Dynamic TCP Proxies: Coping with Mobility and Disadvantaged Hosts in MANETs

by
KYLE G. SCHOMP

Submitted in partial fulfillment of the requirements
For the degree of Master of Science

Thesis advisers:
Dr. Michael Rabinovich
Dr. Shudong Jin

Department of Electrical Engineering and Computer Science
CASE WESTERN RESERVE UNIVERSITY

August, 2010

CASE WESTERN RESERVE UNIVERSITY
SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

_____Kyle G Schomp_____

candidate for the _____Master_____degree *.

(signed)_____Shudong Jin_____
(chair of the committee)

_____Michael Rabinovich_____

_____Vincenzo Liberatore_____

_____

_____

_____

(date) _____6/24/2010_____

*We also certify that written approval has been obtained for any proprietary material contained therein

Table of Contents

List of Figures

List of Equations

Dynamic TCP Proxies: Coping with Mobility and Disadvantaged Hosts in MANETs

Abstract
by
KYLE G. SCHOMP

TCP proxies have been introduced as a method to improve throughput and reduce congestion in mobile ad hoc networks. Proxies split the path into several shorter paths which have higher throughput due to reduced packet loss and round trip time. As a side effect, congestion is reduced because fewer link layer retransmissions occur. In current protocols, proxies are assigned at the start of the transfer and must be used for the duration. Due to mobility and congestion change, pinned proxies can actually reduce throughput.

In this thesis, we present a second version of the DTCP protocol which includes the ability to switch proxies in the middle of a transfer. We demonstrate in the Network Simulator version 2 that the new protocol performs better than other related protocols in simulated mobile ad hoc networks with varying levels of mobility and congestion.

Chapter 1 – Introduction

Applications in mobile ad-hoc networks (MANETs) can suffer from poor link quality. TCP in particular suffers from poor link quality due to frequent congestion window backoff. To resolve this problem, the use of TCP proxies to reduce end-to-end round trip time and packet loss rate has been advocated. DTCP is a protocol which places proxies for this purpose. TCP proxies create an overlay network that remains pinned for the duration of the transfer. Due to factors including mobility and congestion change, the overlay may become suboptimal. We introduce a method for switching proxies during a transfer, which we call migration, and extend the DTCP protocol to include it. The extensions to the protocol detect when nodes move and congestion change. When either occurs, migration is performed and the TCP transfer continues on a new set of proxies.

In addition to migration, we reintroduce TCP end-to-end principles by the addition of an end-to-end session state which is forwarded between proxies. The TCP end-to-end principle of guaranteed delivery is broken by proxies when they acknowledge data before it reaches the ultimate destination. The end-to-end session state acknowledgement returns guaranteed delivery to the protocol. The cost of added end-to-end state signaling is reduced by a method of piggybacking upon existing TCP signaling.

Two prominent methods have been introduced to leverage TCP proxies to improve throughput in MANETs. Kopparty et. al. [Kopparty et. al., 2002] place proxies uniformly along the path in their Split TCP protocol, while Ouyang et. al. [Ouyang et. al., 2009]

encloses disadvantaged links with proxies to mitigate their effect on the rest of the network in their DTCP protocol. Both protocols assign static proxies at the beginning of the transfer. This thesis builds off the work performed by Ouyang et. al.

We introduce DTCPv2 in this thesis. The work performed by Ouyang et. al. is labeled DTCPv1 to prevent confusion. DTCPv2 is designed to maintain the improvements in throughput and reductions in congestion observed in DTCPv1 while adding mobility and congestion change handling. DTCPv2 also reintroduces TCP end-to-end semantics to the protocol. The protocol is validated in the Network Simulator version 2 [NS2, 2009]. We also show results of the protocol in a simulated MANET to demonstrate the potential of DTCPv2.

Chapter 2 presents a detailed look at the protocol and the individual operations it performs. Chapter 3 lays out our method of validation and simulation. In chapter 4, the results of all the NS2 simulations are presented. Chapter 5 contains a discussion of the results and chapter 6 has the conclusions we draw from the results. Finally, chapter 7 highlights some areas for future work.

1.1 – Background

Mobile ad hoc networks (MANETs) are self-configuring networks of wireless nodes. Communication occurs over a path consisting of one or more wireless links. The quality of the links can vary dramatically due to various forms of interference either external to the network or caused by congestion. Due to mobility, links can also break entirely when one node moves away from another.

TCP was designed to perform over existing wired internetworks where packet loss is an indicator of congestion. TCP reacts to packet loss events by cutting throughput to avoid over utilization of the link. In MANETs, there are many sources of packet loss including interference, node movement, data corruption, and congestion. Because TCP reacts to all sources of packet loss in the same manner, the throughput of TCP in MANETs is often substantially lower than the throughput which the network can support.

Since each link in a MANET has the potential to cause packet loss, we can reduce the probability of an individual packet being lost by reducing the hop-distance of a TCP connection [Holland and Vaidya, 1999]. Reduced packet loss improves TCP throughput [Kurose and Ross, 2005]. Furthermore, reducing the number of hops in a TCP connection reduces the round trip time of the connection, again improving TCP throughput. One or more TCP proxies serve the purpose of reducing the hop-distance of TCP connections by splitting them into two or more shorter TCP connections [Kopparty et. al., 2002].

TCP proxies have the highest throughput when they can send data at the same rate at which they receive data [Ehsan and Liu, 2004]. This happens when the throughputs of all the short TCP connections are balanced. Since TCP throughput is affected by packet loss and round trip time, placing TCP proxies more frequently in areas where packet loss is high and less frequently in areas where packet loss is low achieves balanced throughput. This method of proxy placement is the central idea around congestion aware proxy placement in DTCP [Ouyang et. al., 2009].

1.2 - Related Work

Bakre and Badrinath [Bakre and Badrinath, 1995] present I-TCP which is designed to mitigate poor performance experienced by mobile hosts when connecting to existing internetworks. Because of mobility and the unreliable nature of wireless networks which are not handled well by IP-based protocols, they designed I-TCP to separate the wireless network from the wired network. I-TCP allows TCP connections to migrate when the mobile hosts switches cells.

Kim et. al. [Kim et. al., 2005] design RCP to move state data to the recipient side of the connection in mobile wireless networks. When the sender is within existing internetworks and the receiver is a mobile wireless node, RCP is designed for better congestion control, loss recovery, and power management over the last-hop wireless link. In addition, moving state data to the receiver allows for seamless server migration during handoffs.

Shieh et. al. [Shieh et. al., 2005] introduce Trickles, a protocol which includes server state data within the communication effectively removing state from one end of the connection. Trickles enables stateless servers, instantaneous and transparent failover, and connection redirection.

Soenren et. al. [Snoeren et. al., 2001] present techniques for connection failover for replica servers. They use TCP connection migration mechanisms to achieve robust, fast, and fine-grained connection failover without interfering with the operation of the receiver.

Luglio et. al. [Luglio et. al., 2004] demonstrate how TCP proxies can be used to divide terrestrial connections across a satellite network. Proxies are placed between the

terrestrial network and the satellite network and on the satellites to improve connection throughput.

Cohen and Ramanathan [Cohen and Ramanathan, 1997] show that using proxies at the edge of hybrid fiber coaxial (HFC) networks can improve throughput. By handling packet loss which occurs in the HFC locally at low latencies, a proxy server enables faster recovery from packet losses.

Kopparty et. al. [Kopparty et. al., 2002] recognized that TCP in mobile ad hoc networks wrongly attributes packet loss due to line failures as congestion. The problem becomes worse as the hop-distance of TCP connections increases. They introduce Split TCP which places TCP proxies uniformly along the route to break up long hop-distance connections into multiple short hop-distance connections.

Ouyang et. al. [Ouyang et. al., 2009] develop the DTCP protocol to combat the heterogeneous nature of link quality in ad hoc networks. By enclosing poor quality links with TCP proxies, they isolate the effect of poorly performing links on the rest of the network and improve overall performance.

Chapter 2 – Protocol

DTCPv2 improves DTCPv1 by adding the ability to switch proxy placement during communication, rather than solely at the beginning of the connection. Switching proxy placement, which we call proxy migration, allows DTCPv2 to adapt to rapidly changing congestion and routing on MANETs. We also reintroduce TCP's guaranteed delivery end-to-end semantics into DTCPv2. TCP proxies violate end-to-end semantics by acknowledging data before it is delivered to the destination. To accomplish TCP's guaranteed delivery, we introduce a new layer directly above the transport layer in the protocol stack. The new layer handles end-to-end communication, while the transport layer handles communication between proxies. The two ends of a DTCPv2 session maintain two full states: a TCP state with the nearest proxy and a DTCP state with the opposing end of the session.

The connection between the active end of the DTCP connection (the source) and the passive end of the connection (the sink) is known as the global connection. The connections between the source and the nearest proxy, between proxies, and between the final proxy and the sink are known as local connections. Note that each proxy is comprised of a local sink and a local source which are connected with the prior source and post sink, respectively.

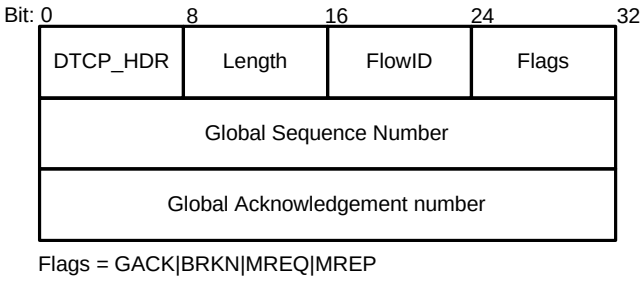

Figure 1

To handle the addition of global connection data to communication, we introduce a DTCP header into the TCP header options. The structure of the header is shown in Figure 1. The DTCP header is 12 bytes including the TCP option number and length field. The data within this header is relevant to the global connection at the global source and global sink. As a result, it is largely ignored by the proxies.

DTCPv2 connection setup begins with a new state called metrics collection. The source uses the collected metrics to compute proxy placements and then proceeds into the proxy setup state which occurs during TCP's three-way handshake. Once TCP enters the established state, the source routinely performs congestion detection and reacts to routing changes. If the connection needs to migrate to a new set of proxies due to either congestion changes or routing changes, the source transitions to the proxy setup state. Once communication is complete, the source closes the connection. Each DTCP extension to TCP is described below. The description below takes the perspective of a one-directional connection where data originates at the source and is received at the sink. Note that DTCP is fully two-way capable. It is described in one-direction for clarity. The implementation described also requires that the Dynamic Source Routing (DSR) protocol be used at the network layer. DTCP uses some cross-layer optimizations to improve

performance. Though DTCPv2 is designed to perform with DSR, DSR is not strictly required. Where cross-layer optimizations exist, alternatives are described which do not require the DSR protocol.

The following description of the DTCPv2 protocol is based of an implementation in the Network Simulator version 2 [NS2, 2009]. As such, it varies from a real world implementation in a few details. A real world implementation would be functionally equivalent to the implementation described here.

2.1 – Metrics Collection

The first communication performed between the global source and sink of a DTCP session is a metrics request (MREQ) from source to sink and a metrics response (MREP) from sink to source. Each node traversed along the path from the source to the sink will attach congestion metrics to the packet. The metrics are then returned to the source in the metrics response packet. Collection uses the initial TCP timeout value. If timeout occurs, the MREQ is retransmitted. The first MREP received completes metrics collection. Once metrics are collected, the source has a list of all nodes traversed in order from the source to the sink along with their respective metrics.

We collect the following metrics: link-layer transmission queue length, MAC layer retransmission count, and MAC layer drop rate. the queue length is a reflection of the load placed upon the node, while the retransmission count and drop rate indicate the level of interference on the link. We assume that high values for any of these metrics indicate congestion. A jump in the metrics between two adjoining nodes is a congestion edge.

Edges are detected by using a threshold parameter which is empirically derived. All three metrics are calculated using the exponentially weighted moving average method to produce smoothed estimates.

The collection of metrics is performed by a packet filter at the network layer of each node in the system. When the source wishes to collect metrics for proxy setup, it sends a data-less TCP segment with the DTCP MREQ flag set. The flag is included in the DTCP header, Figure 1. Each TCP segment received by a node is sent through the packet filter. If the metrics request flag is found, the node inserts its own metrics. If the DTCP header is not found or the MREQ flag is not set, the TCP segment is unchanged. The packet filter is installed upon each DTCP-enabled node before communication begins. The metrics request segment continues all the way to the sink which handles metrics requests received in the TCP listen state by returning a data-less TCP segment with the collected metrics attached and the DTCP MREP flag set. Metrics are collected source-to-sink rather than sink-to-source because of the possibility of the two paths not being equivalent. This is true in some implementations of DSR and other routing protocols where wireless links are not assumed to be bi-directional.

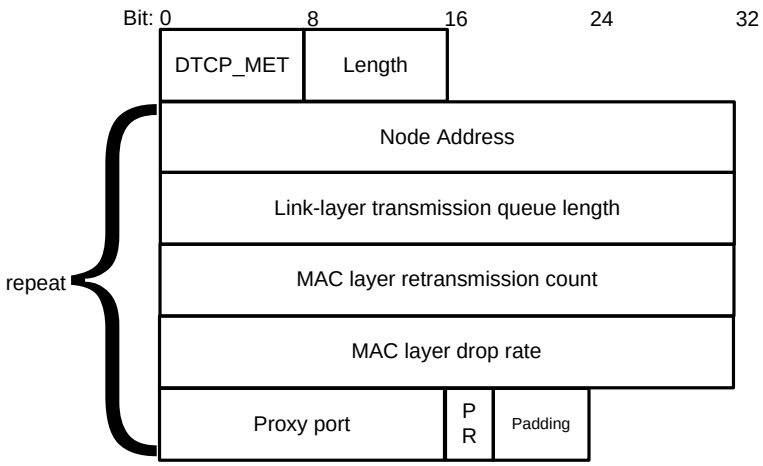

Figure 2

The DTCP metrics TCP header option is shown in Figure 2. The first two fields, DTCP_MET and Length, are the TCP option number and length fields. The rest of the fields are set per node. Each node inserts its metrics at the end of the header option and updates the length field. For each node, the 3 metrics are collected along with the node's IP address. The PR flag is set indicating whether or not there is a proxy for the connection on this node. If there is, the port number of the proxy is also set. In total, 19 bytes of data are collected per node.

The original version of DTCP used DSR route requests (RREQ) and replies (RREP) to gather metrics. Each node, in addition to appending its address upon a RREQ, appends the metrics as well. When a route is cached by a node, it also stores the metrics for the nodes along the route. This is convenient in that the metrics collection and DSR route discovery can be combined, reducing startup time. However, we recognized that routes returned to the source are often from caches within the network and the metrics data may become stale. In order for DTCP to react to a dynamic network, the metrics must be collected in real time. Also, as a secondary goal, we wish to reduce the cross network

layer mechanisms and allow DTCP to function with a variety of network layer protocols. Actively collecting the metrics at startup accomplishes both goals.

2.2 – Proxy Setup

Using the results of metrics collections as input, the source runs the proxy selection algorithm. The output of the algorithm is a list of proxies along the path. DTCPv2's proxy selection algorithm is unchanged from DTCPv1 [Ouyang et. al., 2009]: "We ... use a simple sum of the [link-layer transmission queue length and MAC layer retransmission count] metrics as a combined metric ... We call it the 'proxy-selection metric'. The higher the value of the proxy selection metric of a host the more disadvantaged this host is." The source then runs the algorithm described by the pseudocode in the

Appendix. First, proxies are placed on the less disadvantaged side of each congestion edge. Next, additional proxies are added to zones of 6 or more nodes in which proxies have not already been placed. These extra proxies are placed to provide a fair comparison with Split TCP by placing a similar total number of proxies. Note that there are alternative proxy placement algorithms. The one described here is the one primarily used. Other algorithms are described in [4.2].

The source creates a SYN segment for the connection and inserts the proxy list into the TCP header options. The SYN segment is sent, directed to the global sink's address. A second packet filter in the network layer intercepts TCP SYN segments. When a SYN segment is intercepted by a node along the path and the proxy list contains the node's address, a TCP proxy is installed and receives the segment. The new proxy copies the SYN segments source address as the destination of the proxy sink and the SYN segment's destination address as the destination of the proxy source. The new TCP proxy responds to the SYN with a SYN+ACK using standard TCP conventions. The proxy also sends a SYN segment again to the global sink, inserting the proxy list into the TCP header options minus itself. Once a SYN+ACK segment is received by a source (either the global source or a proxy source), the destination address which was originally the global sink is replaced with the source address of the SYN+ACK segment. This insures future segments will be directed to the correct proxy and avoids protocol brittleness caused by routing changes. Proxies generate a new SYN segment rather than forwarding the original one. This means that the source address will be the proxy's address, allowing the SYN+ACK segment to be directed back to the proxy. This is important because the proxy

may not be on the reverse path to the source. After the initial SYN segment, all future segments are directed to the proxies.

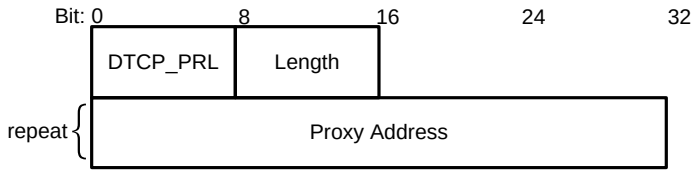

Figure 3

The DTCP proxy list TCP header option is shown in Figure 3. The first two fields are the TCP option number and option length as required by TCP. Following is a list of proxies by the node's IP address. The number of address is the list is determined by the length field. Assuming IPv4 addresses, the length divided by four is equal to the number of addresses in the proxy list.

In addition to the method described above, several other options were explored for performing the metrics collection and proxy setup steps. The possibility of using route discovery to collect metrics was already discussed and discarded due to cross-layer implementation concerns and caching concerns. It is possible to avoid the extra round-trip time imposed during metrics collection by combining it with the SYN segment. Proxies must then be set up after the SYN segment. The sink can potentially handle responsibility for proxy setup by attaching the proxy list to the SYN+ACK segment. Due to the potential divergence of the forward path and the reverse path, this solution is not recommended. Also, the sink would serve the function of assigning proxies. This function is typically reserved for the source or possibly some intermediary. Allowing the sink to assign proxies is a significant paradigm shift which deserves a thorough investigation.

Alternatively, proxy setup may be performed on the source ACK segment. However, such a solution suffers because there is no response to the source ACK. Without a response, the source does not know the address of the next proxy. The protocol becomes brittle in the eventuality of a path change after the source ACK segment and before the first data segment. Any attempt at proxy setup after this point in the connection effectively bypasses the TCP three way handshake and can have unforeseen repercussions.

2.3 – End-to-End Semantics

TCP guarantees delivery of data via acknowledgments from the sink to the source. TCP proxies interfere with the end-to-end semantics of TCP by acknowledging data before it is received by the sink. We resolved this issue by adding global sequence numbers (GSEQNUMs) and global acknowledgements (GACKs) to DTCP. In addition to the TCP sequence number, each segment has a DTCP global sequence number. The global sequence number remains the same when the segment arrives at the sink as it was when it was sent by the source. TCP sequence numbers are local sequence numbers only and can vary depending upon what each proxy's seed value is. We implemented GACKs as cumulative acknowledgements as in TCP Reno, however selective acknowledgements would function equivalently. When data is received by the global sink, a GACK is generated and is piggybacked upon a local ACK. Piggybacking the GACK prevents packet overhead on the network. The GACK is passed between proxies via piggybacking on local ACKs until it arrives at the global source. DTCP uses a timeout on GACKs to

detect a connection loss. If a new GACK is not received before the timer lapses, the connection is assumed to be broken.

Multiple GACKs may be received by a proxy before it is ready to send a local ACK. Multiple GACKs may be combined into a single GACK by select the greatest acknowledgement by value. Selective GACKs require merging logic at this stage since multiple selective GACKs may be received before the proxy is ready to send a GACK. The merging logic would not change the fundamental principles of proxy GACK piggybacking. Another scenario arises when the proxy no longer has any local ACKs to send. This commonly occurs at the end of a connection once all data has been sent by the source, though it may also occur during a connection if the source pauses communication for any reason. Proxies have a delayed GACK timer. Whenever a GACK is received, the timer starts. If no local ACKs are transmitted before the timer expires, a special GACK segment is generated. The segment has the GACK only flag set in the DTCP header. The GACK only flag overrides normal TCP handling of the segment and only the DTCP header is read. GACKs are guaranteed to be delivered to the source eventually and excess traffic is kept to a minimum by the delay timer.

The addition of GACKs allows for a performance improvement to proxy operation. Normally, a TCP proxy will receive TCP segments, reorder them into a data stream and split the data stream into segments to retransmit. However, GSEQNUMs provide a segment ordering independent of TCP sequence numbers. Proxies can send out segments in any order, as long as the GSEQNUMs remain unchanged. Therefore, rather than reordering and generating a data stream, the proxy sink maintains a list of what segments it has received, for the purpose of local acknowledgements, and forwards segments to the

proxy source as soon as they are received. The proxy source has a buffer of segment data and GSEQNUM ordered by GSEQNUM. The proxy source sends segments from this buffer. This can improve proxy sending rate and reduce the amount of buffer space required since the proxy source can send segments immediately which were received out-of-order. We call this new mechanism proxy per-segment forwarding.

The source and sink must keep global state data along with local state data. Data at the sink must be reassembled using the global sequence numbers now. The addition of global sequence numbers also adds a significant amount of overhead to segments in the DTCP header. Each segment carries a GSEQNUM and GACK for a total of 8 additional bytes of header. We justify the overhead with the compliance to TCP end-to-end semantics and offset the overhead with the throughput improvements provided by DTCP.

2.4 – Congestion Change Detection

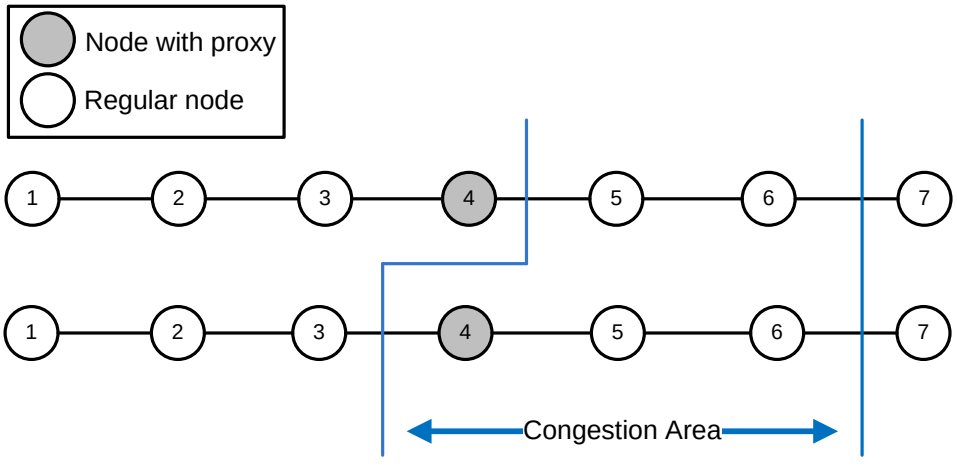

Figure 4

During the course of a long communication, it is possible for the congested zones along a path to change. When congestion change occurs, a proxy placement different than what is currently in use may improve connection throughput. Consider the situation in Figure 4. Originally, a proxy is placed on node 4. Over time, the congestion zone changes to encompass node 4. Our proxy placement heuristic suggests that a proxy at node 3 would perform better in terms of throughput.

To detect congestion change, the source performs the periodic action of requesting updated metrics along the path. It uses the same TCP option as was used during the initial stage metrics collection but piggybacks the header optional on an outgoing data segment. Metrics collection is piggybacked for two reasons: it reduces overhead on the network and, if there are no data segments to send, there is no need to check congestion. The metrics response is piggybacked on a GACK.

The metrics returned to the source included a field which indicates whether each node is a proxy (Figure 2). The SYN segment is always routed to the global sink so that when it is intercepted the proxies know where to send their own SYN segment. There is a window between when the metrics are initially collected and the SYN segment is sent where a routing change may occur. If that happens, the SYN segment will take a different path to the sink which may or may not include the proxies. As a result of this design, a subset of the proxies originally selected by the source may be in use. Metrics collection includes proxy information for an explicit list of what nodes are currently acting as proxies. We use the proxy list contained within the collected metrics as the current proxy list. It is used to determine if migrating will be beneficial.

Based upon the metrics returned, the source may decide to migrate to a new set of proxies. To decide, the proxy selection algorithm is re-run with the new metrics to obtain a new proxy list. Using the new proxy list and the current proxy list, the throughput estimate of each local connection is computed. The formula for the throughput of a TCP connection [Kurose and Ross, 2005] is

$$Throughput_{TCP} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

Equation 1

where maximum segment size (MSS) is constant. The other values in the formula, round trip time (RTT) and loss rate (L), are unknown. However, it is our contention that there is a direct relationship between TCP round trip time and link-layer transmission queue length (TQ) and MAC layer retransmission count (RT). There is also a direct relationship between TCP loss rate and MAC layer drop rate (DR). We estimate TCP throughput as

$$Throughput_{est} = \frac{1.22 \cdot MSS}{(TQ + RT)\sqrt{DR}}$$

Equation 2

The throughput estimate equation is unlikely to give an accurate value of throughput. TQ and RT are counting values which increase when the RTT increases in seconds. They are not the only source of increase in the RTT. MAC layer drops are also not the only source of TCP loss events. However, we expect a connection with a larger value of the throughput estimate to have larger real throughput than a connection with a smaller throughput estimate. We use the values from this formula for comparison of the quality of TCP connections.

The values TQ, RT, and DR are for an entire TCP connection which may span several nodes. To compute TQ, RT, and DR, the individual node values tq, rt, and dr from metrics collection must be summed. Link-layer transmission queue length and MAC layer retransmission count are simple summations shown in the following two formulas, respectively.

$$TQ = \sum_{x=a}^{b} tq_x$$

Equation 3

$$RT = \sum_{x=a}^{b} rt_x$$

Equation 4

MAC layer drop rate are events and must be treated differently. The addition rule for independent events is

$$P(A \cup B) = P(A) + P(B) - P(A)P(B)$$

Equation 5

Therefore, the cumulative drop rate across x nodes is given by the recursive formula

$$DR_a = dr_a$$

Equation 6

$$DR_b = dr_b + DR_{b-1} - dr_b DR_{b-1}$$

Equation 7

The drop rates of adjacent nodes may not be independent. If two nodes attempt to send at the same time, the result will be a collision causing a loss event at both nodes. The drop rate computed here is an upper-bound on the actual drop rate.

The throughput of each local connection is computed using Equation 2. Previous research in TCP proxies demonstrates that the overall throughput is dominated by the local connection with the lowest throughput [Ehsan and Liu, 2004]. The lowest throughput local connections of the new proxy list and the old proxy list are compared. If the estimated throughput computed for the new proxy placement is greater than that computed for the current proxy placement by a parameter threshold, the source migrates to the new set of proxies. See the

Appendix for pseudocode of the complete algorithm.

Several other policies for migration due to congestion changes have been explored. An alternative policy is to always migrate if there is a sufficient amount of data in route. The idea is that when a new proxy list is computed, it is strictly better than the current proxy list and data in route hides the cost associated with rerunning TCP slow start and creating proxies. The former should always be true assuming that our heuristics for proxy placement are correct. This policy does not, however, reflect the possibility of an insignificant improvement by migrating. If the new proxy list provides better performance by a very small margin, the benefits of migration may be outweighed by the costs. The policy chosen can handle insignificant improvements by a tunable threshold value.

2.5 – Path Change Detection

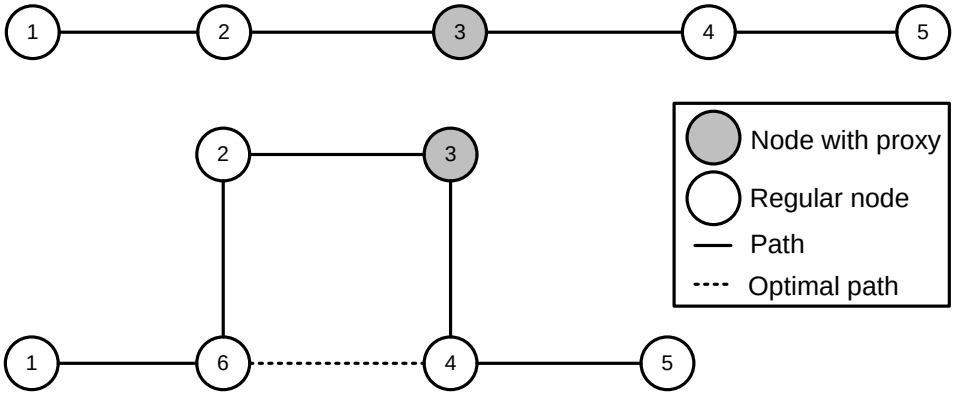

Figure 5

Mobility can cause the path used for routing between two nodes to change with time. For long TCP sessions, the path can change several times. TCP proxies lock a session to the

nodes along the original path. As a result, there may exist preferable paths between the source and the sink which do not include the proxies. Figure 5 shows the case where the proxy moves away from the source and sink. In Figure 5, we would much prefer for the session to follow the dashed path rather than continue through the proxy. We call this case "proxy drift." The ability to adjust proxy selection as the path changes will greatly improve throughput of the connection over its entire life span.

We explored several possible methods for performing path change detection. In addition to normal traffic from the source, we could send a polling packet directly to the sink collecting the path traversed and returning it to the source. Such a method is limited by the polling period and the overhead imposed by the extra packets and routing information, but could be combined with congestion change detection.

We opted to harness DSR Route Error packets. When a packet has been retransmitted the maximum number of times by the MAC layer and no acknowledgement has been received, DSR treats the link as "broken." It sends a Route Error to each node which has routed a packet over the broken link since the last successfully acknowledged packet [Johnson et. al., 2007]. We utilize Route Errors by adding a list of DTCP sessions to the DSR route cache at the source and proxy source. When the DSR layer on the source receives a Route Error, our DSR extension will select the DTCP connection(s) from the list which are routed across the broken link. Those connections will receive an alert from the network layer to the transport layer, indicating that a path change is occurring. When a proxy receives a Route Error alert, it sets the BKRN flag on the next segment returned to the global source. The source acts upon a BRKN flag as if it received a Route Error

itself. The source can thus be alerted to a broken route anywhere along the path to the global sink.

By using Route Error packets, we can detect path changes without adding any additional overhead to the network. It also gives us a more immediate notification of a route change than the previous method described, making the protocol more effective in a highly mobile environment. Our implementation requires DSR at the network level making it less generic, but is substantially more efficient than a polling method.

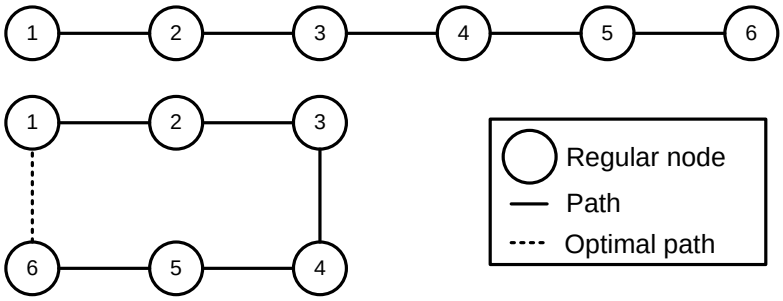

Figure 6

We further recognized that there are situations where the path selected by DSR is suboptimal and will not be broken. Consider Figure 6. The sink and several intermediate nodes move closer to the source such that the path is not broken, but a far better path exists directly between the source and the sink. We could perform better than DSR by detecting the better path. However, we stand by a policy of performing no worse than DSR. We believe that solving situations like Figure 6 is a network layer issue which should be resolved within the DSR protocol.

It is common for DSR Route Errors to be false positives. Particularly when congestion is high on a link, erroneous Route Errors may be frequently sent. DSR uses route requests to handle false positives and may continue to use the same route after receiving an

erroneous Route Error. When DTCP at the source receives a Route Error alert from the network layer, it transitions into the route recovery state during which it attempts metrics collection. TCP operation is not interfered with at this point. The metrics collection packet is directed to the global sink rather than an intermediate proxy. The metrics returned to the source will either be for a new path or the unchanged path. The former indicates the Route Error was valid and migration is immediately performed, while the later indicates a false positive. In the case of a false positive, the collected metrics are used in congestion change detection rather than be wasted.

2.6 – Migration

Migration is the operation of switching from a set of established proxies to a new set of proxies. It occurs when either congestion change detection or path change detection determines that migration is necessary. To perform migration, new local TCP connections need to be established and the global state data at the source and the sink must be connected to the new local connections.

The source begins by creating a new TCP session and copying the global state and any buffered data over to the new session. Calls from the application layer down are intercepted and directed to the new TCP session. Using the metrics collected during the congestion change detection or path change detection, the source enters the proxy setup phase. Proxies are established as described in proxy setup. Note that the previously constructed proxies cannot interfere with the new proxies because the source's port number has changed. Proxies effectively do not distinguish between a second DTCP

connection and a migrated DTCP connection. The source will send data from the buffer at the location where it stopped sending before migration.

NS2 uses a 2-tuple (destination IP and port) for demultiplexing incoming transport layer segments. A real world implementation would use a 5-tuple (source IP and port, destination IP and port, and flow id) for demultiplexing. The implementation here is based off performing a 5-tuple demultiplex within the TCP state handling. A real world implementation would handle segments by demultiplexing to a DTCP state a half layer above the TCP state.

When the SYN segment arrives at the sink node, it is demultiplexed to the current TCP session because the port number has not been changed. When the sink receives a SYN segment from a new source address/port, it performs sink half of migration. To prevent issues with misdirected SYN segments causing migration, we added a flow id to the DTCP header. The value of the flow id is chosen at random by the source at startup. It remains unchanged for the duration of the DTCP connection. Since the IP address, port number, and flow id must be correct to cause migration, the probability of a non-malicious segment causing migration is very low. Another node would have to randomly select the same flow id and direct a SYN segment to the same destination IP address and port. The sink moves the global state and data buffers to the newly created TCP session and returns a SYN+ACK segment to the (proxy) source of the SYN segment with the new sink port number. The buffers do not need to be copied; they may simply be referenced by the new TCP session since the old TCP session will not use them again.

The sink will continue to receive upon the old TCP session until it is closed. Any data already sent by the source at the time of migration will be received by the old TCP session and directed to the new TCP session to be reassembled by global sequence numbers for application delivery. If migration was caused by a proxy becoming disconnected or failing, some data may be lost. We recover from this by introducing global retransmits. Data is retransmitted using new local sequence numbers. At the sink, it is reassembled using the global sequence numbers, resulting in the correct final ordering of the data. Global retransmits are triggered by duplicate GACKs. Experimentally, we found that the threshold for duplicate GACKs must be very high due to GACKs from the sink lagging behind local ACKs from proxies. All data sent after migration is guaranteed to be delivered providing an upper bound on the range which must be retransmitted. Global retransmission mimics the behavior of TCP retransmission by continuing until either the upper bound is reached or a new GACK is received.

Once the new TCP connection is established, the old connection may be safely disposed of via standard TCP shutdown procedure. When a proxy sink is closed, it sends all buffered data from the proxy source and then closes the source as well. Eventually, all proxies will close and the sink will close as well. In the case of a disconnected or failed proxy, the sink on the opposite side of the failed proxy will eventually timeout and close. Because of the timeout required, a new DTCP connection between the source and sink will always be established before the previous connection is closed.

2.7 – Shutdown

Shutdown of a DTCP connection proceeds very similarly to standard TCP. The one exception is that the source must wait for the correct GACK value, rather than local ACK value. Once the GACK value equal to the highest global sequence number sent plus one is received, all data is guaranteed to have been delivered to the global sink. If timeout is reached before the GACK is received, the connection must be assumed to be broken and the amount of delivered data is unknown. When the TCP connection on the sink is closed and there is not another TCP connection already established, the DTCP state on the sink reaches the closed state.

Chapter 3 – Simulation

The protocol was validated and tested via simulation using the Network Simulator v2.31 [NS-2,]. DTCPv2 was built upon the TCP Reno implementation packaged with NS-2. In the validation simulations, wireless nodes are placed in a straight line in a flat grid. Congested nodes are simulated by adding two TCP Reno sessions across the congested node from a node directly above the congested node to another node directly below it. A warm-up time of 10 seconds is used to stabilize metrics data for DTCP's use. Simulations end after 1500 seconds. In almost all simulations, performance is measured during the transfer of a 1.44MB file. When a different file size is used, it will be noted. All simulations use the wireless model parameters listed in the

Appendix. These parameters simulate the Orinoco 802.11b card.

DTCPv2 is compared against DTCPv1, Split TCP [Kopparty et. al., 2002], and TCP Reno without proxies in the three simulations. Split TCP is an alternative method of proxy placement. It is not congestion aware and places a proxy every three nodes between the source and the sink. TCP Reno without proxies is referenced as NoProxy to conserve space and easier reading.

3.1 – Validation simulations

To validate the new protocol, we recreated the simulations used to compare DTCPv1 against established protocols in [Ouyang et. al., 2009]. The three simulations are labeled 9-node, 10-node, and 12-node after the length of the path between the source and the sink. Together, these simulations demonstrate the performance of congestion aware proxy placement along increasingly congested paths. In all simulations, node 1 is the source and the highest number node is the sink. The 9-node simulation has a single point of congestion at the sink, which will cause a proxy to be placed on the congestion edge at node 8. An additional proxy will be placed within the uncongested zone at node 4. The 10-node simulation has a single point of congestion at the 9th node, resulting in the same proxy placement as the 9-node simulation. Note that the 10-node simulation has one more congestion edge than the 9-node simulation and a congestion zone of 2 links, instead of 1 link. The final simulation, 12-node, has a congestion zone from node 4 to node 10. DTCP should place 3 proxies in 12-node: the beginning and end of the congestion zone, and one in the middle. In this idealized simulation, we expect any proxy placement to improve throughput and reduce congestion. Since the proxy placement algorithm used by DTCPv2 is unchanged from DTCPv1, we expect similar performance from both protocols. DTCP should improve performance the most by encapsulating the congestion zones.

To test the congestion change mechanism, we set up a simulation based off of the 9-node simulation. The 8th node is congested for the first 10 seconds of the file transfer. At 10 seconds, the cross traffic at the 8th node is shutdown and congestion is added to the 4th node. By DTCP's proxy placement algorithm, there should initially be a proxy a node 7. Once the congestion change occurs, the proxy placement should switch to node 3 and

node 5. We expect DTCPv2 to detect the congestion change within one metrics collection time (30 seconds) of occurrence and migrate to a new set of proxies. We expect DTCPv2 to perform better than DTCPv1 in this simulation because DTCPv1 will continue to use proxies which no longer encapsulate the congestion.

Path change detection was tested by a simulation with two paths from the source to the sink. One path has 8 hops while the other path has 13 hops. The short path is initially used by the file transfer for the first 10 seconds. After 10 seconds, node 5 along the short path moves away and permanently breaks the short path. The long path follows the short path until node 4, and then proceeds on a tangent for 6 hops and returns to the short path at node 6. This effectively creates a detour around node 5. After the short path is broken, transfer will continue using the long path. The simulation was run several times with the congestion zone along the long path increasing from 0 to 4. The congestion zone on the long path causes migration to be advantageous over maintaining the current proxy placement. The paths are set up and congestion placed so as to avoid breaking any of the protocols. If node 5 was a proxy for either DTCPv1 or Split TCP, the protocols would not finish the file transfer.

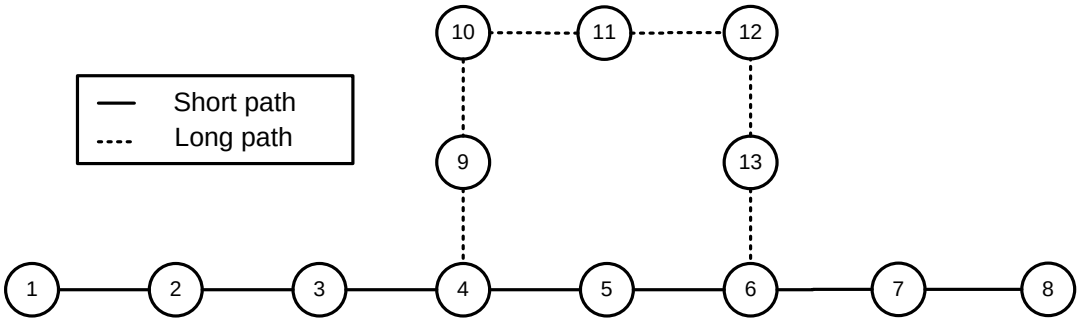

Figure 7

3.2 - Variant simulations

In addition to placing proxies at congestion edges, the proxy placement algorithm of DTCPv1 assigns proxies in uncongested zones every three nodes similarly to Split TCP's proxy placement algorithm. These extra proxies have a clear overhead, but an unclear benefit. To test what effect uncongested zone placement has on performance, we tested DTCPv2 with a new proxy placement algorithm which only places proxies on congestion edges. The variant of DTCPv2 using the modified proxy placement algorithm is labeled NUCP (No uncongested proxies). Proxies in uncongested zones have no effect on throughput and congestion if NUCP performs equivalently to DTCPv2. NUCP would be favorable for having a lower overhead than DTCPv2.

Proxy per-segment forwarding (see 2.3 – End-to-End Semantics) is a new mechanism. We postulate that it can reduce buffer space requirements on proxies and improve throughput by reducing the delay between when a segment is received by a proxy and when it is retransmitted. To test per-segment forwarding, we implemented the Split TCP proxy placement algorithm in DTCPv2 and call this new variant Split DTCP. Split DTCP varies from Split TCP by the start up, header, and metrics collection overhead of DTCPv2 and per-segment forwarding. Per-segment forwarding is the only difference between the two protocols which could have the potential to improve the throughput of Split DTCP over Split TCP. Comparing Split DTCP and Split TCP effectively isolates the performance effects of proxy per-segment forwarding. The buffer space reduction claims of proxy per-segment forwarding can be substantiated by observing buffer size in all simulations.

3.3 - Mobility simulations

The previous simulations are designed to demonstrate and validate the functionality of the DTCPv2 protocol under specific conditions. They do not represent real-world performance of the algorithm. To do this, we set up several simulations of MANETs consisting of 100 nodes. The nodes are initially arranged in a rectangle on a flat grid. The source and sink are selected so that the path between them was sufficiently long to cause proxy placement in all protocols. Six competing TCP Reno sessions are added to the network for congestion and run for the duration of the simulation. Again, a warm up period of 10 seconds is used to stabilize the initial metrics maintained within the nodes. Node movement is simulated using the Probabilistic Random Walk Mobility Model and the Random Waypoint Mobility Model as implemented by [Camp et. al., 2002]. Node movement is constrained within the original placement rectangle to maintain node connectivity. It is expected that the optimal path between the source and the sink will change several times during the simulation. Again, DTCPv2 is compared with DTCPv1, Split TCP, and NoProxy. We run the simulation with several rates of movement to simulate different levels of mobility in the network. The file size of the transfer is increased to 2.4MB to allow the file transfer to progress over several reconfigurations of the network.

The probabilistic random walk mobility model and the random waypoint mobility model are two commonly used methods of simulating random movement. Both models function by moving nodes at a set frequency. The probabilistic random walk model moves individual nodes at each clock tick. The random waypoint model moves a subset of nodes at each clock tick. We tested the probabilistic random walk model with 5, 2, and 1 second

between ticks and the random waypoint model with 50, 30, and 15 seconds between ticks. Note that node movement is not instantaneous. The previous movement may not have completed before the next clock tick.

These two models were chosen to demonstrate DTCPv2 under different potential movement conditions. The mobility simulations test DTCPv2 in an environment designed to be similar to a real-world MANET. Proxy migration should allow DTCPv2 to avoid the performance degradation caused by proxy drift in Split TCP and DTCPv1. Migration imposes a cost in setup time. The simulations are run at increasingly faster rates of movement to reveals whether migration can keep up with the movement and, if it cannot, at what point migration fails.

3.4 – Parameters

The protocol uses three parameters. The congestion edge threshold value remains unchanged from previous research. Of the two remaining parameters, the migration threshold was empirically determined from the congestion change validation test. In the test, the desired outcome is migration. Therefore, the upper bound of the threshold was set as the maximum value which would still result in migration within the simulation. At the upper bound, any congestion change at significant as in the validation test would cause migration. At the lower bound, a threshold of 0 will cause migration whenever congestion changes. We recognize that the cost of migration is significant. Therefore, we chose a migration threshold which is half way between the upper bound and the lower

bound. The threshold value is a balance between reacting to congestion change and the cost of migration.

The other parameter, the metrics collection rate, determines how often the source collects metrics about the path for use in congestion change detection. We observed in simulation that migration can take up to 10 seconds to complete in some simulations with heavy congestion, although the typical migration time was much lower. Metrics collection also increases the size of packets. Again, to strike a balance between reaction time to congestion change and metrics collection cost, we chose a value 30 seconds for the metrics collection rate. We stress that the values used for the migration threshold and metrics collection rate may not be optimal. They are sufficient for demonstrating the performance of the protocol through simulation. Further, any fine tuning of the values can only improve the protocols performance.

Chapter 4 - Results

Performance of the protocols was measured using four metrics. Source transfer finish time is defined as the difference between the time at which the source receives a local ACK for the final data segment and the time at which the first segment is sent by the source. The local ACK comes from the first proxy in the case of DTCP and Split TCP. It comes from the sink when no proxies are being used. The first segment sent by the source is the metrics collection segment for DTCPv2 and a SYN segment for all other protocols. This metric demonstrates the functionality of proxies to limit the time required to transfer on local connections. When the source transfer is finished, the path and nodes along the first local connection become available for other, unrelated traffic. Source transfer finish time offers no insight into overall performance and is the least important of the four metrics.

The overall transfer finish time is defined as the difference between the time at which the local ACK sent from the sink for the final data segment is received and the time at which the first segment is sent by the source. The local ACK is either received by a proxy in the case of DTCP and Split TCP and by the source when no proxies are being used. After the local ACK is received, DTCPv2 continues to send GACKs back to the source. However, DTCPv1 and Split TCP do not have this functionality so a fair comparison including it could not be performed. The time and data transfer required for the final GACK are not significant enough to change the results. Overall transfer finish time demonstrates the throughput experienced by the protocol during the transfer. Proxies are expected to

generally improve throughput and a proxy placement which balances the throughput of the local connections the best should have the best throughput and the smallest overall transfer finish time.

The channel capture cost metric is a measure of the cost in time per node imposed by a transfer upon the network. This metric was first introduced in [Ouyang et. al., 2009]. "We first compute, for each [local connection] (note that regular TCP has a single [connection]), the product of its transfer time and its hop-distance. We then take the sum of these computed products for all path sections. Intuitively, the metric defines the total amount of time when any link is engaged in the current TCP transfer. Note that this is an aggregate metric indicating the overall effect of the transfer on the network, as a particular link will not be continuously busy with an ongoing TCP transfer." A high channel capture cost value indicates a high time cost and high imposed congestion cost for a transfer. Note that channel capture cost does not account for the amount of available bandwidth of a channel that is used by the transfer during the capture time. For this reason, channel capture cost favors short term high bandwidth transfers over long term low bandwidth transfers.

Proxies should reduce the energy requirement of a transfer by reducing the hop-distance of retransmissions. When TCP Reno without proxies must retransmit a segment, it must be transmitted by each node along the path. If a proxy is placed along the path, then any segments needing to be retransmitted upon the local connection between the proxy and the sink do not need to be retransmitted by the nodes along the path between the source and the proxy. The number of node transmissions is reduced and the amount of energy required to complete the transfer is reduced along with them. This is a particularly serious

concern when mobile wireless nodes are running on battery. We measure the energy cost of a transfer by arbitrarily assigning a cost to transmission and reception (including overhearing). The cost is in units per second. A large packet will take longer to transmit and receive than a small packet. Once the transfer is complete, the sum of the energy consumed from each node along the path is the energy cost of the transfer. The protocol which reduces the number of retransmissions the furthest will have the lowest energy cost and impose the least congestion upon the network.

For each simulation, source transfer finish time, overall transfer finish time, energy cost, and channel capture cost are calculated. Related simulations are graphed together to demonstrate trends in performance. Split TCP and TCP Reno without proxies are labeled in the graphs as Split and NoProxy, respectively.

4.1 – Validation simulations

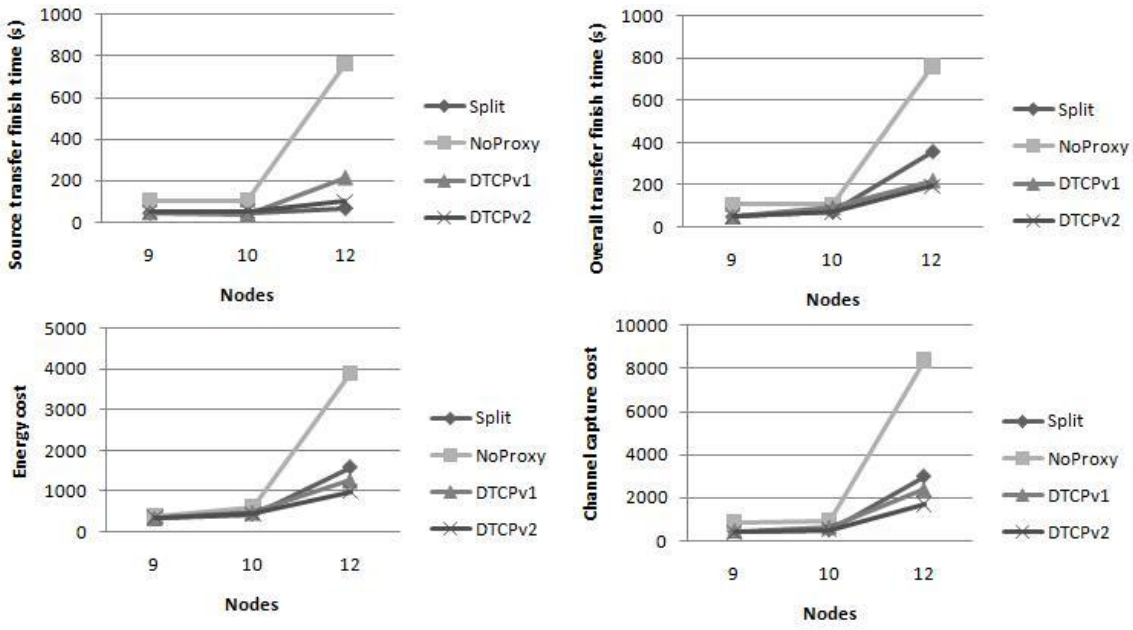

Figure 8

The results for the 9-node, 10-node, and 12-node simulations are shown in Figure 8. Using all four metrics, the three proxy-placing protocols perform significantly better than NoProxy as congestion increases from the 9-node simulation to the 12-node simulation. Split TCP completes source transfer the earliest with its first proxy placed at the third node from the source. Both versions of DTCP perform better than Split TCP in overall transfer finish time, energy cost, and channel capture cost. Because of the differences in metrics collection between DTCPv1 and DTCPv2, the two protocols placed proxies differently in these simulations. DTCPv1 collected metrics at some unfixed time before communication began, while DTCPv2 always collected metrics within one round trip time of when communication began. As congestion increased, DTCPv2 demonstrated an increasing performance improvement over all other protocols in terms of energy cost and channel capture cost.

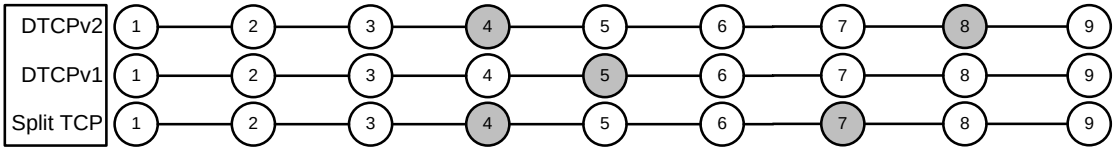

Figure 9

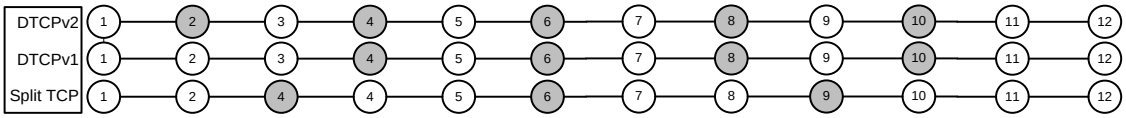

Figure 10

The protocols placed proxies for the 9-node simulation as shown in Figure 9. The 10-node simulation had the same proxy placement as the 9-node simulation. DTCPv1 did not place the same proxies as DTCPv2 because the metrics used by DTCPv1 were collected during the warm up phase before metrics data had stabilized. This is the cause of DTCPv1 performing worse than DTCPv2 in Figure 8. Figure 10 shows the proxy placement for the three protocols in the 12-node simulation.

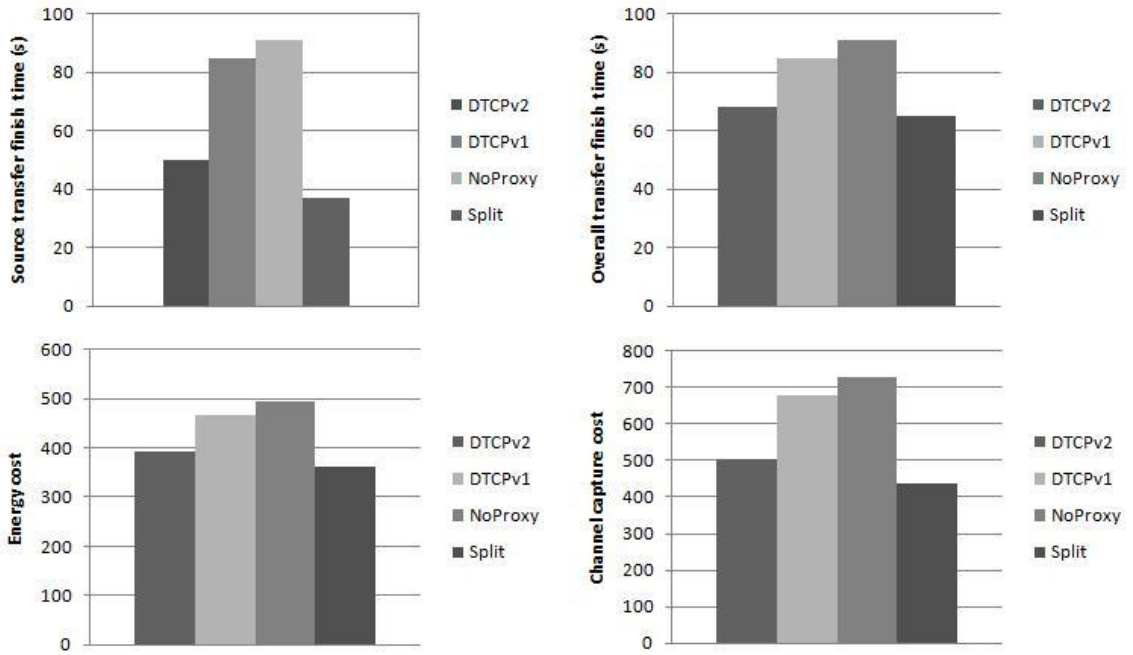

Figure 11

The results for validating the congestion change detection and migration mechanisms are shown in Figure 11. DTCPv2 reacts to the change in network congestion as expected by migrating from a single proxy at node 7 to proxies at both node 3 and node 5. The congestion change occurs 10 seconds after the transfer begins. DTCPv2 detects the congestion change 30 seconds after the transfer begins because the metrics collection rate parameter is set to 30 seconds. The three other protocols do not react to the congestion change. In all metrics, Split TCP has the best performance and DTCPv2 is second best. Because Split TCP places a proxy every three nodes, it performs equally well regardless of where congestion is placed along the path. DTCPv1 performs worse than TCP Reno without proxies in overall transfer finish time. Figure 12 shows the proxy placement of all three protocols within the congestion change simulation. DTCPv2's initial proxy placement is labeled DTCPv2 before, while its post migration proxy placement is labeled DTCPv2 after.

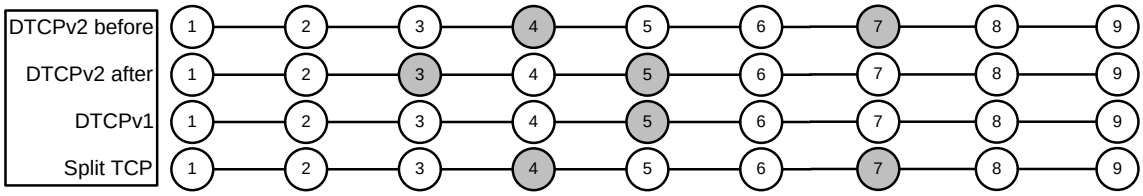

Figure 12

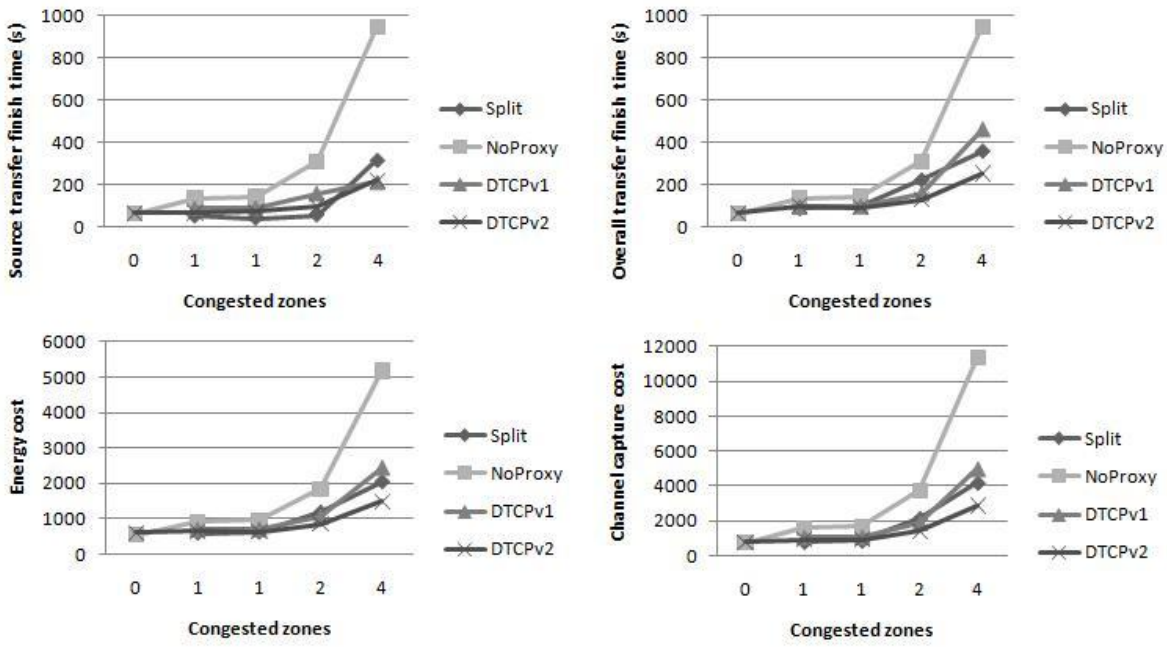

Figure 13

Figure 13 shows the results of the validation simulations for path change detection and migration. In the simulation with a 4 node congestion zone, the DSR Route Error was returned to the source after 12.5 seconds. After 34 seconds, the DSR layer on the source received a route response for the long path. 14 seconds later, DTCPv2 returns metrics and performs migration. 67.5 seconds after the route failure, DTCPv2 completes proxy setup and migration. Compared to No Proxy, which receives its first new ACK 44 seconds after the path change, DTCPv2 takes longer to return to a normal state. However, DTCPv2 is still continuing to send during the migration time period because the current proxies are still connected.

DTCPv2 performs far better than all other protocols in overall transfer finish time, energy cost, and channel capture cost as congestion upon the long path increases. Neither Split TCP nor DTCPv1 react to the path change. All three proxy-placing protocols still

demonstrate a significant improvement in all metrics over TCP Reno without proxies since all three protocols split the path by placing at least 1 proxy.

4.2 - Variant simulations

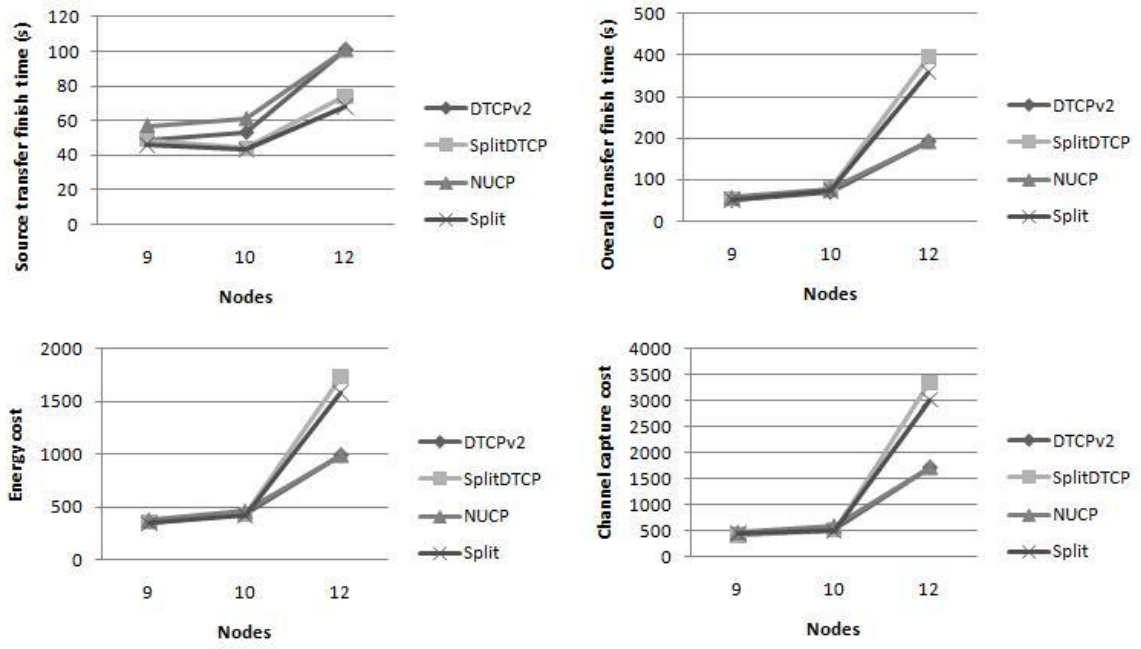

Figure 14

The results in Figure 14 are for three different proxy placement algorithms simulated with the DTCPv2 protocol. Split DTCP is a variant of the DTCPv2 protocol using Split TCP's proxy placement algorithm of a proxy every 3 nodes. NUCP is a variant of DTCPv2 which only places proxies at congestion edges. DTCPv2 and Split TCP are included for comparison. All four protocols are tested in the 9-node, 10-node, and 12-node validation simulations.

Split DTCP performs slightly worse than Split TCP in all simulations and across all metrics. The difference between the two is the added overhead in the DTCPv2 protocol in terms of header data and metrics collection and added start up cost. Split DTCP uses

proxy per-segment forwarding performance improvement. Per-segment forwarding is demonstrated here to not be a significant performance improvement in terms of throughput. The Split TCP proxy placement algorithm demonstrates superior performance in source transfer finish time, but inferior performance in the other three metrics when compared to the DTCP proxy placement algorithm or the NUCP variant.

NUCP performs equivalently to DTCPv2 in all simulations and across all metrics except source transfer finish time. In the 9-node and 10-node simulations, DTCPv2 places a proxy in the uncongested area close to the source. NUCP does not place a proxy there. The 12-node simulation does not have any uncongested zones large enough for proxy placement and NUCP and DTCPv2 perform identically in the simulation.

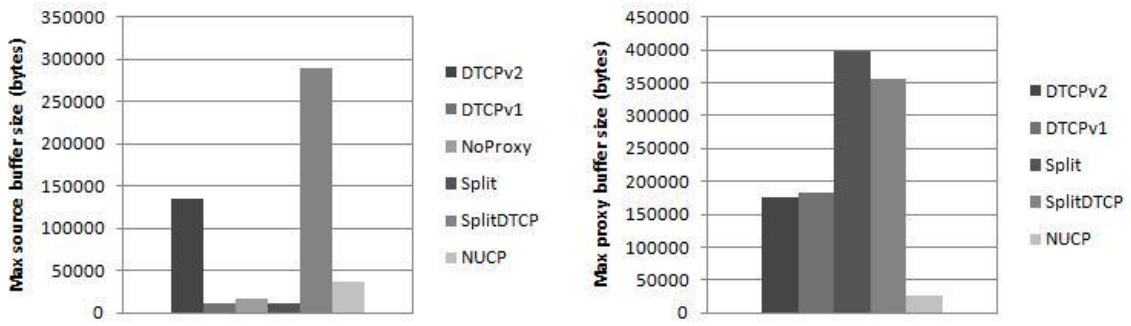

Figure 15

Figure 15 shows the maximum buffer sized required by all protocols in the 9-node simulation. DTCPv2 and Split DTCP require by far the most buffer space on the source because they must buffer data even after it has been locally acknowledged. DTCPv2 slightly reduces proxy buffer space compare to DTCPv1. NUCP significantly reduces buffer space.

4.3 - Mobility simulations

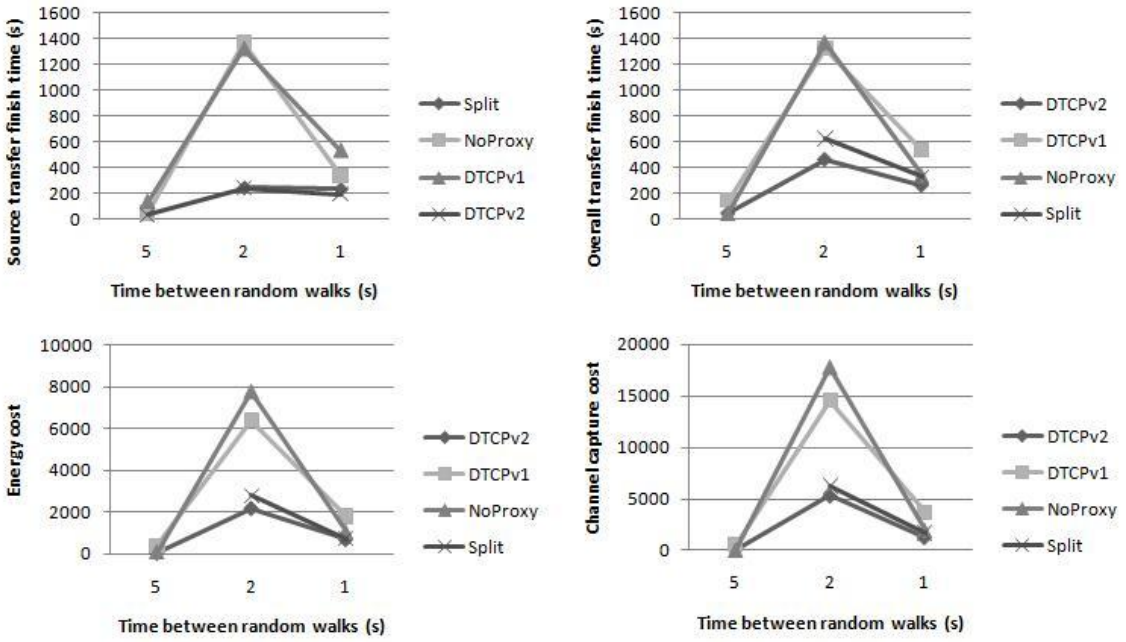

Figure 16

Figure 16 shows the results of simulations using the probabilistic random walk mobility model. The graphs have the results of the three simulations with different node movement frequencies. The simulations are ordered with lowest frequency on the left. Note that with between the 2 second frequency and 1 second frequency simulations, the performance of all the protocols improves. This is due to the unpredictable movement of nodes causing the path between the source and the sink to be shorter and less congested in the 1 second frequency simulation than in the 2 seconds frequency simulation.

DTCPv2 demonstrates performance equal to or superior than the other three protocols in terms of overall transfer finish time, energy cost, and channel capture cost. The 2 seconds frequency simulation illustrates what can happen to DTCPv1 when proxy drift occurs. In this simulation, DTCPv1 experiences only marginal performance improvement over

NoProxy. The proxy which DTCPv1 chose originally was the correct choice. Over time, proxy drift occurred and for a substantial amount of the transfer time DTCPv1's throughput was below that of NoProxy. Note that this can happen to Split TCP as well. DTCPv2 did not experience periods like this due to path change detection and migration.

Split TCP performs well in the 2 and 1 second frequency simulations. The difference in performance between Split TCP and DTCPv1 is caused by the proxies chosen by Split TCP exhibiting localized movement around the source and sink. Split TCP did not complete the 5 seconds simulation because a node which was chosen as a proxy became disconnected. Split TCP has no way of recovering from a failure of this nature.

During the transfer in the 1 second frequency simulation, the source and sink moved within direct communication distance. At this point, DTCPv2 migrated to zero proxies. DTCPv1 and Split TCP each had a single proxy in use which increased the number of hops and congestion imposed by the transfer. During the period when the source and sink were in direct communication, NoProxy performed better than all three proxy-placing protocols. It performed better than Split TCP and DTCPv1 because of the unneeded proxy in each protocol. It performed better than DTCPv2 because of the added header overhead. DTCPv2's performance was much closer to NoProxy than either Split TCP's or DTCPv1's performance.

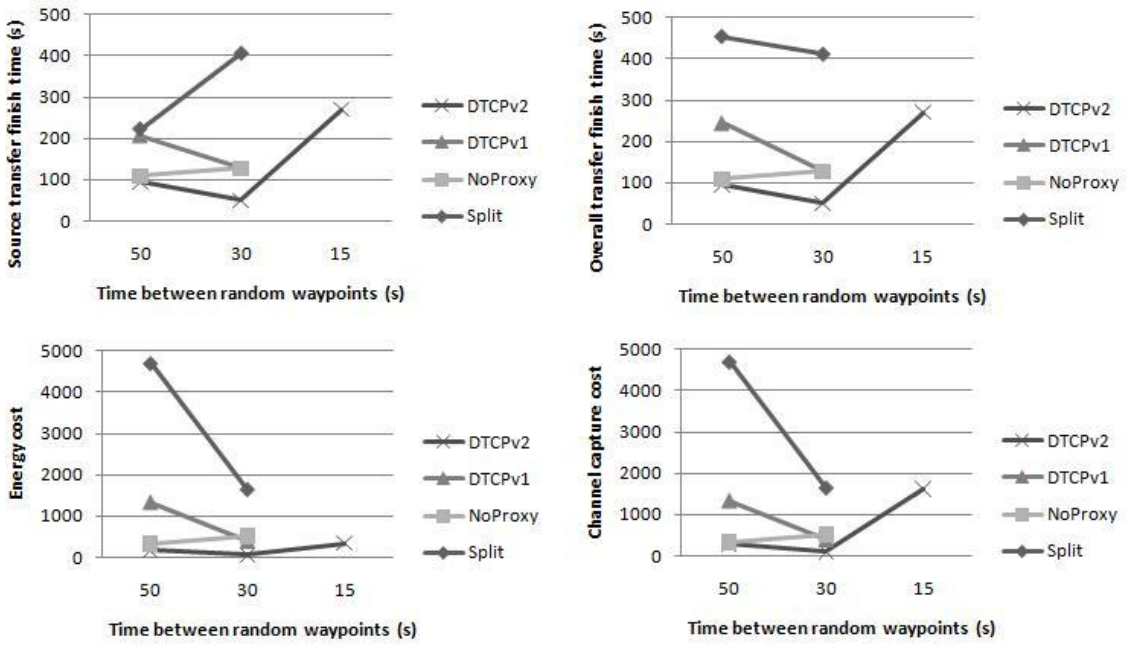

Figure 17

In Figure 17, the results of the simulations using the random waypoint mobility model are shown. Each protocol was run three times using movement patterns with 50, 30, and 15 seconds between movement waves. Again, performance fluctuates with different movement frequency due to the randomness of the system. DTCPv2 demonstrates consistently superior performance to the three other protocols in terms of overall transfer finish time, energy cost, and channel capture cost.

DTCPv2 was the only protocol to complete the 15 seconds between random waypoints simulation. NoProxy did not finish the transfer before the 1500 second simulation terminated. DTCPv1 and Split TCP protocols both broke the connection when a proxy became disconnected. This highlights a major flaw in both protocols. They introduce new points of failure into the system. In TCP, there are two points of failure, the source and the sink. DTCPv1 and Split TCP introduce a new point of failure for each proxy they place. DTCPv2 returns to a two points of failure system.

4.4 - NS2 Bugs

During development of DTCPv2 and simulation, several bugs were found in NS2. First, NS2 fakes TCP segment data with a data length field. The field's value is the imaginary amount of data in bytes contained within the segment. This value is overwritten while in transport causing problems in the cumulative acknowledgements of both the default implementation of TCP Reno and DTCP. To resolve this issue, we added a second data length field to the TCP header which is not included in the header length. The value of the data length field is overwritten by the second data length field value whenever the segment is delivered to the transport layer. Calculations performed by the simulator at lower layers may still use incorrect values of the segment length in computation, but the data length field is typically only inaccurate by a few bytes. Until the bug is resolved, these small inaccuracies are unavoidable.

DSR seems to exhibit self interference in conjunction with very congested links. Consider a transfer between a source and a sink which precedes normally along a path we label path A. There are one or more other paths between the source and the sink which have similar hop distances. Due to the lossy nature of wireless links, the MAC layer may drop packets even when path A is not broken. A DSR Route Error is generated and sent to the source of the packet. The source performs DSR route discovery to recover from a potentially broken path. DSR will receive a route reply first from the path which has the fastest round trip time. However, since the transfer between the source and the sink has been progressing along path A, it has become more congested than the other available

path(s). As a result, DSR will begin using an alternative path, path B. Again, a false positive DSR Route Error may be generated. Now, path B has become more congested than path A and DSR will begin using path A again. This flipping back and forth between paths is the self interference exhibited by DSR. If you include DSR route caches, the selection of path becomes even more random. This problem only occurs under very high congestion when the MAC layer cannot hide packet loss from higher layer protocols. The effect on DTCP is unneeded and expensive migrations.

The MAC layer includes functionality to improve performance when large packets are being transferred. Packets with a length greater than a threshold value use the RTS (ready-to-send) and CTS (clear-to-send) protocol extension to reduce the probability of interference while transmitting. Sending the very small RTS packet and waiting for the equally small CTS packet response before transmitting improvements throughput when dealing with large packets since it prevents most collisions. However, it has the opposite effect when dealing with many smaller packets. The RTS/CTS threshold default value in NS2 is set to zero, meaning that the protocol extension is used with each packet. The data transmitted is increased and delay is added. The threshold value should be set so RTS/CTS are only used with packets greater than 2347 octets [Calhoun et. al., 2009].

Chapter 5 – Discussion

During all simulations of the DTCPv2 protocol, we made several observations about its functionality. First, when a proxy is not placed correctly to balance the throughput of the local connections on either side it, the proxy buffer size can grow extremely large. When the rate at which the proxy receives data is approximately equal to the rate at which it sends data, the buffer remains small and the proxy performs optimally. Balancing the rates of the local connections on either side of a proxy is done placing proxies more frequently in and around zones of high congestion. NUCP reduces buffer space the largest amount, demonstrating how well balanced TCP throughputs on the local connections effectively maintain small proxy buffers. DTCP proxy buffers were typically smaller than or equal in size to the buffers of proxies placed using the Split TCP proxy placement algorithm. While per-segment forwarding was demonstrated to have no benefits in terms of throughput, it decreases the amount of buffer space required by proxies. Combining congestion aware proxy placement with per-segment forwarding, DTCPv2 decreases the buffer space requirement of proxies. Since the buffer is the major cost of a proxy, this represents a significant reduction in the overhead imposed on nodes by DTCP. As the number of proxies increases, the reduction in buffer space becomes more important.

With the inclusion of global acknowledgements, a DTCPv2 source has to buffer data which has already been acknowledged by a proxy but has not yet been acknowledged by the sink. The source buffer size in DTCPv2 was always larger than the source buffer in

DTCPv1 or Split TCP. It was at times larger than the buffer maintained by the source in TCP Reno without proxies. Since GACKs may be delayed at proxies for coalescing and piggybacking, a GACK takes longer to be received by the source than a TCP Reno ACK from the sink. DTCPv2 requires more buffer space on the source than any of the other protocols tested.

Even with congestion aware proxy placement, a perfect balance of throughput between local connections is not always possible. Rate throttling should be used to insure that proxy buffers do not grow too large. Rate throttling would have no effect on overall transfer finish time and throughput. The source transfer finish time would increase as would the channel capture cost since the send rate of the source would be limited to the send rate of the bottleneck local connection. However, both of these metrics have flaws. As mentioned previously, source transfer finish time does not offer any insight into overall performance. Channel capture cost only considers the time period during which a transfer is utilizing a link and not the amount of available bandwidth the transfer is using. A transfer which utilizes 100% of the available bandwidth for 10 seconds would have a lower channel capture cost than a transfer which utilizes 10% of the available bandwidth for 100 seconds even though both consume the same total amount of network resources. In networks with low congestion, the former is clearly advantageous since it completes the transfer quicker. While in networks with competing traffic and congestion, a protocol which shares bandwidth and does not over saturate links is advantageous. The above example suggests that the channel cost metric is flawed. The energy cost metric is a better representation of the congestion cost of a transfer upon the network.

DTCPv2 experiences a period of reduced throughput during migration. The three main expenses of migration are the time required by Dynamic Source Routing (DSR) to perform route discovery, DTCP metrics collection, and TCP slow start. Note that DSR route discovery must be performed regardless of whether migration is performed. Also, a path change will in most cases coincide with one or more TCP loss events. Either TCP fast recovery in the case of three duplicate ACKs or TCP slow start will occur. Considering how long DSR route discovery took in simulation, a TCP timeout and slow start is far more likely than three duplicate ACKs. Therefore, the most significant factor in migration performance is the round trip time required to collect metrics upon the new path. This is confirmed by simulation which shows that DTCPv2 returns to normal state after migration only marginally slower than the protocols return to normal state after the path breaks.

DTCPv2 has a higher cost in terms of DSR route discovery operations than the other three protocols. Whenever the path is broken, a DSR route discovery is performed between the source and the sink. Additionally, a DSR route discovery is performed by the source or a proxy along the old path. The second DSR route discovery is used to flush any buffered data and send the FIN segment to clean up. When a path is broken in DTCPv1, Split TCP, or TCP Reno without proxies only a single DSR route discovery is performed. DSR route discoveries can be expensive operations since they are broadcast.

While DTCPv2 increases the amount of header data at the transport layer by adding TCP header options, it reduces the amount of header data at the network layer. Using DSR and TCP Reno without proxies, the entire path from the source to the sink must be in the DSR header of each packet sent by either the source or the sink. When proxies are used, the

path is broken up so only small chunks of the path must be in the DSR header of each packet. The DTCP header is 12 bytes and, assuming IPv4, each address in the DSR header is 4 bytes. DTCP pays for its own header if it can reduce the number of addresses in the DSR header by three.

5.1 - Validation simulations

The 9-node, 10-node, and 12-node validation simulations clearly demonstrate the potential of DTCP to improve throughput while reducing congestion and energy consumption. Further, the revised metrics collection policy added to DTCPv2 is more reliable in an environment with stabilized metrics than the metrics collection policy of DTCPv1. Metrics collected by DTCPv2 are always less stale than those collected by DTCPv1. It follows that DTCPv2 metrics collection will likely perform better than DTCPv1 metrics collection in an environment where metrics data is constantly changing.

In the congestion change simulation, the proxy placement performed by DTCPv1 reduces throughput once the congestion change occurs. DTCPv1 performs worse than TCP Reno without proxies. Split TCP demonstrates robustness to congestion change along the path and has the best results of the protocols tested. This should be expected since Split TCP places proxies equally along the entire path; it does not matter where the congestion is located. The time required by migration and the overhead of congestion change detection within DTCPv2 are the source of its reduced performance compared to Split TCP. Note that both of these protocols place similar numbers of proxies. If the path were longer with more uncongested zones, DTCPv2 may have lower overhead in terms of number of proxies placed. However, in scenarios with short paths and frequent congestion change, a

congestion unaware proxy placement has less overhead than a congestion aware proxy placement and performs equivalently.

5.2 - Variant simulations

The NUCP variant performs equivalently to DTCP in all metrics showing that the added proxies in uncongested zones have no effect on throughput or congestion. However, the idealized simulations do not consider congestion change. In the previous validation simulation on congestion change, proxies in uncongested zones were shown to serve an important purpose. DTCP must be able to react immediately to congestion change to perform better than Split TCP. Considering the current conservative value for the metrics collection rate parameter, it is highly probable that DTCP's congestion change reaction time can be improve to match performance with Split TCP. If so, then NUCP variant is preferable for reducing overhead.

An alternative protocol would build off of Split TCP and add path change detection. The protocol would remain congestion unaware. The theoretical protocol would have equivalent performance to Split TCP in static networks and worse than DTCP performance. In a network with rapidly changing congestion, the alternative protocol may perform better than NUCP.

5.3 - Mobility simulations

The results of the mobility simulations show that DTCPv2 does not suffer from proxy drift like DTCPv1 and Split TCP. While DTCPv1 and Split TCP frequently perform worse than NoProxy, DTCPv2 consistently performs better than NoProxy. The results show that DTCPv2 is a valid protocol choice for ad hoc wireless networks with mobile nodes and any level of congestion. DTCPv2 improves throughput of the file transfer while reducing congestion and energy consumption. The former is evidenced by the reduction in overall transfer finish time. The later is evidenced by the reduction in energy cost. DTCPv2 does this while maintaining TCP end-to-end semantics which are not available in either DTCPv1 or Split TCP.

The DTCPv2 protocol can co-exist with non-DTCP supporting nodes. Its requires that both the source and the sink support DTCP. Non-DTCP supporting nodes will not attach metrics to DTCP segments with the MREQ flag set and will not be part of DTCP's knowledge of the path and will not be eligible to be proxies. Note that non-DTCP supporting nodes along the path potentially hurt performance by preventing optimal proxy placement. DTCP can co-exist with TCP since TCP congestion avoidance is unchanged.

5.4 – Security

DTCPv2 introduces several new states and operations on top of TCP. As a result, new attacks may exist against the protocol and existing attacks may become more damaging. We performed a brief survey of malicious attacks against DTCPv2. First, all DTCP-enabled nodes will set up a proxy when their address appears in the proxy list of a DTCP

SYN segment. An attacker could use this to expend the resources of a node. This is analogous to SYN flooding a TCP port in the LISTEN state. However, an attacker can exponentially increase the resources expended by picking a circuitous path and selecting all nodes on the path as proxies. Each node will maintain two TCP states while the attacker only maintains a single TCP state for the entire path. This attack can be mitigated by the same methods as SYN flooding but is more damaging. Note that DTCPv1 and Split TCP are also exposed to the attack.

Spoofing the source of a DTCP session can now be done by claiming to act as a proxy for the source. However, assuming the DSR protocol is in use, faking the path back to the source was already a possible way to spoof the source of a TCP session. DTCP introduces a new way to spoof the source, but does not alter the reality of it already being possible. The best way to mitigate spoofing is through authentication which is equally effective against DTCP proxy spoofing and DSR path spoofing.

The sink of a DTCP connection will migrate in all states except the closing states when it receives a SYN segment. The SYN segment must have the correct flow id. Effectively, any attacker able to overhear the flow id can perform an insertion attack, if the global sequence number is known, or break the connection. With the nature of wireless communication, it is very easy to overhear the flow id. If a node can overhear the flow id and global sequence number, it can also overhear local sequence numbers. Insertion attacks are already possible against TCP in ad hoc networks with knowledge of the sequence number. Using the DSR protocol, a malicious node can create a new segment and fake that it was received from the source and forwarded to the sink. The segment either contains data and results in an insertion attack or breaks the connection. DTCPv2

introduces a new method to perform insertion attacks, but the new method is no more viable than existing methods.

Chapter 6 – Conclusion

Previous research in DTCP showed the potential of congestion aware proxy placement to improve TCP throughput and reduce congestion in ad hoc networks. However, proxies harm performance in mobile networks by locking the connection to the nodes where proxies are placed. DTCPv2 enables proxy migration removing the lock. Through simulation, we showed that DTCPv2 has the same benefits as DTCPv1 in static ad hoc networks. DTCPv2 also improves throughput and reduces congestion in mobile ad hoc networks.

DTCPv2 improves over DTCPv1 with a new metrics collection policy which insures more accurate proxy placement. It can also update the proxy placement as congestion changes to maintain the benefits having proxies. DTCPv2 reduces the cost of proxies by reducing their buffer space requirement using out-of-order segment forwarding. Finally, DTCPv2 reintroduces TCP end-to-end semantics which are lost in other proxy-placing protocols. In all simulations and by all four metrics, DTCPv2's performance was no worse than DTCPv1. It often performs better.

All simulations were performed with the Dynamic Source Routing (DSR) protocol. We found that DTCPv2 reacts to path changes in equivalent time to TCP Reno timeout. This suggests that DTCPv2 can handle as frequent node movement as DSR can handle. DTCPv2 throughput is likely to remain better than alternative protocols up to the point of mobility breakdown.

When congestion upon the network is very high, DSR exhibit self-interfering behavior causing frequent path unneeded path changes. This is particularly true when DSR route caches short circuit route discovery. Unfortunately, each path changes causes a DTCPv2 migration at the transport layer significantly reducing throughput. DSR self-interference was only observed at very high levels of congestion. For this reason, DTCPv2 does not perform well when congestion is very high. DTCPv2 may perform better with other network layer protocols in highly congested networks.

Due to the overhead of metrics collection, proxy setup, and header additions, DTCPv2 is not recommended for short-term transfers. In very short-term transfers, DTCPv2 will perform worse than TCP Reno. In our simulations, a 1.44MB transfer was sufficiently large for DTCPv2 use to be beneficial. DTCPv2 is best used in an environment where congestion is present and nodes frequently move. It has been demonstrated to provide an increasing return as the hop distance of the transfer increases. Therefore, DTCPv2 is recommended for large transfers in MANETs where congestion is moderate, nodes are highly mobile, and the network size is large. When there is no network congestion, DTCPv2 and other proxy-placing protocols still improve throughput, but it is only marginally better than the throughput of TCP Reno.

Because DTCPv2 is not always beneficial, we recommend its implementation in real world protocol stacks as a socket option. Applications which intend to perform large transfers over MANETs may turn on the socket option while all other traffic will leave it off.

Chapter 7 - Future work

Global retransmits are unique in that there is both a single point during operation where they are necessary and a bound on how much data must be retransmitted. Global retransmissions are only needed after migration if one or more proxies lose connectivity or go done. This is a rare case in migration with node movement being the cause much more frequently. Also, since data sent after migration is guaranteed to be delivered, the only data which must be retransmitted is the global sequence number (GSEQNUM) greater than the most recent global acknowledgement (GACK) through the last GSEQNUM sent before migration. DTCP uses a large number of duplicate GACKs, 20, to detect data loss during migration. The potential result is a very long delay in delivering data to the application at the sink. The number of duplicate GACKs is very large because duplicate GACKs are created by local ACKs. When several local ACKs are sent by a proxy without an updated GACK, the segments are interpreted as duplicate GACKs by the source. A method of distinguishing between GACKs sent by the sink and duplicates created by proxies should be added to the protocol.

DTCP is handicapped when congestion changes more rapidly than the metrics data is queried. Since metrics collection requires one round trip time and migration requires TCP slow start, reacting to changes in congestion by migrating is not always advisable. Temporal congestion changes or mild congestion should not cause migration. In the simulations, Split TCP was shown to perform well under changing congestion by placing proxies uniformly along the path. A hybrid protocol which places proxies infrequently in

uncongested areas and delays migration until it ascertains that the congestion change is not temporary will likely perform better than either DTCPv2 or Split TCP. The proxies placed in uncongested areas will capture sporadic congestion and eventual migration will capture the addition or removal of long term transfers to network congestion.

A potential problem arises for the DTCP protocol when proxies have buffered data and the sender no longer has any data to send. In this situation, performing migrations to adjust for congestion change and path change is wasteful since at the least the first local connection will never be used. A policy which defers migration to only the local connections still in use would be less costly. To enable this policy change, the source must send a notification when it no longer has any data to send, a local FIN. The proxy which receives the local FIN would takeover congestion change detection and path change detection. Once the proxy empties its own buffer, it would forward the local FIN to the next proxy which repeats the operations. Note that this mechanism may not be necessary if rate throttling successfully maintains low proxy buffer size.

Because DTCPv2 causes congestion and reacts to congestion there is the potential for self interference. Is two DTCPv2 transfer follow the hops and one of the transfers migrates proxies, the congestion edges observed by the other transfer may migrate along with the proxies. The best resolution to this problem is a migration threshold which prevents the congestion caused by a single transfer from causing migration. If the migration threshold is high enough, we can avoid self interference all together.

We have observed that the extra round trip time required by DTCPv2 to collect metrics is the primary factor limiting DTCPv2 performance against other protocols during startup

and path changes. Several more dynamic ways of setting up proxies exist and should be further researched. First, we can combine metrics collection and proxy setup into TCP three-way handshake if nodes make a bid to become proxies. On the SYN segment, each potential proxy creates a dummy TCP state and attaches end point data to the SYN. On the SYN+ACK, the data is returned to the source which then chooses proxies and assigns them on the ACK segment. Another method would have proxies assign themselves on the SYN segment. Assuming each node has a method for collecting the metrics of nodes adjacent to it on the path, nodes assign themselves if they are on a congestion edge. To place proxies at regular intervals in uncongested zones, the node can use the route in the DSR header to determine the hop distance to the previous proxy. Proxy setup is completed entirely during the transmission of the SYN segment to the sink.

Appendix

Proxy Selection Algorithm

```
procedure proxy_selection_alg( M : metrics for nodes along the path )
defined as:
  // place proxies at congestion edges
  for each i in 0 to length(M)
      j = i+1
      metrics1 = M[i].RT + M[i].TQ
      metrics2 = M[j].RT + M[j].TQ
      if metrics1 - metrics2 >= DIFFERENCE_THRESHOLD
            M[j] is a proxy
      else if metrics2 – metrics1 >= DIFFERENCE_THRESHOLD
            M[i] is a proxy
      end if
  end for

  // fill proxies empty areas of 5 or more non-proxies
  do
      add_proxy = false; //assume this round will done the proxy
                         //selection
      first = 0;
      second = 0;
      for each i in 1 to length(M)
            if M[i] is a proxy
                  second = i;
                  if second - first > 6
                  // select a middle node as proxy
                        M[first + (second - first)/2] = 1;
                  end if
                  if ((second - first)/2 + 1 > 6)
                  //can add another proxy next to the one just added
                        add_proxy = true;
                  end if
                  first = second;
            end if
      end for
  while add_proxy
end procedure
```

Conditional Migration Algorithm

```
procedure cond_migrat_alg( M : metrics for nodes along the path )
defined as:

 // M includes the current proxy placement
 // NEW has the updated proxy placement if we migrate
 newM = proxy_selection_alg(M);

 newM_min_throughput = MAX_THROUGHPUT;
 currentM_min_throughput = MAX_THROUGHPUT;
 newM_DR = 0;
 currentM_DR = 0;
 newM_TQRT = 0;
 currentM_TQRT = 0;

 // Compute the throughput of each local connection
```

```
 // to find the minimum throughput local connection
 // for both the current M and new M
 for each i in 0 length(M)

  // Equation 7
  newM_DR = newM_DR + M[i].dr - (newM_DR * M[i].dr);
  currentM_DR = currentM_DR + M[i].dr - (currentM_DR * M[i].dr);
  // Equation 3 and Equation 4
  newM_TQRT += M[i].tq + M[i].rt;
  currentM_TQRT += M[i].tq + M[i].rt;

    if i is a proxy in new proxy placement
      // Equation 2
      throughput = (1.22 * MSS) / (newM_TQRT * sqrt(newM_DR));
            // Found new minimum
            if throughput < newM_min_throughput
              newM_min_throughput = throughput;
            end if
      // Reset for new local connection
      newM_TQRT = 0;
      newM_DR = 0;
    end if

    if i is a proxy in current proxy placement
      // Equation 2
      throughput = (1.22 * MSS) / (currentM_TQRT * sqrt(currentM_DR));
            // Found new minimum
            if throughput < currentM_min_throughput
              currentM_min_throughput = throughput;
            end if
      // Reset for new local connection
      currentM_TQRT = 0;
      currentM_DR = 0;
    end if
 end for

 if newM_min_throughput > currentM_min_throughput + THRESHOLD
  MIGRATE
 end if
end procedure
```

Wireless Model Parameters

```
Antenna/OmniAntenna set Gt_ 1 //Transmit antenna gain
Antenna/OmniAntenna set Gr_ 1 //Receive antenna gain
Phy/WirelessPhy set L_ 1.0 //System Loss Factor
Phy/WirelessPhy set freq_ 2.472e9 //channel-13. 2.472GHz
Phy/WirelessPhy set bandwidth_ 11Mb //Data Rate
Phy/WirelessPhy set Pt_ 0.031622777 //Transmit Power
Phy/WirelessPhy set CPThresh_ 10.0 //Collision Threshold
Phy/WirelessPhy set CSThresh_ 5.011872e-12 //Carrier Sense Power
Phy/WirelessPhy set RXThresh_ 5.82587e-09 //Receive Power Threshold;
Mac/802_11 set dataRate_ 11Mb //Rate for Data Frames
Mac/802_11 set basicRate_ 1Mb
```

References

[Bakre and Badrinath, 1995] A. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95), pages 136–143, Los Alamitos, CA, USA, May 30–June 2 1995. IEEE Computer Society Press.

[Bononi and Di Felice, 2006] Luciano Bononi and Marco Di Felice. Performance analysis of crosslayered multipath routing and MAC layer solutions for multi-hop ad hoc networks. In Proceedings of the 4th ACM international workshop on Mobility management and wireless access (MobiWac), October 2006.

[Calhoun et. al., 2009] P. Calhoun, M. Mountemurro, and D. Stanley. Control and provisioning of wireless access points (CAPWAP) protocol binding for IEEE 802.11. http://www.rfc-editor.org/rfc/rfc5416.txt.

[Camp et. al., 2002] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad hoc network research. Wireless Comm. and Mobile Computing (WCMC) Special Issue on Mobile Ad Hoc Networking, 2(5):483–502, 2002.

[Chakeres and Perkins, 2006] I. D. Chakeres and C. E. Perkins. Dynamic MANET on-demand routing protocol. IETF Internet Draft, 6 2006.

[Cohen and Ramanathan, 1997] Reuven Cohen and Srinivas Ramanathan. Using proxies to enhance TCP performance over hybrid fiber coaxial networks. Technical Report HPL-97-81, HP Labs, 1997.

[De Couto et. al., 2003] Douglas S. J. De Couto, Daniel Aguayo, John Bicket, and Robert Morris. A highthroughput path metric for multihop wireless routing. In Proceedings of MobiCom, 2003.

[Das et. al., 2006] Saumitra M. Das, Himabindu Pucha, and Y. Charlie Hu. Mitigating the gateway bottleneck via transparent cooperative caching in wireless mesh networks. In Proceedings of MobiCom, 2006.

[Draves et. al., 2004] Richard Draves, Jitendra Padhye, and Brian Zill. Comparison of routing metrics for static multi-hop wireless networks. In SIGCOMM, 2004.

[Ehsan and Liu, 2004] Navid Ehsan and Mingyan Liu. Modeling tcp performance with proxies. Computer Communications, 27(10):961 – 975, 2004.

[Haas and Agrawal, 1997] Z.J. Haas and P. Agrawal. Mobile-TCP: an asymmetric transport protocol design for mobile systems. ICC, pages 1054–1058 vol.2, 1997.

[Holland and Vaidya, 1999] G. Holland and N. Vaidya. Analysis of TCP performance over mobile ad hoc networks. In Proceedings of ACM MobiCom, 1999.

[Johnson, 1994] David B. Johnson. Routing in Ad Hoc Networks of Mobile Hosts. In Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications, pages 158–163. IEEE Computer Society, December 1994.

[Johnson et. al., 2007] D. Johnson, Y. Hu, and D. Maltz. The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4. http://www.rfc-editor.org/rfc/rfc4728.txt.

[Johnson and Maltz, 1996] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In Tomas Imielinski and Hank Korth, editors, Mobile Computing, pages 153–181. Kluwer Academic Publishers, 1996.

[Kim et. al., 2005] K.-H. Kim, Y. Zhu, R. Sivakumar, and H.-Y. Hsieh. A receiver-centric transport protocol for mobile hosts with heterogeneous wireless interfaces. Wireless Networks, 11(4):363–382, 2005.

[Kopparty et. al., 2002] Swastik Kopparty, Srikanth V. Krishnamurthy, Michalis Faloutsos, and Satish K. Tripathi. Split TCP for mobile ad hoc networks. In Proceedings of GLOBECOM, 2002.

[Kurose and Ross, 2005] James F. Kurose and Keith W. Ross. Computer networking: a top-down approach featuring the internet. Third edition. Pearson Education, Inc. pages 270-271.

[Luglio et. al., 2004] M. Luglio, M. Y. Sanadidi, M. Gerla, and J. Stepanek. On-board satellite split tcp proxy. IEEE Journal on Selected Areas in Communications, 22(2):326–344, 2004.

[Mitra et. al., 2007] Pramita Mitra, Christian Poellabauer, and Shivajit Mohapatra. Stability aware routing: Exploiting transient route availability in MANETs. In proceedings of 3rd High Performance Computing and Communications, September 2007.

[NS-2, 2009] Network Simulator version 2.31. http://www.isi.edu/nsnam/ns/ns-documentation.htm.

[Ouyang et. al., 2009] T. Ouyang, S. Jin, M. Rabinovich. Dynamic TCP Proxies: Coping with Disadvantaged Hosts in MANETs. In International Conference on Distributed Computing Systems Workshops, 2009.

[Padhye et. al., 1998] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In Proceedings of ACM SIGCOMM, 1998.

[Shieh et. al., 2005] A. Shieh, A. C. Myers, and E. G. Sirer. Trickles: a stateless network stack for improved scalability, resilience, and flexibility. In NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation, pages 175–188, Berkeley, CA, USA, 2005. USENIX Association.

[Snoeren et. al., 2001] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems, pages 19–19, Berkeley, CA, USA, 2001. USENIX Association.