Varun Almaula, Taurin Tan-atichat, Thomas Weng
06/15/2007
CSE 222A

# Improving Internet Mobility: The TCP-to-SCTP Translation Layer

## I. Introduction

The design of the current Internet infrastructure is based on unicast point-to-point communications between fixed hosts. In this model, the sender must know the Internet Protocol (IP) address of the receiver and IP simply routes data packets based on the IP address. Specifically, a client host must know the IP of the server host in order to make a request and the server host assumes the IP of the client host will not change during the lifetime of the communication. IP addressing is assumed to satisfy the need of locating a single fixed host by a single IP address.

However, the advent of new technologies has introduced new communication paradigms that do not fit with current IP addressing techniques. Encoding network topology with an IP address severely hinders communication in these new models. The proliferation of several wireless technologies such as 802.11x, Bluetooth, WiMax, and cellular based 3G data communication often introduces the convergence of multiple networking interfaces on a single host. Many laptops today have multiple wireless interfaces and a wired interface for network communication, with each interface having a separate IP address for the same host. In addition, these types of hosts are mobile and thus, the IP assigned by any single network interface technology can change rapidly.

Generally, it is the client host that is highly mobile with multiple wireless interfaces to choose from. On the server side, it is increasingly common for large organizations to peer with multiple ISPs (multihoming) for connectivity redundancy (fault tolerance) and increased performance. Each ISP will assign a unique IP address for a single large organization (a server host). However, a server host may want to be identified by only one its ISP-assigned IP addresses or may want a separate static IP altogether.

With several changing IP addresses to identify two hosts during a communication session, it becomes increasingly difficult to support uninterrupted data transfer and preservation of communication state at the application level. A large amount of the protocol stack has now come to rely on IP addressing for communication; the network layer (IP), transport layer (TCP/UDP), and all the way up to the application layer use an IP address for uniquely identifying a host.

Introducing flexibility in the IP addressing scheme can be done at various layers. Changes to the IP layer are the most fundamental and address the problem directly, but require the most amount of effort by the networking community to realize. In-network hardware updates and heavy economic investment is required for this approach, making the deployment of new schemes at this layer very slow. Changing existing transport layers (TCP/UDP) is also a viable option. While easier to introduce change at these levels, it is still relatively slow to see widespread adoption of new versions; for example, TCP Vegas [18] has been shown to have superior performance to the existing TCP Reno implementation, yet a decade after its introduction, it has not been widely deployed.

Likewise, introducing a completely new transport layer protocol requires a large investment of application rewriting. Stream Control Transport Protocol, or SCTP, is a transport layer protocol built with IP mobility and multihoming support. While it remains similar to TCP in

that it provides reliable and ordered data communication, it has not had a high adoption rate as it requires rewriting applications and would break compatibility with existing TCP communication. It was introduced in 2000 and initially received much fanfare, including support by various operating systems, but has since seen little use in networking applications.

In this paper, we introduce an intelligent TCP-to-SCTP translation layer. Since TCP has become a dominant protocol for network communications and figures to be so the foreseeable future, allowing the programmer to continue using the familiar interface that TCP exports will spur the adoption of SCTP as an alternative transport layer protocol. This translation layer aims to provide the IP flexibility and performance benefits of SCTP while hiding any changes from the programmer. Changes at the IP level (e.g. IP address changes) do not need to be explicitly dealt with by an end user application; rather, SCTP and the translation layer can manage any IP changes optimally. Normal TCP calls are intercepted by the translation layer and, if possible, are converted into an SCTP equivalent call.

If SCTP is not an option for communication (i.e. the other host does not support SCTP), communication will fall back to a traditional TCP call. Essentially, the translation layer creates a mapping between a TCP socket requested at the application level and the several sockets SCTP creates across all available network interfaces; sockets requested at the application layer are treated as a single logical socket implemented with several sockets (a socket per available network interface) behind the scenes. In addition, an adaptive module interfaces with the translation layer; ideally, this module will record various data (including application type and packet classification) to create future default settings for the same application connections. The adaptive module can dynamically determine optimal settings for SCTP sockets based on saved usage patterns.

This remaining portion of this paper will look at existing approaches to IP mobility and flexibility (section II), the design of the translation layer and adaptive module (section III), the implementation details of the test system (section IV), results and analysis of these tests (section V), opportunities for future research (section VI), and project conclusions (section VII).

## II. Related Work

### Approaches to Internet Mobility

Several approaches towards Internet mobility already exist today including MobileIP, vertical handoff algorithms, transport layer protocols and DNS updating. All provide different levels of support for mobility, multihoming, and ease of deployment, but none truly provide support for all three.

### MobileIP

MobileIP was introduced as a network layer update to IP that provides increased support for mobile clients using heterogeneous networks [1, 2]. Mobile nodes can change network connectivity interfaces without changing their exposed IP address. Transport level (and possibly higher) connections can maintain state across connectivity changes without having to explicitly send routing information in the IP layer. MobileIP's solution introduces an abstraction to the end host addressing model; mobile clients expose both a permanent home address as well as a care-of address (the network the mobile client is currently connected to). A home agent is used to store permanent address of mobile nodes in its

network. A foreign agent is used to store care-of addresses of mobile nodes visiting its network. Communication destined for a mobile client will be directed to the home agent for the mobile client and then packets are tunneled from the home agent to the care-of agent. Communication originating from a mobile client will simply be sent by the care-of agent towards its final address and may employ reverse tunneling.

The MobileIP approach changes the IP layer while preserving the current functionality of end hosts in the Internet. The advantages of this approach mean that current applications and transport protocols do not need to be aware of IP address changes since it is handled in the IP layer. However, there are several disadvantages of this approach. While applications can free themselves of mobility concerns, they also lose control and feedback of network connectivity changes and routing changes. Moreover, this also requires a 'triangle' routing scheme using tunneling which may not be very efficient. MobileIP does not take advantage of multiple connected network interfaces either. Only a single network interface is chosen as the communication channel; additional connected interfaces are not utilized for better performance. Of course, the most daunting problem with mobile IP is that it requires network infrastructure changes and special MobileIP enabled routers to perform the triangle routing. The additional cost and IP layer changes prevent mobile IP from being comprehensively deployed. In essence, MobileIP is a solution geared for mobile clients. While it could be used for a multihomed server host, its triangle routing schemes are certainly not desirable.

Heterogeneous Handover

There are also several schemes proposed for connection management in heterogeneous networks [3, 4, 5, 6, 7]. Lee et. al provide a vertical handoff decision algorithm that aims to provide real-time, high-availability, and instantaneous high-bandwidth in heterogeneous network environments [7]. These seamless vertical handoffs manage connectivity among several network interfaces and optimally choose the correct network connection to choose for all communication. In this scheme, a complex set of formulas distributed across both mobile clients as well as access points are used to optimize load balancing, energy consumption among mobile nodes, route selection, and signal threshold connectivity. Zahran et. al also provide a seamless vertical handoff algorithm based on signal thresholds. Again, a complex set of formulas compromise the ALIVE-HO (lifetime based vertical handoff) algorithm, which takes into account signal strength, handoff latency, mobile client velocity and direction, and application QoS and delay tolerance [6]. This algorithm would run on top of a MobileIP based routing solution. Both solutions require special software running on the wireless APs and on the mobile clients. Furthermore, while both algorithms use a plethora of network level parameters in complex formulas, few specifics are provided on what interface is used to communicate between the high level application and these lower level measurements and handoff decisions.

Transport Layer Support

Yet another set of solutions focus on modifying the transport layer with additional mobility support [8, 9, 10]. Tsukamoto et. al create a Connection Manager for Heterogeneous Networks (CHMN) that sits as a wrapper of multiple TCP connections on behalf of an application [8]. Logically, it is similar to the SCTP implementation, with implementation detail differences, but arrives four years later than SCTP. SCTP already has better OS support and provides the same types of functionality provided by the CHMN, but in kernel space. Snoeren and Balakrishnan utilize DNS as a method of removing the direct mapping between an IP address and a host [9]. Since most hosts utilize a DNS hostname for communication, this can be utilized as host identifier instead of an IP address. DNS

hostnames are not cached (TTL is set to 0) and a new Migrate TCP option is used to handoff state from a previous TCP connection on a different IP address. While this approach provides end-to-end mobility for applications without disturbing the IP layer, it does little for hosts that do not communicate with each other by hostname. Furthermore, it is unclear how DNS would scale if all necessary hosts on the Internet would update their IP addresses frequently. Applications that do a 'one time' resolution of a hostname to an IP address will also need to be modified since IP an address may become stale during a communication session.

SCTP Initial Approach

SCTP was introduced in 2000 as a new transport layer protocol to provide better end-to-end performance in the face of increasingly more mobile clients and multihomed servers [11, 12, 13]. While SCTP provides reliable and ordered delivery of data between endpoints like TCP, it offers specific support for hosts with multiple network interfaces and multiple streams between endpoints. Instead of using a single socket connection between two endpoints, SCTP introduces the idea of an association between two endpoints. Associations utilize multiple network interfaces on each host and thus inherently support multihoming.

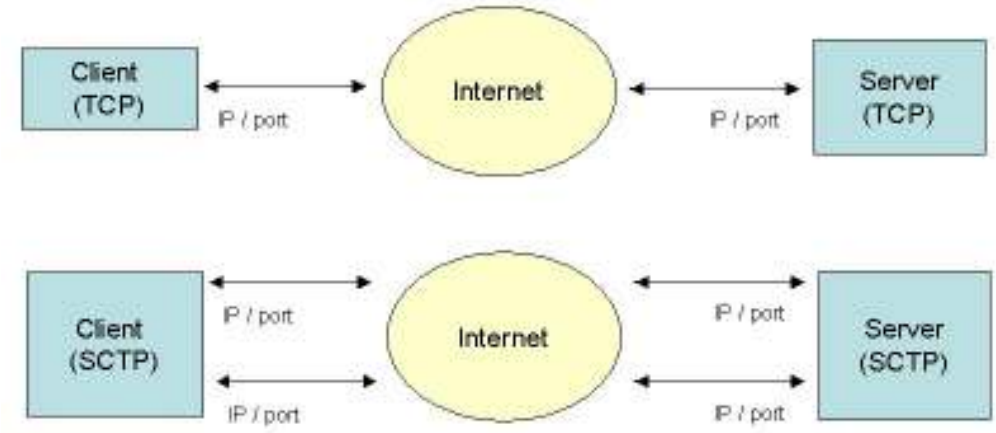


Figure 1: TCP (top) uses one connectivity interface, while SCTP (bottom) can utilize multiple interfaces simultaneously

By using a heartbeat, SCTP monitors the health of each connection, and will redirect traffic on different interface upon detecting a failure. Application sessions are preserved across link failures or interfaces switches. As mobile clients increasingly have several network interfaces to choose from, SCTP provides support for a primary address (preferred interface) for optimizing performance. SCTP also provides multiple streams per association for increased parallelism and performance. Head-of-line blocking problems associated with TCP are avoided.

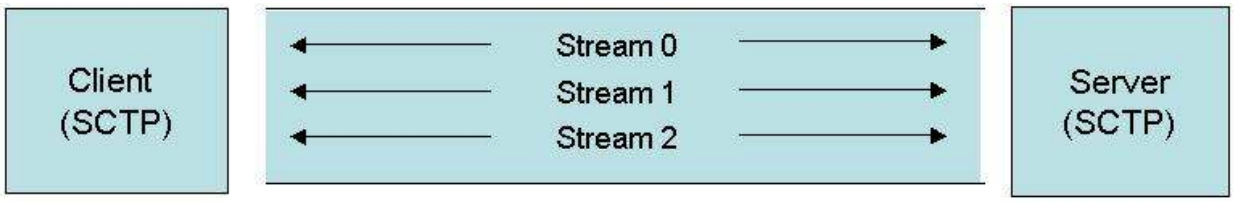


Figure 2: SCTP multi-streaming within a single association

SCTP provides message framing instead of the bytestream data delivery offered by TCP ensuring that the same sized message chunks are read at the sender and receiver. Initial experiments show that SCTP can recover from link failures, provide increased robustness, and higher throughput [12, 13]. Since its introduction it has found OS support in several Linux distributions as well as Microsoft Windows.

SCTP Variants

Since its introduction, several modifications to the SCTP base implementation have been introduced [14, 15, 16, 17]. Kelly, et. al. introduced a different handover mechanism for SCTP; instead of waiting for a failure to start a handover, path delays are monitored and this data is used to make handover decisions [14]. Shaojkian Fu and Mohammed Atiquzzaman combined MobileIP with SCTP and observed far better throughput during handovers [16]. One drawback of the base SCTP implementation is that it requires all known IP addresses for various connected interfaces to be known and exchanged at connection setup. To improve mobility, Joo et. al. introduce mSCTP (mobile SCTP) which introduces the ADDIP extension allowing for dynamic addition and removal of IP addresses during the connection lifetime [17].

These various SCTP modifications improve the performance characteristics and flexibility of SCTP. However, little has been done to improve application compatibility for SCTP. Our proposed solution aims to remove the burden on the programmer to update existing TCP based applications to take advantage of the benefits of SCTP.

## III. System Design

Design Goals

With the realization that SCTP provides an effective solution to the mobile connection issue, we sought to leverage it to better allow for applications to take advantage of it and spur its deployment. Therefore, there are two major design goals for our research. First, we wanted to provide a way for user applications using TCP sockets to utilize SCTP without any changes. Second, we wanted to allow SCTP-aware applications to easily take advantage of some of the features that SCTP supports as well as intelligently deciding which connections to send traffic through when given multiple network connections.

It is difficult to provide a solution that can meet both goals simultaneously; the interface of our SCTP-aware module must transparently look like a TCP socket and yet perform TCP to SCTP translations while allowing for some communication between SCTP-aware programs and the SCTP socket options. Another one of our goals was to take advantage of SCTP as much as possible through our translation layer, with the application not even being aware of SCPT.

We are able to provide this functionality by making SCTP behave in a default mode of operation when no SCTP-specific calls are made. When SCTP-aware applications call SCTP-specific calls, appropriate actions are made to best accommodate the request. In addition, we try to optimize the SCTP link as much as possible on behalf of the application.

Design Architecture

Two different modules were built to meet our goals: a translation layer and an adaptive module. The translation layer is responsible for intercepting all TCP socket function calls and converting them into the appropriate SCTP and/or TCP function calls. The adaptive

module is responsible for dynamically determining the optimum connection to use based on current usage patterns and application hints.

Design Details

*Translation Layer*

Our translation layer module intercepts all TCP socket function calls and makes the appropriate SCTP and/or TCP function calls. For example, when an application first calls *socket()* to create a new socket, the translation layer module actually creates one SCTP socket and one TCP socket. A logical identifier that represents both of these underlying sockets is returned to the caller. The caller can then use this logical identifier as a TCP socket while the translation layer module performs all the appropriate conversion.

The translation layer keeps track of this logical identifier, and maps it to the appropriate TCP and SCTP sockets that were created for it. This means then that for every socket that the application attempts to make, two are created for it at the underlying kernel. Our translation layer is then able to allow for both TCP and SCTP calls through this.

On a *connect()* call the translation layer module will try to connect with an SCTP socket first. If that succeeds, it is determined that the server supports SCTP and that the remaining session should use SCTP. However if it fails, the server does not support SCTP so the translation layer will revert back to using the TCP interface instead. This allows for applications to still connect to TCP servers if no SCTP support was detected at the desired server. It is important to note that there is a slight performance impact, namely the time it takes for the translation layer to realize that it cannot connect using SCTP and to use TCP instead.

The server side is more complex since the server should be able to accept both SCTP and TCP requests. We are able to achieve this by creating both SCTP and TCP sockets, binding both of them to their respective ports, listening with both of them, and finally using a specialized *accept()*. In our *accept()* we know that we are expecting either a SCTP request or a TCP request but not both from an individual client. So we place both sockets into a reader set and call *select()* to figure out which request to actually *accept()*.

In addition to the network function calls, we must intercept all of the associated function calls that might be used for network communications. Examples include *read()* and *write()*. These functions need to check whether the file descriptor passed in through the caller is a logical identifier that corresponds to one of our SCTP/TCP pairs it is keeping track of or if it is a normal file descriptor.

For example, a read function will pass in the file descriptor. Our translation layer intercepts the read function and determines whether the descriptor refers to the logical socket number. If it does, it determines whether the current communication with that logical socket is through SCTP or TCP, and then makes the appropriate read call through the correct socket number. If the descriptor does not correspond to any network socket, then a read call is made with the passed descriptor identifier.

Several system calls and functions are intercepted and are listed in Table 1. Not all the system calls and functions were fully supported. The getsockopt/setsockopt calls for example were difficult to fully map, and another difficulty was in fully supporting close(). This is because it is possible to have a half-closed state with TCP, and some applications

depend on that. SCTP has no such half-closed state. Better support for these functions will be one of the primary goals for any future research.

| Function | Translation Layer Implementation |
|---|---|
| socket() | Creates an SCTP and TCP socket |
| bind() | Binds both sockets to the appropriate port number |
| listen() | Both sockets are placed in listening mode |
| accept() | Uses select so that both TCP and SCTP can be supported |
| connect() | SCTP connection is tried first, followed by TCP if it fails |
| send() | Checks file descriptor and makes appropriate call to either TCP socket, SCTP socket, or passed down descriptor |
| recv() | Same as send() |
| write() | Same as send() |
| read() | Same as send() |
| close() | Closes both SCTP and TCP sockets |
| shutdown() | Shuts down both SCTP and TCP sockets |
| setsockopt() | Sets TCP socket, tries to map options to SCTP if possible |
| getsockopt() | Returns TCP options, or SCTP equivalent if possible |
| getsockname() | Returns TCP sock name, or primary address if SCPT |
| getpeername() | Returns TCP information, or primary peer address if SCTP |

Table 1: System calls and functions that are intercepted by translation layer

*Adaptive Module*

The adaptive module dynamically determines the appropriate network connection to use based upon current usage patterns and hints that the user application has passed to it. There are a few cases to consider. The simplest case occurs when we are not using SCTP sockets but have defaulted to TCP. In this case the adaptive module does nothing. The less trivial case occurs when SCTP sockets are being used but no information has been passed into the adaptive module. In this case the adaptive module uses the default values of the SCTP protocol stack.

A more interesting case is when the application specifies preferences. For example, an application that does not require a lot of bandwidth or low latency, but will be used over a long period of time may request the adaptive module to set the primary interface for that application to be a cellular network connection. This would allow a mobile user to leave a hotspot without any change of connections. A careful reader might question the purpose of selecting a more appropriate network connection for a user application when SCTP would be able to dynamically switch connections upon failure of one connection. There is a cost to switch connections in SCTP. Several timeouts must occur before SCTP will acknowledge that a connection is officially dead. This is to prevent premature switches due to normal network congestion.

Our current implementation of the adaptive module consists of determine packet flows across a particular connection and determining which of the links to use. This requires a-priori information; however future research could improve on this. The adaptive module records the particular settings for each connection, and upon a future connection to the same peer, will automatically provide recorded connection information for the translation layer to use. This allows future connections with that peer to be better optimized.

The ideal case would be to have the adaptive module automatically determine the best

network connection to use based upon observed behaviors. There are several machine learning techniques that could be used to classify types of connections. It is beyond the scope of this paper to discuss such techniques and we leave this as future work.

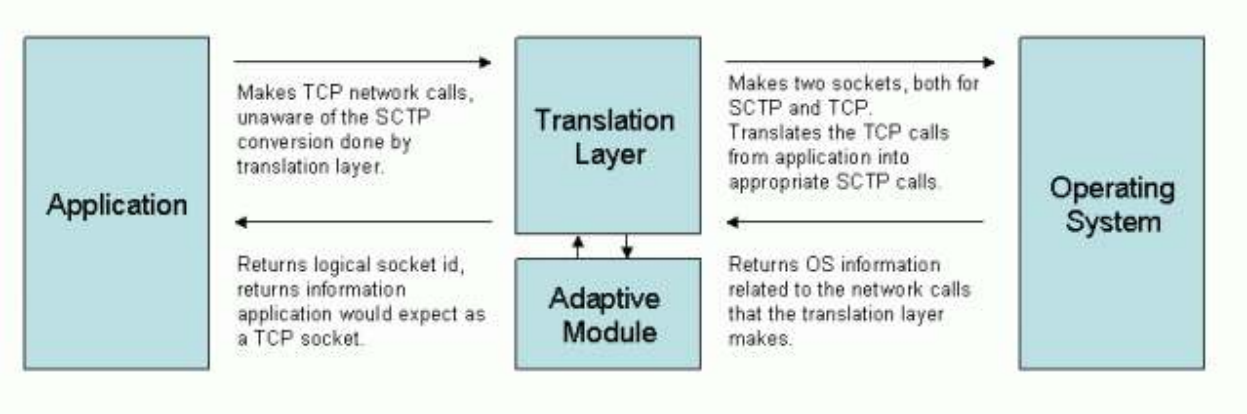


Figure 3: How our SCTP translation layer interacts with application and with the underlying operating system.

## IV. System Implementation

Implementation Details

We developed our system on Fedora Core 7 with the 2.6 Linux kernel version. We ran the lksctp SCTP implementation as a kernel module and ran our modules in userspace. We used an AMD Athlon XP 2400+ for the server and a Dell Latitude D810 with 2.13 GHz Pentium M for the client. We connected the server (one Ethernet interface) and the client (two Ethernet interfaces) together over a Fast Ethernet with a Netgear FS605 unmanaged switch.

Implementation Challenges

The first main challenge was the lack of NAT support. Since NATs typically only support UDP and TCP, our SCTP packets were being dropped by a Linksys router. This forced us to keep servers and clients on the same subnet, going through only a layer 2 switch. The second main challenge was software incompatibilities in Linux. Commonly used utilities such as *netstat*, firewalls, and others did not work, making debugging more difficult.

Another challenge was Linux's poor support for wireless network cards. We were unfortunate to have a laptop installed with a Broadcom chipset whose drivers were not compatible. This forced us to us to use wired Ethernet.The last major challenge was Linux's inability to quickly detect network connection liveliness and properly bring up and down the interface. We needed to run *netplugd* in order for SCTP to recover from a connection loss.

## V. Results and Analysis

Evaluation Methodology

Since one of the main goals of this project was to seamlessly convert applications from TCP sockets to SCTP sockets without a significant impact on performance, we conducted throughput and latency tests of various applications for both TCP sockets and SCTP sockets. Web and file servers are both common data services provided over the Internet, so we

implemented a web server and tested its performance over both native TCP and through our SCTP translation layer.

We tested the scenario that a client was attempting to download a large file from a server on the Internet. The file was a 100 MB file on a 100 Mb/s LAN. A client would download the file either using TCP or SCTP sockets. Download times were tracked to determine the amount of additional overhead SCTP had over TCP. Our results showed that our SCTP translation layer required essentially no more overhead than TCP.

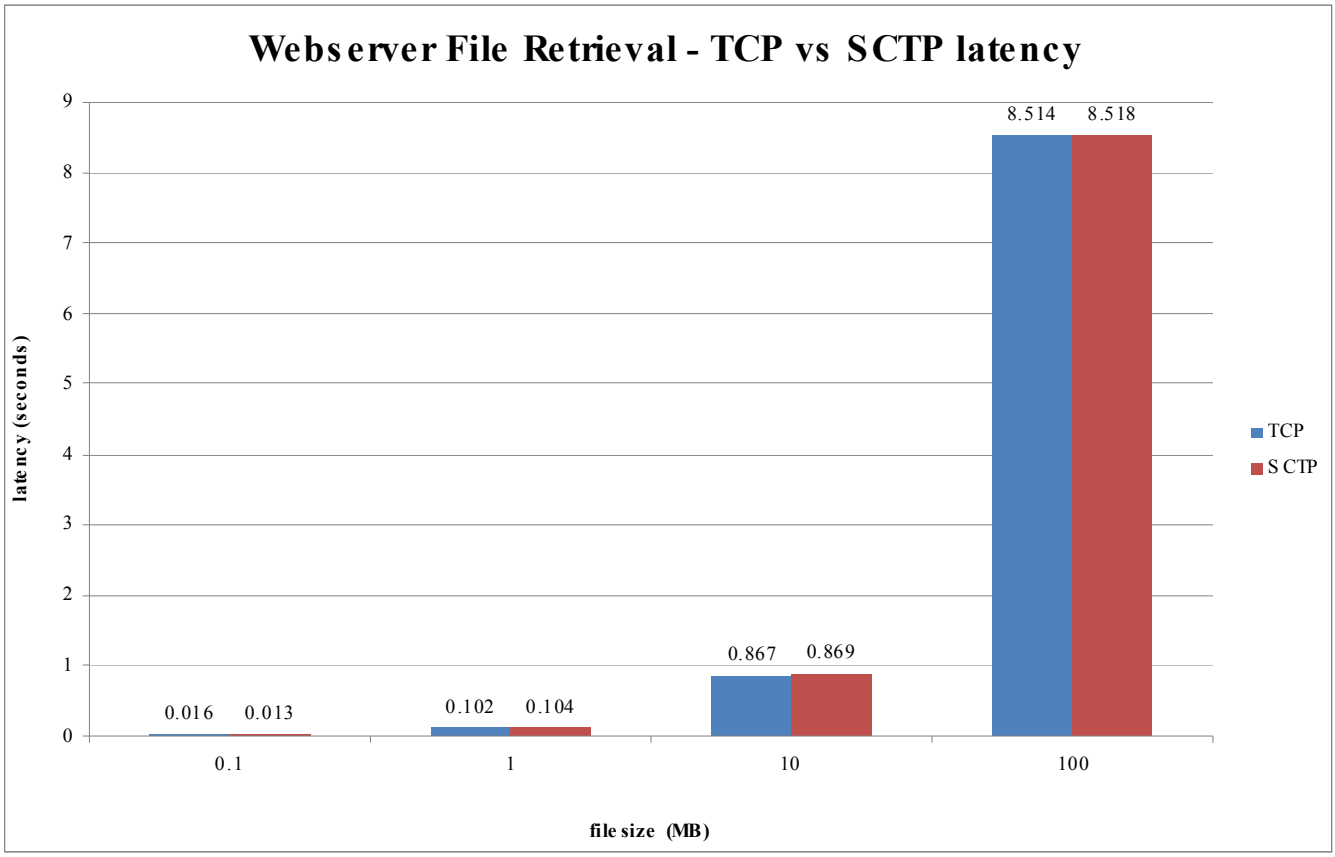


Figure 4: Results comparing the latency of file retrieval using TCP vs. SCTP

The throughput of both TCP and SCTP in our web server test is around 11.7 MB/s across a 100 MB file. Considering this was run on a 100 mb/s switch that can provide a theoretical 12.5 MB/s throughput, our translation layer did not impact the theoretical performance of the application.

Previous studies [20] have demonstrated that web traffic via SCTP does not have any significant impact on performance versus TCP. Our own tests have shown that our translation layer does not impact the performance either, and thus is suitable for real applications that depend on high throughput and low latency.

As our tests were involved on a LAN, we have not shown what happens in a real-world environment where there is congestion and background traffic. However, our focus was on the impact that the layer would have, and not on SCTP itself. Therefore, the tests we performed demonstrated that our translation layer is indeed light weight, and does not exact a significant performance penalty.

In addition to comparison tests between TCP and SCTP, we also conducted tests on the robustness of our SCTP translation layer. Specifically, we sought to measure the time it took for the protocol to detect a network failure and for it to rollover on to a secondary link. During a data transfer, one of the client's network connections goes down, possibly due to traveling out of range of a wireless access point. Upon detection of the failure, SCTP will redirect the remaining part of the transfer over another connection seamlessly.

We simulated this by serving the 100 MB file on a LAN. While a client was downloading the file, the primary CAT5 network cable for the client would be physically disconnected. This would ensure that the connection was lost and that the communication channel was closed.

We measured the time it took for the SCTP implementation to realize that the network connection was down, and how long it took for it to recover (fall back to the secondary link). The amount of time to detect that a network connection went down is highly dependent on certain parameters passed into the SCTP socket. For example, the programmer can set the length of timeouts, the number of timeouts, the maximum length of a timeout (normally doubles after every timeout), and the number of timeouts before concluding a connection failed.

Our non-tweaked implementation took about one minute on average for the SCTP to rollover to the secondary link. While unacceptable for real time communications, our results did prove that our SCTP translation layer did seamlessly provide for IP rollover. Unfortunately, the time it takes to detect network loss and change to the secondary network interface is not a product of our translation layer, but of the particular operating system implementation of SCTP and how it handles network devices. In order to improve this time to a reasonable amount for real time communications, we would seek to modify the underlying kernel.

## VI. Future Work

There are many more extensions and enhancements we would like to make to our translation layer and adaptive module. Some of the system calls are not fully supported in our translation layer, and work would be done here to better implement them. We would also make the translation layer entirely transparent since it currently is running in userspace and there are a few different function naming differences. This would be a very simple fix by using the LD_PRELOAD option in Linux or could be implemented by rewriting part of the networking stack to automatically expand TCP calls.

We would also look into the benefits of implementing the translation layer as part of the operating system. There are many pros and cons to this, and research would be done to determine if this is the appropriate choice. The most significant negative is that the operating system would need to be recompiled.

We would also expand on the adaptive module. We could build a classifier and implement machine learning techniques to dynamically find the best connections to use when multiple are present. We would also like to explore the SCTP parameters to find ways to more easily control which connections should be used for both outgoing and incoming traffic.

If we determine that the translation layer and adaptive module is best implemented in the kernel, we would also look into modifying the operating system to better support the SCTP translation layer. Such modifications would include better allowing the layer to determine network devices and for quicker notification of a connectivity loss. By doing this, we would seek to design a better optimized layer that would allow for true multihoming support for

real time communication applications and still keep the application completely unaware of the underlying mechanics.

This would allow for truly decoupled and separate mobile communications. The ubiquity of TCP [21] means that applications will likely be continued to written with it. Through the use of the translation layer, applications would be able to take advantage of multihoming and mobile communications while still retaining familiar network calls. In addition, the existing code base of network applications could take advantage of this as well. This we feel is the most significant result from this work.

At the translation layer and kernel intersection, an improved interface would allow for better handoff algorithms while still minimizing impact to user applications. The kernel could determine that a particular network interface is superior, communicate that to the translation layer, and have the translation layer initiate the appropriate SCTP request. Future research done here could create a platform that can truly take fulfill the promise of Internet mobility.

## VII. Conclusions

In this research we have explored mobility solutions currently available to provide graceful handover among multiple network connections. We determined that MobileIP required specialized hardware, somewhat complex algorithms, and was focused on client mobility rather than universal mobility and multihoming. We found various vertical handoff algorithms that had extensive theory and mathematical backing, but required hardware and software support and thus had little real world implementation results. We found that using DNS for mobility removes IP addresses as the primary identifier of a host, but potentially strains DNS servers and would not work well for mobile client communication. We considered implementing our own logical socket (VTT sockets) which aggregate TCP sockets, but decided that SCTP is a better protocol to use since it was designed with multihoming and mobility in mind.

We designed and implemented a translation layer that automatically and seamlessly converts TCP calls to SCTP calls. We tested this and determined that it works for IP address rollover and has similar performance characteristics as TCP. We also realize that the support for SCTP is limited since it is a fairly new protocol and various hiccups exist including troubles with NATs, firewalls, and simple utilities like netstat. Our results show that SCTP is a robust, lightweight, transport protocol that can enable multihoming and mobility and that our translation layer can help spur its usage among network applications.

## VII. References

1. Perkins, C.E., and Calhoun, P.R. Mobile IP challenge/response extensions. Internet Draft, IETF, Feb. 2000.

2. Perkins, C.E. IP mobility support. RFC 2002, IETF, Oct. 1996.

3. Brik, Vladimir, Mishra, Arunesh, and Banerjee, Suman. Eliminating handoff latencies in 802.11 WLANs using Multiple Radios: Applications, Experience, and Evaluation. Internet Measurement Conference, 2005.

4. Pesola, Jusso, and Ponkanen, Sami. Location-aided Handover in Heterogenous Wireless Networks. Jusso Pesola, Sami Ponkanen. Wireless Personal Communications, 2004.

5.. Xhafa, Ariton E., and Tonguz, Ozan K. Reducing Handover Time in Heterogenous Wireless Networks. IEEE, 2003.

6. Zahran, Ahmed H., Liang, Ben, and Saleh, Aladdin. Signal threshold adaptation for vertical handoff in heterogeneous wireless networks. Springer Science, 2006.

7. Lee, SuKyoung, et. al. Vertical Handoff Decision Algorithms for Providing Optimized Performance in Heterogeneous Wireless Networks. Submitted to IEEE Trans. on Vehicular Technology.

8. Tsukamoto, Kazuya, Hori, Yoshiaki, and Oje, Yuji. Mobility Management of Transport Protocol Supporting Multiple Connections. MobiWac, 2004.

9. Snoeren, Alex C., and Balakrishnan, Hari. An End-to-End Approach to Host Mobility. MobiCom, 2000.

10. Snoeren, Alex C., Balakrishnan, Hari, and Kaashoek, M. Frans. Reconsidering Internet Mobility. In Proc. 8th on Hot Topics in Operating System (HotOS-VIII).

11. Stewart, Randall, and Metz, Chris. SCTP – New Transport Protocol for TCP/IP. IEEE Internet Computing, November 2001.

12. Ravier, Thomas, et. al. Experimental studies of SCTP multi-homing. Teltec DCU.

13. Shi, Jinyang, et. al. Experimental Performance Studies of SCTP in Wireless Area Networks.

14. Kelly, Andrew, et. al. Delay Centric Handover in SCTP over WLAN.

15. Ma, Li, et. al. A New Method to Support UMTS/WLAN Vertical Handover Using SCTP. IEEE VTCfall, October, 2003.

16. Fu, Shaojian, and Atiquzzaman, Mohammed. Improving End-to-End Throughput of MobileIP using SCTP. IEEE, 2003.

17. Koh, Seok Joom et. al. mSCTP for Soft Handover in Transport Layer. IEEE, 2004.

18. L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. IEEE Journal of Selected Areas in Communication, Vol. 13, No. 8, pp. 1465-1480, October 1995.

19. A. Loukissas, A. Tapadia, and J. Wu. P2P DNS. UCSD CSE 222A Mini-conference, June 2007.

20. Rajamani, Kumar, and Gupta. SCTP versus TCP: Comparing the Performance of Transport Protocols for Web Traffic. University of Wisconsin-Madison Technical Report, May 2002.

21. Kreger, Heather. Fullfilling the Web services promise. Communications of the ACM, 2003.