A Protocol for Network Level Caching

by

Edwin N. Johnson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1998

© Massachusetts Institute of Technology 1998. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and distribute publicly
paper and electronic copies of this thesis and to grant others the right to do so.

Author _____

Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by _____

John V. Guttag
Professor and Associate Head, Computer Science and Engineering
Thesis Supervisor

Certified by _____

Ulana Legedza
Research Assistant
Thesis Supervisor

Accepted by _____

Arthur C. Smith
Chairman, Departmental Committee on Graduate Theses

A Protocol for Network Level Caching

by

Edwin N. Johnson

Submitted to the Department of Electrical Engineering and Computer Science
May 22, 1998
in partial fulfillment of the requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis presents Client-side TCP (CTCP), a new transport protocol that supports network level caching of data packets. By doing so, CTCP enables the retransmission of lost packets from network nodes closer to the client instead of only from the server. Currently, TCP exhibits inefficiencies in terms of bandwidth consumption, retransmission latency, and server processing. CTCP attempts to reduce TCP's transmission inefficiencies by both caching individual data packets in the nodes of the network and by shifting the reliability burden from the server to the client. Network level caching, enabled by CTCP, reduces network traffic near the server as well as packet latency. Specific design and implementation details of CTCP are presented, and an extensive probabilistic analysis of both TCP and CTCP shows CTCP's advantages. Analysis shows that network level caching using CTCP leads to a reduction of up to 88% in redundant bandwidth consumed and redundant packet latency.

Thesis Supervisor: John V. Guttag
Title: Professor and Associate Head, Computer Science and Engineering

Thesis Supervisor: Ulana Legedza
Title: Research Assistant

Acknowledgments

I thank John Guttag for being a great thesis advisor. I am glad to have been a part of his research group and I thank him for his good advice on how to write and present research.

I thank Ulana Legedza for her support and encouragement throughout the entire project. Her advice was invaluable in allowing me to complete my research.

I thank the entire Software Devices and Systems group for providing an excellent research environment and making the year fun and enjoyable.

Finally, I thank my friends and family for their never-ending patience and support.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION 7

CHAPTER 2: DESIGN 9

2.1 Routers and Packets 9
2.1.1 Packet Types 9
2.1.2 Packet Interception and Processing 10
2.1.3 Unique Packet Naming 10
2.1.4 Cache Size 11

2.2 Client-side Transmission Control Protocol (CTCP) 12
2.2.1 Server Independence 12
2.2.2 Connection Establishment and Termination 15
2.2.3 Reliable Data Transfer 16
2.2.4 Flow Control 17
2.2.5 Server Buffer State 18

CHAPTER 3: IMPLEMENTATION 19

3.1 Active Network Platform 19

3.2 Capsule Processing 20

3.3 Client-side Transmission Control Protocol 21
3.3.1 CTCP Server Module 22
3.3.2 CTCP Client Application Module 23
3.3.3 CTCP Client Protocol Module 23
3.3.3.1 Flow Control 24
3.3.3.2 Retransmission Timers 25
3.3.3.3 Congestion Control 26
3.3.3.4 Threads 26

CHAPTER 4: EVALUATION 28

4.1 Bandwidth Analysis 28

4.2 Latency Analysis 34

4.3 Reducing Server Load 37

CHAPTER 5: CONCLUSION 40

5.1 Future Work 41
5.1.1 CTCP Functionality 41
5.1.2 Operating System Support for CTCP 41

5.1.3 Limited Caching 42
5.1.4 CTCP Acknowledgments 42

5.2 Conclusion 42

NOMENCLATURE 43

BIBLIOGRAPHY 44

APPENDIX A: CONNECTION ESTABLISHMENT AND TERMINATION 46

A.1 TCP Connection Management 46

A.2 CTCP Connection Management 47

TABLE OF FIGURES

Figure 1: TCP Reliable Transfer Model *13*
Figure 2: CTCP Request-Data Model *14*
Figure 3: CTCP Retransmission Example *15*
Figure 4: Fast Retransmit Algorithm Under TCP and CTCP *17*
Figure 5: Module Diagram *22*
Figure 6: CTCP Client Threading Model *27*
Figure 7: Model of an N-link Network Path *29*
Figure 8: Server Packet Processing *38*
Figure 9: TCP Connection Establishment *46*
Figure 10: TCP Connection Termination *47*

CHAPTER 1: INTRODUCTION

The Transmission Control Protocol (TCP) was developed in the late 1970's to transmit data reliably in the presence of Internet packet loss due primarily to network congestion. This protocol later became the standard transport protocol for the Internet. TCP and other reliable transport protocols handle lost packets by having the sender detect the loss and then retransmit the lost packet. TCP also uses a congestion control algorithm to dynamically react to changing bandwidth limits of the Internet.

While TCP's sender-based packet retransmission and congestion control guarantee reliable transfer of data, they exhibit three inefficiencies. First, since packets can be lost at any point between the server and the client, packets that are retransmitted end up traversing at least some parts of the network twice. This extra traffic is typically in the form of large data packets and consumes extra bandwidth. Since TCP connections must share the bandwidth available on the Internet, this extraneous bandwidth consumption degrades the throughput on all data transfers. Second, TCP adds to the latency of lost packets by requiring the server to detect the packet loss and retransmit the packet over a portion of the network that the packet has already traversed. High latency of a single packet is affects the quality of interactive sessions more than the high throughput of a bulk data transfer. Lastly, the server must spend processing time to both detect and retransmit lost packets and to perform congestion control. This extra processing burdens the server and limits the number of clients that the server can handle.

This thesis explores the idea of using network-level caching to combat these inefficiencies of TCP. Network level caching is the process of saving individual data packets in the nodes of the network. This allows data retransmission to originate from a network node instead of from the server. The node that is closest to the client that has the packet in its cache is chosen as the source of the retransmission. Consequently, network level caching reduces the amount of network traffic near the server. In addition, the overall retransmission latency may be reduced, as the packet does not need to travel the entire path from the server to the client. Existing protocols such as Snoop TCP [2] have used network level caching to improve throughput over wireless networks. This thesis expands the node caching idea to standard

wired networks. Our analysis shows a reduction of up to 88% in redundant bandwidth and redundant packet latency with network node caching.

To support the concept of network level caching over standard networks, a new transport protocol is needed. This thesis describes the development of a new protocol, Client-side TCP (CTCP), that supports and uses network level caching. Under CTCP, the burden of ensuring reliable transmission is shifted from the server to the client. To do so, CTCP replaces TCP's Data-Acknowledgment network communication scheme with a Request-Data scheme. Instead of the server sending another data packet to the client if the client's acknowledgment is not received, the client sends another request to the server if the data is not received. Specific design and implementation details are discussed, and an analysis of both TCP and CTCP shows CTCP's advantages. CTCP is shown to support network level caching and to reduce the load on the server by requiring less processing per packet received by the server.

Chapter 2 presents the design of CTCP, including network requirements such as appropriate cache sizes and motivations behind particular design choices. The prototype implementation of CTCP is covered in Chapter 3, which discusses the underlying network platform and the packet processing performed. Chapter 4 evaluates and compares the performance of TCP and CTCP. Lastly, Chapter 5 concludes with a summary and discusses future research directions.

CHAPTER 2: DESIGN

In order to incorporate network-level caching into reliable data transfer, network nodes need to have the ability to cache and retransmit data packets. This new network functionality requires the node to examine all packets, a unique name identifying each packet, and the ability to retransmit independent from the server. Other design considerations that are discussed include connection establishment and termination, reliable data transfer, and server buffer state.

2.1 Routers and Packets

The most fundamental requirement of network node caching is the existence of caches on network nodes (or routers) and new packet processing routines that access the cache. Currently, routers are designed simply to route packets from their source to their destination. In particular, traditional routers do not contain caches and do not allow additional packet processing to be specified.

There are two solutions to the problem of limited existing router functionality. One choice is to build new routers that incorporate the requirements for network level caching. Alternatively, a programmable network, such as an active network, could be used. The choice between these two options is an implementation issue discussed in Chapter 3. Regardless of the implementation chosen, the design solutions presented below assume the presence of a cache in the router and an ability to specify packet processing actions.

2.1.1 Packet Types

In general, there are two types of packets used by reliable transport protocols: data packets and control packets. Data packets contain the user data being transferred from the sender to the receiver. In addition to the data, data packets contain a name, usually represented as a large integer, to identify the data contained within the packet. This name is located in the header of the data packet and is intended to be unique for each data packet.

Control packets transmit protocol information between the sender and the receiver. This information is used to handle things such as connection management, network reliability, and congestion control. Depending on the transport protocol, control packets used for reliability can be one of two types: Requests or Acknowledgments. Request packets request more data to be sent. Acknowledgment packets verify the receipt of data while implicitly requesting more data to be sent. Both Requests and Acknowledgments also contain the identifier of the data packet to which they refer.

2.1.2 Packet Interception and Processing

Given that we can specify additional packet processing for routers to perform as mentioned above, we modified the nodes to be able to cache data packets and to retrieve them when a control packet is received. We decided to cache all data packets that are transmitted through them. Caching every data packet means that our routers will take slightly longer to forward every packet than traditional routers because even those that are unlikely to need retransmission are still cached. A more efficient alternative (discussed in more depth in Section 5.1.3) might be to cache packets only when data loss is relatively frequent.

When a packet is received by a router, the router identifies the packet as either a data or a control packet. Once the packet type is identified, the router chooses an action based on what type of packet was received. If a data packet is received, the router extracts the name from the packet and stores the data in the cache using the name as a lookup key for later retrieval. For control packets such as Requests, the node again retrieves the name from the packet. This name is then used as a key to lookup in the cache. If the key is present, a new data packet is constructed using the data from the cache that was associated with that key. This data packet is then retransmitted to the client. If the name was not present in the cache, then the router simply forwards the control packet onto the next node towards the server using the routing tables.

2.1.3 Unique Packet Naming

In order to successfully cache data as a key-value pair, unique names are needed for each packet that is transmitted through the network. Without a naming scheme, the nodes in the middle of the network would be unable to distinguish between packets containing data from different documents.

TCP uses a single name, called a sequence number. This sequence number uniquely names a packet within a connection. Each consecutive data packet receives a consecutive number. CTCP also uses unique sequence numbers to refer to the data packets of a particular connection.

In a network level caching scheme, however, it is not sufficient to use only sequence numbers to uniquely identify data packets. Since data packets are present in the node's cache, they may be accessed by any connection that is routed through that node. Without further identification, a packet from the wrong document may be retrieved from the cache because of a collision of sequence numbers.

Because of the need for a name unique for each data transfer, CTCP requires the server to assign an identifier to the document when the connection is established. The server generates this document identifier by calculating a hash of the document's contents. While a hash function does not guarantee uniqueness, it reduces the probability of collision to 1 in 2^n, where n is the size in bits of the hash. The hash size used for CTCP is 64 bits resulting in a 1 in 2^{64} probability of collision. Hashing the document's contents allows the server to assign the same name to a document until the contents of the document change. This dynamic naming scheme reduces the likelihood of receiving stale data from a cache. In addition, using an identifier based on the contents of the document allows for the possibility of using cached data for subsequent connections. However, the probability of this occurring depends on the size of the cache and is expected to be small, as described in Section 2.1.4.

CTCP uses the combination of the document identifier and the sequence number as the packet's unique name. Every packet that is transmitted by CTCP contains both of these names thus allowing the routers to uniquely cache data packets using the names as a unique lookup key.

2.1.4 Cache Size

An important design parameter is the size of each node's cache. The size of the cache determines the length of time that packets will stay in the cache before being replaced by new packets. The cache should be small for a cheaper implementation, but must be large enough to hold packets long enough for lost packets to be retransmitted. If packets are overwritten in

the cache before they are needed, then the cache is not being used efficiently. The optimal size of a node cache in terms of price and performance is one in which packets stay in the cache just long enough for retransmission to occur if necessary. If retransmission is not necessary, then the packet will be replaced by newly arriving packets.

The optimal cache size can be calculated by multiplying the bandwidth of the router by the time that it takes the transport protocol to detect and retransmit a lost packet. This represents the number of packets that will be routed by the node before it is safe to assume that the packet was successfully delivered. For both TCP and CTCP, this length of time is approximately the round-trip time. While the optimal cache size depends on the specific router's bandwidth capabilities, several "back of the envelope" calculations can be made. High-end Cisco routers currently range from speeds of 155 Megabits per second to 2.4 Gigabits per second [3]. Round-trip times for typical Internet paths range from a few milliseconds to hundreds of milliseconds depending on traffic levels. Multiplication of these parameters leads to cache sizes of 1MB-200MB. The same Cisco routers currently have 64MB-256MB of memory for routing tables and 32MB-128MB of memory for buffering incoming packets. These numbers indicate that caches are already feasible on today's routers.

2.2 Client-side Transmission Control Protocol (CTCP)

A transport protocol that supports network level caching needs to have the ability to retransmit data packets from network nodes instead of solely from the server. TCP, the transport protocol currently used on the Internet, was found to be inappropriate for this node retransmission task. Therefore, in order to take advantage of the new network caching service described in Section 2.1, a new transport protocol was developed.

2.2.1 Server Independence

CTCP uses a packet retransmission scheme that is independent of the server. This is a significant departure from TCP's communication scheme. TCP guarantees reliable data transfer using a Data-Acknowledgment (or Data-Ack) model. TCP requires the client to acknowledge all data received from the server by transmitting an Ack packet. The server, in turn, is in charge of making sure the data (via Data packets) arrives at the client. A simplified picture showing these Data-Ack operations is presented in Figure 1. The server (S) sends Data

packet #1 and #2 to the client (C) via nodes 1, 2, and 3. The client acknowledges each of the data packets as they are received. If Data Packet #2 is lost, the server must retransmit the packet once its loss is detected. TCP's operations are therefore directly tied to the server's actions as the server is responsible for reliable transmission to the client.

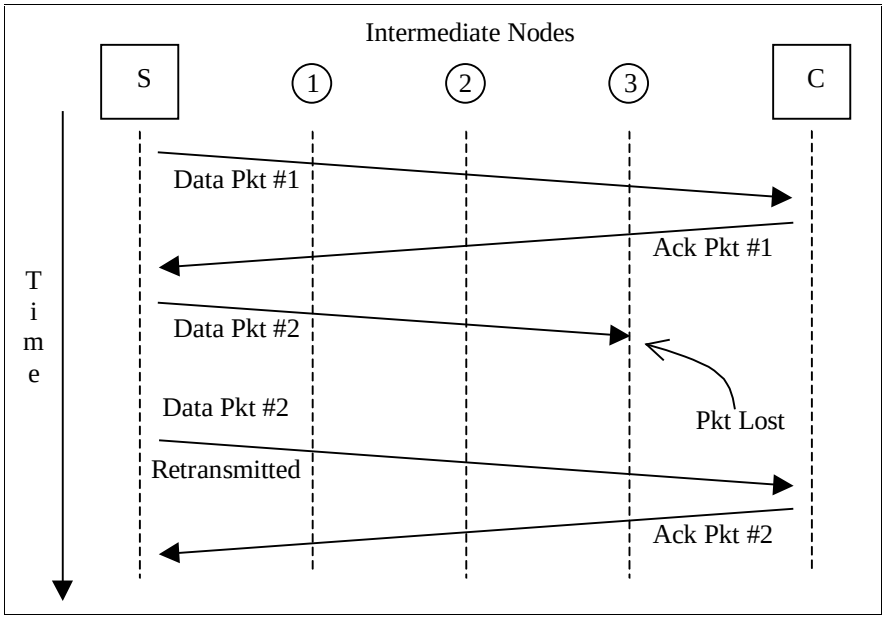

Figure 1: TCP Reliable Transfer Model

Although TCP is dependent on the server, it could be used in a network node caching scenario. Since the server is responsible for detecting packet loss under TCP, the server would have to tell the intermediate node with the cached data that the data had been lost. This loss information would then cause the intermediate router to retransmit the packet in order to use the cached data. However, to inform the router, the server would have to send some sort of control packet to the node. This would reduce network traffic, since control packets are typically much smaller than data packets. However, the retransmission problems of increased latency and server overloading problems are not addressed by this scheme. CTCP addresses these problems by removing the server from the retransmission process entirely.

In order to make retransmit lost packets without server intervention, CTCP transfers the burden of reliable transmission from the server to the client. CTCP uses a Request-Data model to transmit data across a network. The client sends Request packets to the server and

the server replies with Data packets. Figure 2 shows an example of CTCP's explicit request mechanism. Data Packet #1 is requested by the client and the server responds.

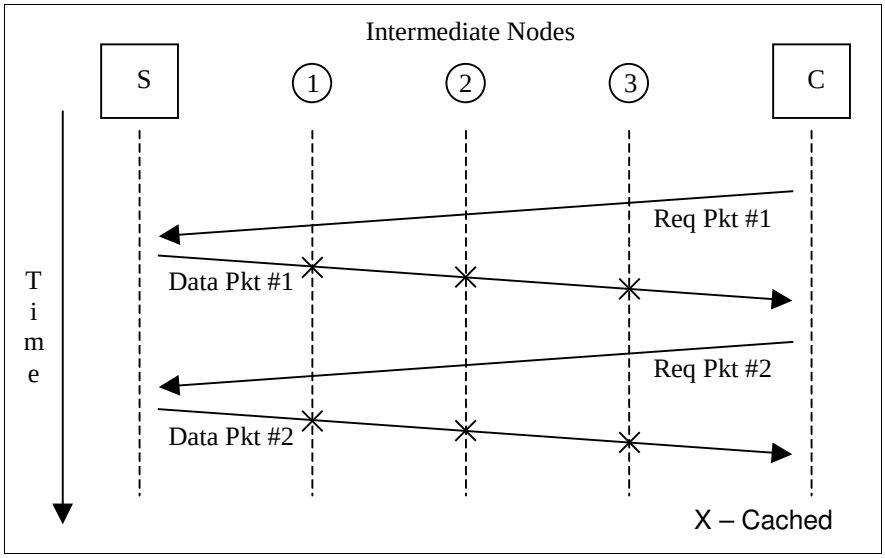

Figure 2: CTCP Request-Data Model

The advantage of having an explicit Request-Data model is that intermediate nodes are allowed to short-circuit the round-trip client-to-server loop and satisfy the requests with data located in their cache. Figure 3 shows the client sending Request #1 asking the server for Data packet #1. Data packet #1 is successfully sent to the client in response. The client then asks for the next data packet via Request #2. Data packet #2 is lost before it is received by the server. The client detects the lost packet (described below in Section 2.2.2) and requests the data again. This second request is simply routed by node 3 since it does not have a copy of the correct data packet in its cache. Node 2 is able to intercept the request and retrieve Data packet #1 from its cache to be retransmitted to the client. For this retransmission procedure to work effectively, it is assumed that the Request packets are routed along the same network path as the Data packets. If the routing is asymmetric, then the Requests will be retransmitted back to the server without finding the cached data. Assuming a symmetric network path, under CTCP, each node effectively acts as a server for the data packets contained within their cache and independence from the original server is achieved

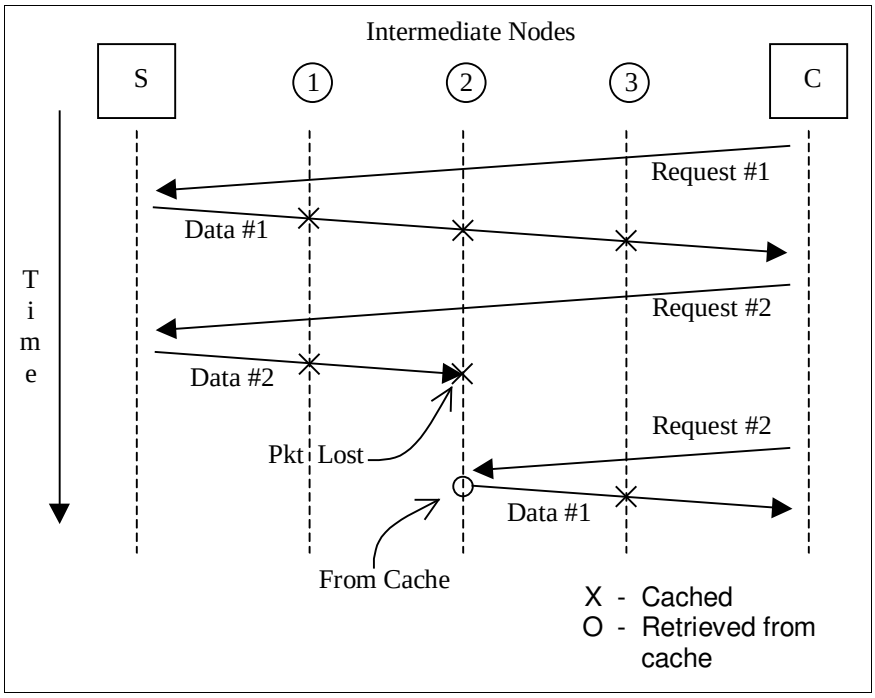

Figure 3: CTCP Retransmission Example

2.2.2 Connection Establishment and Termination

CTCP's client-server connection establishment is similar to that of TCP. To initiate a connection between the client and the server, several packets are exchanged prior to any transmission of data. CTCP uses a three-way handshake to establish a connection between the client and server. The primary difference between CTCP's and TCP's connection establishment is that in CTCP, the client is required to send the filename of the document being requested as protocol information. This information is contained in the header fields of the initial packet sent from the client to the server. The filename is part of the protocol so that a document identifier can be generated by the server as discussed in Section 2.1.3. The fact that a filename is required to initiate a connection means that CTCP is dependent on the existence of a name for the data to be transmitted. TCP, on the other hand, does not include a filename as part of the connection establishment and therefore does not have this dependence.

As with connection establishment, a few packets are exchanged to terminate a connection once the data is reliably received. In CTCP, since the client is in charge of retrieving the data from the server, the client is also responsible for terminating the connection from the server.

TCP allows the client and server to terminate its half of the connection independently of each other. In practice, however, a TCP server initiates the connection termination by closing its half of the connection and then, upon receiving the server's termination packets, the client does the same. Further details on connection establishment and termination can be found in Appendix A. CTCP terminates a connection by having the client send a termination packet to the server. The server then replies by sending an acknowledgment back to the client. The connection is closed once the client receives the acknowledgment from the server.

2.2.3 Reliable Data Transfer

As a transport protocol, CTCP must provide reliable data transfer between two end-points of a connection. CTCP guarantees this reliability in the same fashion as TCP: by detecting a lost packet and then retransmitting it.

In order for TCP to discover packet loss, the server maintains a timer based on an estimation of how long it takes a packet to travel from the server to the client and back again. This estimation is dynamically obtained by monitoring the time between sending a data packet and receiving an acknowledgment for the data. The server uses this round-trip time estimation to set a conservative timer each time a packet is sent. The timer is reset whenever any packets are received. If this retransmission timer ever goes off, then the server assumes that the packet was lost and responds by retransmitting the data.

In addition to simple timeouts, TCP also incorporates the Fast Retransmit algorithm developed by Jacobson [5] to avoid idle timeout delays. TCP's Fast Retransmit algorithm is show schematically in the left-hand side of Figure 4. Data Packet #2 is lost creating a hole in the data received by the client. Given that TCP allows only a single cumulative Ack, the client continues to acknowledge Data Packet #1 upon receiving Data Packets #3, #4, and #5. Once the server receives the third duplicate Ack #1, the Fast Retransmit algorithm takes effect. The server assumes that Data packet #2 was lost and retransmits it. The Fast Retransmit algorithm under TCP states that if the server receives three duplicate Acks in a row, then the server should assume that the packet was lost and not simply delayed and retransmit the packet.

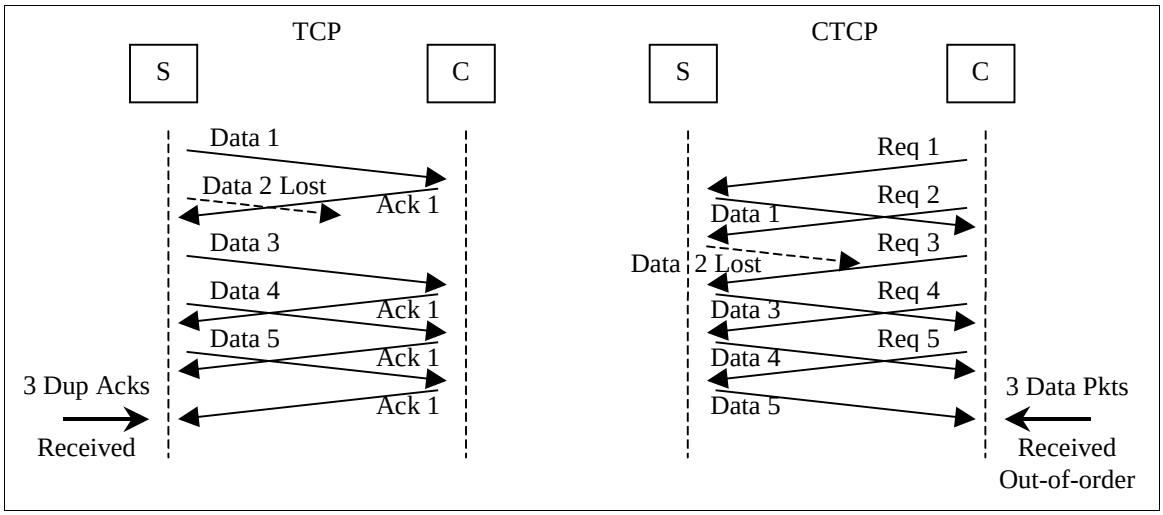

Figure 4: Fast Retransmit Algorithm Under TCP and CTCP

Like TCP, CTCP also keeps track of round-trip time estimators as a way to discover packet loss. The CTCP client monitors the amount of time between sending a request and receiving the data for the request and uses this data as an estimation of round-trip times. Timers are set based on these estimations in order to detect lost packets. As a second detection mechanism, CTCP also has a Fast Retransmit algorithm. Since there are no acknowledgments, the client assumes a packet has been lost if it receives three or more Data packets that are out of order. The right hand side of Figure 4 shows this Fast Retransmission scenario. Data packet #2 is lost on its way from the server to the client. The client, unknowingly, continues to request more data. Once the client receives Data packets 3, 4, and 5, without receiving Data packet #2, the client assumes that it was lost. Once a lost packet is detected, retransmission is straightforward—the client simply retransmits the Request to the server asking for the lost data.

2.2.4 Flow Control

Flow control prevents a slow client from being overrun by a fast server. Since the client only has a fixed amount of buffer space for incoming packets, the client must restrict the rate at which the server can send data. In TCP, the client advertises the amount of buffer space that it has available in every acknowledgment sent to the server. This advertisement is called the flow control window.

In CTCP, the client manages the flow control and the transmission of the packets. Therefore, the client is not required to inform the server of its flow control window. Instead, the client simply limits the number of requests sent to the server. Therefore, to implement flow control, the CTCP client only requests the number of data packets that can be saved in its internal buffer.

2.2.5 Server Buffer State

A TCP server only needs to maintain enough buffer space to hold the portion of the document it is trying to send. This buffer can then be reused for the next portion of the document after all the data that has been sent is acknowledged. The initial design of CTCP, however, requires a CTCP server to maintain a larger buffer. CTCP doesn't currently allow the client to inform the server about the receipt of data. The client can only request more data. Since the server can't tell if the client has received any of the data, it cannot get rid of the buffer containing the document data. Therefore, the server must hold onto the entire document until the client closes the connection. The solution to this problem is to have the client acknowledge received data to the server in addition to requesting new data. This would allow the server to manage its buffer state more intelligently similar to TCP.

CHAPTER 3: IMPLEMENTATION

This section describes the current prototype implementation of CTCP. It is written in Java and runs on an ANTS-based active network. The network programmability provided by ANTS was crucial for implementing the packet caching and retrieval that CTCP requires to be performed at the network nodes. Java provided a high-level language thereby aiding in development, testing, and debugging.

3.1 Active Network Platform

The purpose of designing and implementing CTCP is to take advantage of caching at the network level. As mentioned in 2.1, in order to cache packets in routers, it is necessary to have the ability to specify additional processing for the router to perform. In addition, routers need to dedicate memory as a cache for data packets. The routers in the Internet do not contain either of these capabilities. In fact, the role of computation within traditional packet networks, such as the Internet, is extremely limited. Although routers in traditional networks may modify a packet's header, such as the time-to-live field, they transfer the user data opaquely without examination or modification.

One solution to limited functionality within current router technology is to build new routers that contain all of the new capabilities desired. This option was discarded as being impractical and prohibitively expensive for experimentation.

Current research in active networks presents a more flexible alternative. In an active network, the routers of the network are programmable and perform customized computations on the packets travelling through them. These networks are active in the sense that nodes can perform computations on, and modify, the packet contents. While active networks do not perform caching at the network level by default, their programmable nature allows the desired functionality to be added. The active network solution was chosen because of its inherent flexibility and because of its availability in our lab. As such, CTCP was implemented on an active network prototype called ANTS (Active Network Transfer System) [13].

ANTS provides two important functions with respect to network level caching. First, ANTS allows the protocol to define a packet processing (or forwarding) routine for each packet type in the protocol. This combination of packet data and packet processing routine is called a capsule in ANTS terminology. Specifically, when a capsule arrives at an active node, the node executes the processing routine that is associated with the type of capsule received.

The second major feature of ANTS is the existence of node storage organized as a soft-state cache. The cache is available from the nodes as a `NodeCache` object. Two methods exist in order to operate on the cache: `get()` and `put()`. The `put()` method allows objects to be inserted into the cache as a key-value pair, while the `get()` method retrieves the objects associated with the specified key. These methods were used by the capsule processing routines to cache and retrieve data.

3.2 Capsule Processing

Implementing CTCP required the definition of two types of capsules: Request capsules (`CTcpReqCapsule` objects) and Data capsules (`CTcpDataCapsule` objects). Both of these capsule objects implements a processing routine, called `evaluate()`.

Whenever a node receives a capsule, it calls the `evaluate()` method on whichever type of capsule is received. Calling `evaluate()` allows CTCP to respond to the receipt of both Request and Data capsules by each active node.

Data capsules contain the user data to be transferred and are sent from the server to the client. When a Data capsule is received by an active node, it calls the forwarding routine for `CTcpDataCapsule`. Instead of simply routing the capsule toward its destination, the `evaluate()` method saves a copy of the Data capsule in the node's cache. As an optimization, the capsule is not cached at the client or server nodes since these two nodes represent either the source or the destination of the data.

Request capsules contain the control information used to manage the reliable transfer of data. The client sends Request capsules to the server to ask for data. The server is in charge of responding to the client's requests. The forwarding routine for a Request capsule performs the other half of the caching procedure. The `evaluate()` method for a Request capsule looks in

the node's cache for the requested Data capsule. If an entry is found, then the `CTcpDataCapsule` is retrieved from the cache. This new Data capsule's source and destination addresses and ports are set according to the received request. The `CTcpDataCapsule` object is then routed by the node. If no corresponding entry is found in the cache, then the request capsule is simply routed onto the next node.

3.3 Client-side Transmission Control Protocol

While the caching performed at network nodes serves to improve the protocol's performance, it is the endpoint processing at the client and server that ensures the correctness and reliability of CTCP. In the current CTCP implementation, only one-way data transfers from the server to the client are implemented. In this model, the client asks for a document and the server sends it to the client. This is in contrast to a more general two-way model that allows data to be sent in either direction. In a full implementation, however, the server and client modules would be integrated to support the two-way data transfer model.

The server needs to take requests from the client, obtain the requested data, and respond by sending out a Data capsule. This functionality was implemented as a single module, called `CTcpServerApplication`, shown in the right side of Figure 5.

A CTCP client has many more duties than the server. The client generates requests and sends them to the server, while setting a timeout timer when a capsule is sent. The client also has to reorder any data that is received out of order and drop any duplicate data that is received. To maintain independent application and protocol functionality, the client implementation was divided into an application level and a protocol level, shown in the left side of Figure 5.

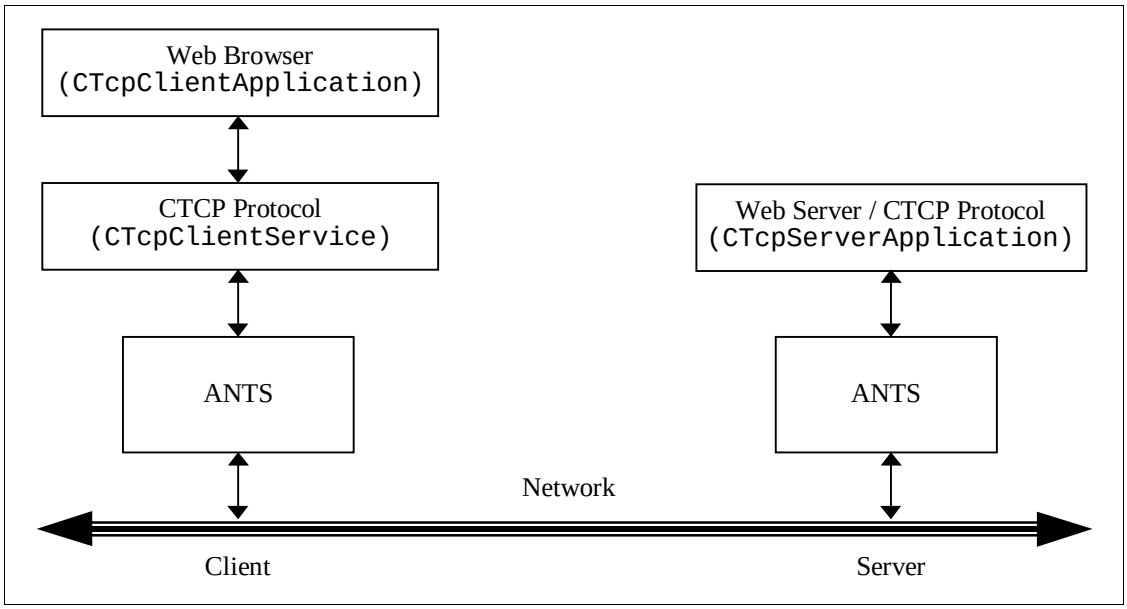

Figure 5: Module Diagram

3.3.1 CTCP Server Module

The primary responsibility of the server is to respond to a client's requests. When the server receives a Request capsule from the client, it responds with a Data capsule. The server also handles connection establishment and termination (described in general in Section 2.2.2 and in more detail in Appendix A). The single object implementing these duties, called `CTcpServerApplication`, is a small and simple object since most of CTCP's functionality is performed by the client.

The server was implemented using a single thread of control. This was possible because the server only requires a request-response type of action. For every Request capsule received by the server, the server replies with a Data capsule containing the requested data.

In order to be capable of serving many clients simultaneously, the server maintains a list of connections that are open. The server keeps track of connections by using a Java object called `CTcpServerConnection`. Each connection object is uniquely identified by three fields: the client address, the client port, and a connection identifier chosen by the client. Every packet sent from the client to the server contains these three fields allowing the server to uniquely identify the connection for a given capsule.

In order to receive capsules from the underlying network, ANTS requires the server to implement the `receive()` method. In the CTCP server, the `receive()` method determines which connection the capsule is for by searching through the connection list. Once `receive()` determines the connection, it determines the type of incoming capsule. If the capsule is a Request capsule, the function `recvRequest()` is called. Except for capsules required to initiate and terminate connections, the server should only receive Request capsules. Any other capsules received generate an error.

The `recvRequest()` procedure performs the protocol portion of the server. First, it verifies that the data requested by the capsule is within the range of the actual data. This verification is accomplished by comparing the requested sequence number to the size of the buffer of user data stored in the connection object. After verifying that the data requested is valid, a new Data capsule is created by the server. The data is then copied into the capsule and is sent back to the client.

The CTCP server module is therefore seen to accept requests and generate valid data packets by using only two primary functions: `receive()` and `recvRequest()`.

3.3.2 CTCP Client Application Module

The client half of the CTCP implementation was written as two separate modules. A protocol level object (`CTcpClientService`) provides the reliable transfer capability to the application object (`CTcpClientApplication`).

In the prototype implementation of CTCP, we have written a simple application that allows the user to retrieve a file from the server. The user is allowed to specify both the name of the file to retrieve from the server and the name under which to save it on the local system.

3.3.3 CTCP Client Protocol Module

The CTCP client protocol module performs multiple functions. First, the protocol object is responsible for reliably transferring data from the server. Second, it provides an interface to this reliable transfer mechanism to applications.

The client application is provided access to the client protocol object using two methods: `connect()` and `recv()`. These methods allow the application to create a connection to a server and to transfer the data.

The `connect()` procedure initiates a connection from the client to the server. The application specifies the IP address and port number of the server as well as the path name of the file to be retrieved, as arguments to the procedure `connect()`. The `connect()` method connects to the server using the connection establishment sequence discussed in Section 2.2.2 and Appendix A. A connection identifier is returned to the application in order to distinguish between distinct connections.

The client service object was designed to support use by many clients having many simultaneous connections open. Each connection's information is stored as a client-side object called `CTcpClientConnection`. These connection objects contain many fields including the server address and port number, a buffer containing received data, round-trip time estimators, and congestion control information.

Once the client is connected to the server, the application begins the retrieval data from the server by calling the protocol object's `recv()` method. The connection identifier returned from the `connect()` call and a user buffer in which to receive the data are passed as arguments to the `recv()` method. Once `recv()` is called, the client begins to request data from the server and does not return from the `recv()` method until either all data is received from the server or the user buffer is filled.

During the call to `recv()` the client's tasks can be broken down into three main categories: packet flow control, retransmission timers, and congestion control.

3.3.3.1 Flow Control

Flow control restricts the number of outstanding requests that can be sent to the server, as limited by the client's buffer space. Unlike TCP, flow control is implemented entirely within the client in CTCP. The flow control window is simply the amount of unallocated buffer space in the client. Each outgoing request allocates enough space in the CTCP client's internal buffer to hold the requested data thereby reducing the flow control window. When a Data

capsule is received by the client, it copies the data from the capsule into the buffer space allocated by the corresponding Request capsule. Later the data is copied from the client's internal buffer to the user buffer supplied by the application in the call to `recv()`. At this point, the buffer space previously containing the data is marked as unallocated in order to be reused by more incoming data capsules. Since more buffer space is now available, the client's flow control window is increased allowing more requests to be sent to the server.

3.3.3.2 Retransmission Timers

In the CTCP implementation, retransmission timers are implemented using a counter for each connection. Each counter represents the number of half-seconds that has elapsed since a packet was sent or received by the client. Every 500ms, the client increments each connection's counter and compares their values to the connection's timeout threshold. If a counter is larger than the connection's timeout value, the CTCP client determines that a capsule was lost somewhere and retransmits the appropriate request.

To set the timeout value for each connection, the client needs an estimate of how long it takes a data packet to travel from the client to the server and then for a response to travel back from the server. This estimate, called a round-trip time estimate, is calculated by the client using a method similar to TCP's timestamp option [11, 12]. Each Request capsule the client sends to the server is stamped with the current time before transmission. The server is then responsible for echoing this timestamp on the Data capsule sent in reply. When this capsule is received, the client uses the difference from the current time and the capsule's timestamp to maintain an estimate of the maximum round-trip time necessary. The estimate is recorded in two parts: the smoothed average and the mean deviation. The first step to maintaining these statistics is to calculate the error by subtracting the newly measured round-trip time from the previous value for the average. The smoothed average is then calculated by taking the previous average and adding one eighth of the error. The mean deviation is updated by adding in one fourth of the difference between the absolute value of the error and the previous value for the mean deviation. The retransmission timer is then set for a conservative value according to these statistics. The new timeout value is set to the average plus four times the mean deviation. By setting the timer for this conservative value, the client reduces the probability of performing an unnecessary retransmission due to the network delaying packets.

3.3.3.3 Congestion Control

The CTCP client implements congestion control using the same algorithm used by TCP [11, 12]. The client maintains a congestion window (`cwnd`) and a slow start threshold (`ssthresh`) for each connection. The congestion window limits the number of requests sent to the server in the same fashion as the flow control window.

Whenever the `cwnd` is less than `ssthresh`, the client performs slow start. During slow start, the `cwnd` is incremented by one for every data capsule received by the client. This causes the client to increase its requesting rate exponentially. If `cwnd` is greater than `ssthresh`, the client performs congestion avoidance during which `cwnd` is incremented by `1/cwnd` for each data capsule.

When congestion occurs (indicated by either a timeout or the reception of out of order data), one-half of the current window size (with a minimum of two) is saved in `ssthresh`. In addition, if the congestion is indicated by a timeout, `cwnd` is reset to one capsule thereby forcing slow start.

3.3.3.4 Threads

Because of the necessity of monitoring timeouts, the CTCP client was not implemented using a single thread. ANTS was designed to give control to the protocol whenever a capsule was received by calling the protocol's `receive()` method. In order to implement functionality that occurs independently of receipt of capsules, it is necessary to use a separate thread of control. The CTCP client protocol object needs a second thread since it may be required to retransmit capsules even while capsule aren't being received. Figure 6 summarizes the threading model used in implementing the CTCP client. The node thread calls `receive()` every time a capsule is received. ANTS imposes the constraint that while a capsule is being processed by a protocol's `receive()` method, no other capsules can be received. This constraint necessitates a quick return from `receive()`. Therefore, the `receive()` method simply copies the data from the Data capsule and updates the connection's sequence numbers. In particular, no capsules are sent from within the call to `receive()`. Instead, a second thread, the protocol thread, executes a loop of monitoring the timeouts and sending any data necessary. This thread originates from the application's call to `recv()`.

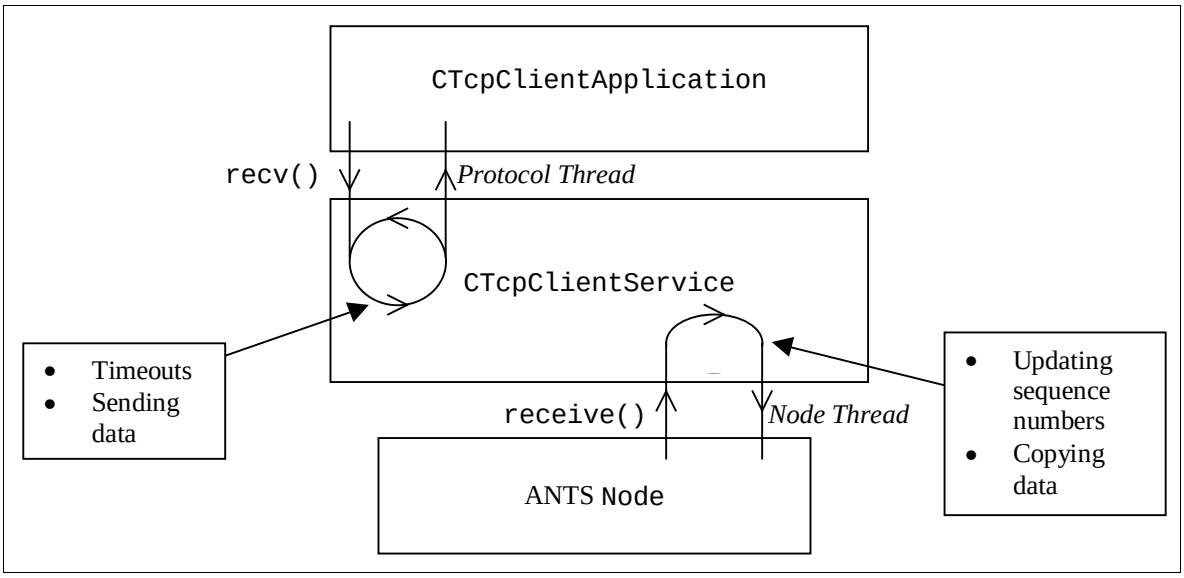

Figure 6: CTCP Client Threading Model

CHAPTER 4: EVALUATION

Client-side TCP and network node caching were evaluated in three areas. First, a theoretical analysis was made of the effectiveness of network node caching in typical data transfers regarding bandwidth consumption. This type of analysis was then repeated for end-to-end packet latency improvements. Lastly, the packet processing routines of both CTCP and TCP were compared.

4.1 Bandwidth Analysis

One of the goals of network node caching is to reduce the traffic in the network links nearest the server. This reduction results from the fact that when data packets are cached at each node along the path between the client and the server, they do not need to be retransmitted from the server if they are lost. In order to analyze the effectiveness of network node caching on bandwidth, a probabilistic model of packet loss in both CTCP and TCP data transfers was made. This model was used to determine, for both TCP and CTCP, the expected number of hops taken by each packet. Our analysis shows that CTCP transmits packets over fewer hops than TCP thereby reducing the bandwidth consumed by each packet's transmission.

In order to compare the bandwidth utilized by TCP and CTCP, a simple model of network packet loss was developed. The model is pictured in Figure 7. It shows a server (S) connected to a client (C) via n-1 intermediate routers. These routers form the path of n links traversed by packets sent from the client to the server and vice versa. It is assumed that for a given TCP or CTCP connection, the network path would stay the same for the duration of the connection. At each link, there is a probability that a packet is lost due to a congested router. It is assumed that this probability of loss (P) is the same for each link as well as for each packet traversing the link. In addition, it is assumed that packet losses are independent of each other. In reality, packet losses tend to occur in bursts causing many packets to be lost in a row. However, bursts of packet losses are accounted for by using the average packet loss over an entire connection.

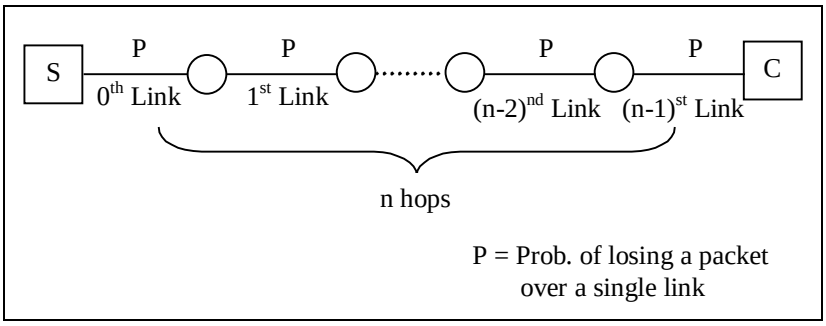

Figure 7: Model of an N-link Network Path

This model represents a Bernoulli process of n trials of losing a packet with probability P. As such, the relationship between P and P_{path}, the probability that a packet is lost somewhere along the path, is:

$$\begin{aligned} P_{path} &= P + (1-P)\cdot P + (1-P)^2 \cdot P + \ldots + (1-P)^{n-1} \cdot P \\ &= P\sum_{i=0}^{n-1}(1-P)^i \\ &= 1-(1-P)^n \end{aligned} \qquad (1)$$

Data from Internet traffic studies done by Paxson [7, 8] was used to determine realistic values of P and P_{path}. These studies analyze traffic of TCP transfers along various paths through the Internet, including the Trans-Atlantic link between the U.S. and Europe. They indicate that approximately 50% of all connections do not experience any packet loss at all while the other half lose at least one packet. The patterns of packet loss demonstrated an unsurprising diurnal behavior of periods of high packet loss during working hours and little or no loss during periods late at night, especially early morning hours. The 50% of connections that exhibited packet loss averaged 4-17% packet loss. The higher packet losses tended to occur on the Trans-Atlantic link between the U.S. and Europe. Purely domestic U.S. connections exhibited a 4.4% packet loss rate.

Bounds for the network path length (n) were determined by measuring some common Internet paths of various geographic lengths. For example, the average path length between a computer in MIT's Laboratory of Computer Science and MIT's web server is 5 links while the distance between MIT's web server and Berkeley's web server is 14 links. Trans-Atlantic

network paths tend to be only a little longer in terms of hops since the ocean is bridged by a single link. For this analysis, however, attention is focused on U.S. domestic connections. Given Equation 1 and Paxson's results of 4.4% packet loss over a domestic U.S. network path (equal to P_{path}), the probability of losing a packet over a single link (or P) is 0.89% for short network paths (n=5) and 0.30% for longer paths (n=15).

The goal of this bandwidth analysis is to compare the total number of redundant hops (TRH) of a packet transmitted using TCP to that of a packet transmitted using CTCP. Total redundant hops is defined as the number of links a packet traverses more than once. In the normal case where a packet is not lost, the TRH is equal to zero. However, the expected value for TRH is greater than zero due to packet loss. We also define and calculate total hops (TH), or the total number of hops a packet must take to reach the client, for both TCP and CTCP. TH is larger than TRH by the length of the network path ($\mathrm{TH} = \mathrm{TRH} + \mathrm{n}$) because TRH disregards the n non-redundant hops that are necessary to get the packet to the client and only counts the redundant ones. While TH is a measure of the overall benefits of network level caching, we focus on TRH since redundant hops represent the inefficient use of bandwidth that can actually be eliminated.

We calculate the expected TRH by counting the number of times a packet must be retransmitted over each link due to packet loss. In addition, the number of times a packet must be retransmitted over the links nearest the server is calculated as it shows the greatest discrepancy between TCP and CTCP for a single link. Therefore, the link closest to the server is the link that would receive the most benefit by using network level caching.

To calculate the total number of redundant hops for a packet, a summation of the number of times the packet is retransmitted over each link is computed. Since a probabilistic model is being used, the actual number of retransmissions is unknown, so the expected number is used instead. The formula for total hops is:

$$\mathrm{TRH} = \sum_{i_0=0}^{n-1} \mathrm{E}[r | i = i_0] \qquad (2)$$

where TRH is the total number of redundant hops a packet must make to reach the client, r is the number of retransmissions, i is the number of the link, and $\mathrm{E}[r|i=i_0]$ is the expected number of times a packet must be retransmitted over the i^{th} link due to the packet being lost further down the path.

The definition of expected value leads to the following equation for $\mathrm{E}[r|i=i_0]$:

$$\mathrm{E}[r|i=i_0] = \sum_{r_0=0}^{\infty} r_0 \cdot P_{rtx,link}(r_0, i_0) \tag{3}$$

where r is the number of retransmissions and $P_{rtx,link}(r,i)$ is the probability of retransmitting a packet r times over the i^{th} link and is given in Equation 4.

$$P_{rtx,link}(r,i) = P_1 \cdot P_2 \tag{4}$$

where P_1 is the probability of losing a packet r times over links past the i^{th} link (e.g. i+1, i+2, …, n), and P_2 is the probability that the packet is then successfully transmitted through the rest of the network to the client.

Because of their differing retransmission strategies, P_1 and P_2 take on different values for TCP and CTCP. TCP must retransmit a packet from the server regardless of where the packet is lost in the network. This leads to the links nearest the server seeing more retransmitted packets than links close to the client. Thus P_1^{TCP} equals the probability that r packets are lost in the portion of the network after the current (or i^{th}) link. P_2^{TCP} equals the probability that the packet is transmitted successfully over the rest of the network (e.g. the links from the i^{th} to the $(n-1)^{st}$ link).

$$\begin{aligned} P_1^{TCP} &= \left(1-(1-P)^{n-i}\right)^r \\ P_2^{TCP} &= (1-P)^{n-i} \\ P_{rtx,link}^{TCP}(r,i) &= \left(1-(1-P)^{n-i}\right)^r \cdot (1-P)^{n-i} \end{aligned} \tag{5}$$

CTCP, on the other hand, was designed to retransmit a lost packet from the last node it successfully reached. Since each packet is cached for possible retransmission at each node, once a packet is sent successfully across a link, it never needs to be sent over the link again. For this analysis of CTCP, it was assumed that the cache was large enough that a lost packet would still be present in the cache if it needed to be retransmitted. In other words, a sending node is only responsible for getting a packet to the next node. After that, the next node caches the packet and responsibility shifts to that node. This lock-step process of moving a packet toward the client one link at a time means that the number of retransmissions over a specific link is independent of the link's distance from the server. P_1^{CTCP} is equal to the probability that a packet is lost over the next link r times. P_2^{CTCP} is the probability that the packet is then transmitted successfully over that single link.

$$\begin{aligned} P_1^{CTCP} &= P^r \\ P_2^{CTCP} &= 1-P \\ P_{rtx,link}^{CTCP}(r,i) &= P^r \cdot (1-P) \end{aligned} \tag{6}$$

Evaluating Equation 3 for TCP, we find that the expected number of retransmissions for a given link is:

$$\mathrm{E}^{TCP}[r|i] = \frac{1-(1-P)^{n-i}}{(1-P)^{n-i}} \tag{7}$$

Similarly, for CTCP:

$$\mathrm{E}^{CTCP}[r|i] = \frac{P}{1-P} \tag{8}$$

For the link next to the server, the expected number of times a packet must be retransmitted using TCP over a 10-link path is $\mathrm{E}^{TCP}[r|i=0] = 0.046$. For a link halfway from the server (i=5), the expected number of retransmissions is given as $\mathrm{E}^{TCP}[r|i=5] = 0.023$. For CTCP, a packet is expected to be retransmitted $\mathrm{E}^{CTCP}[r|i] = 0.0045$ times, independent of the network path length or location of the link. These numbers were computed using P=0.45% as

determined from Equation 1 and Paxson's analysis. This scenario shows that a CTCP server is expected to retransmit a packet 90.2% less times across the link connected to the server than a TCP server thereby saving bandwidth for non-redundant data. In the middle of the network, CTCP still results in an 80.4% reduction in retransmissions.

Using $E^{TCP}[r|i]$ and $E^{CTCP}[r|i]$, Equation 2 may be solved to find the total hops taken by a packet using TCP and CTCP, respectively

$$\mathrm{TRH}^{TCP} = \frac{1-(1-P)^n - nP\cdot(1-P)^n}{P\cdot(1-P)^n} \tag{9}$$

$$\mathrm{TRH}^{CTCP} = \frac{nP}{1-P} \tag{10}$$

The difference between TRH^{TCP} and TRH^{CTCP} gives the expected benefit of CTCP over TCP in terms of links traversed. Continuing the example above, using TCP over a 10-link network path with P=0.45% results in $\mathrm{TRH}^{TCP} = 0.252$ total redundant hops required for a packet to be transmitted from the server to the client. CTCP, on the other hand, is only expected to take $\mathrm{TRH}^{CTCP} = 0.0452$ total redundant hops. This represents an 82% saving in the number of redundant hops a packet is expected to traverse before reaching the client.

Since it was assumed that each packet transmission was independent of successive transmissions, the bandwidth consumption scales 1-for-1 with the number of packets transmitted. For instance, in order to transfer a 200KB file using 400 packets of 512 bytes each, TCP is expected to transmit a total of 101 redundant packets over the 10 links. CTCP, on the other hand, is only expected to transmit a total of 19. Therefore, caching packets at the network level and using CTCP would give at least a 67% and up to an 88% saving of redundant bandwidth over using TCP for network path lengths ranging from 5 to 15 links.

While TRH represents the amount of bandwidth that can be eliminated and therefore potentially affected by network level caching, it is also important to examine the difference between the total bandwidth consumed by TCP and CTCP. The metric for comparing total bandwidth was defined above as TH or total hops. TH is similar to TRH except that it

includes the n hops that are necessary in order to reach the client. The equation for TH is given as:

$$\begin{aligned} \mathrm{TH} &= \sum_{i_0=0}^{n-1} (1 + \mathrm{E}[r | i = i_0]) \\ &= \mathrm{TRH} + n \end{aligned} \tag{11}$$

Over a 10-link network path with packet loss probability P=0.45%, $\mathrm{TH}^{TCP} = 10.252$ and $\mathrm{TH}^{CTCP} = 10.0452$. This represents a 2.0% reduction in number of hops for every packet that is transmitted.

For internal U.S. connections, the expected reduction in the number of hops for every packet is 2.0%. This percentage may seem low, but that is because packet loss rates in the U.S. are some of the lowest on the Internet. However, Paxson's results [8] show that the packet loss rates are rising. Over 1995, U.S. packet loss rates increased 21% from 3.6% to 4.4%. Other countries outside of the U.S. already face higher loss rates. For instance, connections between Europe and the U.S. experience as much as 16.9% packet loss. This packet loss rate over paths ranging from 12 to 20 hops leads to a reduction of between 8.3% and 8.6% in total hops taken by packets when using CTCP.

4.2 Latency Analysis

The second goal of CTCP is to lower the latency of data packets being sent from the server to the client. TCP must retransmit lost packets from the server regardless of the point at which they are lost. This results in lost packets traversing the portions of the network multiple times delaying receipt by the client. CTCP, by caching in the network nodes, retransmits lost packets from the point where they were lost. This reduces the number of links that lost packets must traverse compared to TCP. To analyze the effectiveness of caching at the network level on packet latency, the probabilistic model mentioned above was used again. This time, the expected number of links that a packet must traverse in the face of packet loss was calculated for both TCP and CTCP. CTCP was shown to transmit packets over fewer links on average than TCP.

While lower latency does cause an increase in data throughput, its effect is limited in a bulk data transfer. Lower latency tends to benefit interactive sessions more than bulk transfers like those for which CTCP was designed. This is because in a bulk data transfer, packets are transmitted in rapid succession to achieve a high throughput. This means that several packets are in transit at any point in time and latency only affects the time to receive the first packet. Successive packets are received at the rate they are transmitted independent of the latency. Therefore, latency is an important metric to analyze for interactive sessions rather than bulk data flow.

In order to analyze the improvements with respect to latency, the N-link Network Path model presented in Figure 7 was used again to represent a typical network path used in a TCP or CTCP connection. Instead of attempting to determine the number of packets transmitted on a given link, the goal is to determine the expected redundant latency required for a packet to reach the client. Again, it is assumed that each link has an equal and constant probability (P) of dropping packets due to congestion and that packet losses are independent of each other. Most importantly, each link is assumed to require an equal and constant amount of time for a packet to traverse (t_{link}).

In order to compare the latency of TCP and CTCP packets, total latency (TL) and total redundant latency (TRL) are defined similar to TH and TRH from the previous section. Total latency is the total amount time it takes for a packet to travel from the server to the client. Total redundant latency, on the other hand, discounts the time to traverse the path once, as it is necessary in order to transfer the packet. Again, TRL would be zero except for packet loss. As with the bandwidth study, TRL is the focus of this analysis, but TL is also be derived.

TRL is related to the number of hops it takes for the packet's location to go from the server to the client defined as TH in the previous section. TRL is equal to the number of redundant hops multiplied by the length of time each hop takes. Specifically, the expected total redundant latency is given as:

$$TRL = TRH \cdot E[t] \tag{12}$$

where l is the latency, TRH is the number of total redundant hops that a packet must take to reach the client, and E[t] is the expected length of time for each hop.

The time each hop takes depends on whether or not the packet is lost. If the packet is transmitted successfully, the traversal time is simply t_{link}, the time to traverse a link. If the packet is lost, however, additional time (t_{detect}) is spent detecting the loss. Given that a loss occurs with probability P, the expected length of time for each hop is:

$$\begin{aligned} \mathrm{E}[t] &= (1-P)\cdot t_{link} + P\cdot(t_{link} + t_{detect}) \\ &= t_{link} + P\cdot t_{detect} \end{aligned} \tag{13}$$

Before TCP can retransmit a lost packet from the server, the server needs to detect the packet's loss. TCP has two methods for detecting lost packets: timeouts and Fast Retransmit. During bulk data transfer, Fast Retransmit is used almost exclusively to detect lost packets. Timeouts are used more often in interactive sessions where TCP has a limited amount of data to send. The time for the Fast Retransmit algorithm to take effect is approximately the round-trip time of the network path. Therefore, the round-trip time is used as the length of time to detect a lost packet for TCP as shown in Equation 14.

CTCP also must detect a packet's loss before it can retransmit it. Like TCP, during bulk data transfer, CTCP detects lost packets by using its version of the Fast Retransmit algorithm. Therefore, the length of time for CTCP to detect lost packets using Fast Retransmit (t_{detect}) is also approximately the round-trip time.

$$t_{detect} \cong 2n\cdot t_{link} \tag{14}$$

Using the formula for total redundant hops using TCP (TRH^{TCP}) and substituting in for $\mathrm{E}[t]$, the expected end-to-end for TCP is:

$$\mathrm{TRL}^{TCP} = \frac{1-(1-P)^n - nP\cdot(1-P)^n}{P\cdot(1-P)^n}\cdot(2nP+1)\cdot t_{link} \tag{15}$$

where TRL^{TCP} is the expected total redundant latency for TCP.

Similarly for CTCP:

$$\mathrm{TRL}^{CTCP} = \frac{n}{(1-P)} \cdot (2nP+1) \cdot t_{link} \tag{16}$$

where TRL^{CTCP} is the expected the latency from a CTCP server to the client.

Using Equations 15 and 16, the expected redundant latency of a 10-link network path with the probability of losing a packet P=0.45% is $\mathrm{TRL}^{TCP} = 0.275 \cdot t_{link}$. CTCP, under the same conditions, has a lower redundant latency at $\mathrm{TRL}^{CTCP} = 0.0493 \cdot t_{link}$. Thus using CTCP in this scenario results in an expected reduction of end-to-end packet latency of 82.1%. For networks with between 5 and 15 links, the latency reduction vary between 67.1% and 87.7%.

While TRL is the latency that can be affected by network level caching, it is also important to examine the reduction in total latency. The equation for total latency is given as:

$$\mathrm{TL} = \mathrm{TH} \cdot \mathrm{E}[t] \tag{17}$$

Solving Equation 17, the expected total latency of a 10-link network path with the probability of losing a packet P=0.45% using TCP is $\mathrm{TL} = 11.17 \cdot t_{link}$ while using CTCP is $\mathrm{TL} = 10.95 \cdot t_{link}$. Thus using CTCP instead of TCP in this scenario results in a 2.0% expected reduction of end-to-end packet latency.

Just like the bandwidth analysis above, latency savings are much more dramatic outside of the U.S. This is because of the higher packet loss rates over Internet connections in other countries. For example, connections between Europe and the U.S., which face packet loss rates of up to 16.9%, can expect to experience up to a 8.6% reduction in overall latency. This percentage was calculated using a network path length of 20 hops and P=0.92%.

4.3 Reducing Server Load

In addition to the bandwidth and latency benefits of CTCP associated with network node caching, CTCP also has the potential to reduce the load on network servers such as web

servers. CTCP reduces the load on servers by distributing some of the work previously performed by the server to the clients. Instead of requiring the server to manage the reliability of a data transfer, the client is required to do so. This shift in responsibility lightens the server's load substantially. This can be seen comparing a TCP and a CTCP server's duties. Figure 8 lists the processing that must occur for each packet that is received by both TCP and CTCP.

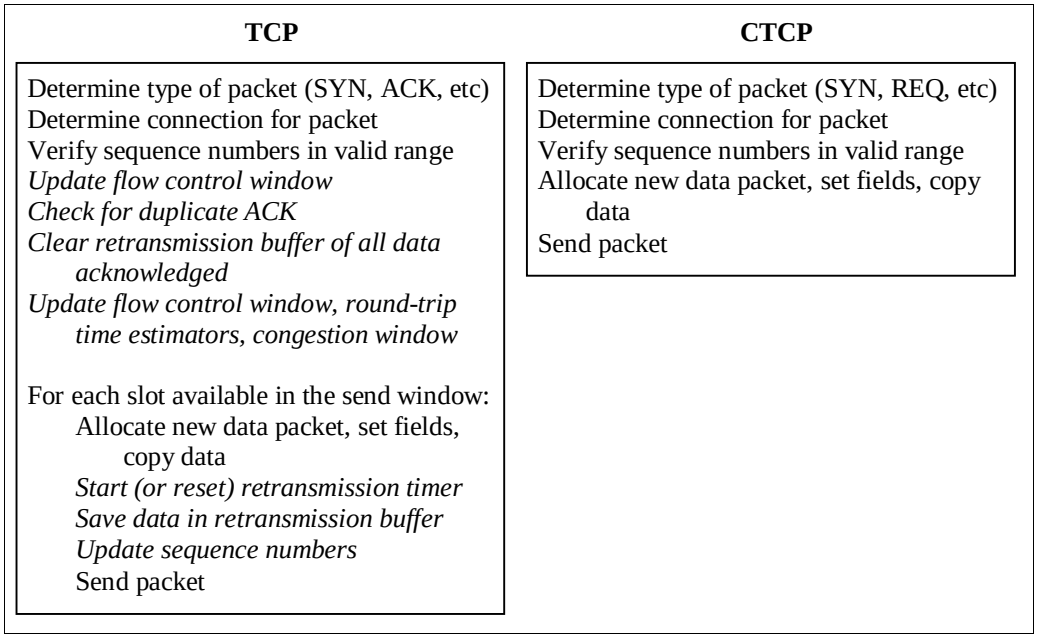

Figure 8: Server Packet Processing

The time required to perform each of the tasks listed in Figure 8 is dependent on not only the implementation of the software but also on the underlying hardware running the software. It is typically assumed that the most time consuming task is copying data from the user-supplied buffer to the new packet. However, the data copy task cannot be eliminated from the data transfer. Instead, the discussion is limited to control processing instead of memory copies, as the amount of control processing can actually be reduced as a result of the transport protocol. TCP's most significant tasks are recalculating the round-trip time estimators and managing congestion control every time a packet is received. CTCP also performs these tasks, but does so on the client instead of the server. In addition to the processing shown in Figure 8, TCP must also update and check timers periodically for every connection in order to detect packet loss.

TCP clearly has more processing to perform than CTCP for every packet that is received. All of the extra processing that the TCP server has to do has been relocated to the client in CTCP. The result is that a CTCP server can both serve more clients simultaneously as well as server each client faster than a TCP server.

CHAPTER 5: CONCLUSION

This thesis presented network node caching as a solution to combat the overhead associated with retransmissions of lost packets. Retransmissions cause increased bandwidth utilization, higher packet latency, and increased server load. To alleviate these problems, individual packets within a connection were cached in the nodes of the network. These cached packets were then used in the case that the packet was lost and needed to be retransmitted.

TCP was determined to be inappropriate to support network node caching because of the server's involvement with retransmitting lost packets. Client-side TCP was designed as an alternative to TCP to take advantage of network level caching functionality. CTCP put the data transfer reliability burden on the client instead of on the server. Since the client maintained the reliability state associated with the connection, CTCP was free to remove the server from the retransmission process if requested data was found at an intermediate node.

A probabilistic analysis of TCP and CTCP was made with respect to bandwidth consumption and end-to-end packet latency. CTCP was found to consume up to 88% less redundant bandwidth than TCP under typical Internet data transfers. CTCP was also shown to reduce an individual packet's redundant latency up to 88%. This reduction in bandwidth utilization allows more "useful" traffic to traverse the Internet while the reduction in latency improves response time as seen by the user.

The packet processing routines for TCP and CTCP servers were also compared. A CTCP server was found to do less computation for each packet that was received than a TCP server. This reduction in processing will allow CTCP servers to server more clients simultaneously.

CTCP was therefore shown to successfully utilize network level caching, having benefits over TCP that include lower bandwidth consumption, smaller packet latencies, and a decreased server load.

5.1 Future Work

There are several interesting future research opportunities regarding CTCP and network node caching. These areas are related to adding additional functionality to CTCP's design and implementing a higher performance version of CTCP.

5.1.1 CTCP Functionality

CTCP was designed for bulk data flow such as file transfers. Several modifications would need to be made to CTCP's design in order to support interactive data flow such as telnet sessions. When a user presses a key in a telnet session using TCP, the sender (the user's machine) transmits a packet specifying the key pressed to the receiver (the machine running the telnet daemon). The receiver then sends an acknowledgment back to the sender verifying that the data was received correctly. In CTCP, the sender does not transmit the data unless the receiver requests it. In an interactive mode, however, the receiver does not know when the next data (generated by the user pressing a key) will be available. In short, CTCP does not have a mechanism by which the sender can tell the receiver to begin requesting data.

There are two potential solutions to CTCP interactive data sessions. First, operating systems could support both CTCP and TCP. Whenever an interactive session was requested, TCP could be used. However, due to the benefits of CTCP over TCP in bulk data transfer, CTCP would be used whenever a file transfer session was requested. This has the benefit that CTCP could remain a relatively simple protocol thus aiding in the development of bug-free implementations.

The other option would be to add interactive support to CTCP's design. For instance, a control packet could be added to CTCP that the sender could use to tell the receiver to begin requesting a certain amount of data. The problem with this plan is that it increases traffic in the form of these extra control packets. However, control packets are usually fairly small and are typically "piggybacked" on a data packet, so that the increased traffic would be minimal.

5.1.2 Operating System Support for CTCP

The current implementation of CTCP was written in Java at the application level. Java, as a high level language, made the implementation easy to design and debug. These benefits were at the cost of performance. Operating system protocols such as TCP are normally

implemented at the kernel level to achieve high networking performance. In order to compare CTCP to current industry TCP implementations, CTCP would need to be written at the operating system level.

5.1.3 Limited Caching

The initial design for network level caching states that routers are to cache all data packets that they receive. This means that even packets that are not lost and therefore do not need to be retransmitted must spend time being cached at each router. Since packet loss rates are typically small (on the order of 5%), the cost of caching every packet is likely to be greater than the benefit that is gained. Instead of caching every data packet that is routed, it might be possible to construct a more complicated algorithm that only caches packets that are more likely to be lost. This kind of algorithm would have to dynamically recalibrate itself to changing network conditions.

5.1.4 CTCP Acknowledgments

A CTCP server must hold onto the data for the entire file being transferred. This is because the server does not receive information regarding what data the client has actually received. The server only receives the currently requested data packet's sequence number. Instead, CTCP could be modified such that the requests double as acknowledgments. Each time the client sends a request to the server, it could also inform the server about how much data has actually been received. This would allow the server to incrementally free up data buffer space, thereby allowing the server to handle more connections.

Therefore, future improvements on CTCP would address it current limitations regarding interactive sessions and operating system level support.

5.2 Conclusion

Client-side TCP and network level caching represent two techniques that attempt to reduce both the amount of bandwidth consumed by data transfers and the end-to-end packet latency. In addition, Client-side TCP attempts to reduce the load placed on servers. In the future, as Internet packet loss rates rise due to increased usage and congestion, the benefits of caching techniques such as network level caching will continue grow.

NOMENCLATURE

P	Probability of losing a packet over a single link
P_{path}	Probability of losing a packet over an entire network path
r	Number of retransmissions
i	Link number. Links are numbered starting with link 0 connecting the server to the first intermediate node.
TRH^{TCP}	Expected number of total redundant hops a packet must take from the server to the client using TCP
TRH^{CTCP}	Expected number of total redundant hops a packet must take from the server to the client using CTCP
$\mathrm{E}^{TCP}[r\|i]$	Expected number of times a packet is transmitted over the i^{th} link using TCP
$\mathrm{E}^{CTCP}[r\|i]$	Expected number of times a packet is transmitted over the i^{th} link using CTCP
$P_{rtx,link}^{TCP}(r,i)$	Probability of a packet being retransmitted r times over the i^{th} link using TCP
$P_{rtx,link}^{CTCP}(r,i)$	Probability of a packet being retransmitted r times over the i^{th} link using CTCP
TH^{TCP}	Expected number of total hops a packet must take from the server to the client using TCP
TH^{CTCP}	Expected number of total hops a packet must take from the server to the client using CTCP
TRL^{TCP}	Expected total redundant latency for a single packet transmitted using TCP
TRL^{CTCP}	Expected total redundant latency for a single packet transmitted using CTCP
t_{link}	Time for a packet to be transmitted across a single link
t_{detect}	Time for a transport protocol to detect and retransmit a lost packet
TL^{TCP}	Expected total latency for a single packet transmitted using TCP
TL^{CTCP}	Expected total latency for a single packet transmitted using CTCP

BIBLIOGRAPHY

[1] H. Balikrishnan, V. Padmanabhan, S. Seshan, M. Stemm, R. Katz. "TCP Behavior of a Busy Internet Server: Analysis and Improvements." *Proc. IEEE Infocom*, San Francisco, CA, USA, March 1998.

[2] H. Balakrishnan, S. Seshan, E. Amir, R. H. Katz. "Improving TCP/IP Performance over Wireless Networks." *Proc. of 1st ACM Conf. on Mobile Computing and Networking*, Berkeley, CA, November 1995.

[3] Cisco Web Page. http://www.cisco.com.

[4] J-C. Bolot. "Characterizing End-to-End Packet Delay and Loss in the Internet." *Journal of High-Speed Networks*, vol. 2, no. 3, pp. 305-323, December 1993.

[5] V. Jacobson, R. T. Braden, L. Zhang. "TCP Extensions for High-Speed Paths." RFC 1185. October 1990.

[6] J. Padhye, V. Firoiu, D. Towsley, J. Kurose. "Modeling TCP Throughput: A Simple Model and its Empirical Validation." To appear in *Proc. of SIGCOMM '98*. Vancouver, CA, September 1998.

[7] V. Paxson. "End-to-End Internet Packet Dynamics." *ACM SIGCOMM '97*, September 1997, Cannes, France.

[8] Paxson, V. "Measurements and Analysis of End-to-End Internet Dynamics." Ph.D. Dissertation, 1997.

[9] J. B. Postel. "Transmission Control Protocol." RFC 1793. September 1981.

[10] D. Sanghi, A. K. Agrawala, O. Gudmundson, B. N. Jain. "Experimental Assessment of End-to-End Behavior on Internet." *Proc. of the Conference on Computer Communications*, pp.867-874, March/April 1993.

[11] W. Stevens. TCP/IP Illustrated, Volume 1: The Protocols. Professional Computing Series. Addison-Wesley, Reading, MA, 1994.

[12] W. Stevens. TCP/IP Illustrated, Volume 2: The Implementation. Professional Computing Series. Addison-Wesley, Reading, MA, 1995.

[13] D. J. Wetherall, J. Guttag, D. L. Tennenhouse. "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols." *IEEE OPENARCH '98*, San Francisco, CA, April 1998.

TCP and CTCP are connection-based protocols. Before data can be transmitted from the server to the client, a connection must be established between them. In addition, once the data transfer is complete, the connection must be closed to free up resources for future connections. CTCP's connection management is similar in design to TCP and is described in detail here.

A.1 TCP Connection Management

The connection establishment procedure using TCP is as follows:

1. The client initiates the connection by sending a SYN packet to the server. The SYN packet specifies the port number of the server that the client wants to connect to and the client's initial sequence number.
2. The server responds with its own SYN packet containing the server's initial sequence number. In addition, the server marks the SYN packet as an ACK in order to acknowledge the client's initial SYN.
3. The client then acknowledges the receipt of the server's SYN-ACK by transmitting an ACK back to the server.

These three packets complete the connection establishment. This process is typically called the three-way handshake and is illustrated in Figure 9.

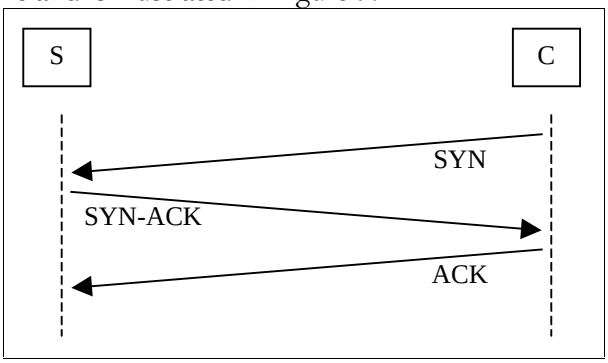

Figure 9: TCP Connection Establishment

While connection establishment requires three packets, terminating a connection requires four. This is because each direction of the data transfer is closed independently from the other. One side of the connection indicates that it is finished transferring data by sending a FIN packet to the other endpoint. The receiver of the FIN packet responds with an acknowledgment. At this point, data transfer in that direction is finished and the connection is considered only "half-open." In order to completely close the connection, the same procedure occurs again in the reverse direction. This is illustrated in Figure 10.

For more information regarding TCP's connection establishment and termination procedures, refer to [11, 12].

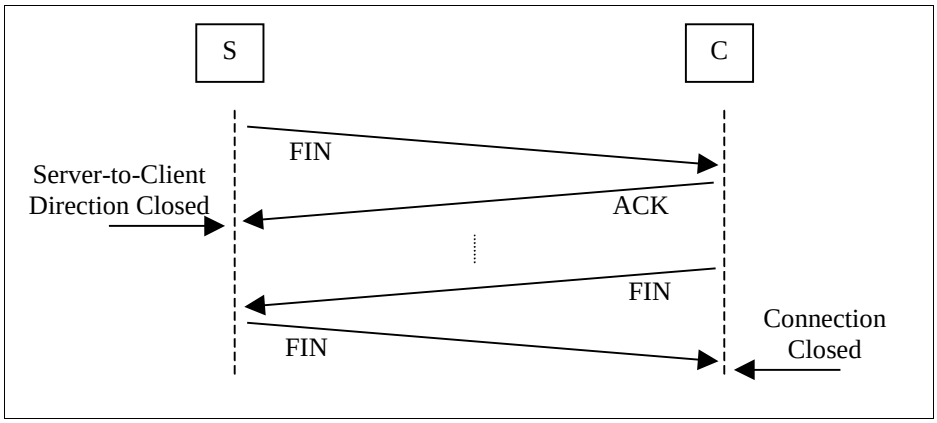

Figure 10: TCP Connection Termination

A.2 CTCP Connection Management

CTCP's connection mechanisms are almost identical to TCP's. The three-way handshake is used for connection establishment. However, CTCP requires the client to send the name of the document as part of the initial SYN packet sent from the client to the server. This allows the server to respond with the unique document identifier in the SYN-ACK that is sent back to the client. In addition, the server is also required to tell the client the size of the requested file. This is because the client is in charge of requesting portions of the data and must therefore know the size of the data.