MSOCKS: An Architecture for Transport Layer Mobility

David A. Maltz
Carnegie Mellon University
dmaltz@cs.cmu.edu

Pravin Bhagwat
IBM T.J. Watson Research Center
pravin@watson.ibm.com

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. We present an architecture called *Transport Layer Mobility* that allows mobile nodes to not only change their point of attachment to the Internet, 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.**

Keywords— **mobile networking, proxies, TCP, connection redirection, SOCKS, firewalls**

I. 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 [7]. 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 the ability to specify which interface each individual type of traffic should be carried on. In this paper, we present both a flexible system that gives mobile nodes control over which interface data flows to them and 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 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* (TLM). Greatly simplified, the architecture provides a means for redirecting the endpoints of an existing transport session (e.g., a TCP connection or a series of UDP packets) to arbitrary addresses.

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 its correspondent hosts. For each data stream from a mobile node to a correspondent host, the proxy is able to maintain one stable data stream to the correspondent host, isolating the correspondent host 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.

II. OVERVIEW

Many proxy-based architectures have been proposed to manage the interactions between resource-poor mobile nodes and servers on correspondent hosts [16]. The typical proxy-based architecture places an intermediate host 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 (shown in Figure 1) 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

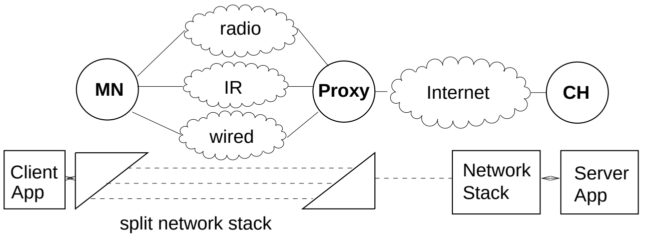

Fig. 1. A common network topology showing the location of a proxy between the mobile node and correspondent host.

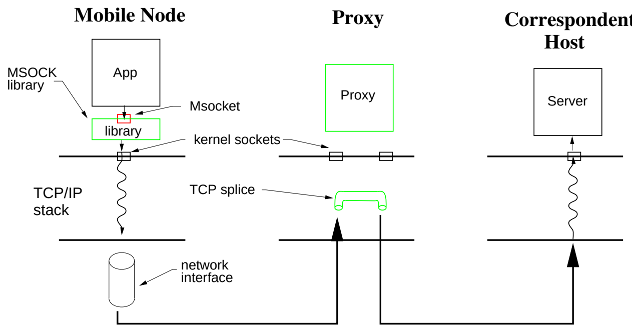

Fig. 2. The MSOCKS architecture. Parts shown in gray are where MSOCKS alterations are made to the standard parts of proxy based client/server system.

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 where mobile nodes desiring to communicate with a 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 server and then loops: it 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 in the hosts they serve by providing a way to switch the mobile-proxy connection while maintaining the proxy-server 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 another.

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 [3][11][15].

Since each network interface on the mobile node has its own IP address, we can control which interface data will move from server to mobile node over by choosing which IP address the mobile node uses as its endpoint address in the mobile-proxy connection. We control which network interface the data moves from mobile node to server over 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 the its endpoint address and proxy's endpoint address to use for the mobile-proxy connection half of a session, it chooses on a on a session-by-session basis which interfaces data will flow over both to and from the mobile node.

All solutions that seek to control which interface data flows into the mobile node over must involve a proxy or some other network entity between the correspondent host and the mobile node, since it is the sender of a packet that chooses the packet's destination address. In designs without a proxy, the mobile node can control over which interface it sends packets out, but unless the correspondent 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.

A communication session composed of two TCP connections spliced together appears to the client and 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 doing, normally risks the loss of any data in flight while the reconnection happens, which would break the end-to-end semantics of 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* [9]. 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.

III. 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 server, the server machine, the

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

mobile node kernel, or the client application (though to take maximum advantage of the TLM service, the application must be programmed to be mobility-aware, as described in Section **??**).

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.

A. The MSOCKS Protocol

The MSOCKS protocol is built on top of the SOCKS protocol [8] for firewall traversal. Only two additions to the SOCKS protocol are needed to support MSOCKS's basic 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` request. 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. The server on the correspondent host and the application on the mobile node need not be aware the reconnection has happened. As explained below, our splicing technique allows us to perform reconnections even when there is data in flight between correspondent host and mobile node, or when there is no warning the mobile node will need to change addresses, such as during hard hand-offs. 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.

Figure 3 shows the packets exchanged when an MSOCKS client application connects to a 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 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-server connections

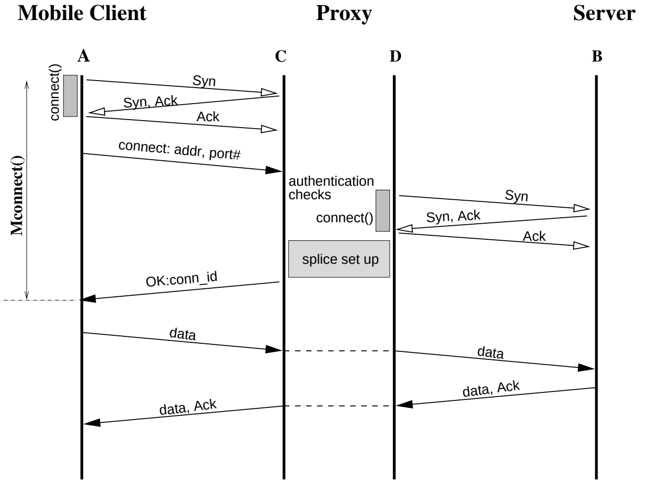

Fig. 3. Packet exchange diagram for connection establishment between a MSOCKS client and a correspondent host via a MSOCKS proxy.

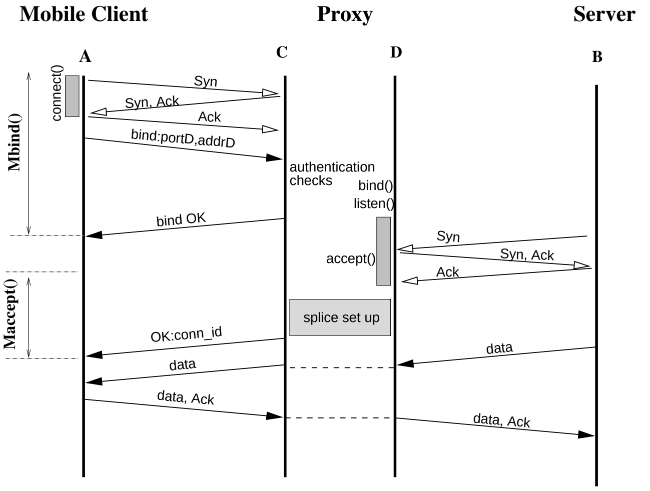

Fig. 4. Packet exchange diagram for a MSOCKS client accepting a connection from a server on a correspondent host via a MSOCKS proxy.

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 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

[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.

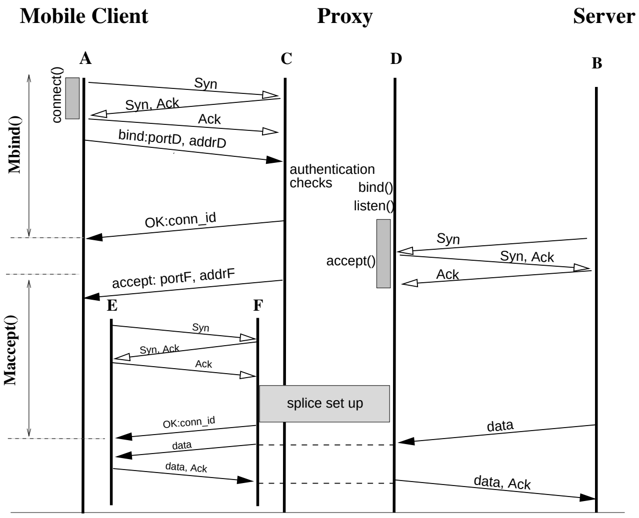

Fig. 5. Packet exchange diagram for complete BSD semantics.

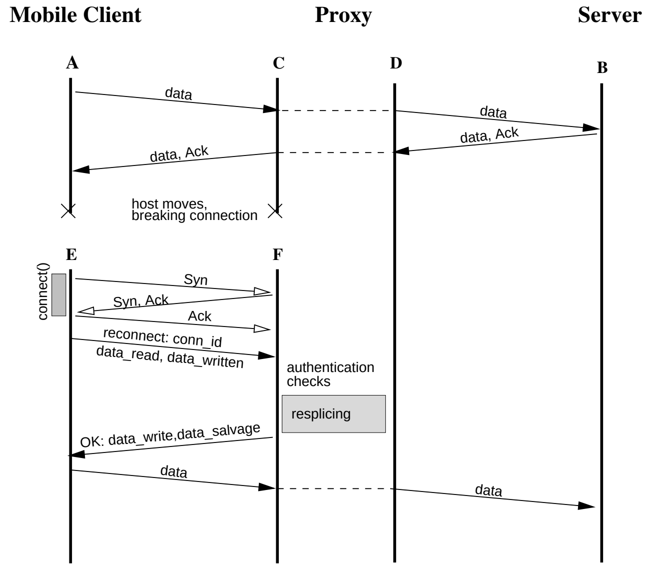

Fig. 6. Packet exchange diagram for a mobile node reconnecting to an existing connection.

MSOCKS library receives the OK message from the server, the mobile-proxy and proxy-server 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. As shown in Figure 5, 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 6 shows the packets exchanged when a connection between a mobile node and proxy breaks for some reason (e.g., the mobile node moves and obtains a new 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 `data_read` counter, telling the proxy how many bytes of data the application has read from the connection, and a `data_written` counter, telling the proxy how many bytes of data the application has written to the connection. 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 an `OK` message to the MSOCKS library, along with `data_write` and `data_salvage` counters directing the MSOCKS library how to complete the splice at the mobile node's end. The application and server are completely unaware the switch has happened. The `data_read`, `data_written`, `data_write`, and `data_salvage` counters are explained in detail later.

B. *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 behind TCP Splice is simple: data can be lost in split connection proxy schemes because the proxy acknowledges the receipt of data to the correspondent host 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.

We implement the splice by altering all the packets sent on one connection, including the acknowledgments, so the packets appear to belong to the second connection, and then sending the packets out over the second connection. 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 data acknowledgments, TCP end-to-end semantics are preserved between the two endpoints. Only after the mobile node receives data and transmits an ACK can the correspondent host possibly receive an ACK, since the proxy only relays the mobile node's ACK.

B.1 TCP Background

Before describing TCP Splice, some background on TCP is required (see [14] for more detailed information). Figure 7 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:

- `snd_nxt`: The sequence number of the next data byte to be sent.
- `snd_una`: The sequence number of the first unacknowledged data byte (equivalent to the sequence number of the greatest ACK received).

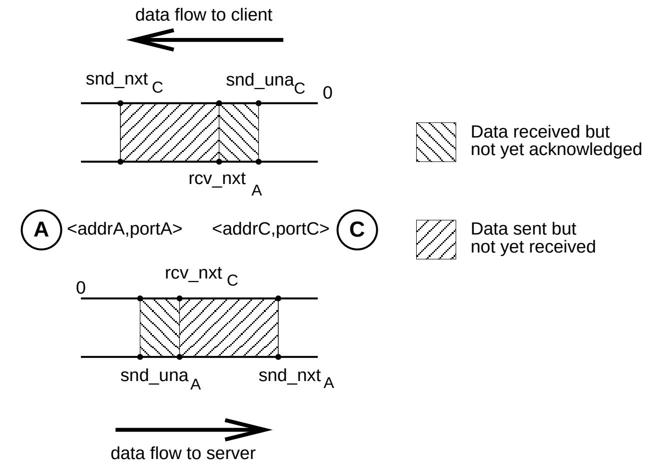

Fig. 7. A normal TCP connection between sockets **A** and **C** with state counters labeled.

- `rcv_nxt`: 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. Without loss of generality, we assume for this explanation that each sequence space begins numbering at 0. Data bytes with sequence numbers greater than `snd_nxt` have not yet been sent. Data bytes with sequence numbers less than `rcv_nxt` have been received by the TCP stack, but perhaps not yet read by the application. We say that data is acknowledged when the sender of the data receives an acknowledgment for it: `snd_una` will be less than the associated `rcv_nxt` counter (i.e., $\texttt{snd\_una}_C < \texttt{rcv\_nxt}_A$) whenever an ACK is in flight, delayed, or lost.

B.2 Mapping Sequence Spaces and Moving Packets

If the proxy has two connections it is to splice together, one to **A** and one to **B**, the next data byte the proxy expects to receive from **A** must be transmitted on the connection to **B** with the sequence number that **B** next expects to receive from the proxy. We call the sequence number of the next byte the proxy expects to receive from **A** the *splice initial receive sequence number* (`splice_irs`) and the sequence number **B** next expects to receive from the proxy the *splice initial send sequence number* (`splice_iss`). Together, the pair <`splice_irs`,`splice_iss`> define a mapping between the sequence number spaces of the spliced connections from **A** to **B**. For example, the datum with sequence number $\texttt{splice\_irs} + N$ on the **A**-to-proxy connection maps to sequence number $\texttt{splice\_iss} + N$ on the proxy-to-**B** connection. A second pair similarly defines the mapping of the spliced connections from **B** to **A**.

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.

B.2.a 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.

B.2.b Alter TCP header. The following steps are used to alter the TCP header:

- Change source and destination port numbers to match outgoing connection.
- Map sequence number from incoming sequence space to outgoing space:
  $seq\_num = (seq\_num - in\text{->}splice\_irs) + out\text{->}splice\_iss$
- Map ACK number from incoming sequence space to outgoing space:
  $ack\_num = (ack\_num - in\text{->}splice\_iss) + out\text{->}splice\_irs$
- 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. A special check must be made to ensure a segment being mapped does not contain data with sequence numbers less than splice base point `splice_irs`. If such a segment is received, the data up to `splice_irs` is simply chopped out of the segment.

B.2.c 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.

B.3 Choosing the Basepoints

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 6). The basepoints for the splice must be carefully chosen to prevent any overlaps or gaps forming in the sequence space of the logical session between the client and server. There are three cases to be concerned with: two covering the data flow from server to client, and one covering the data flow from client to server.

B.4 Splicing the Server to Client Flow

Figure 8 shows how basepoints are chosen for data flowing from the server to client in the case in which data is arriving faster than the client application reads it. Let us introduce a

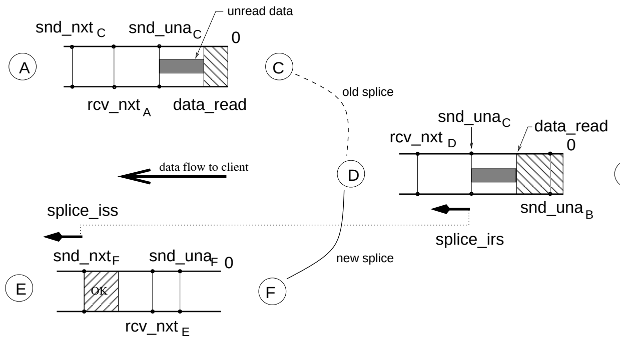

Fig. 8. The selection of basepoints for resplicing data flowing from server to client. The thin, shaded box represents unread data present in the socket of the old connection that has not yet been read by the application.

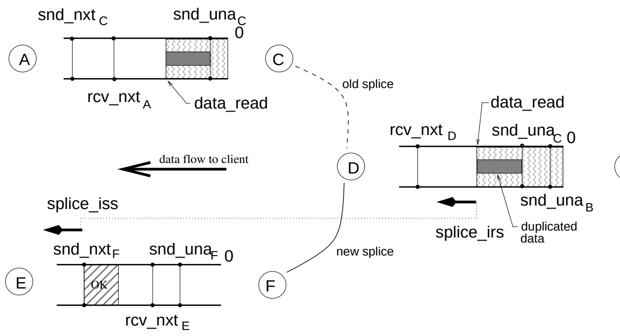

Fig. 9. The selection of basepoints for resplicing data flowing from server to client. The thin, shaded box represents data read by the application, but not yet ACK'd to the sender.

pointer data_read which keeps track of how much of the sequence space has actually been read by the application. The proxy is sure that the mobile node has received at least all the data up to $\texttt{snd\_una}_C$ (the largest ACK the proxy has received from the client) since it has seen a cumulative ACK from the mobile node for that data. If $\texttt{snd\_una}_C$ is greater than data_read, then at least $\texttt{snd\_una}_C - \texttt{data\_read}$ bytes of data are present in the socket buffers at **A**. Since an ACK has been sent for the data, the proxy must assume the the server may have received this ACK and so will never retransmit the data again. The MSOCKS library must therefore drain the $\texttt{data\_salvage} = \texttt{snd\_una}_C - \texttt{data\_read}$ bytes out of the old socket **A** before freeing it and must offer those data to the application when it next reads from its MSOCKS socket. As shown by the thick arrows in the figure, the proxy sets splice_irs so that the next byte of unacknowledged data from the server is the next byte of data sent to the client after the OK message. The data between $\texttt{recv\_nxt}_D$ and splice_irs can be retransmitted by the server if they were lost during the hand-off.

Figure 9 shows how basepoints are chosen for data flowing from the server to the client in the case in which data is read faster than it is ACK'd to the server ($\texttt{snd\_una}_C < \texttt{data\_read}$). To avoid duplicating data, we choose splice_irs to be at the data_read pointer. 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 OK message (which is read by the MSOCKS library). Any data the application has already already read that is retransmitted by the server will fall before the splice_irs

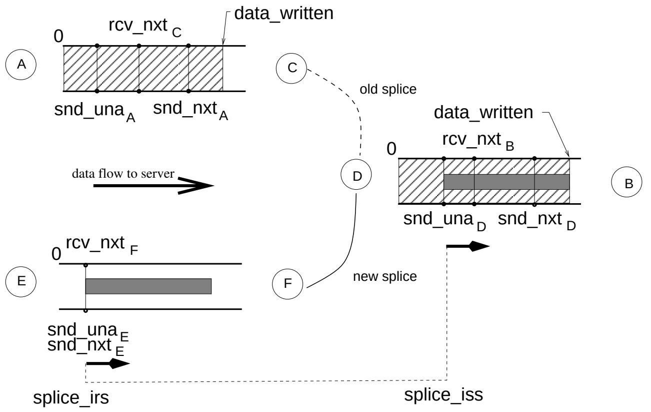

Fig. 10. The selection of basepoints while resplicing data flowing from client to server. The thin, shaded box represents written by the mobile node's application, but unacknowledged by the correspondent host.

point and will be chopped off and dropped by the proxy.

B.5 Splicing the Client to Server Flow

Figure 10 shows how basepoints are chosen for data flowing from the client to the server. The thin, shaded box in the figure depicts the data the application has written that the proxy has not yet seen acknowledged by the server. While this data is present in the old socket **A**, it is a "hole" in the sequence space of the new socket **E**, the socket that will be responsible for retransmitting the data once the new **E-F** connection is spliced in. Since some of the data in the hole may have been lost in flight, the mobile node must rewrite the data into the new connection between sockets **E-F**. The rewritten data, shown as the thin, shaded box in the figure, "covers the hole" left by the previous connection. In the case in which all the data in the hole has safely made it to the correspondent, the next ACK from the correspondent will acknowledge all the rewritten data. After the MSOCKS library has written data into socket **E** to cover the hole, new application data can be sent via **E** as normal. The proxy is able to calculate how many bytes of data the MSOCKS library must rewrite into the new connection as $\texttt{data\_write} = \texttt{data\_written} - \texttt{snd\_una}_D$.

B.6 Urgent Data

TCP defines a notion of urgent data [14] 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 data_read pointer. Since the MSOCKS proxy sees all out-of-band data as it flows through, it is able to correct the data_read value reported by the MSOCKS library to reflect all the data that has traversed the connection.

C. The MSOCKS Library

The MSOCKS library sits between the application and the kernel on the mobile node. It's 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 III-A. We will now describe how the MSOCKS library maintains or uses the `data_read`, `data_written`, `data_salvage`, and `data_write` counters.

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 `data_read` counter, 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 $\texttt{snd\_una}_C > \texttt{data\_read}$. The proxy calculates how many bytes of data are left in the old socket as $\texttt{data\_salvage} = \texttt{snd\_una}_C - \texttt{data\_read}$ and sends the MSOCKS library the `data_salvage` counter as part of the `OK` message for the `RECONNECT`. While an Msocket's `data_salvage` 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 `data_written` and `data_write` counters and the hole 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 hole, 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 `data_written` field in `RECONNECT` messages. The proxy calculates the size of the hole and returns the value as the `data_write` field in the `RECONNECT OK` message. In order to cover the hole, the MSOCKS library must then write into the new socket the last `data_write` 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.

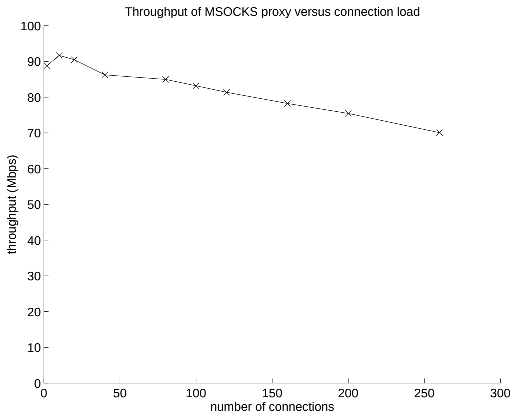

Fig. 11. TCP throughput supported by the MSOCKS proxy as a function of number of simultaneous connections.

IV. 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 client and a server program on the same machine as the MSOCKS proxy. By using the loopback interface to carry the traffic from client to proxy and from proxy to server, 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 11 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 pack-

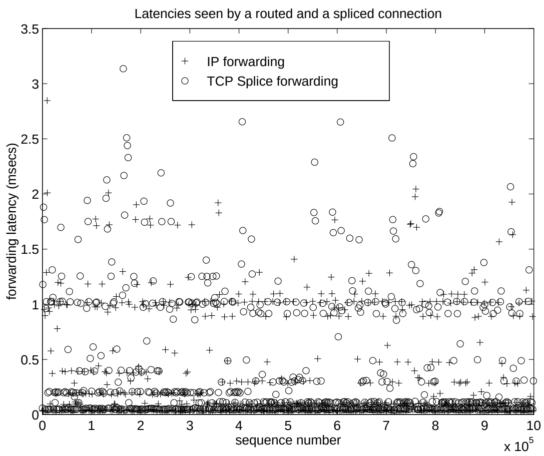

Fig. 12. Comparison of the latency of IP packet forwarding with the latency of TCP Splice forwarding.

TABLE I
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

ets traveling to or from mobile nodes. Figure 12 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. The X-axis in the figure is the sequence number of the packet. The data from a larger run is summarized in table I. 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 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.

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.

V. 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 [10] 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 [12][6] 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 [5][13], 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 [5], 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 [13] 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 [4] 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.

VI. CONCLUSIONS

Providing mobility support at the transport layer has both strategic and technical benefits. Strategicly, adding mobility support to the transport layer allows us to add mobility support to applications running on operating systems, like Windows 95, 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 correspondent host, something no other split connection protocol does. 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. Badrinath. Handoff and system support for indirect TCP/IP. In *Second USENIX Symposium on Mobile and Location-Independent Computing Proceedings*, Ann Arbor, Michigin, April 10-11 1995. Usenix.

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

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

[4] WinSock Group. Windows Sockets 2 Service Provider Interface. Web White Paper — Revision 2.2.2, August 1997. Available as ftp://ftp.microsoft.com/bussys/winsock2/wsspi22.doc.

[5] C. Huitema. Multi-homed TCP. IETF Working Draft - work in progress, May 1995. draft expired November 1995.

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

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

[8] M. Leech, David Koblas, et al. SOCKS protocol version 5. Internet Request For Comments RFC 1928, April 1996.

[9] David A. Maltz and Pravin Bhagwat. TCP splicing for firewall and HTTP cache performance. Technical report, 1998. To be published.

[10] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, Eric J. Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, St. Malo, France, October 1997.

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

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

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

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

[15] S. Thomson and T. Narten. IPv6 stateless address autoconfiguration. Internet Request For Comments RFC 1971, August 1996.

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