MSOCKS+: An Architecture for Transport Layer Mobility*

Pravin Bhagwat† David A. Maltz‡ Adrian Segall§

October 28, 2001

Abstract

Mobile nodes of the future will be equiped with multiple network interfaces to take advantage of overlay networks, yet no current mobility systems provide full support for the simultaneous use of multiple interfaces. The need for such support arises when multiple connectivity options are available with different cost, coverage, latency and bandwidth characteristics, and applications want their data to flow over the interface that best matches the characteristics of the data. In this paper we introduce and analyze an architecture called *Transport Layer Mobility* that allows mobile nodes to not only change their point of attachment to the Internet within a corporate domain, but also to control which network interfaces are used for the different kinds of data leaving from and arriving at the mobile node. We implement our transport layer mobility scheme using a split-connection proxy architecture and a new technique called TCP Splice that gives split-connection proxy systems the same end-to-end semantics as normal TCP connections. We introduce the architecture, present its system aspects, investigate its performance and prsent its reliability properties. The analytical aspects of the protocol, in particular its pseudo-code, its properties and its validation are given in a related Technical Report.

1 INTRODUCTION

Current mobile nodes can choose between many types of wireless network interfaces, each with wildly different bandwidth, error-rate, cost, and latency characteristics. Mobile nodes of the future will each carry multiple network interfaces in order to take advantage of overlay networks [8]. Yet, current mobility support efforts do not enable applications to fully take advantage of more than one interface at a time. While current mobility support allows mobile nodes to move between subnets, or to change which interface they use as wireless services become unavailable, it does not support the simultaneous use of multiple interfaces, nor does it have the ability to specify which interface each individual type of traffic should be carried on. In this paper, we present a flexible system that gives mobile nodes control over which interface data flows to/from them, and an enabling technique, called *TCP Splice*, that preserves TCP's end-to-end reliability and correctness semantics while allowing connections to be redirected.

Given a diverse networking environment, application designers need a networking infrastructure that allows them to specify how particular streams of data should be communicated between a mobile node and a static correspondent host. Applications need to be able to specify the network interfaces over which each data stream should be sent and received. Since data streams correspond most closely to entities in the transport or session layers of the OSI network model, we call our architecture *Transport Layer Mobility*. Greatly simplified, the architecture provides a means for redirecting one of the endpoints of an existing transport session (e.g., a TCP connection or a series of UDP packets) to a new arbitrary address.

*A preliminary version of this paper was preented at INFOCOM'98

†Reefedge, Inc., Fort Lee, NJ, pravin@reefedge.com

‡Carnegie Mellon University, dmaltz@cs.cmu.edu

§Technion, Israel Institute of Technology, segall@ee.technion.ac.il

For an application designer, the natural way to think about the desired quality of service for the application's data packets is on a stream-by-stream basis. Consider the case of a video-conferencing application which deals with video, audio, and text streams (with the text being a transcript of the video). The application designer might want to express the notion that packets in the video stream should be sent over the link with highest available bandwidth and not sent at all if no cheap interface is available, while the audio packets should be sent over a low latency interface, and the transcript should be sent over the interface with the greatest geographic coverage.

Our Transport Layer Mobility architecture is built around a proxy that is inserted into the communication path between a mobile node and a static correspondent host. For each data stream from a mobile node to a static correspondent host, the proxy is able to maintain one stable data stream to/from the static host, isolating it from any mobility issues. Meanwhile, the proxy can simultaneously make and break connections to the mobile node as needed to migrate data streams between network interfaces or subnets.

Our protocol is designed to allow roaming within an Enterprise Network. It may be possible to extend it to handle global roaming, by designing an inter-proxy protocol, but the need for such an extension seems to be limited at this point, since we do not expect users to keep their laptops powered while moving to remote locations.

2 OVERVIEW

Many proxy-based architectures have been proposed to manage the interactions between resource-poor mobile nodes and servers on static hosts [15]. The typical proxy-based architecture places an intermediate unit called a proxy in the communication path between a mobile node and the servers with which the mobile node's applications converse. The proxy can then mediate the communication between server and client, and provide services on behalf of either. As examples of possible proxy services, proxies can: provide processing resources the client may not have; reformat information from the server to fit the mobile node, such as resizing GIF images for small screens; or use compression to reduce the bandwidth required between the mobile node and proxy, which is frequently a low quality link. Since the proxy is typically under the control of the same organization that owns the mobile nodes it serves, the proxy can be configured to support the peculiarities of its population of mobile nodes. The servers that mobile nodes access may be under the control of other organizations, who have little incentive to change their code to support the newest mobile nodes.

The networks of many corporations and schools with wireless networks follow a similar pattern, in which there is one wiring closet where the wired connections to the base stations of various wireless networks come together and connect to the building's wired networks. This interconnection point is a perfect place to put a proxy that supports mobile nodes, as it is already on the path that packets will travel between servers and mobile nodes. If the forwarding latency of the proxy is kept to around that of router forwarding latencies, the proxy architecture does not even increase the latency seen by the mobile node. In the general case, the network stack of the mobile can be thought of as actually being split between the proxy and the mobile node. Any transport protocol can be used to exchange information between the proxy and mobile node, so long as both the client and server see the expected end-to-end semantics from the communications session between them.

Most proxies operate on a *split connection* model (shown in Figure 1), where mobile nodes desiring to communicate with a static server first make a connection to the proxy and tell it which server they want to communicate with. The proxy makes a second connection to the static server and then reads data from one connection and writes it into the other, thereby allowing the client and server to communicate. Each logical *communication session* between mobile node and server is split into two separate *TCP connections*.

Proxies can support mobility by providing a way to switch the mobile-proxy connection while maintaining the proxy-static connection unchanged. Imagine a mobile node starting a TCP connection while using its wired network interface, so that the connection between the mobile node and the proxy uses the mobile node's wired IP address as its endpoint. If the mobile node is disconnected from its wired network, it could potentially contact the proxy using the IP address of its radio interface and ask the proxy to subsequently copy data from the server-proxy connection to the new mobile-proxy-via-radio connection, instead of the old mobile-proxy-via-wire connection. In this way, the mobile node can migrate its sessions from one network interface to the other.

The case in which a mobile node moves from one subnet to another subnet on the same interface can be handled in the same way, so long as the mobile node can obtain an address[1] for use on that subnet via a protocol such as DHCP or stateless address autoconfiguration in IPv6 [10][11][14].

Each network interface on the mobile node has its own IP address. We can therefore control over which interface data will travel from the static to the mobile node by choosing the IP address used by the mobile node as its endpoint in the mobile-proxy connection. We can also control the network interface over which data moves in the opposite direction, from mobile to the static server, by assigning the proxy several IP addresses — it does not matter which interface on the proxy the addresses are bound to. Both BSD and Windows define a notion of a "host route" that specifies to which network interface packets to a specific host address should be routed. The mobile node creates a host route for each of the proxy's addresses, such that packets sent to that address will go out a different interface of the mobile node. The mobile node can choose which interface that data to the proxy will flow out by picking the appropriate proxy address as the peer address for the mobile-proxy connection. Taking the two methods together, when the mobile node chooses its endpoint address and proxy's endpoint address to use for the mobile-proxy connection, it chooses on a session-by-session basis over which interfaces will data flow both to and from the mobile node.

All solutions that seek to control the interface over which data flows into the mobile node must involve a proxy or some other network entity between the static host and the mobile node, since it is the sender of a packet that chooses the destination address of the packet. In designs without a proxy, the mobile node can control over which interface it sends packets out, but unless the static host is modified to include all the mobility support features of a proxy, the mobile node cannot control which interface it will receive each packet over. Assuming mobile nodes will typically run client applications that receive more data than they send, it is critical to allow mobile nodes to control the inflow of packets.

Obviously, enabling mobility has its price. For example, consider the situation when the static server is located within the same domain as the users. In order to allow user mobility, we have to require that even in this case, the traffic will flow through the proxy, with the danger of proxy bottlenecks. The solution in this type of configurations may be proxy enhancements, like proxy clusters connected to a high speed back plane. Another limitation is that even when the mobile units are static and connected through the wired network, the MSOCKS+ procedures must be performed. With the popularity of 802.11, the trend may be changing and users may prefer wireless connections anyway. Even if not, one may consider augmented solutions when this problem becomes critical, where manual or automatic actions are requird when switching from the static connection through the wired network to the mobile one via the wireless network and viceversa.

A communication session composed of two TCP connections spliced together appears to the mobile and to the static server as a single TCP connection, and so is defined in terms of the IP addresses and port numbers of the connections' endpoints. Thus, changing the endpoint address of an existing session effectively breaks the session. Connecting and reconnecting two connections, as we propose in this paper, normally risks the loss of any data in flight while the reconnection happens, which would break the end-to-end semantics of

[1] This address is called a *co-located care-of address* in Mobile IP terms.

the logical mobile-to-server communication session. Our transport layer mobility solution, which we call *MSOCKS+*, is built around a technique we call *TCP Splice*. TCP Splice allows the machine where two independent TCP connections terminate to splice the two connections together, effectively forming a single end-to-end TCP connection between the endpoints of the two original connections.

Note that although the two TCP connections appear as a single TCP connection, the protocol cannot cope with IP layer cryptography, like IPSEC. However our solution can be used with higher layer crypto protocols like SSL and TSL, which are more prevalent in the Internet today.

The reasons behind the need for a layer 4 protocol as opposed to enhancements to MobileIP are:

- many enterprise IT managers are reluctant to deploy MobileIP because it requires upgrades to routers and client stacks.
- IT managers are more comfortable with a client-server style solution which minimizes changes to their existing infrastructures.

3 MSOCKS

As shown in Figure 2, the MSOCKS+ architecture consists of three pieces: a user level *MSOCKS proxy* process running on a proxy machine; an in-kernel modification on the proxy machine to provide the *TCP Splice service*; and a shim *MSOCKS library* that runs under the application on the mobile node. No modifications are needed in the static server, the server machine, the mobile node kernel, or the Mobile application (though to take maximum advantage of the TLM service, the application must be programmed to be mobility-aware, as described in Sec. 6).

In the remainder of this section, we first describe the protocol used between the MSOCKS library and the MSOCKS proxy. We then explain our TCP Splice technique that allows TCP connections to be arbitrarily reconnected, and describe our implementation of the MSOCKS library.

3.1 The MSOCKS+ Protocol

The MSOCKS+ protocol is built on top of the SOCKS protocol [9] for firewall traversal. Only two additions to the SOCKS protocol are needed to support MSOCKS's ability to redirect TCP streams to a mobile node's changing location.

First, we introduce the notion of connection identifier by which logical sessions between the mobile node and the proxy are tracked. The MSOCKS proxy issues a new connection identifier every time a mobile node makes a `BIND` or `CONNECT` request to the MSOCKS proxy asking to be connected to a correspondent host. The connection identifier is sent to the mobile node along with the normal SOCKS reply message that indicates the success of the request.

Second, we add the new MSOCKS `RECONNECT` and `RESTORE` requests. When the MSOCKS library wants to change the address or network interface that a TCP connection uses to communicate with the MSOCKS proxy, it simply opens a new connection to the proxy and sends a `RECONNECT` message specifying the connection identifier of the original connection. A connection's identifier contains the proxy port number the library should connect to when reconnecting to the identified connection. Upon receiving a `RECONNECT` message, the proxy unsplices the old mobile-to-proxy connection from the proxy-to-server connection, and splices in the new mobile-to-proxy connection. It also sends a `REPLY` message on this new connection. The server on the static correspondent host and the application on the mobile node need not be aware the reconnection has happened.

Another situation considered in this paper is the case when the disconnection of the old mobile-to-proxy connection occurs because of a proxy intermitent failure. If the proxy subsequently recovers within a reasonable amount of time, or an alternative proxy is found, the mobile-server connection can be reestablished. To accomplish that, the `REPLY` message sent by the proxy in response to `RECONNECT` reflects this situation by containing an ∞ variable. In this case, the mobile will follow up with a `RESTORE` message, that will allow the proxy to reestablish the required splice parameters.

As explained below, our splicing technique allows us to perform reconnections even when there is data in flight between the static correspondent host and the mobile node. Without care, these packets in flight may be lost or duplicated. The MSOCKS+ reconnect protocol together with TCP Splice ensures that the end-to-end reliable, in-sequence semantics of TCP are maintained. The reliability property is maintained across many classes of exceptions: a) no warning that the mobile node will need to change addresses, such as during hard hand-offs; b) repeated unsuccessful attempts to reconnect, followed by a successful reconnection; c) proxy failure, followed by proxy recovery or proxy hot-replacement.

Figure 3 shows the packets exchanged when an MSOCKS mobile client application connects to a static server on a correspondent host. The application's `connect()` call is intercepted by the MSOCKS library and turned into a call to `Mconnect()`. `Mconnect` first uses the mobile node's normal TCP stack to make a connection to the proxy, using whatever addresses are appropriate to the data the connection will carry. Over this connection, the library sends to the proxy the server's address and port number that the application gave as arguments to `Mconnect()`, along with any authentication information the proxy requires. Our splicing technique supports an arbitrary authentication negotiation with packets sent both from and to the proxy, although only a single packet is shown in the figure. After authenticating the mobile node, the proxy connects to the desired server and then splices the mobile-proxy and proxy-static connections together. When the splice is set up, the proxy transmits a final `OK` message to the mobile node to synchronize the MSOCKS library. The `OK` message contains the connection identifier the proxy has assigned to this session for use should the mobile node later want to reconnect it.

Figure 4 shows the packets exchanged when an application on a mobile node wishes to accept a connection from a server (e.g., the data connection during an FTP transfer). The application's call to `bind()` is intercepted and the code in `Mbind()` carried out. `Mbind()` connects to the proxy, which opens another socket (labeled **D** in the figure) and prepares it to receive a connection from the static server. The proxy then tells the MSOCKS library which address and port number the proxy is listening on, and the library returns this information to the application. When `Maccept()` is called, it blocks waiting for a message from the proxy notifying it of a server's connection[2]. By the time the MSOCKS library receives the `OK` message from the server, the mobile-proxy and proxy-static connections are spliced together so the application can communicate with the server as normal.

The MSOCKS+ protocol for accepting connections is roughly equivalent to the SOCKS protocol, and shares with it the drawback that it can accept single connections, but it cannot publish a port/address pair to which many connections are made. Since most mobile nodes will be acting as clients, this is not a significant limitation. A more complicated protocol can be used that gives the application on the mobile node the complete Berkeley Sockets semantics, allowing it to accept multiple connections to the same port, but we have not implemented this.

Figure 5 shows the packets exchanged when a connection between the mobile node and the proxy breaks for some reason, except for a proxy failure. Examples are when the mobile node moves and obtains a new

[2]A fortuitous side effect of this approach is that nonblocking applications, which `select()` on the bound socket before calling `accept()`, continue to work since `select()` indicates a socket has a pending connection by marking it as readable, which the `OK` message will do as well.

IP address, or it wishes to switch the session from one network interface to another. After the connection to the proxy is broken, the MSOCKS library opens a new socket, labeled **E** in the figure, and connects to the proxy using it. The MSOCKS library transmits a `RECONNECT` message to the proxy giving the connection identifier of the old connection to the server, along with a $ReadPntr_S$ variable, telling the proxy how many bytes of data the application has read from all previous connections, and a $WritePntr_M$ variable, telling the proxy how many bytes of data the application has written to the connection[3]. The proxy then splices the new connection to the proxy-server connection in place of the old mobile-proxy connection and closes the old connection. Once the splice is setup, the proxy sends a `REPLY` message to the MSOCKS library, along with variables $ReadAmount_S$ and $WriteAmount_M$, that direct the MSOCKS library how to complete the splice at the mobile node's end. If the `REPLY` message arrives at the Mobile before the latter moves again, the splice is completed, while the Mobile Application and the server at the static node are completely unaware the switch has happened. The proxy resumes data transmission to the mobile after sending the `REPLY`, the mobile resumes transmission of data afer receiving the `REPLY`. The $ReadPntr_S$, $WritePntr_M$, $ReadAmount_S$ and $WriteAmount_M$ variables are explained in detail in Sec. 3.2.3.

If the above exchange of messages is interrupted by another move of the mobile, we say that the previous reconnection is *incomplete* and the above procedure is reinitialized. In this case extra care is necessary because the proxy may have sent data on the first connection before it learns that the Mobile has moved again. The exact portion of this data must be appropriately salvaged so as to preserve the end-to-end reliability semantics. More details about the procedure appears in Sec. 3.3.

If on the other hand the connection is broken due to a proxy intermitent failure, that causes the proxy to lose its reference point, then upon recovery the proxy cannot send any meaningful values of the variables $ReadAmount_S$ and $WriteAmount_M$. The `REPLY` message will contain infinite values for these variables, that signal to the MSOCKS library in the mobile node that a recovery procedure is required. The Mobile responds with a `RECOVER` message that contains the connection reference points. The details are given in Sec. 3.4.

3.2 TCP Splice

The goal of a TCP Splice is to make it appear to the endpoints of two separate TCP connections that those two connections are, in fact, one. From the point-of-view of the endpoints, it should appear that they are directly connected by a single TCP connection with all the end-to-end properties of a normal TCP connection. The insight of the scenario that must be corrected by the TCP Splice is simple: data can be lost in non-Splice split connection proxy schemes because the proxy acknowledges the receipt of data to the static server node before receiving an acknowledgment (ACK) from the mobile node. Data which is ACK'd to the server but lost in transmission to the mobile node or mired in the kernel socket buffer of a broken connection, is lost forever. Similarly, the proxy acknowledges data to the mobile before receiving an ACK from the static server.

We implement the splice by altering all the packets received on one connection, including the acknowledgments, before sending them on the second connection, so the packets appear to belong to the latter. Since the alterations are a simple mapping function and require no storage, they can be done quickly in the kernel. Since the TCP Splice code itself does not generate acknowledgments, TCP end-to-end semantics are preserved between the two endpoints. Only after one end of the end-to-end connection receives data and transmits an ACK can the other end possibly receive an ACK, since the proxy only relays the ACKs.

[3]The subscript S indicates that the variable belongs to the Static-Mobile data flow direction, whereas a subscript M indicates variables belonging to the data flow in the opposite direction, from Mobile to Static.

3.2.1 TCP Background

Before describing TCP Splice, some background on TCP is required (see [13] for more detailed information). Figure 6 depicts a normal TCP connection with data in flight between endpoints. Each normal TCP connection is point-to-point and terminates at a *TCP socket* which is named by an address and a port number. A TCP connection is uniquely identified by the names of the two sockets at its endpoints. For each TCP socket, the normal TCP state machine maintains the following three counters:

- $SndNext$: The sequence number of the next data byte to be sent.
- $FirstUna$: The sequence number of the first unacknowledged data byte (equivalent to the sequence number of the greatest in-sequence ACK received).
- $RcvNext$: The sequence number of the next byte of data the socket expects to receive (equivalent to one more than the greatest consecutive sequence number received so far).

Using these counters, TCP assigns each byte of data sent over the connection a sequence number, so TCP can detect and recover from data loss, reordering, or duplication.

These counters define a *sequence space* associated with the socket. Data bytes with sequence numbers greater than or equal to $SndNext$ have not yet been sent. Data bytes with sequence numbers less than $RcvNext$ have been received by the TCP stack, but perhaps not yet read by the application. We say that data sent with sequence number N is acknowledged when the sender of the data receives an acknowledgment for it, namely when $FirstUna > N$. Denote by A and B respectively the sender and receiver of some data that was sent with sequence number N. While the data is in flight or lost, holds $RcvNext(B) \leq N$. Whenever the ACK is in flight or lost, holds $FirstUna(A) \leq N < RcvNext(B)$.

3.2.2 Mapping Sequence Spaces and Moving Packets

Consider first the data flow in the direction from the Static Server to the Mobile. Variables associated to this data flow contain subscript S. The proxy has two connections to splice together, one from Static and one to Mobile. Then the first data byte the proxy expects to receive from Static that will be forwarded on this connection to Mobile must be transmitted on the connection to the Mobile with the sequence number that Mobile next expects to receive from the proxy. The sequence number of the byte the proxy expects to receive from Static that is transmitted first on the current connection to Mobile is denoted by $MapStatic_S$. The initial sequence number on the current connection to Mobile will be denoted by $MapMobile_S$. Together, the pair $<MapStatic_S, MapMobile_S>$ define a mapping between the sequence number spaces of the spliced connections from Static to Mobile. In particular, the datum with sequence number N on the Static-to-proxy connection maps to sequence number $N - MapStatic_S + MapMobile_S$ on the proxy-to-Mobile connection.

We use similar notations, $MapMobile_M, MapStatic_M$ for the initial sequence numbers for the Mobile-to-Static direction. The datum with sequence number N on the Mobile-to-proxy connection maps to $N - MapMobile_M + MapStatic_M$ on the proxy-to-Static connection.

As each TCP segment is received at a spliced socket on the proxy, the segment's IP headers are altered to address the segment to the socket at the other end of the spliced connection. The segment's TCP headers are altered so the segment will be intelligible to the end system when it arrives — the segment will look like a continuation of the normal TCP connection that the end system first started with the proxy. To alter a segment for forwarding, the proxy needs only the state from the two sockets located on it (labeled **C** and **D** in the figures). In the discussion below, all variables referred to are those kept by the proxy. Processing a segment requires three steps: altering the IP and TCP headers, and checking for connection closing.

Alter IP header The following steps are used to alter the IP header:

- Change source and destination address to that of outgoing connection.
- Remove IP options from incoming packet.
- Update IP header checksum.

Alter TCP header The following steps are used to alter the TCP header:

- Change source and destination port numbers to match the outgoing connection.
- Map sequence number from incoming sequence space to outgoing space for Static to Mobile connection: $N \Rightarrow N - MapStatic_S + MapMobile_S$
- Map ACK number for Static to Mobile connection: $ack.N \Rightarrow ack.N - MapMobile_S + MapStatic_S$
- Map sequence number from incoming sequence number to outgoing space for Mobile to Static connection: $N \Rightarrow N - MapMobile_M + MapStatic_M$
- Map ACK number for Mobile to Static connection: $ack.N \Rightarrow ack.N - MapStatic_M + MapMobile_M$
- Update TCP header checksum.

The TCP and IP headers are updated incrementally, saving the time required to recompute them, while also preserving the checksum's error detection ability. TCP represents the urgent pointer as an offset from the segment's sequence number; it is not changed during the mapping procedure. In the Static-Mobile direction, a special check must be made to ensure that a segment being mapped does not contain data with sequence numbers less than splice base point $MapStatic_S$. If such a segment is received, the data up to $MapStatic_S$ is chopped out of the segment and an appropriate ACK is sent to the Static.

Connection Teardown As TCP segments are passed through the splice, they are examined for indications that the end systems are closing their connections. When the end systems finalize their connection, the TCP Splice code tears down the splice between the two sockets and frees the sockets on the proxy:

- If each side sends a FIN and ACKs the other side's FIN, then tear down the splice because the end systems have closed.
- If either side sends a reset (RST), tear down the splice.

3.2.3 Selecting the Base points

During the reconnect operation, the proxy must unsplice the connection between sockets **A-C** and **D-B** and splice in the connection between sockets **E-F** (see Figure 5). The base points for the splice must be carefully chosen to prevent any overlaps or gaps forming in the sequence space of the logical session between the Mobile and the Static server. There are three cases to be concerned with: two covering the data flow from Static to Mobile, and one covering the data flow from Mobile to Static.

Splicing the Static to Mobile Flow When data arrives at the TCP socket in the Mobile library, it is acknowledged to the proxy and placed in the buffers to be read by the Application. When a Mobile-proxy connection fails, the basepoint of the new connection is established by the proxy, depending on the relative situation between the acknowledged bytes and the bytes read by the Application.

In addition to the physical sequence numbers on each of the spliced connections, we define a *Virtual sequence number*, that corresponds to the absolute sequence space of the transmitted bytes on the end-to-end connection. We point out that virtual sequence numbers are not carried in messages. The first transmitted byte on the end-to-end connection has virtual sequence number 0. Subsequent bytes have consecutively increasing virtual sequence numbers, irespective of the proxy-to-Mobile connection they are sent over. The proxy knows the mapping from the Virtual to the physical sequence numbers by remembering the parameter irs_S, the *Initial physical receive sequence number* on the connection from the Static server.

We introduce a pointer $ReadPntr_S$ at the Mobile library, which keeps track of how much of the sequence space has actually been read by the application. $ReadPntr_S$ is initialized to 0 when the end-to-end connection is established and is incremented every time a byte is read by the Mobile application from a TCP socket buffer. Thus $ReadPntr_S$ represents one plus the Virtual sequence number of the last byte that has been read by the Mobile application. Another pointer is $SndUna_S$, at the proxy, which is initialized to 0 when the end-to-end connection is establshed and is increased according to the number of bytes acknowledged every time an ACK traverses the proxy from the Mobile to the Static. The variable $SndUna_S$ is one plus the Virtual sequence number of the last ack forwaded by the proxy from the Mobile to the Static. The proxy is sure that the mobile node has received at least all the data up to (but not including) $SndUna_S$, since it has seen a cumulative ACK from the mobile node for that data. If the Mobile-proxy connection fails, the proxy finds out the value of $ReadPntr_S$ when it receives the `RECONNECT` message ot the new connection. It then can compare it with the value of $SndUna_S$.

First consider the case when acks are sent by the MSOCKS library to the proxy faster than the Mobile application reads the data, namely $SndUna_S > ReadPntr_S$. Then $SndUna_S - ReadPntr_S$ bytes of data or more are present in the socket buffers at the Mobile. There can be more than $SndUna_S - ReadPntr_S$ bytes because new bytes may have been received after the Mobile has sent the `RECONNECT`. Since an ACK has been sent for the data up to $SndUna_S$, the proxy must assume that the static node may have received this ACK and so will never retransmit the data. The MSOCKS library must therefore drain the $ReadAmount_S = SndUna_S - ReadPntr_S$ bytes out of the old socket at the Mobile before freeing it and must offer those data to the application when the latter next reads from its MSOCKS socket. The value of $ReadAmount_S$ is sent by the proxy to the Mobile in the `REPLY` message. The base point for the new connection is now set such that the first byte sent by the proxy on the new connection to the mobile is the byte whose Virtual sequence number is $SndUna_S$. The variable $MapStatic_S$, which denotes the physical sequence number of the byte the proxy receives from Static that corresponds to the *first* byte sent on the new connection to Mobile, is therefore set to $SndUna_S + irs_S$. If the proxy has received bytes whose virtual sequence number is beyond $SndUna_S$, and thus $RcvNext_S(\mathbf{D}) > SndUna_S + irs_S$, these bytes are retransmitted by the Static if they were lost during the hand-off.

The second case is when data is read faster than it is acknowledged to the server, namely $SndUna_S < ReadPntr_S$. To avoid duplicating data, we choose $MapStatic_S$ to correspond to the $ReadPntr_S$ pointer, namely $MapStatic_S = ReadPntr_S + irs_S$. The next byte of data the application has not read will then be the first byte of data to be read from the new socket after the MSOCKS library receives the `REPLY` message. In this case the library does not need to drain any previous data, thus $ReadAmount_S = 0$. Any data the application has already read that is retransmitted by the server will fall before the $MapStatic_S$ point and will be chopped off and dropped by the proxy.

Splicing the Mobile to Static Flow In order to establish the basepoint for the data flowing from Mobile to Static, we must consider the data the Mobile application has written that the proxy has not yet seen acknowledged by the Static. When a Mobile-proxy connection fails, this data is present in the old socket but it is a gap in the sequence space of the new socket , the socket that will be responsible for retransmitting the data once the new connection is spliced in. Since some of the data in the gap may have been lost in flight, the mobile node must rewrite the data into the new connection. In order to be able to do this, the Mobile application must keep in a buffer all data for which the socket has not seen an acknowledgement yet and in a variable . The number of bytes written so far by the application library into the TCP sockets is kept in a variable $WritePntr_M$. The rewritten data, covers the gap left by the previous connection. In the case in which all the data in the hole has safely made it to Static, the next ACK from Static will acknowledge all the rewritten data. After the MSOCKS library has written data into the new socket to cover the gap, new application data can be sent via the new connection as normal. The proxy is able to calculate how many bytes of data the MSOCKS library must rewrite into the new connection upon receiving the pointer $WritePntr_M$ in the `RECONNECT` message. The amount is $WriteAmount_M = WritePntr_M - SndUna_M$, where $SndUna_M$ is a variable kept by the Proxy and denotes the number of acks that have traversed the Proxy. The value of $WriteAmount_M$ is sent by the proxy to the Mobile in the `REPLY` message. The base point for the new connection is now set such that the first byte received by the proxy on the new connection from the mobile is the byte whose Virtual sequence number is $SndUna_M$. The variable $MapStatic_M$, which denotes the physical sequence number of the byte the proxy sends to Static that corresponds to the *first* byte received on the new connection to Mobile, is therefore set to $SndUna_M + iss_M$.

3.2.4 Urgent Data

TCP defines a notion of urgent data [13] that is typically received either *inline* or *out-of-band* by the user-level application. The MSOCKS proxy must handle connections with out-of-band urgent data slightly different from those with inline urgent data, since there is the potential for pending out-of-band data to be overwritten by newly arriving out-of-band data before the MSOCKS library can include the pending data in the $ReadPntr_S$ pointer. Since the MSOCKS proxy sees all out-of-band data as it flows through, it is able to correct the $ReadPntr_S$ value reported by the MSOCKS library to reflect all the data that has traversed the connection.

3.3 Incomplete Reconnections

If the Mobile MSOCKS library does not receive the `REPLY` on a new connection within a preassigned time, it times out and tries to establish a new connection. On the other hand, the proxy starts sending normal data after sending the `REPLY` (see Fig. 5), so that data may arrive and be acknowledged on a proxy-Mobile connection on which the library has timed out. We refer to those connections as *incomplete reconnections*. `RECONNECT` messages sent on consecutive incomplete reconnections carry the same value of $ReadPntr_S$, but the `REPLY` messages may carry different values of $ReadAmount_S$, since acknowledged data at the proxy increases $SndUna_S$. When the MSOCKS library finally receives a `REPLY` before timing out, it will drain the required bytes from the sockets of the previously timed-out reconnections. The calculation of the correct number of bytes that needs to be drained from each of the incomplete reconnections is illustrated in the following example.

Denote the byte with virtual sequence number n by $b[n]$. Suppose that the MSOCKS library receives data on connection **A-C** in Fig.5 and times out after having received bytes $b[0]$ and $b[1]$. Thus, the value of $ReadPntr_S$ upon timing out, denoted by $\bar{R}$, is 2. We also assume that the library times out on connections

E-F and **G-H** (the latter not shown in the Figure) before receiving any message on those connections. Finally, the `RECONNECT` message on the subsequent connection **I-J** succeeds and the Mobile receives the `REPLY` message and the bytes following it. In this example, the library sends `RECONNECT(2)` to the Proxy on sockes **E,G** and **I**. Suppose that at the times when these messages are received, $SndUna_S = 3, 5$ and 9 respectively. The Proxy will send the difference between $SndUna_S$ and $\bar{R}$ in the `REPLY` messages, namely it sends `REPLY(1)`,`REPLY(3)` and `REPLY(7)` respectively. The first byte sent after `REPLY` will be $b[3], b[5]$ and $b[9]$ respectively. Since on connections **E-F** and **G-H**, the MSOCKS library times out before receiving the `REPLY` message, the contents of these messages waits in the buffers (we denote them by *init*). When the library receives the `REPLY(7)` message on socket **I**, it proceeds as follows: it reads all bytes from the buffer of socket **A**, in our example $b[2], b[3], b[4]$ and remembers that $b[5]$ is the next byte it should read; since $init[\mathbf{E}] = 1$, it knows that the first byte on **E-F** is $b[\bar{R} + 1] = b[3]$ and thus it discards the first 2 bytes on **E-F**, reads the rest and remembers that the next byte it should read is $b[8]$; since $init[\mathbf{G}] = 3$, it knows that the first byte on **G-H** is $b[\bar{R} + 3] = b[5]$ and thus it discards the first 3 bytes on **G-H**; since the `REPLY` received on **I-J** contains 7, it reads only one byte from **G-H**, this is byte $b[8]$, discards the rest and continues reading from connection **I-J**. The first byte on **I-J** is $b[9]$ and from now on connection **I-J** is the current connection. Note that the `REPLY` messages contain only the difference between $\bar{R}$ and $SndUna_S$. From these the library is able to figure out exactly how many bytes it should read from each connection.

3.4 Intermittent Proxy Failures

Failure of the proxy results in loss of state, in particular $SndUna_S, SndUna_M, irs_S$ and iss_M. In order to allow recovery, we keep a backup copy of the initial connection mapping variables irs_S and iss_M in the Mobile. These values are communicated by the proxy to the Mobile as soon as the end-to-end Static-Mobile connection is established. The MSOCKS library learns that the Proxy has lost synchronization from the `REPLY` message, because after failure and recovery, the Proxy responds to `RECONNECT` with `REPLY`(∞) . In this case, the MSOCKS library reads all bytes in its buffers and responds with a RESTORE message. The latter contains the irs_S and iss_M parameters, as well as the new values of $ReadPntr_S$ and $WritePntr_M$, which allow the Proxy to resume transmission and receipt on the new connection.

3.5 The MSOCKS Library

The MSOCKS library sits between the application and the kernel on the mobile node. Its task is to provide an interface to the application identical to that of the Berkeley Sockets API, while internally using the normal TCP stack of the kernel to provide mobility functions. We call the sockets exported by the MSOCKS library *Msockets*. The MSOCKS library works as a shim library. It intercepts calls made by the application to networking functions such as connect(), send(), recv(), and getsockopt(), and replaces those calls with code from the MSOCKS library.

There are numerous ways to insert a shim library between an application and a kernel; the best method depends on factors such as the ability to recompile the application, and OS support for shared libraries. On Windows platforms, we are implementing the MSOCKS library as a DLL that fits between the application and the WinSock DLL. On our BSD OS implementation, we are looking at using the shared library support, although we currently recompile the application.

In our BSD implementation, each Msocket is an integer, identical to how normal BSD sockets are represented to applications. The integer is the index of an entry in a table kept by the MSOCKS library — a kind of user level file descriptor table. The entry, in turn, contains all the MSOCKS data associated with that socket and its connection. Underlying each Msocket in our implementation is a real socket, and we

chose Msocket table entries such that the Msocket file descriptor is the same as the underlying, real file descriptor. This allows applications to use Msocket descriptors in the same way as all other file and socket and descriptors.

The basic operation of the library was already discussed indirectly in the Section 3.1. We will now describe how the MSOCKS library maintains or uses the $ReadPntr_S$, $WritePntr_M$, $ReadAmount_S$, and $WriteAmount_M$ variables.

As explained above, the proxy must know how many bytes of data the application has read from the server-to-client flow in order to properly choose the mapping basepoint for a respliced connection. To maintain this `ReadPntr` variable, the MSOCKS library must intercept all the calls that read from the Msocket, make the appropriate call to the underlying real socket, and update the counter with the number of bytes transfered. There is no data copying overhead, as the library does not make a pass over the data transfered.

If, while resplicing, the proxy finds that the application has been reading data more slowly than it has been arriving, the MSOCKS library will have to salvage the unread data from the old connection's socket before closing the old socket. This is the case described above when $SndUna_S > ReadPntr_S$. The proxy calculates how many bytes of data are left in the old socket as $ReadAmount_S = SndUna_S - ReadPntr_S$ and sends the MSOCKS library the $ReadAmount_S$ counter as part of the `OK` message for the `RECONNECT`. While an Msocket's $ReadAmount_S$ is greater than 0, the MSOCKS library directs all read calls to the old socket. When it drops to 0, read calls are directed to the new socket.

The final major task of the MSOCKS library concerns the $WritePntr_M$ and $WriteAmount_M$ counters and the gap described above that can form when resplicing the data flow from client to server. In order for the proxy to calculate the size of the gap, the MSOCKS library must count the number of bytes the application has sent into its Msocket so it can provide the proxy with the count as the $WritePntr_M$ field in `RECONNECT` messages. The proxy calculates the size of the gap and returns the value as the $WriteAmount_M$ field in the `REPLY` message. In order to cover the gap, the MSOCKS library must then write into the new socket the last $WriteAmount_M$ bytes that were sent by the application. This implies that the MSOCKS library must keep a copy of all data sent by the application into an Msocket, as well as writing it into the Msocket's underlying real socket. We use a circular buffer to store the data, and the buffer size is set as the minimum of the maximum TCP window size and the underlying socket buffer size. Luckily, most mobile nodes receive significantly more data than they send, so the data copying requirement imposed is minimal in practice.

By intercepting the `setsockopt()` call, the MSOCKS library can determine whether the application is receiving urgent data inline or out-of-band and can notify the proxy as appropriate.

4 FORMALIZATION and RELIBILITY

Reliable transport protocols make three guarantees with respect to the data they carry: that data will be delivered eventually, that the data will be delivered in the order it was sent, and that exactly the data that was sent will be delivered without omission or duplication. The first property is referred to (see [16],[17]) as the **Delivery** property, the combination of the other two as the **FIFO** property. All such reliable services are built around two basic elements: a numbering scheme by which each unit of data carried by the service is assigned a sequence number, and an acknowledgement scheme by which the data receiver tells the sender which pieces of data were correctly received. Thus another essential property is that data for which an acknowledgement has been received at the sender has indeed been received correctly by the receiver. This property is referred to as the **Confirm** property.

Formally, the definition of reliability of a connection [16],[17]as appropriate to our system is:

a) **FIFO** :

- Bytes that are read by the Mobile Application are read in order, with no gaps or duplicates.
- Bytes that are read by the Static are read in order, with no gaps or duplicates.

b) **Confirm** :

- A byte for which acknowledgement is received at the Static node has been placed in a MSOCKS library buffer.
- A byte for which acknowledgement has been received at the MSOCKS library has been received at the Static.

c) **Delivery** : Suppose that after a split connection is established and after an arbitrary finite number of failures of the proxy and/or of the proxy-Mobile session, the proxy and the session are up for a sufficiently long time. Suppose also that the timeout and the communication delays are such that `REPLY` is eventually received before the MSOCKS library times out. Then

- all bytes introduced by the Static node into the split connection are delivered in finite time to the MSOCKS library and are acknowledged in finite time at the Static node.
- all bytes introduced by the MSOCKS library into the split connection are delivered in finite time to the Static and are acknowledged in finite time at the MSOCKS library.

The pseudo-code of the protocol is given in [18]. Its main property is given in Theorem 1 below, whose proof appears in [18].

Theorem 1

The split connection is reliable, namely it satisfies the **FIFO, Confirm and Delivery** *properties.*

5 PERFORMANCE

All proxy architectures cause a concentration of traffic at the proxy, which raises scalability concerns. If each mobile node needed its own proxy to serve it, the transport layer mobility architecture would not be practical. Testbed evaluation shows that because the forwarding operation at the proxy is so cheap for spliced connections, the primary limitation on how many mobile nodes a proxy can handle is the link bandwidth in and out of the proxy.

To discover how many simultaneous connections an MSOCKS proxy can support, we ran both a Mobile library and a Static server program on the same machine as the MSOCKS proxy. By using the loopback interface to carry the traffic from Mobile to proxy and from proxy to Static, we avoided limitations on throughput resulting from a physical link and maximally stressed the proxy. The test machine was a 200 MHz Pentium Pro with 256KB of cache running BSDI BSD/OS. Figure 7 shows the total throughput achieved by the proxy as the number of connections through it was increased. Considering that fast wireless technologies support 1-3Mbps, we believe this data shows that the MSOCKS proxy is scalable.

In addition to supporting many connections, an ideal MSOCKS proxy would add minimal latency to the path of packets traveling to or from mobile nodes. Table 1 compares the latency seen by packets in a TCP connection routed via IP forwarding through our test machine with the latency seen by packets in a TCP connection spliced at our test machine. Latency measurements were made by configuring the test machine as a router between two Ethernet interfaces. The `tcpdump` program was used to record the time at which

Table 1: Summary of forwarding latencies created by TCP Splice and IP routing

	mean (msecs)	median (msecs)
IP forwarding	0.4038	0.0960
TCP Splice forwarding	0.4444	0.1120

each packet was received by an interface and the time at which that packet was written into the `IF_QUEUE` of the outgoing interface. While our test machine is not a commercial-strength backbone router, the rough equality of the forwarding latencies shows that inserting a TCP Splice between two connections does not create burdensome latency overhead. Note that `tcpdump` is a tool for sniffing packets and does not carry out any processing at the TCP layer. It intercepts packets at the device driver layer and timestamps arrival and departure of each packet. Nonetheless, the `tcpdump` tool does add some noise in the measurements reported in Table 1, but that noise is insignificant because even at full line speed the CPU is not the bottleneck in our system. In any case, the same noise is added in both observations (IP forwarding Vs TCP splice forwarding).

Another issue of concern is how quickly MSOCKS+ will be able to reconnect TCP connections after the decision to reroute a connection has been made. The time taken by the proxy to resplice two connections is insignificant. The greatest latency in reconnection results from the time required to establish the new TCP connection and transmit the `RECONNECT` message, which in turn depends critically on the roundtrip time (RTT) of the particular network technology being switched to. A protocol level analysis shows the greatest rate at which a mobile node can reasonably reconnect TCP sessions is limited to once per 2.5 round trip times: 1.5 RTT for the connection establishment, and 0.5 RTT for transmission of the `RECONNECT OK` message, and 0.5 RTT for the data.

6 DISCUSSION

We have been deliberately vague in describing the mechanism applications use to set and change the policies between mobile node and proxy, such as which network interfaces are used for which traffic. We see transport layer mobility as a generic mechanism that can be used along side others in a complete connectivity management solution for mobile nodes.

For example, there is a natural fit between the Odyssey project at CMU [4] and our transport layer mobility architecture. Odyssey is a system of wardens, one per data type, overseen by a viceroy that negotiates with user level applications to determine resource usage policies. Each warden is responsible for the control of all streams of some data type (e.g., movies, files, audio) entering or leaving the mobile node, and based on policy passed down from the viceroy, it determines fidelity levels for the data. Odyssey implicitly assumes that data servers will be able to alter the fidelity of the data they transmit based on signals from wardens on the mobile node. The transport layer mobility architecture enhances Odyssey wardens by allowing them control over which interface their data is sent over. Additionally, since few or no servers today are capable of negotiating data fidelity levels with clients, the TLM proxy provides an ideal place to put the transcoding services mobile nodes need to interoperate with unmodified servers.

At a philosophical level, mobility support systems can be tuned to support either *local mobility* or *global mobility*. Transport Layer Mobility is tuned to support local mobility, as we feel many mobile computer users, such as office workers, will not want to keep their connections up and valid during long moves. They may move often inside their buildings or between home and work, but they will shutdown their machines before leaving on a trip. While inside their buildings, the machine may move between a desk with a wired

network connection, meeting rooms with diffuse IR constrained to the room, and all the while never leaving the range of a building-wide radio network. Given this environment, we focused on a design that allows individual data streams to be rerouted, rather than rerouting packets.

Mobile IP [3][7] is concerned with global mobility, that is, maintaining a mobile node's connections by rerouting packets to it, regardless of where in the world the it happens to wander. As currently defined, Mobile IP largely assumes that there is only one way to reach a mobile node, and that all packets sent to the mobile node have equal priority. Since Mobile IP is a network layer function, it does not distinguish between the different types of data present in the packets it carries, and it has no way to handle them differently. Mobile IP can not support handling each transport session differently without violating the network stack layering.

Additionally, Mobile IP traffic can not currently cross corporate firewalls, as the protocol is firewall unaware, which limits the movement options of company mobile nodes to networks inside the corporate firewall. Since our mobility proxy functions can be integrated with other proxies, such as firewalls, MSOCKS mobile nodes can move outside their corporate firewall while retaining communication with both internal and external correspondents. MSOCKS+ by itself, however, does not solve the problem of traversing multiple firewalls.

Many researchers have focused on the TCP address matching function as the root of TCP's mobility problems. Several have proposed schemes which identify TCP connections by a unique identifier not based on the endpoint addresses [6][12], similar to the way our MSOCKS proxy assigns connections identifiers. These unique-identifier schemes typically require significant modifications to the correspondent hosts, which make the schemes hard to deploy. Furthermore, schemes based solely around connection identifiers are extremely insecure, in that any host overhearing the connection identifier can "capture" that connection by issuing a forged reconnect message. Since MSOCKS+ is built on top of the SOCKS firewall authentication system, it is already as secure as SOCKS itself.

Some unique-identifier schemes, like [6], require modifying the transport layer header so each packet carries the connection identifier. In MSOCKS+, packets are demultiplexed based on port number at the proxy, and connection identifiers are used only when reconnecting TCP connections. Since all connection identifiers used by a proxy are issued by that proxy, there is no problem maintaining identifier uniqueness. The Mobile Socket Layer design [12] describes a virtual port that performs the same functions of byte counting as the Msocket in our MSOCKS library. However, the Mobile Socket Layer is based on a unique-identifier scheme, where MSOCKS+ is based on TCP Splice and so is compatible with unmodified correspondent hosts.

The use of the MSOCKS shim library to support mobility on mobile nodes without requiring any mobile node kernel changes is not one of the major contributions of this paper. However, it dovetails well with TCP Splice and the overall Transport Layer Mobility architecture to support mobility on systems like Microsoft Windows95 where the code for the transport protocols (like TCP) comes from a third-party and can not be modified. Even the WinSock2 specification, which defines a Service Provider Interface [5] hook point above the transport protocol, does not provide sufficient access to easily implement mobility.

Bakre and Badrinath [1] proposed using a split TCP connection architecture both for mobility and for improving TCP's performance over wireless links, though their system significantly violates the normal end-to-end semantics of TCP. Their mobile support router (roughly equivalent to our proxy) acknowledges data to the server before it has been received by the mobile node, which could cause serious data integrity problems in the case of failed hand-offs. To the best of our knowledge, TCP Splice is the first technique that enables split TCP connection architectures while maintaining end-to-end TCP semantics. We see the issues of transport layer mobility and TCP's wireless performance as largely orthogonal. MSOCKS+ and

TCP Splice concentrate on providing mobility with correct end-to-end semantics. Systems like Snoop TCP [2] could be used underneath TCP Splice to improve the wireless performance.

7 CONCLUSIONS

Providing mobility support at the transport layer has both strategic and technical benefits. Strategicaly, adding mobility support to the transport layer allows us to add mobility support to applications running on operating systems, like Windows 98, where we do not have access to the network source code but can can intercept data above the transport layer. Technically, adding mobility support at the transport layer allows us to provide applications with a qualitatively different kind of control over their sessions: they can specify which interfaces are used for each type of traffic they exchange. The architecture allows mobile nodes to specify the both the network interface that packets in a session should be sent out on, and the interface that packets for the session should be received on.

Additionally, we have shown how our TCP Splice technique can be used to implement MSOCKS+ while preserving TCP's end-to-end reliability and semantics between a mobile node and a Static correspondent host, something no other split connection protocol does. These properties are stated precisely and proved in an associate Technical Report [18]. Finally, given the low cost of forwarding packets between two spliced connections, TCP Splice enables simple and scalable proxy services, like MSOCKS+.

References

[1] Ajay Bakre and B.R. Badrinaft. Handoff and system support for indirect TCP/IP. In *Second USENIX Symposium on Mobile and Location-Independent Computing Proceedings*, Ann Arbor, Michigan, April 10-11 1995.

[2] Hari Balakrishnan, Srinivasan Seshan, and Randy H. Katz. Improving reliable transport and handoff performance in cellular wireless networks. *ACM Wireless Networks*, 1(4), Dec. 1995.

[3] R. Droms. Dynamic host configuration protocol. Internet Request for Comments RFC 2131, April 1997.

[4] Brian D. Noble et Al. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, St. Malo, France, Oct. 1997.

[5] WinSock Group. Windows sockets 2 service provider interface, Aug 1997. Web White Paper - Revision 2.2.2, available as ftp://ftp.microsoft.com/bussys/winsock2/wsspi22.doc.

[6] C. Huitema. Multi-homed TCP, May 1995. IETF Working Draft - work in progress.

[7] David B. Johnson and David A. Maltz. Protocols for adaptive wireless and mobile networking. *IEEE Personal Communications*, 3(1):34–42, Feb. 1996.

[8] Randy H. Katz and Eric A. Brewer. The case for wireless overlay networks. In *SPIE Multimedia and Networking Conference (MMNC'96)*, San Jose, CA, Jan. 1996.

[9] M. Leech, D. Koblas, and et Al. SOCKS protocol version, April 1996. Internet Request For Comments RFC 1928.

[10] Charlie Perkins. IP mobility support. Internet Request For Comments RFC 2002, October 1996.

[11] Charlie Perkins and Tangirala Jagannadh. DHCP for mobile networking with TCP/IP. In *Wireless IEEE Internation Symposium on System and Communication, Alexandria, Egypt*, June 1995.

[12] Xun Qu, Jeffery Xu Yu, and Richard P. Brent. A mobile TCP socket. Technical Report Technical Report TR-CS-9708, The Australian National University, April 1997.

[13] W. Richard Stevens. *TCP/IP Illustrated, The Protocols*, volume 1. Addison-Welsley, 1994.

[14] S. Thomson and T. Narten. lPv6 stateless address autoconfigumtion, August 1996. Interet Request For Comments RFC 1971.

[15] Bruce Zenel and Dan Duchamp. General purpose proxies: Solved and unsolved problems. In *Proceedings of Hot-OS VI*, May 1997. http://www.mcl.cs.columbia.edu/ baz/ps/hot-os-vi.ps.

[16] A.E. Baratz and A. Segall. Reliable link initialization procedures. *IEEE Transactions on Communications*, 36(2):144–152, Feb 1988.

[17] G. Grover and A. Segall. A full duplex protocol on two links. *IEEE Trans. on Communications*, 40(1):210–223, Jan 1992.

[18] A. Segall, P. Bhagwat, and D. Maltz. Proxy-Based Mobile Networking: End-to-End Reliability and Surviving Proxy Failures. Technical Report CCIT Report Pub. 283, Technion, Israel Inst. of Technology, 1999. available from http://www-comnet.technion.ac.il/segall/Reports.html.

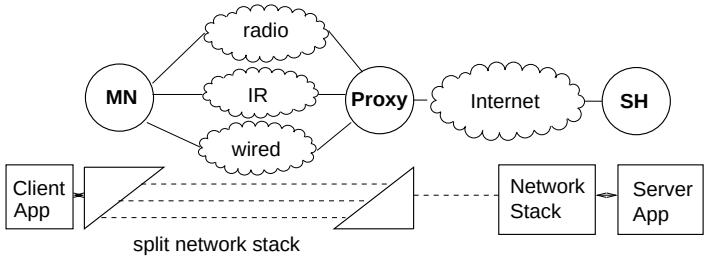

Figure 1: A common network topology showing the location of a proxy between the mobile node and the server host.

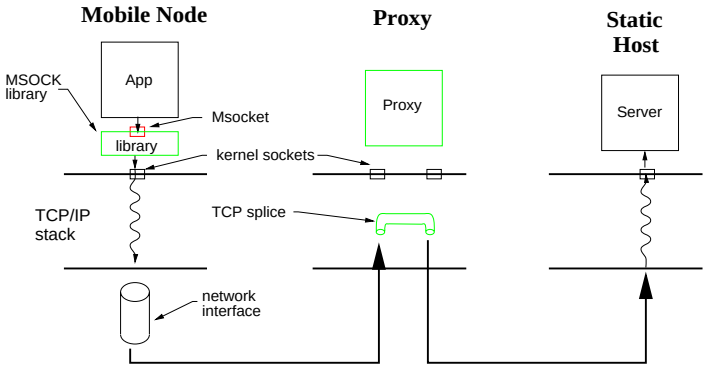

Figure 2: The MSOCKS+ architecture. Parts shown in gray are where MSOCKS alterations are made to the standard parts of proxy based client/server system.

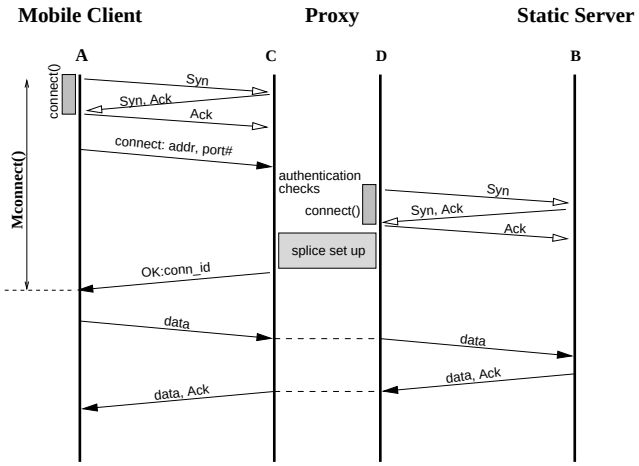

Figure 3: Packet exchange diagram for connection establishment between a MSOCKS client and a correspondent host via a MSOCKS proxy.

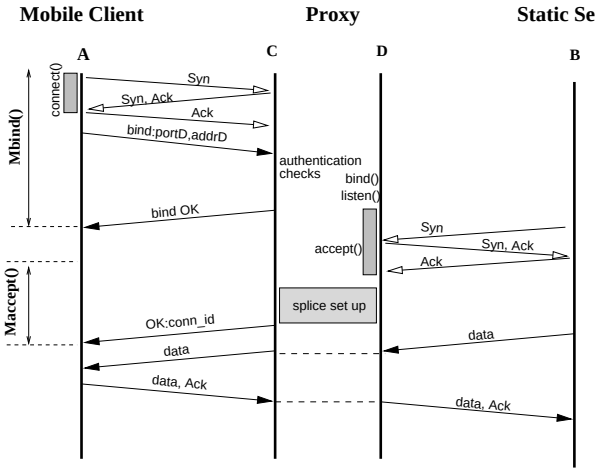

Figure 4: Packet exchange diagram for a MSOCKS client accepting a connection from a server on a correspondent host via a MSOCKS proxy.

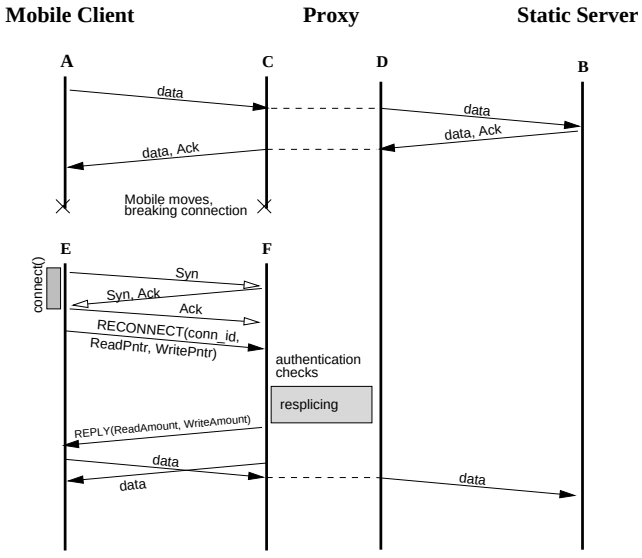

Figure 5: Packet exchange diagram for a mobile node reconnecting to an existing connection.

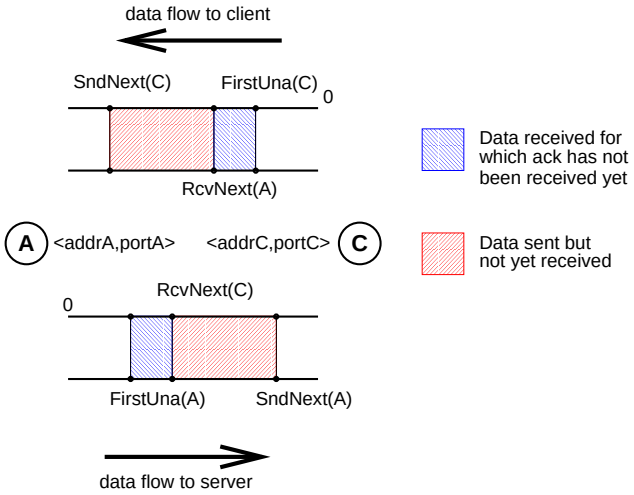

Figure 6: A normal TCP connection between sockets **A** and **C** with state counters labeled.

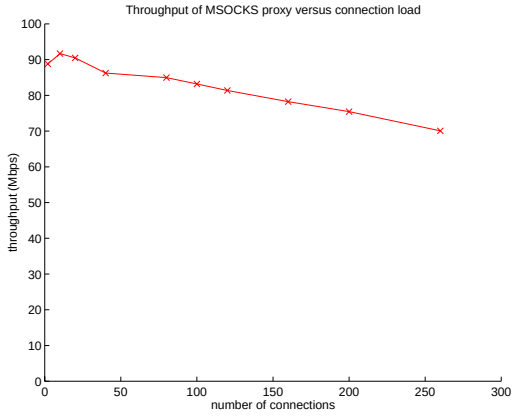

Figure 7: TCP throughput supported by the MSOCKS proxy as a function of number of simultaneous connections.