that if the 6th segment is seen (as for a trace collected at monitoring point 2), the best estimate is the time t_2 between 5th and 6th segments. A data acquisition algorithm capable of handling these two cases cannot simply rely on counts and data directionality changes, but must keep track of the start of the current ADU, the highest sequence number seen so far, and the timestamp of the last data segment. In our analysis, rather than trying to handle every possible case of loss and retransmission, we rely on a basic property of TCP to conveniently reorder segments and still obtain the same ADU sizes and inter-ADU quiet time durations. This makes our analysis simpler and more robust.

3.3.2 Logical Order of Data Segments

A fundamental invariant that underlies our previous ADU analyses is that every byte of application data in a TCP connection receives a sequence number, which is unique for its direction[7]. This property also means that data segments transmitted in the same direction can always be *logically ordered* by sequence number, and this order is independent of both the time at which segments are observed and any reordering present in the trace. The logical order of data segments is a very intuitive notion. If segments 6 and 7 in Figure 3.10 carried an HTML page, segment 6 carried the first 1,460 characters of this page, while segment 7 carried the remaining 1,095. Segment 6 logically preceded segment 7. When the same page is transmitted in Figure 3.11, the first half of the HTML is in segment 6 (which was lost) and again in segment 9. Both segments 6 and 9 (which were identical) logically precede segment 7, which carried the second half of the HTML page.

[7]This is true as long as the connection carries 4 GB or less. Otherwise, sequence numbers are repeated due to the wraparound of their 32-bit representation. We discuss how to address this difficulty at the end of Section 3.3.3.

The notion of logical order of data segments can also be applied to segments flowing in opposite directions of a sequential TCP connection. Each new data segment in a sequential connection must acknowledge the final sequence number of the last in-order ADU received in the opposite direction. If this is not the case, then the new data is not sent in response to the previous ADU, and the connection is not sequential (*i.e.*, two ADUs are being sent simultaneously in opposite directions). In the previous examples in Figures 3.10 and 3.11, we can see that both data segments comprising b_1 acknowledge the final sequence number of a_1. Intuitively, no data belonging to b_1 can be sent by the server until a_1 is completely received and processed. The data in a_1 logically precede the data in b_1, and therefore the segment carrying a_1 logically precedes the segments carrying b_1. Given the sequence and acknowledgment numbers of two data segments, flowing in the same or in opposite directions, we can always say whether the two segments carried the same data, or one of them logically preceded the other.

Connections that fit into the sequential a-b-t model are said to preserve a *total order of data segments* with respect to the logical flow of data:

> For any pair of data segments p and q, such that p is not a retransmission of q or *vice versa*, either the data in p logically precedes the data in q, or the data in q logically precedes the data in p.

In the example in Figure 3.11, the data in segment 9 logically precedes the data in segment 7 (*e.g.*, segment 9 carries the first 1460 bytes of a web page, and segment 7 carries the rest of the bytes). We know this because the sequence numbers of the bytes in segment 9 are below the sequence numbers of the bytes in segment 7. The first monitoring point observes segment 7 before segment 9, so temporal order of these two segments did not match their logical data order. A total order also exists between segments that flow in opposite directions. In the example in Figure 3.11, the data in segment 4 logically precede the data carried in the rest of the data segments in the connection. Timestamps and segment reordering play no role in the total order that exists in any sequential connection.

Logical data order is not present in data-concurrent connections, such as the one shown in Figure 3.8. For example, the segment that carried the last b-type ADU (the `438 don't send` ADU) may have been sent roughly at the same time as another segment carrying some of the new data of the data unit sent in the opposite direction (such as a `CHECK` ADU). Each segment would use new sequence numbers for its new data, and it would acknowledge the data received so far by the endpoint. Since the endpoints have not yet seen the segment sent from the opposite endpoint, the two segments cannot acknowledge each other. Therefore, there is no causality between the segments, and no segment can be said to precede

the other. This observation provides a way of detecting data concurrency purely from the analysis of TCP segment headers. The idea is that a TCP connection that violates the total order of data segments described above can be said to be concurrent with certainty. This happens whenever a pair of data segments, sent in opposite directions, do not acknowledge each other, and therefore cannot be ordered according the logical data order.

Formally, a connection is considered to be concurrent when there exists at least one pair of data segments p and q that either flow in opposite directions and satisfy

$$p.seqno > q.ackno \tag{3.1}$$

and

$$q.seqno > p.ackno, \tag{3.2}$$

or that flow in the same direction and satisfy

$$p.seqno > q.seqno \tag{3.3}$$

and

$$q.ackno > p.ackno. \tag{3.4}$$

, Where $p.seqno$ and $q.seqno$ are the sequence numbers of p and q respectively, and $p.ackno$ and $q.ackno$ are the acknowledgment numbers of p and q respectively. Note that, for simplicity, our *.ackno* refers to the cumulative sequence number accepted by the endpoint (which is one unit below the actual acknowledgment number stored in the TCP header [Pos81]). The first pair of inequalities defines the *bidirectional test* of data concurrency, while the second pair defines the *unidirectional test* of data concurrency. We next discuss why a connection satisfying one of these tests must necessarily be associated with concurrent data exchanging.

We consider first the case where p and q flow in opposite directions, assuming without loss of generality that p is sent from initiator to acceptor and q from acceptor to initiator. If they are part of a sequential connection, either p is sent after q reaches the initiator, in which case p acknowledges q so $q.seqno = p.ackno$, or q is sent after p reaches the acceptor in which case $p.seqno = q.ackno$. Otherwise, a pair of data segments that do not acknowledge each other exists, and the connection exhibits data concurrency.

In the case of segments p and q flowing in the same direction, we assume without loss of generality that

$p.seqno < q.seqno$. The only way in which $q.ackno$ can be less than $p.ackno$ is when p is a retransmission sent after q, and at least one data segment k with new data sent from the opposite direction arrives between the sending of p and the sending of q. The arrival of k increases the cumulative acknowledgment number in p with respect to q, which means that $q.ackno < p.ackno$. In addition, k cannot acknowledge p, or p would not be retransmitted. This implies that the connection is not sequential, since the opposite side sent new data in k *without* waiting for the new data in p.

Thus, only data-concurrent connections have a pair of segments that can simultaneously satisfy inequalities (3.1) and (3.2) or inequalities (3.3) and (3.4). These inequalities provide a formal test of data concurrency, which we will use to distinguish sequential and concurrent connections in our data acquisition algorithm. Data-concurrent connections exhibit a *partial order of data segments*, since segments flowing in the same direction can always be ordered according to sequence numbers, but not all pairs of segments flowing in opposite directions can be ordered in this manner.

Situations in which all of the segments in a concurrent data exchange are actually sent sequentially are not detected by the previous test. This can happen purely by chance, when applications send very little data or send it so slowly that concurrent data sent in the reverse direction is always acknowledged by each new data segment. Note also that the test detects *concurrent exchanges of data* and not concurrent exchanges of segments in which a data segment and an acknowledgment segment are sent concurrently. In the latter case, the logical order of data inside the connection is never broken because there is no data concurrency. Similarly, the simultaneous connection termination mechanism in TCP in which two FIN segments are sent concurrently is usually not associated with data concurrency. In the most common case, none of the FIN segments or only one of them carries data, so the data concurrency definition is not applicable. It is however possible to observe a simultaneous connection termination where both FIN segments carry data, which is considered concurrency if these segments satisfy inequalities (3.1) and (3.2).

3.3.3 Data Analysis Algorithm

We have developed an efficient data analysis algorithm that can determine whether a connection is sequential or concurrent, and can measure ADU sizes and quiet time durations in the presence of arbitrary reordering, duplication, and loss. Rather than trying to analyze every possible case of reordering, duplication/retransmission, and loss, we rely on the logical data order property, which does not depend on the original order and timestamps.

Given the set of segment headers of a TCP connection sorted by timestamp, the algorithm performs two passes:

1. Insert each data segment as a node into the data structure `ordered_segments`. This is a list of nodes that orders data segments according to the logical data order (bidirectional order for sequential connections, unidirectional order for concurrent connections). The insertion process serves also to detect data exchange concurrency. This is because connections are initially considered sequential, so their segments are ordered bidirectionally, until a segment that cannot be inserted according to this order is found. No backtracking is needed after this finding, since bidirectional order implies unidirectional order for both directions.

2. Traverse `ordered_segments` and output the a-b-t connection vector (sequential or concurrent) for the connection. This is straight-forward process, since segments in the data structure are already ordered appropriately.

The first step of the algorithm creates a doubly-linked list, `ordered_segments` in which each list node represents a data segment using the following four fields:

- $seqno_A$: the sequence number of the segment in the initiator to acceptor direction (that we will call the A direction). This sequence number is determined from the final sequence number of the segment (if the segment was measured in the "A" direction), or from the cumulative acknowledgment number (if measured in the "B" direction).
- $seqno_B$: the sequence number of the segment in the acceptor to initiator direction.
- dir: the direction in which the segment was sent (A or B).
- ts: the monitoring timestamp of the segment.

The list always preserves the following invariant that we call *unidirectional logical data order*: for any pair of segments p and q sent in the same direction D, the `ordered_segments` node of p precedes the `ordered_segments` node of q if and only if $p.seqno_D < q.seqno_D$. At the same time, if the connection is sequential, the data structure will preserve a second invariant that we call *bidirectional logical data order*, which is the opposite of the data concurrency conditions defined above: for any pair of segments p and q, the `ordered_segments` node of p precedes the `ordered_segments` node of q if and only if

$$(p.seqno_A < q.seqno_A) \wedge (p.seqno_B = q.seqno_B)$$

or

$$(p.seqno_A = q.seqno_A) \wedge (p.seqno_B < q.seqno_B).$$

Insertion of a node into the list starts backward from the tail of the `ordered_segments` looking for an insertion point that would satisfy the first invariant. If the connection is still being considered sequential, the insertion point must also satisfy the second invariant. This implies comparing the sequence numbers of the new segment with those of the last segment in the `ordered_segments`. The comparison can result in the following cases:

- The last segment of `ordered_segments` precedes the new one according to the bidirectional order above. If so, the new segment is inserted as the new last element of `ordered_segments`.
- The last segment of `ordered_segments` and the new segment have the same sequence numbers. In this case, the new segment is a retransmission and it is discarded.
- The new segment precedes the last segment of `ordered_segments` according to the bidirectional order. This implies that network reordering of TCP segments occurred, and that the new segment should be inserted before the last segment of `ordered_segments` to preserve the bidirectional order of the data structure. The new segment is then compared with the predecessors of the last segment in `ordered_segments` until its proper location is found, or inserted as the first segment if no predecessors are found.
- The last segment of `ordered_segments` and the new segment have different sequence numbers and do not show bidirectional order. This means that the connection is concurrent. The segment is then inserted according to its unidirectional order.

Since TCP segments can be received out of order by at most W bytes (the size of the maximum receiver window), the search pass (third bullet) never goes backward more than W segments. Therefore, the insertion step takes $O(s\ W)$ time, where s is the number of TCP data segments in the connection.

The second step is to walk through the linked list and produce an a-b-t connection vector. This can be accomplished in $O(s)$ time using `ordered_segments`. For concurrent connections, the analysis goes through the list keeping separate data for each direction of the connection. When a long enough quiet time is found (or the connection is closed), the algorithm outputs the size of the ADU. For sequential connections, the analysis looks for changes in directionality and outputs the amount of data in between the change as the size of the ADU. Sufficiently long quiet times also mark ADU boundaries, indicating

an epoch without one of the ADUs.

Reordering makes the computation of quiet times more complex than it seems. As shown in Figure 3.11, if the monitor does not see the first instance of the retransmitted segment, the quiet times should be computed based on the segments with sequence numbers 341 and 2555. This implies adding two more fields to the list nodes:

- min_ts: the minimum timestamp of any segment whose position in the order is not lower than the one represented by this node. Due to reordering, one segment can precede another in the bidirectional order and at the same time have a greater timestamp. In this case, we can use the minimum timestamp as a better estimate of the send time of the lower segment.
- max_ts: the maximum timestamp of any segment whose place in the order is not greater than the one represented by this node. This is the opposite of the previous min_ts field, providing a better estimate of the receive time of the greater segment.

These fields can be computed during the insertion step without any extra comparison of segments. The best possible estimate of the quiet time between two ADU becomes

$$q.min\_ts - p.max\_ts$$

for p being the last segment (in the logical data order) of the first ADU, and q being the first segment (in the logical data order) of the second ADU. For the example in Figure 3.11, at monitoring point 1, the value of min_ts for the node for the 9th segment (that marks a data directionality boundary when segment nodes are sorted according to the logical data order) is the timestamp of the 7th segment. Therefore, the quiet time ta_1 is estimated as t_1. Note that the use of more than one timestamp makes it possible to handle IP fragmentation elegantly. Fragments have different timestamps, so a single timestamp would have to be arbitrarily set to the timestamp of one of the fragments. With our algorithm, the first fragment provides sequence numbers and usually min_ts, while the last fragment usually provides max_ts.

Other Issues in Trace Processing

Our trace processing algorithm makes two assumptions. First, it assumes we can isolate the segments of individual connections. Second, it assumes that no wraparound of sequence numbers occurs (otherwise, logical data order would not be preserved). These two assumptions can be satisfied by preprocessing the

trace of segment headers. Isolating the segments of individual TCP connections was accomplished by sorting packet header traces on five keys: source IP address, source port number, destination IP address, destination port number, and timestamp. The first four keys can separate segments from different TCP connections as long as no source port number is reused. When a client establishes more than one connection to the same server (and service), these connections share IP addresses and destination port numbers, but not source port numbers. This is true unless the client is using so many connections that it reuses a previous source port number at some point. Finding such source port number reuses is relatively common in our long traces, which are at least one hour long. Since segment traces are sorted by timestamp, it is possible to look for pure SYN segments and use them to separate TCP connections that reuse source port numbers. However, SYN segments can suffer from retransmissions, just like any other segment, so the processing must keep track of the sequence number of the last SYN segment observed. Depending on the operating system of the connection initiator, this sequence number is either incremented or randomly set for each new connection. In either case, the probability of two connections sharing SYN sequence numbers is practically zero.

Segment sorting according to the previous 5 keys requires $O(s\ log\ s)$ time (we use the Unix `sort` utility for our work). It is also possible to process the data without an initial sorting step by keeping state in memory for each active connection. On the one hand, this can potentially eliminate the costly $O(s\ log\ s)$ step, making the entire processing run in linear time. On the other hand, it complicates the implementation, and increases the memory requirements substantially[8]. Detecting the existence of distinct connections with identical source and destination IP addresses and port numbers requires $O(s)$ time, simply by keeping track of SYN sequence numbers as discussed above. In our implementation, this detection is done at the same time as segments are inserted into `ordered_segments` data structure, saving one pass.

TCP sequence numbers are 32-bit integers, and the initial sequence number of a TCP connection can take any value between 0 and $2^{32}-1$. This means that wraparounds are possible, and relatively frequent. One way to handle sequence number wraparound is by keeping track of the initial sequence number and performing a modular subtraction. However, if the SYN segment of a connection is not observed (and therefore the initial sequence number is unknown), using modular arithmetic will fail whenever the

[8]The well-known `tcptrace` tool [Ost], provides a good example of the difficulty of efficiently implementing this technique. `tcptrace` can analyze multiple connections at the same time, by keeping separate state for each connection, and making use of hashing to quickly locate the state corresponding to the connection to which a new segment belongs. When this tool is used with our traces, we quickly run out of memory on our processing machines (which have 1.5 GB of RAM). This occurs even when we use `tcptrace`'s real-time processing mode, which is supposed to be highly optimized. We believe it is possible to perform our analysis without the sorting step, but it is certainly much more difficult to develop a memory-efficient implementation.

connection suffers from reordering of the first observed segments. In this case the subtraction would start in the wrong place, *i.e.*, from the sequence number of the first segment seen, which is not the lowest sequence number due to the reordering. One solution is to use backtracking, which complicates the processing of traces.

A related problem is that representing sequence numbers as 32-bit integers is not sufficient for connections that carry more than 2^{32} bytes of data (4 GB). The simplest way to address this measurement problem is to encode sequence numbers using more than 32 bits in the `ordered_segments` data structure. In our implementation we use 64 bits to represent sequence numbers, and rely on the following algorithm[9] to accurately convert 32 bit sequence numbers to 64-bit integers even in the presence of wraparounds. The algorithm makes use of a wraparound counter and a pair of flags for each direction of the connection. The obvious idea is to increment the counter each time a transition from a high sequence number to a low sequence number is seen. However, due to reordering, the counter could be incorrectly incremented more than once. For example, we could observe four segments with sequence numbers $2^{32} - 1000, 1000, 2^{32} - 500$, and 2000. Wraparound processing should convert them into $2^{32} - 1000, 2^{32} + 1000, 2^{32} - 500$, and $2^{32} + 2000$. However, if the wraparound counter is incremented every time a transition from a high sequence number to a low sequence number is seen, the counter would be incremented once for the segment with the sequence number 1000 and again for the segment with sequence number 2000. In this case, the wraparound processing would result in four segments with sequence numbers $2^{32} - 1000, 2^{32} + 1000, 2^{32} - 500$, and $2^{32} + 2^{32} + 2000$. The second increment of the counter would be incorrect.

The solution is to use a flag that is set after a "low" sequence number is seen, so the counter is incremented only once after each "crossing" of 2^{32}. This opens up the question of when to unset this flag so that the next true crossing increments the counter. This can be solved by keeping track of the crossing of the middle sequence number. In our implementation, we use two flags, `low_seqno` and `high_seqno`, which are set independently. If the next segment has a sequence number in the first quarter of 2^{32} (*i.e.*, in the range between 0 and $2^{30} - 1$), the flag `low_seqno` is set to true. If the next segment has a sequence number in the fourth quarter of 2^{32} (*i.e.*, in the range between 2^{31} and $2^{32} - 1$), the other flag`high_seqno` is set to true. These flags are unset, and the counter incremented, when both flags are true and the next segment is not in the first or the fourth quarter of 2^{32}. Sequence numbers in the first quarter are incremented to 2^{32} times the counter plus 1. The rest are incremented by 2^{32} plus the

[9]We have not addressed the extra complexity that TCP window scaling for Long-Fat-Networks (RFC 1323 [JBB92]) introduces. It is often the case that TCP options are not available in the traces, so the use of window scaling and TCP timestamps has to be inferred from the standard TCP header. This is a daunting task. If the options are available, it is straightforward to combine regular sequence numbers and timestamps to handle this case.

counter. This handles the pathological reordering case in which the sequence number of the first segment in a connection is very close to zero, and the next segment is very close to 2^{32}. In this case the low sequence number would be incremented by 2^{32}. This algorithm requires no backtracking, and runs in $O(s)$ time. In our implementation, the sequence number conversion algorithm has been integrated into the same pass as the insertion step of the ADU analysis.

Our data acquisition techniques have been implemented in the analysis program `tcp2cvec`. The program also handles a number of other difficulties that arise when processing real traces, such as TCP implementations that behave in non-standard ways. In addition, it also implements the analysis of network-level parameters described in the next chapter.

3.4 Validation using Synthetic Applications

The data analysis techniques described in the previous section are based on a number of properties of TCP that are expected to hold for the vast majority of connections recorded. For example, the logical data order property should always hold, since TCP would fail to deliver data to applications otherwise. There are, however, a number of possible sources of uncertainty in the accuracy of the data acquisition method, and this section studies them using testbed experiments.

The concept of an ADU provides a useful abstraction for describing the demands of applications for sending and receiving data using a TCP connection. However, the ADU concept is not really part of the interface between applications and TCP. In practice, each TCP connection results from the use of a programming abstraction, called a socket, that receives requests from the applications to send and receive data. These requests are made using a pair of socket system calls, `send()` (application's write) and `recv()` (application's read). These calls pass a pointer to a memory buffer where the operating system can read the data to be sent or write the data received. The size of the buffer is not fixed, so applications are free to decide how much data to send or receive with each call and can even use different sizes for different calls. As a result, applications may use more than one send system call per ADU, and there may be significant delays between successive calls belonging to the same ADU. These operations can further interact with mechanisms in the lower layers (*e.g.*, delayed acknowledgment, TCP windowing, IP buffering, etc.) creating even longer delays between segments carrying ADUs. Such delays distort the relationship between application-layer quiet times and segment dynamics, complicating the detection of ADU boundaries due to quiet times.

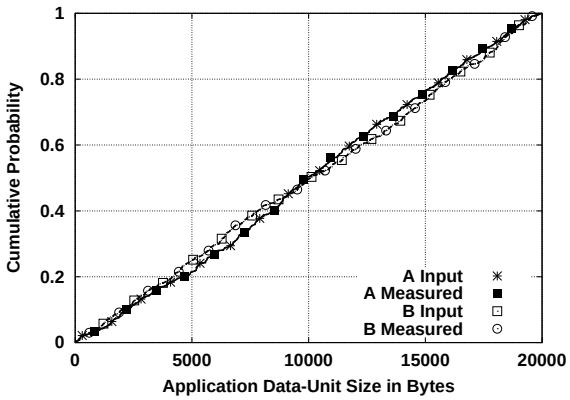

Figure 3.12: Distributions of ADU sizes for the testbed experiments with synthetic applications.

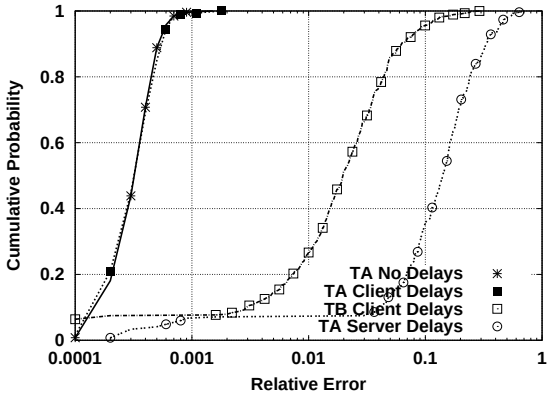

Figure 3.13: Distributions of quiet time durations for the testbed experiments with synthetic applications.

To test the accuracy of our data acquisition techniques, we constructed a suite of test applications that exercise TCP in a systematic manner. The basic logic of each test application is to establish a TCP connection and send a sequence of ADUs with a random size, and with random delays between each pair of ADUs. In the a-b-t model notation, this means creating connections with random a_i, b_i, ta_i and tb_i. As the test application runs, it logs ADU sizes and various time intervals as measured by the application. In addition, the test application can set the socket send and receive calls to random I/O sizes, and can introduce random delays between successive send or receive calls within a single ADU. In our experiments, the test application was run between two real hosts, and traces of the segment headers were collected and analyzed using our measurement tool. Our validation compared the result of this analysis and the correct values logged by the applications.

We conducted an extensive suite of tests, but limit our report to only some of the results. Specifically we only show the results with the most significant deviations from the correct values for ADU sizes or quiet time durations. Figure 3.12 shows the relative error, defined as

$$\frac{\text{value} - \text{approximation}}{\text{value}}$$

, in measuring the randomly generated ADU sizes when random send/receive sizes and random delays between socket operations were used in the test applications. The distribution of sizes of a-type ADUs as logged by the application is labeled "A Input", while the distribution of sizes of a-type ADU measured from segment headers is labeled "A Measured". There is virtually no difference between the correct and inferred values. Figure 3.12 also shows the same data for the b-type distributions which appear equally accurate. This means that our analysis will correctly infer ADU sizes even though send/receive sizes

and socket operation delays are variable.

In general, we found only two cases that expose limitations in the data acquisition method when analyzing sequential connections. While random application-level send and receive sizes, and random delays between successive send operations within a data unit do not have a significant effect, random delays between successive receive operations produce errors in estimating some quiet time durations. In this case, the application inflates the duration of a quiet time by not reading data that may already be buffered at the receiving endpoint. The consequence is a difference between the quiet time as observed at the application level and the quiet time observed at the segment level. The quiet time observed by the application is the time between the last read used to receive the ADU a_i (or b_i) and the first write used to send the next ADU b_i (a_{i+1}). The quiet time observed at the segment level is the time between the arrival of the last segment of a_i (b_i) and the departure of the first segment of b_i (a_{i+1}). If the application reads the first ADU slowly, using read calls with significant delays between them, it will finish reading a_i (b_i) well after the last segment has reached the endpoint. In this case, the quiet time appears significantly shorter at the application level than at the segment level.

For example, a data unit of 1,000 bytes may reach the receiving endpoint in a single segment and be stored in the corresponding TCP window buffer. The receiving application at this endpoint could read the ADU using 10 `recv()` system calls with a size of only 100 bytes, and with delays between them of 100 milliseconds. The reading of this ADU would therefore take 900 milliseconds, and hence the application would start measuring the subsequent quiet time 900 milliseconds after the arrival of the data segment. Our measurement of quiet time from segment arrivals can never see this delay in application reads, and would therefore add 900 milliseconds to the quiet time. For most applications we claim there is no good reason to delay read operation more than a few milliseconds. Therefore, the inaccuracy demonstrated here should be very infrequent. Nonetheless we have no direct means of assessing this type of error in our traces.

Figure 3.13 shows the relative error in the measurement of quiet time duration when there are random delays between successive read operations. The worst error is found when measuring quiet times between a_i and b_i (*i.e.*, within an epoch) when random read delays occur on the connection acceptor (receiver of a_i and b_i). Even in this case, 70% of values have less than 20% error in an experiment with what we considered severe conditions of delays between read operations for a single ADU (random delays between 10 and 100 milliseconds).

We also studied the impact of segment losses on the accuracy of the measurements. In general,

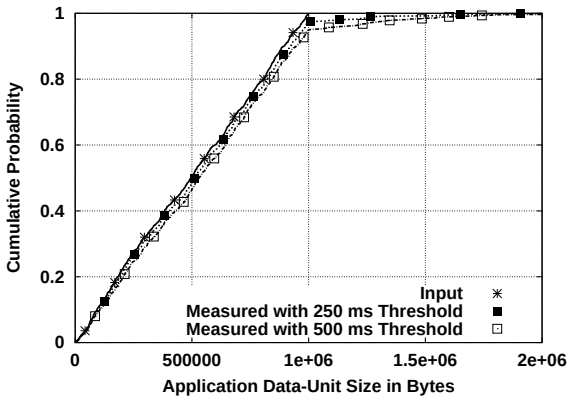

Figure 3.14: Distributions of ADU sizes for the testbed experiments with synthetic applications.

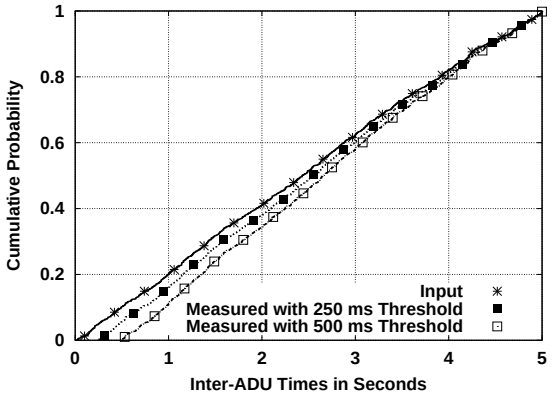

Figure 3.15: Distributions of quiet time durations for the testbed experiments with synthetic applications.

the algorithm performs well, but the analysis helped us to identify one troublesome case. If the last segment of an ADU is lost, the receiver side does not acknowledge the last sequence number of the ADU. After a few hundred milliseconds the sender side times out and resends the last segment. If the loss of the segment occurs before the monitoring point, no retransmission is observed for this last segment. If the time between this last segment and its predecessor is long enough (due to the TCP timeout), the ADU is incorrectly divided into two ADUs. Other types of segment loss do not have an effect on the measurement, since the algorithm can use the observation of retransmission and/or reordering to identify quiet times not caused by source-level behavior. The troublesome case is so infrequent that we did not try to address it. However, we note that it seems possible to develop a heuristic to detect this type of problem. The idea would be to estimate the duration of the TCP retransmission timeout, and ignore gaps between segments that are close to this estimate. The implementation of this heuristic would be complicated by the need to take into account differences in the resolution of the TCP retransmission timers, round-trip time variability and the possibility of consecutive losses.

Measuring the size of ADUs in concurrent connections is generally more difficult. This is because a change in the directionality of sequence number increases does not constitute an ADU boundary and thus we have to rely instead on quiet times to split data into ADUs. Figure 3.14 compares the input distribution of ADU sizes (from both a-type and b-type ADUs) and the measured sizes when the sizes of socket reads/writes and the delays between them are random. The measurement is generally very accurate, although some ADUs that were sent with small quiet times between them are mistakenly joined into the same measured ADU. This creates a longer tail in the measured distributions. Reducing the quiet time threshold from 500 to 250 milliseconds does little to reduce the measurement inaccuracy.

The measured quiet times are also quite close to those at the application level, as shown in Figure 3.15. The small inaccuracy comes again from ADUs that are joined together when their inter-ADU times are short. This inaccuracy biases the measured distribution of quiet times against small values (notice that the measured distributions start at a higher value). Reducing the minimum quiet time threshold to 250 milliseconds makes the measured distribution closer to the actual distribution.

3.5 Analysis Results

The a-b-t model provides a novel way of describing the workload that applications create on TCP connections. Thanks to the efficiency of the analysis method presented in Section 3.3, we are able to process large packet header traces from several Internet links. This section presents our results. The analysis of the a-b-t connection vectors extracted from disparate traces reveals that certain distributional properties remain surprisingly homogeneous across links and times-of-day, while others change substantially. To the best of our knowledge, this is the first characterization of the behavior of sources driving TCP connections that considers the entire mix of application traffic rather than just one or a few applications.

Our results come from the five traces shown in Table 3.1. This table reports statistics that compare the number of connections that are determined to be sequential and those that are determined to be concurrent according to the analysis algorithm described in section 3.3. The main lesson from Table 3.1 is the very different view of aggregate source-level behavior that counting connections or counting bytes provide. In terms of the number of connections, concurrent connections appear insignificant, accounting for a mere 3.6% of the connections in the Leipzig-II trace. The picture is completely different, however, when we consider the total number of bytes carried in those concurrent connections. In this case, concurrent connections account for 21.7% of the Leipzig-II workload, clearly suggesting that concurrency is frequently associated with TCP connections that carry large amounts of data. Abilene-I provides an

	Sequential Connections				**Concurrent Connections**			
Trace	**Count**	%	**GB**	%	**Count**	%	**GB**	%
Abilene-I	2,335,428	98.4	400.36	68.1	39,260	1.7	187.95	31.9
Leipzig-II	1,836,553	96.4	46.08	78.3	68,857	3.6	12.77	21.7
UNC 1 AM	529,381	98.5	90.35	82.4	8,345	1.6	19.34	17.6
UNC 1 PM	2,124,431	99.1	189.75	87.9	18,855	0.9	26.11	12.1
UNC 7:30 PM	808,857	98.7	102.04	76.8	10,542	1.3	30.83	23.2

Table 3.1: Breakdown of the TCP connections found in five traces.

even more striking illustration, where 31.9% of the bytes were carried by concurrent connections, which only accounted for 1.7% of the total number of connections in the trace. This is not surprising given that one of the motivations for the use of data exchange concurrency is to increase throughput. Applications with a substantial amount of data to send can greatly benefit from higher throughput, and this justifies the increase in complexity that implementing concurrency requires. On the contrary, applications which generally transfer small amounts of data have less incentive to complicate their application protocols in order to support concurrency. In this fashion, interactive traffic (*e.g.*, telnet, SSH, IRC), which tends to be associated with large numbers of small ADUs, does not usually profit from concurrency.

It is important to note that two types of TCP connections are not included in the statistics in Table 3.1: unidirectional connections and connections that carried no application data (*i.e.*, no segment carried a payload). Unidirectional connections are those for which the trace contains only segments flowing in one direction (either data or ACK segments). There are two major causes for these types of connections[10]. First, attempts to contact a nonexistent or unavailable host may not receive any response segments. In this case, the trace would show only one or a few SYN segments flowing in one direction, and no communication of application data between the two hosts. Attempts to connect to firewalled hosts also result in similar unidirectional connections. Second, routing asymmetries, that are known to be frequent in the Internet backbone, may result in connections that traverse the measured link only in one direction. Among our traces, routing asymmetries are only possible for the Abilene-I trace. The UNC and Leipzig-II traces were collected from border links that carry all of the network traffic to and from these two institutions. Two other possible causes of unidirectionality, that we believe have a much smaller impact on the count of unidirectional connections, are the effects of trace boundaries, which can limit the tracing to only a few segments flowing in one direction; and misconfigurations, where incorrect or spoofed source addresses are used.

In the UNC and Leipzig-II traces, the number of unidirectional connections was relatively high. We found between 249,923 (Leipzig-II) and 1,963,511 (UNC 1 AM) unidirectional connections. Since these are traces without any routing asymmetry, it is clear that a substantial number of attempts to establish a TCP connection failed. For example, the UNC 1 AM trace has approximately one million more unidirectional connections than the other two UNC traces. These connections are likely related to

[10]It is very unlikely that any of these connections was measured as unidirectional due to measurement losses. The traces studied in this section were collected using a high-performance monitoring device, a DAG card [Pro], that did not report any losses during data acquisition.

some traffic anomaly, such as malicious network scanning[11] and port scanning[12]. We have not studied this phenomenon further, but it is clearly important to filter out unidirectional connections to produce the results in Table 3.1. Otherwise, the percentages would be misleading, since this table is about connections that exchanged one or more ADUs during TCP application communication, and unidirectional connections did not engage in any kind of useful communication. Furthermore, unidirectional connections accounted for less than 0.15% of the bytes in the Leipzig-II and UNC traces.

The number of unidirectional connections in the Abilene-I trace was even larger: 2.6 millions in the Indianapolis to Cleveland direction and 22.3 millions in the opposite direction. Unlike the UNC and Leipzig-II traces, these connections accounted for a significant fraction of the bytes in each direction (1.63% and 14.42%). This fact, and a closer examination of the connections[13], confirmed that routing asymmetry is present in the Abilene-I trace. Asymmetric connections can carry application data, and therefore should be considered in source-level studies. However, our concurrency test requires bidirectional measurements, so the type of breakdown shown in Table 3.1 cannot be performed with the unidirectional connections in the Abilene-I trace.

Our traces also include a significant number of connections that did not carry any application data (*i.e.*, TCP connections that were established and terminated without transmitting a single data segment[14]). The number of connections without any data units varied between 75,522 in the UNC 1 AM trace and 400,853 in the Abilene-I trace. These "dataless" connections can again be due to network and port scanning, and also to failed attempts to establish TCP connections. These failures can come from attempts to contact endpoint port numbers on which no application is listening[15]. They can also come from aborted connections which are due to high loss rates, excessive round-trip times, or implementation problems. While the number of connections without application data is relatively high when compared with the number of connections in Table 3.1, these connections accounted for less than 0.11% of the bytes.

[11] Network scanning is a technique for discovering the hosts attached to a network by probing each possible IP address in a network domain. The basic technique is to send a packet which generally requires a response from the host that received it (*e.g.*, an ICMP echo request, a TCP SYN segment). Malicious users often scan remote networks to find hosts before trying to break into them. Network scanning with TCP segments is available in many popular tools, *e.g.*, `nmap`.

[12] Port scanning is similar to network scanning, but it involves probing a range of port numbers (for a single IP address) rather than probing a range of IP addresses. The goal of port scanning is to discover active services, which could potentially have vulnerabilities. Port scanning is performed using any TCP segment (or UDP datagram) that elicits a response from the victim (*e.g.*, a SYN segment requires a SYN-ACK in response, a malformed segment requires a RST segment in response).

[13] We found numerous connections that had data segments with increasing sequence numbers.

[14] In some cases, these connections showed some data segments with a sequence number above that of the FIN segments. These cases seemed to be caused by TCP implementation errors.

[15] In this case the destination endpoint responds with a TCP reset segment, and no application-level communication takes place.

The rest of this section examines the distributional properties of the connection vectors derived from the traces. Connection vectors constitute a rich data set that can be explored along different axes. We have chosen to first compare traces collected at different sites. This helps us study variability in source-level behavior originating from differences in the populations of users and services. The second part of the section studies the three traces from UNC, analyzing the changes in source-level behavior due to the strong time-of-day effects that most Internet links exhibit. At the same time, this section illustrates the significant difference between TCP connections initiated from one side of the link (by clients inside UNC) and those initiated from the other side (by clients outside UNC that contacted servers inside UNC).

Note that the analysis below reports only on those connection vectors derived from TCP connections that were *fully captured*, *i.e.*, those for which we believe that every segment was observed. In practice, we consider that a connection was fully captured when we observe both the start of the connection, marked by SYN and SYN-ACK segments, and the end of the connection, marked by FIN or RST segments. This does not necessarily mean that we observed every single segment of the connection[16], but it does imply that the full source-level behavior of the connection is observed. Another reason to work only with fully captured connections is that the absence of connection establishment segments prevents us from identifying the connection initiator. It is often the case that the acceptor is listening on a reserved port number (< 1024), which provides a way to address this difficulty. However, there is still a large fraction of the connections that use dynamic port numbers, and for which the initiator cannot be identified with certainty.

3.5.1 Variability Across Sites

Sequential Connections

We start our statistical analysis with the characterization of sequential connections from different sites. Figure 3.16 examines the distributions of the sizes of the ADUs for three traces: Abilene-I, Leipzig-II and UNC 1 PM. We use the letter "A" to refer to a distribution of a-type ADU sizes, and the letter "B" to refer to a distribution of b-type ADU sizes. The distributions in this figure only include samples from sequential connection vectors. We can distinguish two regions in this plot. For sizes of ADUs above 250 bytes, the shape of the A distributions is remarkably similar for all three traces, and

[16] In some (rare) cases, we may miss some segments before connection establishment (*e.g.*, we miss the first SYN segment but observe its retransmission), or we may miss some segments after connection establishment (*e.g.*, we miss the retransmission of the final FIN segment and its acknowledgment).

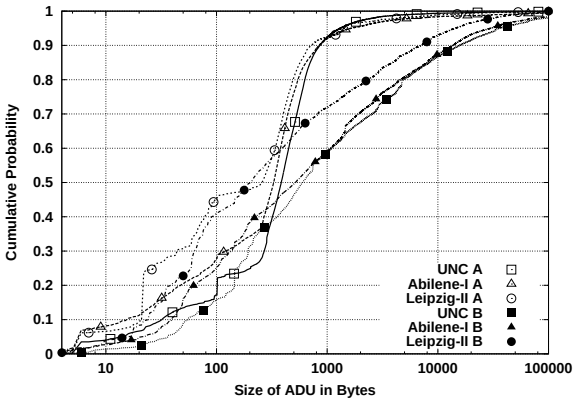

Figure 3.16: Bodies of the A and B distributions for Abilene-I, Leipzig-II and UNC 1 PM.

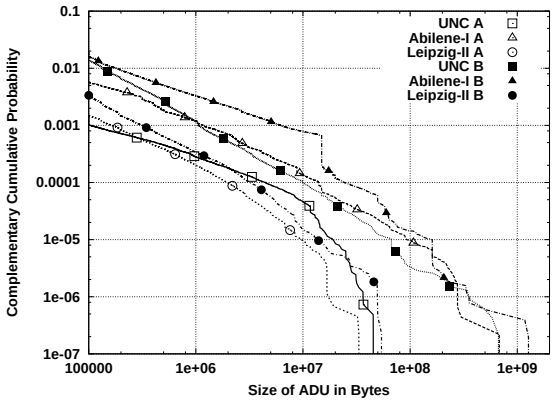

Figure 3.17: Tails of the A and B distributions for Abilene-I, Leipzig-II and UNC 1 PM.

quite different from the shapes of the B distributions. The vast majority of the ADUs sent from the connection initiator (92%) had a size below 1,000 bytes. This is consistent with the idea that a-type ADUs mostly carry small requests and control messages. Most a-type ADUs can therefore be carried in a single standard-size segment of 1960 bytes. The shape of the B distributions is also consistent with our intuition, although the Leipzig-II distribution is significantly lighter than the others. The B distributions are heavier than the A distributions. Between 38% and 27% of the b-type ADUs are larger than 1460 bytes, so they require two or more segments to be transported from the connection acceptor to the connection initiator. Only 8% to 12% of the b-type ADUs carried 10,000 bytes or more. We also note that for ADU sizes below 250 bytes, the plot shows less similarity among distributions of the same type. However, the logarithmic scale on the x-axis can be misleading. The large separation between the curves corresponds to only a few tens of bytes, and this has little impact on TCP performance. ADUs as small as 250 bytes can always be transported in a single (small) segment.

Figure 3.17 shows the tails of the A and B distributions using complementary cumulative distribution functions. It shows that even a-type ADUs can be quite large, and that the distributions are consistent with heavy-tailness (*i.e.*, exhibits linear decay in the log-log CCDF). For this reason, Pareto or Lognormal models could provide a good foundation for analytical modeling of the distributions[17]. Interestingly, when we compare A and B distributions for the same trace, we find that B distributions are only slightly heavier than A distributions, especially for Abilene-I and Leipzig-II. This implies that there are protocols in which the initiator sends large ADUs to the acceptor. For example, web browsers are often used to upload files and email attachments for web-based email accounts. It is also interesting to

[17]The tail of a Pareto distribution is always linear in a CCDF, and the tail of a Lognormal distribution can be linear for an arbitrary number of orders of magnitude.

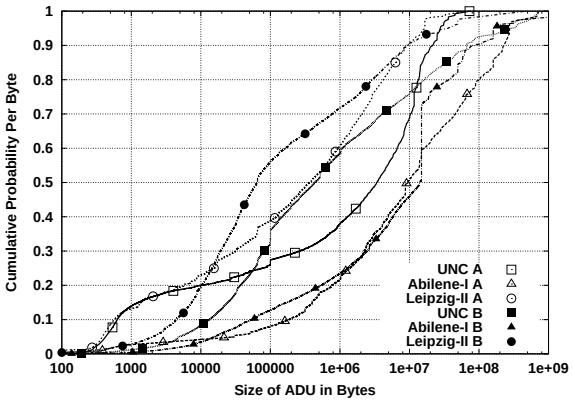

Figure 3.18: Bodies of the A and B distributions with per-byte probabilities for Abilene-I, Leipzig-II and UNC 1 PM

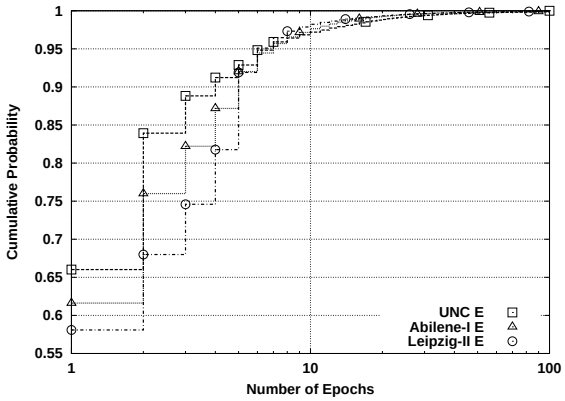

Figure 3.19: Bodies of the E distributions for Abilene-I, Leipzig-II and UNC 1 PM.

note that Abilene-I's A distribution is heavier than UNC's and Leipzig-II's B distributions, and that UNC's B distribution is significantly heavier than Leipzig-II's B distribution. We believe this reflects the type of network measured and/or the population of users. Transferring large ADUs is more feasible in higher capacity networks, and this fosters the use of more data-intensive applications and more data-intensive uses of applications. Abilene is a well-provisioned backbone network that carries traffic between well-connected American universities, so it seems more likely to exhibit connections with larger ADUs.

The small probabilities of finding large ADUs shown in Figures 3.16 and 3.17 can give the false impression that only small ADUs are important. Figure 3.18 corrects this view by plotting the probability that a byte is carried in an ADU of a given size. The figure shows that the majority of the bytes in the network were carried in large ADUs. For example, the probability that a byte was carried in an ADU of 100,000 bytes or more was as high as 0.9 for Abilene-I. This is in stark contrast to the corresponding Abilene-I distribution in Figure 3.16, where the probability of an ADU of 100,000 bytes or more is as low as 0.01 for the three traces.

The three networks show remarkably different distributions in Figure 3.18. This is in part due to the impact of sampling on this type of analysis, which is rather sensitive to the number of samples in the tail of the distribution. Adding a single very large sample can shift the entire distribution downward, since the probability of finding a byte in the rest of the ADU sizes decreases significantly. However, we can still make interesting observations about the bodies of these distributions based on their shapes (which are not affected by sampling artifacts). The distributions for UNC and Leipzig-II show two striking crossover points, the first one around 10 KB and the second one around 10 MB. The curves before the first crossover point show that the ADUs carrying 20% of the a-type bytes tended to be much smaller

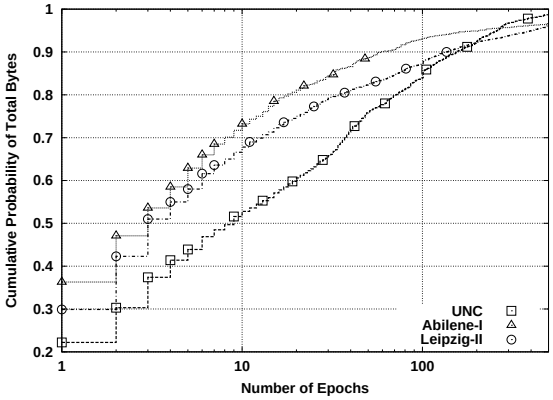

Figure 3.20: Bodies of the E distributions with per-byte probabilities for Abilene-I, Leipzig-II and UNC 1 PM.

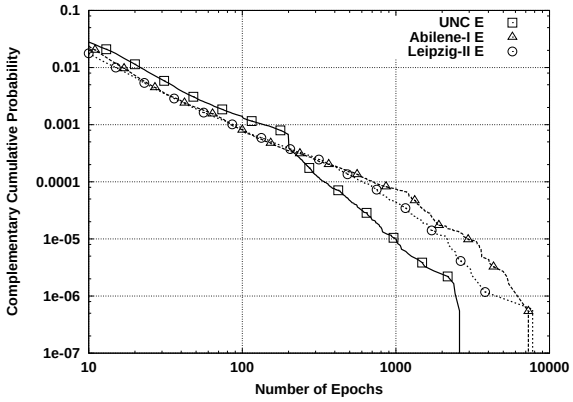

Figure 3.21: Tails of the E distributions for Abilene-I, Leipzig-II and UNC 1 PM.

than those carrying 20% of the b-type bytes. The curves between the two crossover points show the opposite for larger ADUs. Here 50% of the a-type bytes are carried in ADUs that tended to be much larger than those ADUs carrying b-type bytes. The situation reverses again after the second crossover point. This shows that the A distributions are strongly bimodal: objects are either much smaller or much larger than the average b-type ADU. The same phenomenon is found in the Abilene-I distributions between 10 KB and 1 MB, but the difference in probability is much smaller here (and could be explained by tail sampling artifacts). In addition, there is a third crossover point in the Abilene-I distributions, which defines a new region between 15 and 250 MB.

The distribution of the number of epochs E in each set of connection vectors is shown in Figure 3.19. Between 58% and 66% of the connection vectors have a single epoch. This includes a significant number of connections with a single half-epoch that come from FTP-DATA connections. Only 5% of the connections have more than 10 epochs. This does not mean that connections with a large number of epochs are unimportant. As Figure 3.20 shows, connections with a large number of epochs are responsible for a large fraction of the bytes. For example, connections with 10 epochs or more, which represent 3% of the connections, carried between 30% and 50% of the total bytes, depending on the trace.

Figure 3.20 shows that UNC's E distribution is substantially heavier than the ones for the other two traces when probability is computed over the total number of bytes. This suggests that the type of traffic in the UNC trace includes applications that make more use of multi-epoch connections. This also provides evidence that connections with moderate numbers of epochs can fit within the shorter duration (1 hour) of this trace. Otherwise, the Abilene-I trace (2 hours long) and the Leipzig-II traces(2 hours and 45 minutes long) would show heavier bodies. On the contrary, the tails of the E distributions shown

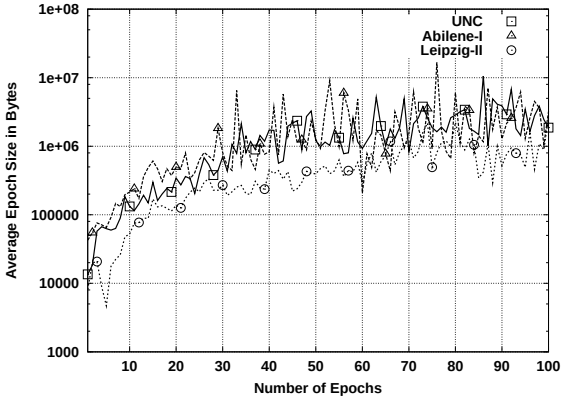

Figure 3.22: Average size $a_j + b_j$ of the epochs in each connection vector as a function of the number of epochs, for UNC 1 PM, Abilene-I and Leipzig-II

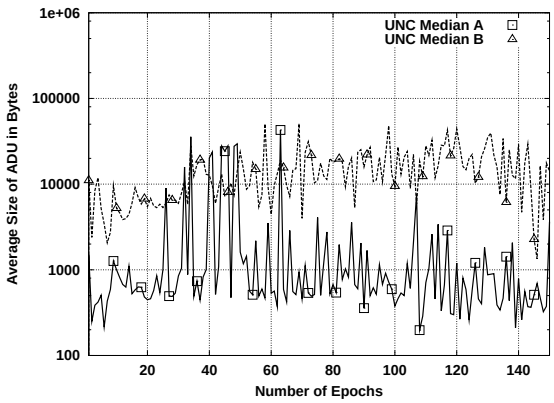

Figure 3.23: Average of the median size of the ADUs in each connection vector as a function of the number of epochs, for UNC 1 PM.

in Figure 3.21 are significantly heavier for Abilene-I and Leipzig-II than for UNC. This perhaps suggests that 1-hour traces are too short to observe connections with thousands of epochs. The sharp change in the slope of the tail of UNC's E distribution could be explained by a common application that has a fixed limit on the number of epochs (perhaps 110). However, we know of no such application.

One interesting modeling question is whether there is any dependency between the size of the ADU in one epoch and the number of epochs in the connection. If these are independent, it would be straightforward to generate synthetic connection vectors simply by first sampling a number of epochs E and then assigning ADU sizes by sampling from A and B. Figure 3.22 shows that this independence does not exist. The average size of an epoch (*i.e.*, $a_j + b_j$) increases very quickly for connections up to 30 epochs (notice the logarithmic y-axis). Connections with more epochs show high variability in the average size of their epochs. UNC and Abilene-I have quite similar averages that are much larger than those found in Leipzig-II (but note the sharp increase in average sizes for connections with 60 to 80 epochs).

Figures 3.24-3.26 provide further evidence against the independence of ADU sizes and number of epochs, and illustrate some remarkable complexity and site dependence. The plots illustrate how the number of epochs changes the size of the typical ADU, where "typical" is defined as the median of the sizes of the ADUs in each connection vector. Since a large number of connection vectors have the same number of epochs, we summarized these data by plotting the average of the median sizes *vs.* the number of epochs. Unlike the data in Figure 3.22, we analyzed median ADU sizes for a-type and b-type ADUs separately.

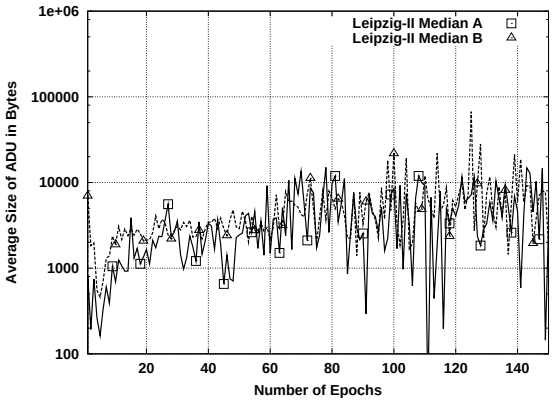

Figure 3.24: Average of the median size of the ADUs in each connection vector as a function of the number of epochs, for Leipzig-II.

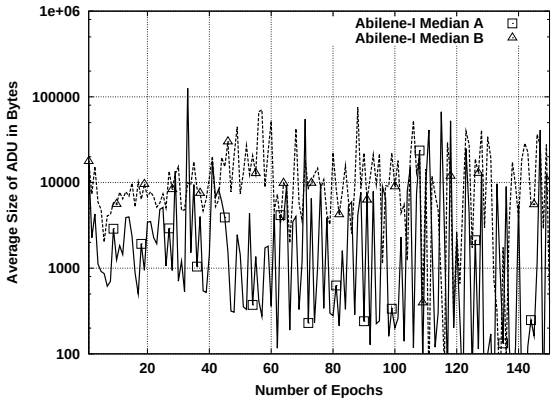

Figure 3.25: Average of the median size of the ADUs in each connection vector as a function of the number of epochs for Abilene-I.

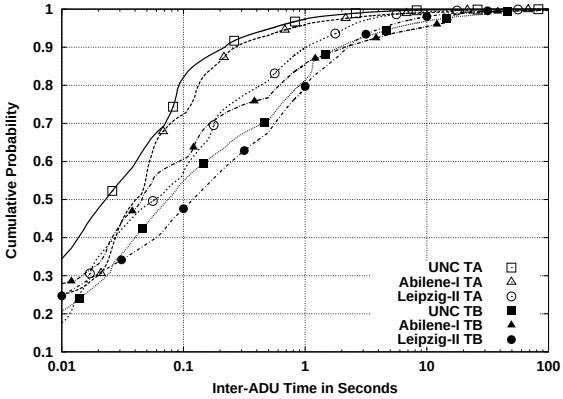

Figure 3.26: Bodies of the TA and TB distributions for Abilene-I, Leipzig-II and UNC 1 PM.

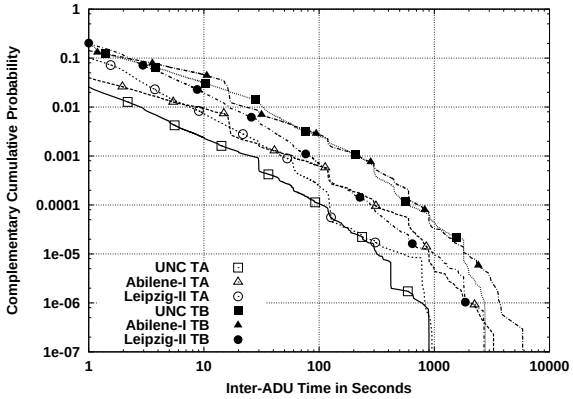

Figure 3.27: Tails of the TA and TB distributions for Abilene-I, Leipzig-II and UNC 1 PM.

The two distributions for UNC trace in Figure 3.23 are completely different (the median sizes for b-type ADU are much larger). There are, however, some epochs sizes between 25 and 50 for which a-type data units can be as large as b-type data units. Leipzig-II shows a completely different structure in Figure 3.24, where a-type ADUs are shown to be as large as b-type ADUs, and both are larger than UNC's a-type ADUs, and smaller than UNC's b-type ADUs. Abilene-I's distribution of b-type ADUs is similar to that of UNC. On the contrary, Abilene-I's distribution of a-type ADUs shows extreme variability for 60 epochs or more, and this phenomenon is completely absent in UNC's distribution. The conclusion of these four plots is clear: it is quite unrealistic to generate synthetic connection vectors using a simple model that assumes independence between ADU sizes and number of epochs.

Figure 3.26 examines the distributions of quiet times between ADUs. Shown are the distributions TA for ta_j and TB for tb_j. Note that the quiet times between the last ADU and connection termination,

i.e., tb_j for the last epoch, are not included in TB. The plot shows that, as the durations of the quiet times increase, the bodies of the TA distributions become increasingly lighter than those of the TB distributions. This is consistent with our understanding of client/server applications. Inter-epoch quiet times (TB) are usually user-driven, while intra-epoch quiet times (TA) are usually due to server processing delays. Server processing delays should generally be far shorter than user think times. For UNC and Abilene-I, most of the probability mass of TA is below 100 milliseconds, while that of TB is spread more widely. This is a strong indication that quiet times on the order of a few hundred milliseconds mostly reflect source-level quiet times. Observing TA being significantly lighter than TB is explained by the presence of user think times. Neither network delays nor the location of the monitor can provide an alternative explanation of the difference, since both factors have exactly the same impact on both distributions. The bodies Leipzig-II's TA and TB distributions are substantially heavier than the corresponding bodies of the other two traces. This could be due in part to network-level components of these distributions. Since Leipzig is in Europe, clients in the Leipzig-I trace suffer far longer round-trip times to US servers than clients found in the UNC and Abilene-I traces.

Unlike the bodies, the tails of the distributions shown in Figure 3.27 do not show the same difference between Leipzig-II and the other traces. This is consistent with the expectation that these longer quiet times are completely dominated by source-level behavior, and not by the impact of network location (*i.e.*, Europe *vs.* U.S.A.). We observe that Abilene-I's and UNC's TB are both substantially heavier than Leipzig-II's TB. Also, Leipzig-II's TA becomes lighter than Abilene-I's TA for quiet times above 11 seconds. Interestingly, we also find a similar shape for the two heaviest tails, Abilene-I's and UNC's TB, which came from traces of very different durations (2 hours *vs.* 1 hour). This provides strong evidence that trace boundaries are not introducing artifacts in our characterization of inter-ADU quiet times, despite the hard upper limit that trace duration imposes on quiet time duration.

Figure 3.28 shows the distribution of extra quiet times between the last ADU in a connection and TCP's connection termination. In the UNC and Abilene-I traces, 84% of the connections had extra quiet times below 1 second. The extra quiet time is actually zero for 83% of the cases, where the last segment of the last ADU had the FIN flag enabled. Leipzig-II showed an even higher percentage, 65%, of long quiet times after the last ADU. In all cases, we find large jumps in the probability for some values (*e.g.*, 7, 11 and 15 seconds). Moreover, the tails are surprisingly long. Since most connections transfer small amounts of data, this high frequency of extra quiet times has an important impact on the lifetimes of TCP connections observed from real links, and play an important role in realistic traffic generation.

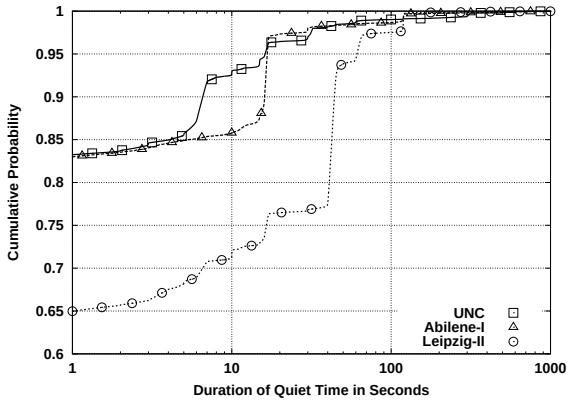

Figure 3.28: Distribution of the durations of the quiet times between the final ADU and connection termination.

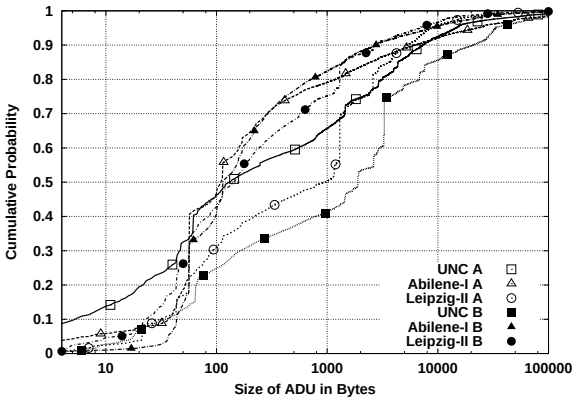

Figure 3.29: Bodies of the A and B distributions for the concurrent connections in Abilene-I, Leipzig-II and UNC 1 PM.

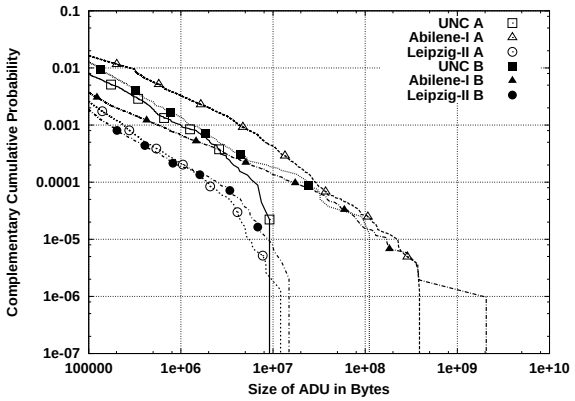

Figure 3.30: Tails of the A and B distributions for the concurrent connections in Abilene-I, Leipzig-II and UNC 1 PM.

Concurrent Connections

Concurrent connections exhibit substantially different distributions. Figure 3.16 showed distributions of a-type ADU sizes with bodies that were clearly lighter than those of b-type ADU sizes. In contrast, Figure 3.29 shows that concurrent connections made use of larger a-type ADUs, and that the shapes of A and B are not consistent across sites. Abilene-I does not show any significant difference between A and B, while Leipzig-II and UNC distributions do show a heavier B. The tails of these distributions shown in Figure 3.30 are as heavy as those for sequential connections, with the same three distributions (Abilene-I's A and B and UNC's B) having much longer tails that the other three. This phenomenon is far more striking for concurrent connections.

The distributions of quiet time durations shown in Figure 3.31 reveal that concurrent connections do

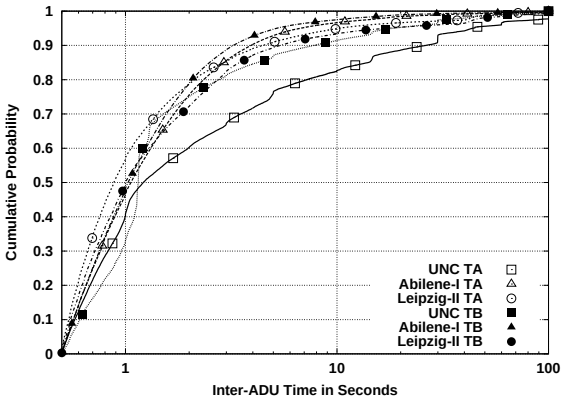

Figure 3.31: Bodies of the TA and TB distributions for the concurrent connections in Abilene-I, Leipzig-II and UNC 1 PM.

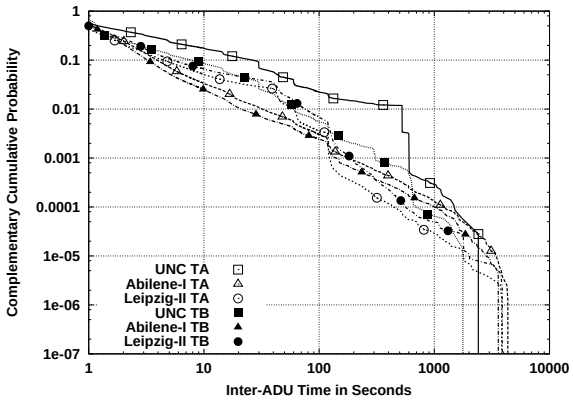

Figure 3.32: Tails of the *TA* and *TB* distributions for the concurrent connections in Abilene-I, Leipzig-II and UNC 1 PM.

not exhibit the clear separation between TA and TB that was observed for the sequential connections in Figure 3.26. This is consistent with the motivations for using concurrent data exchanges given in section 3.2. Connections that use concurrency to improve throughput by keeping the pipeline full do so to reduce the impact of user delays and client processing, thereby making TB lighter. Connections used by applications that are naturally concurrent should not exhibit any systematic difference between TA and TB distributions. Note that the minimum quiet time was 500 milliseconds, which was the duration of our threshold separating ADUs in concurrent connections.

The TA distribution for concurrent connections is significantly heavier for UNC. This suggests the presence of a concurrent application at UNC that is rather asymmetric and that is not so common in Abilene-I and Leipzig-II. The tails of the TA and TB distributions for concurrent connections shown in Figure 3.32 exhibit similar shapes and lengths to those found for sequential connections.

3.5.2 Time-of-Day Variability and Workload Directionality

The previous analysis illustrated the variability of the a-b-t distributions when several sites are compared. It also pointed out a number of features that are consistent with the communication patterns that motivate our models. TCP workloads at the same site can also exhibit significant differences, as the set of dominant applications changes throughout the day. For example, we expect to find substantial traffic from applications that are used for study and work activities (*e.g.*, e-business, research digital libraries) from 8 AM to 5 PM in the academic environment. In contrast, our guess is that traffic from gaming and other leisure time applications should be more common after 5 PM, mostly coming from

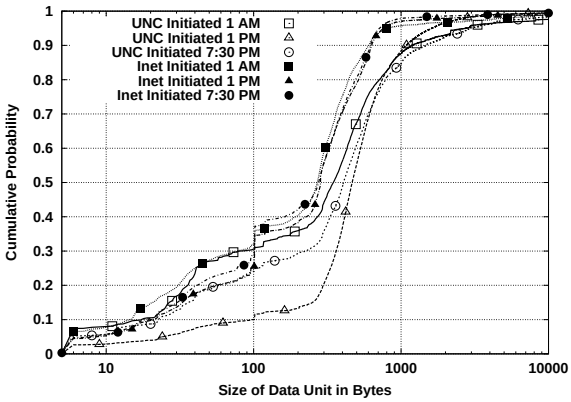

Figure 3.33: Bodies of the A distributions for UNC 1 AM, UNC 1 PM and UNC 7:30 PM.

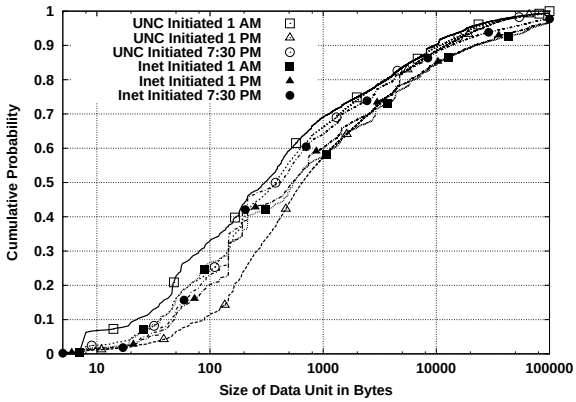

Figure 3.34: Bodies of the B distributions for UNC 1 AM, UNC 1 PM and UNC 7:30 PM.

the dorms where students live. This change in the mix of applications should have an impact on the source-level properties of the traffic.

Another important dimension of traffic variability that was not considered in the previous section was the fact that traffic may be asymmetric. For example, traffic created by UNC clients is representative of the network activity of a large population of users (30,000) that can access any kind of service on the Internet. On the contrary, traffic created by clients from outside UNC is representative of the type of services that an academic institution offers to the rest of the Internet. This dichotomy should have an impact on the source-level properties of the traffic, as traffic from UNC's connection initiators is expected to be driven by a rather different mix of applications than that of UNC's connection acceptors.

Figure 3.33 provides a first illustration of the impact of these two kinds of variability on source-level properties. The plot shows A distributions for sequential connections observed at UNC during three different intervals (1 to 2 AM, 1 to 2 PM, and 7:30 to 8:30 PM). The plots separate data from connections initiated by UNC clients (labeled "UNC Initiated") and data from connections initiated by clients outside UNC (labeled "Inet Initiated"). The significant difference between A distributions for UNC initiators is in sharp contrast with the quite similar A distributions for UNC acceptors. This shows that time-of-day variation is substantial for connections initiated at UNC, but not for connections initiated outside UNC. This is consistent with the observation that UNC services, such as the large software repository `ibiblio.org`, are available 24 hours a day, and they serve clients from different parts of the world throughout the entire day. On the contrary, the activities of UNC clients are a function of campus activity and its evolution along a diurnal cycle. The distributions of b-type ADU sizes in Figure 3.34 also reflect this dichotomy. The B distributions on UNC initiated connections for

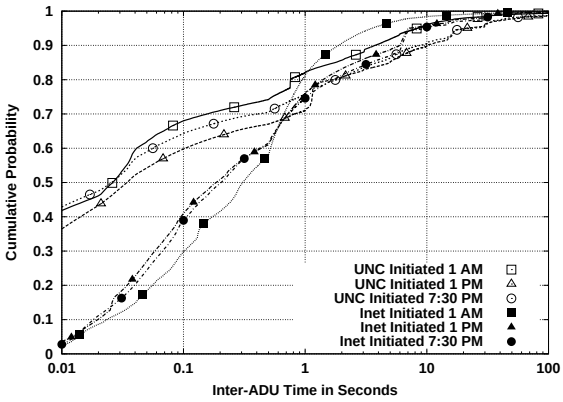

Figure 3.35: Bodies of the TB distributions for UNC 1 AM, UNC 1 PM and UNC 7:30 PM.

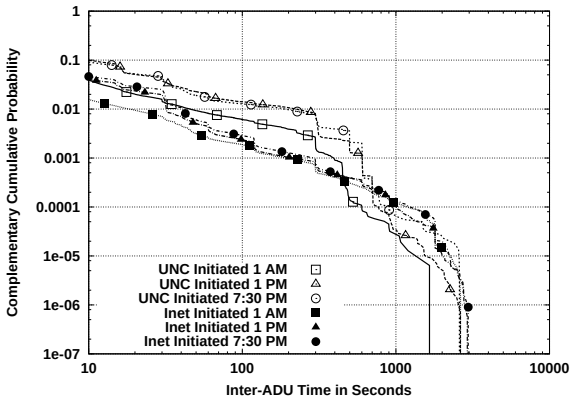

Figure 3.36: Tails of the TB distributions for UNC 1 AM, UNC 1 PM and UNC 7:30 PM.

the 1 AM and 1 PM traces form an envelope around the other distributions, while the three distributions for non-UNC initiators are remarkably similar.

Figure 3.35 serves to illustrate the impact of monitor location on the measurement of quiet times. UNC traces were collected on the border link between UNC and the rest of the Internet. This means that the monitoring occurred very close, in terms of delay, to UNC clients and UNC servers. Going back to the diagram in Figure 3.10, this means that connections initiated from UNC are seen from the first monitoring point (very close to the client), while those initiated from outside UNC are seen from the second monitoring point (very far from the client). As a consequence, TB distributions from UNC clients, which measure the time between the end of a response b_j and the beginning of a new request a_{j+1}, are observed much closer to the clients, and are characterized very accurately. TB distributions from non-UNC clients are measured much further from the client, so they tend to overestimate true quiet times. As discussed before, this type of inaccuracy is a function of round trip time. This is clearly shown in Figure 3.35, where TB distributions from UNC initiators are much lighter than those for non-UNC initiators for quiet times below 1 second. As quiet times get larger and larger, the inaccuracy due to the placement of the monitoring point becomes less and less significant. The crossing points of the distributions between 500 milliseconds and 1 second suggest that the characteristics of applications and user behavior start to dominate measured quiet times above a few hundred milliseconds.

The same observations regarding the impact of the monitoring point also holds for the TA distributions in Figures 3.37 and 3.38. Here the effect of the monitoring point is reversed: ta_j is observed far from the client for UNC initiated connections, and close to the client for non-UNC initiated connections).

Time-of-day effects are less clear in Figure 3.35. If we look at quiet times above 1 second (the relevant

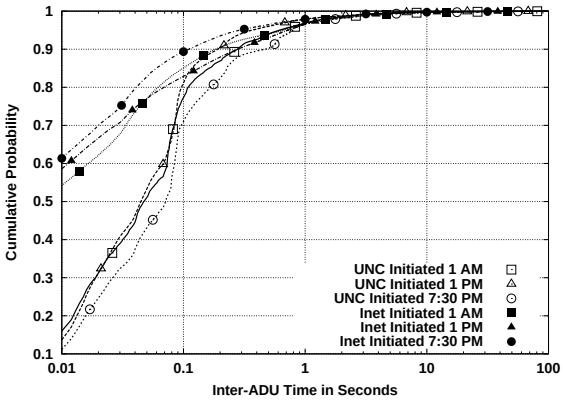

Figure 3.37: Bodies of the TA distributions for UNC 1 AM, UNC 1 PM and UNC 7:30 PM.

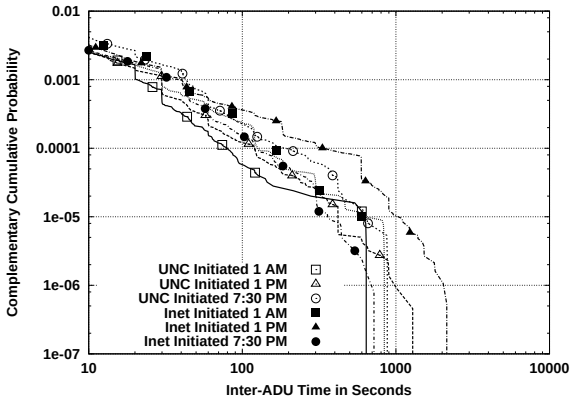

Figure 3.38: Tails of the TA distributions for UNC 1 AM, UNC 1 PM and UNC 7:30 PM.

ones), we can see that the distributions for 1 PM and 7:30 PM are quite similar for both directions, while those for 1 AM are lighter and not consistent with each other (especially for UNC acceptors). This is also true for the tails of these distributions shown in Figure 3.36 for quiet times below 500 seconds. The tails of the TA distributions in Figure 3.38 do not show any consistent pattern (*i.e.*, no grouping based on time-of-day or directionality). They are also somewhat lighter than the TB distributions.

3.6 Summary

This chapter presented our method for describing source-level behavior in an abstract manner using the *a-b-t model*. The basic observation behind this model is that the job of a TCP connection is to transfer one or more application data units (ADUs) between two network endpoints. TCP is sensitive to the sizes of these ADUs, which determine the number of segments required to transfer them, but it is insensitive to the actual semantics of each ADU. Consequently, we proposed to describe the source-level workload of TCP connections in terms of ADUs, characterizing their number, order, and sizes. Additionally, we also observed that applications may remain inactive during long periods of time (*e.g.*, during user think times), which often results in TCP connections that last far longer than required to transfer their ADUs. This motivated us to also incorporate quiet times into our generic descriptions of source-level behavior. We formulated these ideas into the a-b-t model, which describes source-level behavior in abstract terms common to all applications. The model distinguishes a-type ADUs, sent from the connection initiator to the connection acceptor, and b-type ADUs, sent in the opposite direction the connection. It also distinguishes between quiet times due to inactivity on the initiator endpoint and due to inactivity on the acceptor endpoint.

Our analysis of TCP connections observed on real Internet links revealed two types of source-level behavior, which motivated us to develop two different versions of our a-b-t model. Most TCP connections exchange ADUs in a sequential, alternating manner, where *a-type* ADUs usually play the role of request from client and *b-type* ADUs usually play the role of responses from servers. We describe this first type of source-level behavior using the *sequential version* of our a-b-t model, which consists of a sequence of epochs, where each epoch captures one exchange of ADUs (*i.e.*, one a-type ADU and one b-type ADU). The rest of the TCP connections exhibit *data exchange concurrency*, where their endpoints send at least one pair of ADUs simultaneously. We describe this second type of source-level behavior using the *concurrent version* of our a-b-t model, where the ADUs and the quiet times from each endpoint are described independently. The examples from real applications examined in this chapter demonstrated the ability of the a-b-t model to provide a detailed description of source-level behavior for both sequential and concurrent data-exchanges. This means that our approach is able to characterize the source-level behavior of *entire traffic mixes* without any need to understand the specific semantics of each individual application present in the mix.

A fundamental strength of abstract source-level modeling is the possibility of acquiring data from packet header traces in an efficient manner. This is critical to make the approach widely applicable. Packet header traces do not contain any application-level payload, so they are easy to anonymize simply by replacing IP addresses. As a consequence, many organizations have made packet header traces of their Internet links public [nlab]. We proposed a data analysis algorithm that can transform the set of segment headers observed for each connection in a trace into an a-b-t connection vector. The cost of this algorithm is $O(sW)$, where s is the number of segments and W the maximum window size. The algorithm relies on the concept of *logical data order* (*i.e.*, the order of data as understood by the application layer) to robustly handle segment reordering and retransmission. This approach enables us to measure the real size of ADUs at the application level, to distinguish between source-level quiet times and quiet times due to losses, and to identify data exchange concurrency without false positives. We validated this algorithm using synthetic applications, studying the impact of the sizes of socket reads and writes, delays between socket operations and packet loss. The results demonstrated that our data acquisition algorithm is very accurate. Our validation also studied the accuracy of our data acquisition when our basic algorithm is extended with a quiet time threshold to separate consecutive ADUs flowing in the same direction. Even in this case, we only uncovered minor inaccuracies in the measured inter-ADU quiet times when arbitrary delays between socket reads are used and when connections suffered from packet loss.

We concluded the chapter with a statistical analysis of the a-b-t connection vectors in five packet header traces. Three of these traces came from our own data collection effort at the University of North Carolina at Chapel Hill, and the other two traces, Leipzig-II and Abilene-I, came from NLANR's public repository of packet header trace. Before we presented the analysis, we pointed out the need to filter out the following two types TCP connections:

- Connections for which no observed segment carried application data, and therefore had no ADUs. They corresponded to failed attempts to establish a TCP connection (*e.g.*, due to closed ports), denial-of-service attacks (*e.g.*, SYN attacks), and port scanning activity. These connections were very numerous, but they carried an insignificant fraction of the total traffic in each trace. Properly characterizing these "ADU-less" connections is outside the scope of this dissertation.

- Connections for which segments are observed in only one direction. We found a significant number of unidirectional connections only in the case of Abilene-I, since this trace was collected traffic in a backbone network where asymmetric routing was common. Distinguishing between sequential and concurrent connections require to observe both directions of a connection, so we ignored unidirectional connections in our later analysis and traffic generation.

In addition, our statistical analysis of the traces considered only fully-captured TCP connections, those for which we observed both the segment performing connection establishment and connection termination. We therefore ignored partially-captured connections, which contained only partial information about source-level behavior. Our results considered sequential and concurrent connections separately. We can highlight the following observations from these results:

- Every trace showed a small fraction of concurrent connections, at most 3.6%, but they account for a far more substantial fraction of the total bytes, between 18% and 32%. This is consistent with our observation that concurrency can increase throughput, so it is often implemented in bulk applications that transfer large amounts of data.

- Regarding the bodies of distributions of ADU sizes, sequential connections showed a substantial difference between a-type and b-type ADUs. The sizes of 90% of the a-type ADUs were at most 1,000 bytes, while the sizes of 90% of the b-type ADUs were at most 10,000 bytes. The observed differences across sites paled in comparison to this phenomenon. On the contrary, the tails of the distributions appeared similar for a-type and b-type ADUs, being consistent with heavy-tailness in both cases. Concurrent connections did not show a systematic difference between a-type and

b-type ADUs, but their size distributions varied widely for the three sites and also exhibited heavy-tailness. Another interesting observation is that between 80% and 90% of the bytes were carried in ADUs whose size was above 10,000 bytes.

- Regarding the distribution of the number of epochs, we found a large fraction of connections, between 57% and 65%, with only one epoch. However, these connections accounted for a far smaller fraction of the total bytes, between 22% and 38%. Most of the remaining connections had a moderate number of epochs, between 2 and 10. Connections with tens or hundreds of epochs represented only 5% of the connections, but they carried 30% to 50% of the bytes.

- Our joint analysis of ADU size and number of epoch revealed a complex inter-dependency. The average amount of data in an epoch and the median size of ADUs showed substantial variability for different values of the number of epochs in a connection, without any apparent pattern. In addition, the results of the joint analysis are very different across sites. It does not seem possible to develop a simple parametric model for these data.

- Regarding the bodies of the distributions of quiet times, sequential connections showed a larger fraction of durations above 1 second for quiet times on the client side, between a b-type ADU and the a-type ADU that follows it. Quiet times on the server side, between an a-type ADU and the following ADU, were less substantial but also significant. This motivated us to incorporate server-side quiet times on our model. Both distributions showed substantial tails. The difference between the two distributions of quiet time durations appear less significant for concurrent connections.

- A significant percentage of connections, between 65% and 83%, showed a quiet time between the last ADU and TCP's connection termination with a duration above 1 second. This quiet time often increased the duration of the connection dramatically, since connections with little data completed their data transfer very quickly, but remained idle waiting to be closed. This finding justified the addition of a final quiet time duration to our a-b-t model.

- Our comparison of the distributions from the three UNC traces, which were collected at three different times of the day, revealed clear differences in the data. These differences are however less dramatic than those observed when traces from three different sites are compared.