Wireless Networks 11, 99–114, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

A Transport Layer Approach for Achieving Aggregate Bandwidths on Multi-Homed Mobile Hosts

HUNG-YUN HSIEH
Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, R.O.C.

RAGHUPATHY SIVAKUMAR *
School of Electrical and Computer Engineering, Georgia Institute of Technology, 777 Atlantic Drive NW, Atlanta, GA 30332, USA

Abstract. Due to the availability of a wide variety of wireless access technologies, a mobile host can potentially have subscriptions and access to more than one wireless network at a given time. In this paper, we consider such a multi-homed mobile host, and address the problem of achieving bandwidth aggregation by striping data across the multiple interfaces of the mobile host. We show that both link layer striping approaches and application layer techniques that stripe data across multiple TCP sockets, do not achieve optimal bandwidth aggregation due to a variety of factors specific to wireless networks. We propose an end-to-end transport layer approach called *pTCP* that effectively performs bandwidth aggregation on multi-homed mobile hosts. We show through simulations that pTCP achieves the desired goals under a variety of network conditions.

Keywords: multi-homed mobile host, striping, inverse multiplexing, parallel sockets, head-of-line blocking

1. Introduction

The explosive growth in the number of mobile Internet users has been accompanied by the equally staggering increase in the number of wireless access technologies. A mobile user today can choose from a myriad of options ranging from networks such as Globalstar or Iridium for satellite access, CDPD, GPRS, EDGE, or 3G for wide area access, Ricochet for metropolitan area access, and IEEE 802.11 or HiperLAN based networks for local area access. An interesting and obvious challenge that arises when such independent options exist is: *How can these technologies co-exist in providing the best wireless access possible to mobile users?*

As a first step toward addressing this challenge, approaches have been proposed for performing *vertical handoffs* from one network to another as the mobile user migrates across coverage areas [14,22]. When the coverage areas of two networks overlap (say wide area and local area), the user is provided access through the higher data rate connection. In this paper, we consider an identical scenario of a mobile host having multiple wireless interfaces. However, instead of the mobile host being provided access only through one of its interfaces, we consider the problem of providing simultaneous access through all of its active interfaces. For example, if a user is subscribed to both a wide-area wireless network with a data rate of 2 Mbps indoors (e.g., 3G systems), and a local-area wireless network with an effective data rate of 5 Mbps (e.g., IEEE 802.11 WLANs), assuming that the user is within range of the access points of both networks we address the following question: *Can an application that requires reliable and sequenced delivery of data be provided a data rate of 7 Mbps through the use of both interfaces?* Since TCP is by far the most dominant protocol used for reliable and sequenced data delivery, we use TCP in all of our discussions henceforth and use the generic term *sockets* to refer to TCP sockets.

A simple approach to aggregate bandwidths would be to use multiple sockets, one each for every active interface, and use application layer striping and resequencing. In fact, similar schemes have been proposed to improve application layer throughput, albeit in a different context where such striping is done on multiple sockets that share a long-fat path, and the goal is to fill the bandwidth-delay product of the path (which otherwise will be impossible without both ends supporting the window scaling option) [2,19]. However, in the context of multi-homed mobile hosts, it turns out that such an approach not only fails to achieve the aggregate data rate, but in the specific case of the connections having vastly differing bandwidth-delay products can result in the effective aggregate data rate being lower than the data rate of the slowest connection! While we identify the reasons involved and discuss them in detail in section 2, briefly, the performance degradation occurs due to head-of-line blocking at the resequencing buffer of the receiving application. Pending packet arrivals at the receiving end of the slower connection stall the TCP sender of the faster connection, which eventually enters persist mode because of zero window advertisement by its receiving end [26].

Link layer striping schemes that assume stable link characteristics have been proposed earlier for bandwidth aggregation [6,20,25]. However, such schemes are unfortunately inapplicable to wireless links. Even with an adaptive mechanism that monitors the quality of wireless links and stripes accordingly [21], optimal bandwidth aggregation still cannot

* Corresponding author.
E-mail: siva@ece.gatech.edu

be achieved in the context of multi-homed mobile hosts where different interfaces potentially belong to different access networks. In [1], the authors propose a "channel" striping algorithm where a channel is defined as a logical FIFO path at any layer of the protocol stack including the transport layer. We discuss in section 7 why this approach exhibits drawbacks and limitations that specifically pertain to link layer striping schemes.

In this paper, our goal is to study the problems involved in achieving bandwidth aggregation when an application on a mobile host uses multiple interfaces simultaneously, and propose a transport layer approach that effectively addresses the problems. To this end, we propose a purely end-to-end transport layer approach called *pTCP* (parallel TCP), and present it as a wrapper around a slightly modified version of TCP that we refer to as *TCP-v* (TCP-virtual). For each pTCP socket opened by an application, it creates and maintains one TCP-v connection for every interface over which the connection is to be striped on. pTCP manages the send buffer across all the TCP-v connections, decouples loss recovery from congestion control, performs intelligent striping of data across the TCP-v connections, does data reallocation to handle variances in the bandwidth-delay product of the individual connections, redundantly stripes data during catastrophic periods (such as blackouts or resets), and has a well-defined interface with TCP-v that allows different congestion control schemes to be used by the different TCP-v connections. We show through *ns-2* [24] based simulations that pTCP outperforms both simple and sophisticated schemes employed at the application layer. Although we present pTCP as a transport layer solution for simplicity, it can be implemented as a session layer solution if sufficient support is provided by the transport layer. The contributions of this work can thus be summarized as follows:

(1) We consider mobile hosts that have multiple interfaces corresponding to independent wireless access networks, and investigate why using multiple sockets with application layer support does not result in the desired bandwidth aggregation.

(2) We propose an end-to-end transport layer approach called pTCP that effectively provides applications with the aggregate bandwidths available through the multiple interfaces at a mobile host.

The rest of the paper is organized as follows. Section 2 discusses why using multiple sockets does not result in aggregate bandwidths. Section 3 presents the assumptions and key tenets of the pTCP design. Section 4 describes the pTCP protocol in detail along with the pTCP state diagram, handshakes, and packet header formats. Section 5 provides simulation results of the pTCP protocol. Section 6 revisits several assumptions, and considers the impact of relaxing them; it also discusses several deployment issues for pTCP. Finally, section 7 discusses related work, and section 8 concludes the paper.

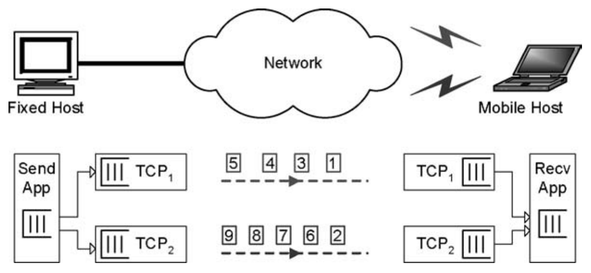


Figure 1. Application layer striping.

2. Motivation

In this section, we consider an approach wherein the application opens multiple TCP sockets, one each for every interface, and performs striping of data across the different sockets to achieve bandwidth aggregation. Figure 1 illustrates such an application layer striping technique. We consider both an *unaware* application that has no knowledge of the underlying connection data rates, and a *smart* application that has some knowledge of the available data rates (which will consequently enable it to stripe more intelligently). In the former case, when a socket blocks on a write, the sending application moves to the next socket and so on. In the latter case, the sending application stripes data based on a ratio determined by estimation of the available rates on the different connections (or pipes[1]). Since the goal is to perform reliable in-sequence data delivery, the receiving application does the resequencing using a finite resequencing buffer. For simplicity, we assume application level sequence numbers to facilitate the resequencing process, and restrict our discussions to packet streams as opposed to byte streams. The receiving application continues to read packets from each socket as long as its resequencing buffer has available space. When the application buffer is full, it stops reading from sockets that have already delivered packets with sequence numbers larger than the next expected application level sequence number. It then enters a *peek mode*[2] where it peeks into the next available packet in each of the other sockets, and reads a packet only when it is the next in-sequence packet. Note that if the application buffer size is zero, the application will always read in-sequence packets from the sockets. On the other hand, increasing the size of the application buffer has the effect of reducing the chances of the faster pipe being stalled by the slower one. We elaborate on this phenomenon later in this section when we discuss the impact of data rate differential among the multiple pipes.

We now proceed to identify the key constraints of such an application layer striping approach.

[1] We refer to the individual connections as pipes hereafter differentiate them from the aggregate connection.

[2] The *recv*() socket call and its variants support a peek flag that when set allows the receive operation to retrieve data from the beginning of the receive buffer without removing that data from the buffer [26].

Data rate differential

When the data rates of the pipes used by the unaware application are different, the aggregate bandwidth achieved by the simple approach remains a tight function of the data rate of the slowest pipe. This can intuitively be explained as follows. Consider two pipes with data rates of 10 Mbps and 2 Mbps, respectively. Since the application stripes data by keeping the send buffer of each pipe filled, a send-buffer's worth of application data will be injected to the first (10 Mbps) pipe (let this block of data be B_1). Blocked by the first pipe, the application will then proceed to inject data into the second (2 Mbps) pipe (let this block of data be B_2). Because the first pipe will drain data faster, the application will, after filling the second pipe, inject more data into the first pipe (let this block of data be B_3). Assume that because of the data rate difference, the first pipe delivers B_3 before B_2 is drained out by the second pipe.

Since the additional data (block B_3) will be out-of-order, it will be queued up in the resequencing buffer of the receiving application pending the arrival of the entire block of data B_2 through the second pipe. Because the first pipe will continue to transfer data at a faster rate, this will eventually result in the application's resequencing buffer overflowing. The receiving application will thereupon stop reading data from the first pipe, which in turn will cause the first pipe's TCP receiver buffer to fill up. The TCP receiver will then advertise a window size of zero, completely stalling the first pipe. Once the in-sequence data (block B_2), sent originally through the second pipe, reaches the receiver and hence releases space in the resequencing buffer, the first pipe will become active again. Note that such head-of-line blocking is indeed an artifact of the unaware striping mechanism used by the application. One way of reducing the above coupling between the faster and slower pipes is to increase the resequencing buffer size at the application layer. The larger the buffer size, the more the time for which the faster pipe can remain active without being inhibited through flow control. Specifically, if the two pipes have bandwidths of R_1 and R_2 ($R_1 < R_2$) respectively, and equal delays, the application buffer required in steady state to effectively aggregate bandwidths is $(R_2/R_1) \cdot W$, where W is the default socket buffer size. Even assuming that the above buffer requirements can be accommodated, such buffering still cannot handle stalls that occur due to losses in the slower pipe. Also, even if the application does smart striping, such a problem will exist as long as the striping ratio does not exactly match the data rate ratio of the different pipes. We elaborate on this issue in the next constraint.

Furthermore, in TCP, the performance degradation for the simple approach is severe because of another phenomenon: persist timers. When the sender of the faster pipe receives a window advertisement of zero, it enters persist mode. If the single *window update* from the receiver happens to be lost (either due to congestion or random wireless losses), the sender probes the receiver only after the persist timer expires next (5 seconds). The persist timer value doubles after every unsuccessful probe and is capped only at 60 seconds [26]. While this effectively brings down the progress of the faster pipe to a crawl, the impact is more severe as the slower pipe can potentially enter persist mode because of the persist-timer induced stalling of the faster pipe! Hence, in TCP the effect of the data rate differential among the different pipes can potentially be catastrophic to the application, resulting in the aggregate throughput being lower than the data rate of the slowest pipe.

Data rate fluctuations

Although the problem due to data rate differential can be overcome by employing an intelligent striping scheme, performing such intelligent striping is inherently a difficult problem because of two reasons: (i) The pipes are end-to-end pipes that traverse multiple hops between the sender and the receiver, and the available bandwidth is likely to fluctuate dynamically; and (ii) Given the dynamic nature of wireless link characteristics, it is very likely that the pipes will exhibit highly varying data rates. When the application stripes based on the estimated data rates of the pipes, and the data rates change, the very purpose of intelligent striping is defeated resulting in degraded performance. Note that the dynamic characteristics of the wireless link, and the consequent difficulty in performing accurate rate estimation are only part of the reason for the degraded performance. The coupling of congestion control and loss recovery (for the aggregate connection) that exists because of the individual TCP pipes functioning independent of each other is also a contributing factor. For example, packets assigned to a TCP pipe by an application cannot be "withdrawn" from that pipe, notwithstanding any bandwidth reduction the pipe may experience. Thus, if bandwidth reduction occurs, packets assigned to the pipe that have not yet been transmitted due to lack of space in the reduced congestion window cannot be reassigned to another active pipe.

Blackouts

Blackouts are extreme cases of rate fluctuations where the available data rate falls to zero and remains at zero for an extended period of time. Causes for such phenomena include temporary loss in connectivity (e.g., when the user is passing through a tunnel), fading, interference from a moving source, etc. Observations on the occurrence of such phenomena have been made in related work [18]. In the multiple sockets approach, such blackouts on one or a subset of the pipes will stall the entire aggregate connection because of buffer overflow at the receiving application. This is obviously an undesirable phenomenon. While the only solution to this problem is to have some feedback mechanism at the application layer (for the application to realize that a particular pipe has stalled), this will substantially increase the overhead and complexity in the application.

Application complexity

Although the above application layer approaches are simple in the sense that they do not require any protocol changes at the transport layer, the complexity and overheads at the application layer are considerable. Essentially the application has to implement a resequencing mechanism over the reordering already performed within each pipe by TCP. Sequence numbers that facilitate the resequencing have to be included in application defined headers, and the application has to explicitly ensure that the application layer "segments" (that have unique application layer sequence numbers) do not get fragmented. One conceivable way the application can ensure that application layer segments are not fragmented is to write exactly one MSS (maximum segment size) worth of data during every write. If nagling [13] is enabled, this would achieve the desired goal. Similarly, in order to stripe intelligently, the application will have to redundantly implement a bandwidth estimation mechanism in spite of the bandwidth estimation already performed by TCP through its congestion control mechanism. Furthermore, in order to solve the problems identified as consequences of blackouts, the application will have to implement a feedback mechanism to recover from pipes that are stalled, and in effect duplicate both the reordering and loss recovery mechanisms already implemented by TCP for the individual pipes. It is clearly undesirable to overload applications in such a manner when all applications on the mobile host would require similar functionality. Note that the above arguments would also hold for session layer approaches in the absence of appropriate interfaces between the session layer and the transport layer.

Multiple congestion control schemes

Since different wireless network technologies possess very diverse characteristics in terms of throughput, delay, jitter, loss rates, etc., approaches to improve TCP performance over wireless networks have typically been proposed for specific scenarios. For example, while *snoop* [5] has been proposed primarily for WLANs, [18] shows that snoop is inappropriate in WWANs due to its key assumption that wireless link delays are insignificant when compared to the end-to-end delays. WTCP, proposed in [18] for WWANs, however, will stand inappropriate in WLANs due to its reliance on inter-packet separation as the key congestion metric. WWANs have low data rates that result in the inter-packet delay being large, which in turn makes it a robust and realistic metric to use. However, in WLANs where bandwidths can be in the order of tens of megabits per second, it no longer serves as a reliable congestion metric. Similarly, approaches such as [8] to specifically improve TCP's performance over satellite links that possess very large bandwidth-delay products have also been proposed. When a mobile host has multiple interfaces, a conceivable scenario (until a unified transport layer framework is derived) is one where interface-specific transport protocols will be used. In the simple application layer striping approach, besides the numerous roles assigned to the application, the task of choosing an appropriate transport protocol for the different pipes will also rely on the application.

3. The pTCP design

In this section, we present the key design elements of pTCP that overcome the drawbacks identified earlier for the application layer approaches. The following assumptions are made for the basic design of pTCP: (i) Mobile hosts have multiple interfaces, which they would ideally like to use simultaneously for a single application connection; (ii) Both the sender and the receiver support pTCP; (iii) The bandwidth bottlenecks are purely in the wireless links for the individual pipes; and (iv) The application should ideally be unaware of the striping process. Briefly, assumption (iii) is to ensure TCP-friendliness of the aggregate connection in the backbone Internet where paths of multiple pipes may merge. While we make the assumptions primarily for simplifying the resentation of pTCP, we revisit assumptions (ii) and (iii) in section 6, and discuss the required modifications to pTCP if the assumptions are to be relaxed.

The pTCP protocol is based on the following five design elements.

Decoupled congestion control and reliability

As described in section 1, pTCP is a *wrapper* around a modified TCP that we refer to as *TCP-v*. While we present the details of the interaction between pTCP and TCP-v in section 4, briefly pTCP maintains and controls a single send buffer across all the TCP-v pipes for the aggregate connection. The individual TCP-v pipes perform congestion control and loss recovery just like regular TCP. However, any segment transmission by a TCP-v is preceded by an explicit call to pTCP requesting for application data. Since pTCP has control over the buffer, a retransmission at the TCP-v level does not need to be a retransmission at the pTCP level. On the other hand, the amount of data that can be sent out through each TCP-v pipe is strictly determined by the TCP congestion control algorithm employed by each respective pipe. Therefore, TCP-v controls the *amount* of data that can be sent while pTCP controls *which* data to send. In this fashion, pTCP decouples congestion control and reliability. We describe as we go along how the decoupling contributes to improved performance and functionality in pTCP.

Congestion window based data striping

When a TCP-v pipe has space in its congestion window for transmissions, it requests pTCP for data. If there exists no unsent data, pTCP registers the concerned TCP-v pipe as an *active* pipe and returns a *freeze* value. The TCP-v pipe then waits for a subsequent *resume* call from pTCP before requesting for data again. When pTCP receives new data from the application, it issues the resume call only to those TCP-v pipes that are registered as active. Note that such striping is different from striping that is conditional on buffer availability (as seen in the unaware application layer approach). In pTCP, *data will be given to a TCP-v pipe only when there is space in its congestion window for the data to be sent.* Note that this inherently assumes the congestion window to be a true representative of the bandwidth-delay product of the pipe. While the TCP congestion window *is* an approximation of the bandwidth-delay product, it is possible that it is an incorrect estimation (say, for example, due to deep buffers in the network). We revisit this issue in section 6. The striping of data based on the congestion window of the individual pipes removes the problem that arises due to differences in the rates of the pipes, *provided there is no fluctuation in the available bandwidth.*

Dynamic reassignment during congestion

Recall that it is possible for the congestion window to be an over-estimate especially just before congestion occurs. This can result in an undesirable hold up of data in pipes where the congestion window was reduced recently. For example, consider a scenario in which the congestion window of pipe p_i is $cwnd_i$. If $cwnd_i$ worth of data is assigned to p_i, and the window is cut down to $cwnd_i/2$ due to bandwidth fluctuations, the $cwnd_i/2$ worth of data that falls outside the congestion window of p_i will be blocked from transmission till the $cwnd_i$ opens up. In the meantime, this is equivalent to a static scenario in which the application undesirably assigned more data than what a pipe can carry and in the process slows down other faster pipes. pTCP solves this problem by leveraging the decoupling that exists between congestion control and reliability. When a pipe experiences congestion, irrespective of whether the detection is through duplicate acknowledgements or a timeout, the window is reduced (by half in the former and to one in the latter). If the congestion window of a pipe is thus reduced, pTCP immediately *unbinds* the data that was bound to the sequence numbers of the concerned pipe that fall outside the current congestion window. Thus, if another pipe has space in its congestion window and requests for data, the unbound data is now available for reassignment to that pipe. When the original pipe requests for data corresponding to the same sequence number that was unbound, new application data is bound by pTCP and returned to the pipe. Such a reassignment strategy greatly improves the performance of pTCP under dynamic conditions, as we illustrate through simulation results in section 5. The trade-offs of the reassignment strategy is the potential overhead of performing unnecessary retransmissions. Our simulation results show that this overhead is insignificant.

Redundant striping for blackouts

While the strategy described above reassigns data that falls out of a pipe's congestion window, it does not deal with the one MSS worth of data (the first MSS in the congestion window) that will never fall out of the congestion window irrespective of the state of the pipe. Failure to deliver that one MSS worth of data can potentially stall the entire aggregate connection if the concerned pipe undergoes multiple timeouts or suffers a blackout. Hence, pTCP *redundantly stripes the first MSS of data in a congestion window that has suffered a timeout, onto another pipe.* In doing so, the binding of the data is changed to the new pipe, although the old pipe has access to a copy of the same data. The reason for leaving a copy behind instead of a regular reassignment is that the old pipe will require at least one MSS worth of data to send in order to recover. At the same time, providing it with a new MSS worth of data is a potential pitfall because of the chances of blocking, given that the pipe is experiencing severe conditions.

Selective acknowledgments

The pTCP design does not impose any requirements on the design of the TCP-v protocol used by the individual pipes. However, the use of TCP–SACK helps the performance of pTCP under certain conditions. We present here an argument for why TCP-v in pTCP should preferentially use TCP–SACK. When there are multiple losses within a congestion window, TCP–Reno and TCP–NewReno will recover from the losses at the rate of one loss per round-trip time. Thus, if multiple losses occur within a congestion window of a pipe, the time taken to recover from a loss farther down in the congestion window increases. In default TCP, this is acceptable as the alternative is to take a pessimistic attitude like in TCP–Tahoe that by default treats all packets after a hole to be lost, and hence starts retransmitting them. However, in pTCP, such a delayed recovery can make the hole potentially stall the entire aggregate connection. This makes the rate at which loss recovery is done critical. When TCP–SACK is used, loss recovery is done faster with multiple holes filled within one round-trip time because of the SACK information exchanged. This results in pTCP experiencing better performance. Since TCP–SACK is preferred in a general Internet setting [12], and more so in wireless environments to enable faster recovery from random channel errors [18], we believe that the recommendation for the use of SACK in pTCP is a reasonable one.

4. The pTCP protocol

4.1. Overview

Figure 2 provides an architectural overview of the pTCP protocol. pTCP acts as the central engine that interacts with the application, IP, and TCP-v, respectively. For each interface used by the application to achieve bandwidth aggregation, pTCP creates and maintains one TCP-v pipe. *We assume that the choice of the number of interfaces to use is an external decision, and is conveyed to pTCP through a socket option.* The figure also illustrates the key data structures maintained for every aggregate connection. pTCP controls and maintains the send and receive socket buffers for the connection. Application data writes are served by pTCP, and the data is copied onto the `send_buffer`. A list of active TCP-v pipes (that have space in the congestion window to transmit) called

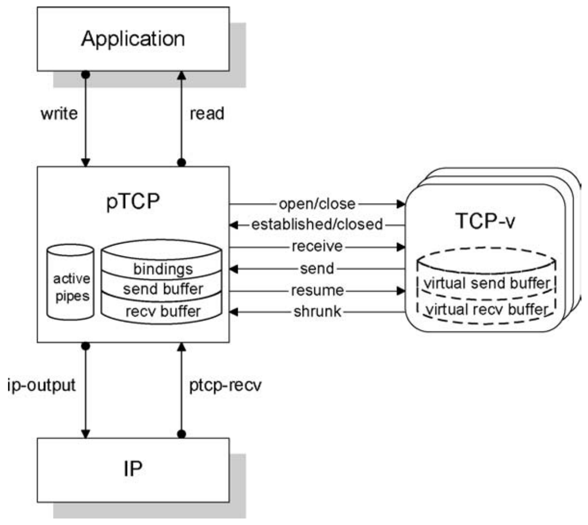


Figure 2. Overview of pTCP and key data structures.

`active_pipes` is maintained by pTCP. A TCP-v pipe is placed in `active_pipes` initially when it is created by pTCP. Upon the availability of data that needs to be transmitted, pTCP sends a *resume*() command to the active TCP-v pipes. Once a resume is issued to a pipe, the corresponding pipe is removed from `active_pipes`. A TCP-v pipe that receives the command builds a regular TCP header based on its state variables, and gives the segment (sans the data) to pTCP through the *send*() interface. pTCP binds an unbound data segment in the `send_buffer` to the header of the "virtual" segment TCP-v has built, maintains the binding in the data structure called `bindings`, inserts its own header, and sends it to the IP layer. A *resumed* TCP-v continues to issue *send*() calls till there is no more space left in the congestion window, or pTCP responds back with a *freeze* return value to "freeze" the concerned pipe (note that the TCP-v pipe needs to perform a few rollback operations to account for the unsuccessful transmission). When pTCP receives a *send*() call, and has no unbound data left, it returns a *freeze* value, and adds the corresponding pipe to `active_pipes`.

When pTCP receives an *ACK*, it strips the pTCP header, and hands over the packet to the appropriate TCP-v pipe (through the *receive*() interface). The correct TCP-v pipe is recognizable from the TCP 4-tuple. The TCP-v pipe processes the *ACK* in the regular fashion, and updates its state variables including the virtual send buffer. The virtual buffer can be thought of as a list of segments that have only appropriate header information. The virtual send and receive buffers are required to ensure regular TCP semantics for congestion control and connection management within each TCP-v pipe. The pTCP header carries cumulative pTCP level ACK information that pTCP uses to purge its receive buffer if required. When pTCP receives an incoming data segment, it strips both the pTCP header and the data, enqueues the data in the `recv_buffer`, and provides the appropriate TCP-v with only the skeleton segment that does not contain any data. TCP-v treats the segment as a regular segment except that no application data is queued in the virtual receive buffer.

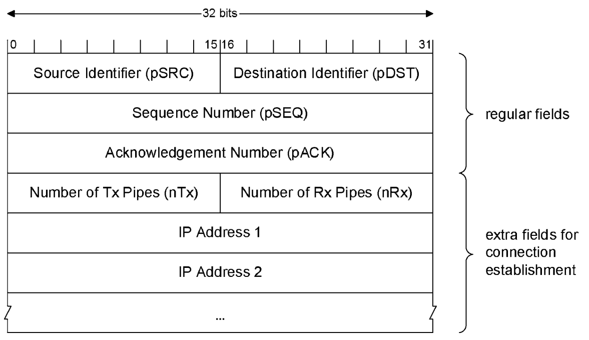


Figure 3. pTCP header format.

In the rest of the section, we describe the different components of the pTCP protocol including interfaces with TCP-v, header formats, connection management, congestion and flow control, and reliability.

4.2. TCP-v interface

As seen in figure 2 the following eight functions act as the interface between pTCP and TCP-v: *open*(), *close*(), *established*(), *closed*(), *receive*(), *send*(), *resume*(), and *shrunk*(). pTCP uses the *open*() and *close*() calls as inputs to the TCP-v state machine for opening and closing a TCP-v pipe, respectively. TCP-v uses the *established*() and *closed*() interfaces to inform pTCP when its state machine reaches the **ESTABLISHED** and **CLOSED** states, respectively [26]. The *send*() call is used by TCP-v to send "virtual" segments to pTCP which will bind the segments to real data. The *receive*() interface on the other hand is used by pTCP to deliver "virtual" segments to TCP-v. pTCP uses *resume*() to inform TCP-v that additional unbound data is available. TCP-v, upon receiving the call, attempts to send as much data as possible till it gets a *freeze* return value on its *send*() call and freezes. Finally, TCP-v uses the *shrunk*() interface to inform pTCP of any change in its congestion window so that pTCP can perform reassignment as described in section 3.

4.3. Header formats

Figure 3 presents the header formats for the pTCP protocol. Note that the header is in addition to the regular TCP header that will be used by TCP-v. The regular pTCP header consists of the following four fields: (i) source connection identifier (*pSRC*), (ii) destination connection identifier (*pDST*), (iii) pTCP sequence number (*pSEQ*), and (iv) pTCP acknowledgement number (*pACK*). The connection identifiers are used to uniquely identify the aggregate pTCP connection at both ends. The *pSEQ* is the sequence number at the aggregate connection level and is independent of the TCP sequence number. The *pACK* is a cumulative acknowledgement similar to the TCP acknowledgement (ACK) field. Note that the individual TCP-v pipes will use the TCP ACK fields to perform congestion control (recall that congestion control and loss recovery are coupled in TCP), and hence they cannot be reused by pTCP. Because pTCP is responsible for performing flow control (given that it controls the buffer), it requires

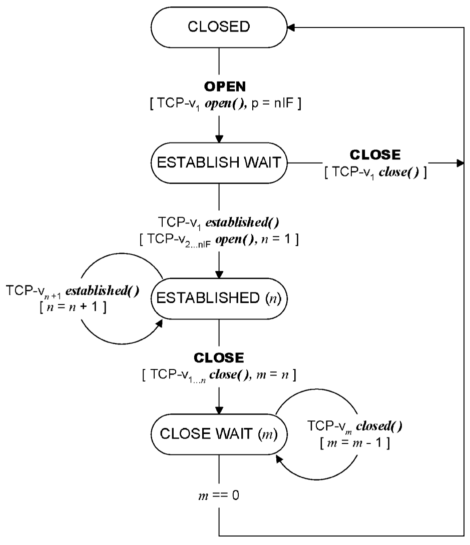


Figure 4. pTCP state machine.

a field for window advertisement as in TCP. However, since TCP-v pipes do not have to perform flow control (they merely maintain virtual buffers), pTCP reuses and overrides the TCP window advertisement field for performing flow control. The reuse does not interfere with the progress of the individual TCP-v pipes due to the fact that the pTCP advertised window will always be greater than the actual window of an individual pipe (we elaborate on this in section 4.5).

In addition to the regular pTCP header fields, the header format for the connection establishment phase is further augmented with the following fields: (i) number of transmitting interfaces to be desirably used (*nTx*), (ii) number of receiving interfaces that can be used (*nRx*), (iii) list of IP addresses corresponding to *nTx* (*ipTx*), and (iv) list of IP addresses corresponding to *nRx* (*ipRx*). The *nTx* field is the number of interfaces the source would ideally like to use for its transmissions (which in effect will require *nTx* pipes to be maintained at both ends), and the *nRx* field is the maximum number of interfaces on which the source is willing to serve the reverse path. Note that even if *nRx* is 1 at one end, the other end can still use multiple interfaces, but all pipes will terminate in the one interface at this end. (This would be the typical setup when a multi-homed mobile host communicates with an Internet backbone host that has only one interface.)

4.4. Connection management

We use the state machine of pTCP and the connection establishment handshake presented in figures 4 and 5, respectively, for the following discussions. Note that the state machine for TCP-v is the same as that of default TCP, and the interface between pTCP and TCP-v is presented in section 4.2. We assume the number of interfaces to be *nIF* at both ends.

Establishment

When an active open is issued by the client application, a pTCP socket with a transmission control block (TCB) similar to TCP's TCB, but with the additional state variables introduced earlier in section 4.1, is created. After the pTCP socket is created, pTCP creates one TCP-v TCB and issues the *open*() call to it. When the TCP-v *SYN* packet is sent out, pTCP sets *nIF* in the *nTx* field and the corresponding IP addresses in *ipTx*, and appends additional pTCP connection management header information to the packet. When the pTCP at the server end receives the *passive* open, it checks to see if it is willing to support *nIF* TCP-v pipes.[3] Assuming that the receiver can support the required number of pipes, it creates the first TCP-v TCB, issues the passive open to it, and in the process takes it to the **SYN_RCVD** state [26]. When the *SYN+ACK* is sent out by the first TCP-v at the server end, the destination IP address is appropriately set based on the information received in the first *SYN*, and the source address reflects the local host interface the first TCP-v pipe is bound to. The *SYN + ACK* message carries *nIF* in the *nRx* field that the server has agreed to support, and the corresponding IP addresses in *ipRx*.

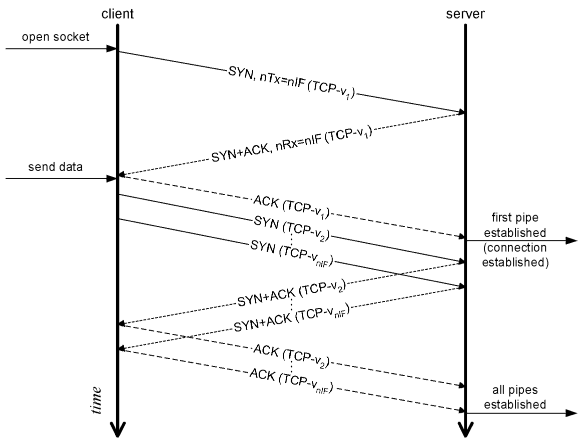


Figure 5. Connection establishment handshake.

When the client pTCP receives the *SYN + ACK*, it creates the remaining *nIF* − 1 TCP-v TCBs and issues *open*() calls to each of them. Also, the first TCP-v pipe at this stage enters the **ESTABLISHED** state after sending back an *ACK* to the server. pTCP thus goes into the **ESTABLISHED (1)** state and can start accepting data from the application. *Hence, even if some of the pipes are experiencing connection setup problems, pTCP will still ensure data flow between the client and the server.*

The source IP address of each of the outgoing *SYN*s is set to the local interface the TCP-v pipe is bound to. The destination address is set to one of the addresses in *ipRx* in the *SYN + ACK* sent from the server. When the first TCP-v pipe at the server receives the *ACK*, it enters the **ESTABLISHED** state and can thus participate in the data exchange with the client. The pTCP at the server end also enters the **ESTABLISHED (1)** state. When the server receives the *SYN* messages from each of the remaining *nIF* − 1 TCP-v

[3] There might be several reasons including memory or processor limitations, security considerations, etc., because of which the receiving host might desire to limit the number of pipes.

pipes, it creates the corresponding TCP-v TCBs and assigns the respective *SYN*s to the TCBs, taking each of them to the **SYN_RCVD** state. From there on, the exchange of information between each server TCB and the corresponding client TCB is similar to that of TCP.

As and when each of the individual TCP-v pipes enter the **ESTABLISHED** state, they issue the *established*() call to pTCP making pTCP move down the state machine shown in figure 4. Finally, when all the individual pipes enter the **ESTABLISHED** state, pTCP enters the **ESTABLISHED (nIF)** state.

Termination

The teardown of a pTCP connection is relatively simpler than the connection establishment. When an application closes the connection, pTCP uses the *close*() interface to make the individual TCP-v pipes close. Each pipe closes using TCP's regular closing handshake. When a TCP-v pipe enters the **CLOSED** state in its state machine, it invokes the *closed*() callback to pTCP. For every *closed*() message pTCP receives, it appropriately keeps track of the number of closed TCP-v pipes. Upon successful completion of all TCP-v pipes, pTCP enters the **CLOSED** state of figure 4 and confirms the close to the application layer.

4.5. Congestion control and flow control

pTCP by itself does not perform any congestion control. The individual TCP-v pipes are solely responsible for controlling the amount of data transferred through each pipe. On the other hand, flow control in pTCP is performed at the pTCP layer. While the primary reason is the fact that pTCP has control over the receive buffer, it also helps in better utilization of the buffer across the multiple pipes. For example, in the case of the unaware application approach, irrespective of the bandwidth-delay product (BDP) of the individual TCP pipes, each pipe would have a constant buffer (of 64 KB by default). This will result in wastage of buffer space for pipes with smaller BDPs and wastage of capacity for pipes with larger BDPs. However, in pTCP the buffer space will be shared by the individual pipes based on their respective BDPs. Note that this property can also be achieved using some approaches proposed in related work [17].

We assume the buffer space (both send buffer and receive buffer) available at the pTCP layer is equal to $n \cdot B$, where n is the number of TCP-v pipes, and B is the default TCP buffer size. Every segment that belongs to a pTCP connection always carries the available space in the pTCP receive buffer, irrespective of which pipe it belongs to. The pTCP sender keeps track of the number of outstanding bytes for the connection, and ensures that the receive buffer never overflows. Although all individual TCP-v pipes see the same available buffer space and hence can contend simultaneously for that space (provided there is space in their congestion windows), since pTCP has control over all data transmissions, it prevents any excess data from being transmitted. For example, consider a scenario in which the receiver has advertised a window size of 1000 bytes. Assuming that there exist three TCP-v pipes at the sender and each of them has 1000 bytes space left in the congestion window, each of the pipes will attempt to transmit 1000 bytes worth of data. However, except for the first pipe that succeeds in transmitting the 1000 bytes, the other pipes would have a *freeze* value returned for their *send*() calls since pTCP would be aware of the global situation.

4.6. Reliability

As we discuss in section 4.1, pTCP maintains the bindings between the actual data segments and the TCP-v virtual segments. Once the application data is bound to a particular TCP-v pipe, it is the concerned TCP-v pipe's responsibility to reliably deliver the data to the receiver by using its own (essentially TCP's) reliability mechanism. Therefore, reliable transport of the application data is achieved in pTCP as long as every data segment in pTCP's send buffer is bound to a virtual segment in one of its TCP-v pipe's virtual send buffer. However, note that the binding between the actual data segment and TCP-v virtual segment can be altered when pTCP performs dynamic reassignment or redundant striping (see section 3). We now discuss how pTCP can still ensure reliability under these two special conditions.

- *Dynamic reassignment*. Whenever pTCP does dynamic reassignment of a data segment from a particular TCP-v pipe, it unbinds the data segment from that pipe, and reassigns (binds) it to the next available pipe (which can be the same pipe). From then on, the new pipe will assume the responsibility of reliably delivering the reassigned data segment to the peer TCP-v. Note that when the old pipe "retransmits" the previously bound virtual segment, it will be reassigned a different data segment to carry.
- *Redundant striping*. Redundant striping in pTCP is a special case of the dynamic reassignment where the old pipe still keeps a copy of the data segment. Since the data segment will be bound to a new pipe, its reliable delivery will be guaranteed by the new pipe. However, since the old pipe will also attempt to deliver the same data segment to its peer, there might be duplicates at the receiving pTCP. Such duplicates can be easily detected via the sequence number field (*pSEQ*) in the pTCP packet header.
  We have thus far described the key components of the pTCP protocol. In the next section, we present performance of pTCP and compare it with the performance of both the simple and sophisticated application layer techniques.

5. Performance evaluation

5.1. Simulation model

We evaluate the performance of pTCP using the *ns-2* [24] network simulator and the network topology shown in figure 6. The network topology consists of 5 backbone routers (R0–R4) and 20 access nodes. For simplicity the backbone routers

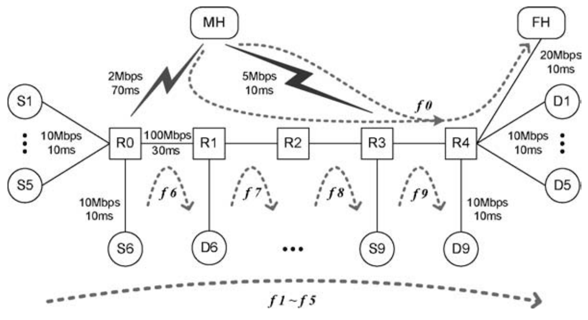
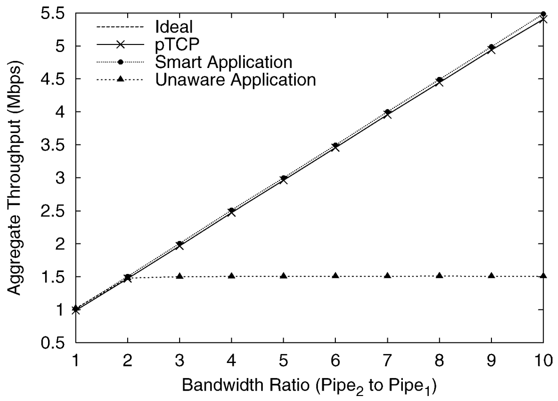


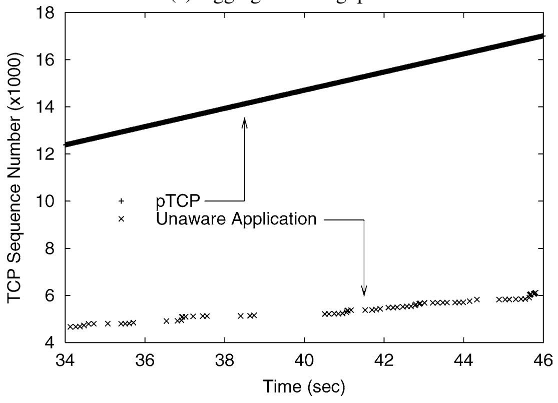
Figure 6. Network topology.

also double up as wireless access points (or base stations) that the mobile host (MH) can use to communicate with a fixed host (FH) in the backbone network. The mobile host primarily uses two different types of wireless links: (i) link MH–R0 with bandwidth equal to 500 Kbps or 2 Mbps (depending upon the scenario) and a 70 ms access delay, and (ii) link MH–R3 with bandwidth ranging from 500 Kbps to 5 Mbps (depending upon the scenario) and a 10 ms access delay. Together with the delay experienced in the backbone network, these values simulate a WWAN connection (through a macro-cell or a pico-cell) with 400 ms round-trip time, and a shared WLAN connection with 100 ms round-trip time, respectively. We introduce random bandwidth fluctuations (between 20% and 100% of the maximum link capacity), blackouts (as long as 25 seconds), and random losses (from 0.01% to 1% packet loss rate) in the wireless links to capture the link characteristics. Besides the concerned flow ($f0$) between MH and FH, we also include 9 other flows ($f1$–$f9$) to simulate the background traffic in the backbone network: 5 long TCP flows with 280 ms round-trip time (S1–D1 to S5–D5), and 4 short UDP flows with 10 Mbps data rate and 100 ms round-trip time (S6–D6 to S9–D9) as shown in the figure.

To observe the effect of bandwidth aggregation, we consider a backlogged application at the mobile host that uses active network interfaces to perform data striping, and measure the throughput delivered to the peer application at the fixed host. We compare the performance of pTCP with two application layer striping techniques: an unaware application and a smart application. The *unaware* approach, as described in section 2, represents the simplest form of the application layer striping where the sending application writes to each socket, in turn, until blocked, and the receiving application reads from each socket only in-sequence packets. On the other hand, the *smart* approach relies on a sophisticated application to perform bandwidth estimation of the end-to-end path, such that it can stripe according to the ratio of the available data rates of individual pipes. For reference, we also present the "ideal" performance of bandwidth aggregation where we add up the individual throughputs of independent TCP connections, one each for every interface, that do not experience any head-of-line blocking. We use TCP-SACK for all TCP flows including the TCP-v pipes in the simulation. Unless otherwise specified, all data points are averaged over 10 samples using random seeds, and each sample is run for a period of 600 seconds. We present the following results for different



(a) Aggregate throughput.



(b) Sequence number progression.

Figure 7. Scalability with rate differential.

striping approaches in the rest of the section: (i) scalability with respect to rate differential, (ii) scalability with respect to the number of pipes, (iii) resilience to rate fluctuations, (iv) resilience to blackouts, and (v) co-existence of different congestion control schemes.

5.2. *Rate differential*

We show in this section the impact of rate differential between the two pipes of the aggregate connection. We fix the bandwidth of one of the wireless links to 500 Kbps (link MH–R0), and increase the bandwidth of the other link from 500 Kbps to 5 Mbps in increments of 500 Kbps (link MH–R3). Note that since the wireless link is the bottleneck of the end-to-end path, the data rate at which the pipe can send is equal to the bandwidth of the wireless link. Figure 7(a) shows the ideal aggregate bandwidths and the performance of the 3 different striping approaches when the rate differential between the two pipes increases. The x-axis value represents the bandwidth ratio of the second pipe to that of the first pipe, while the y-axis value shows the aggregate throughput achieved at the receiving application. It is clear from the figure that both pTCP and the smart application achieve near ideal performance, but the unaware approach performs significantly worse and exhibits a non-increasing aggregate throughput beyond a ratio of 2.

The non-performance of the unaware application, while explained in section 2, can be further illustrated by the results presented in figure 7(b), where we compare the sequence

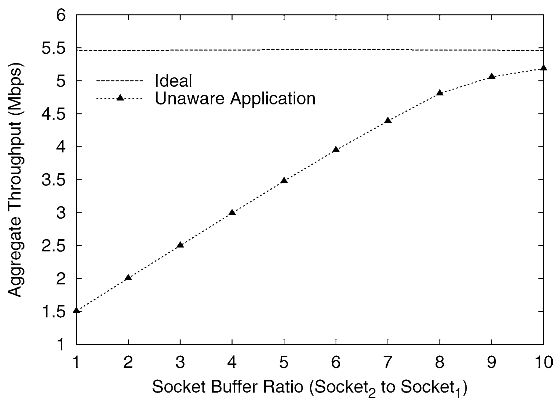


Figure 8. Buffer requirement for the unaware application.

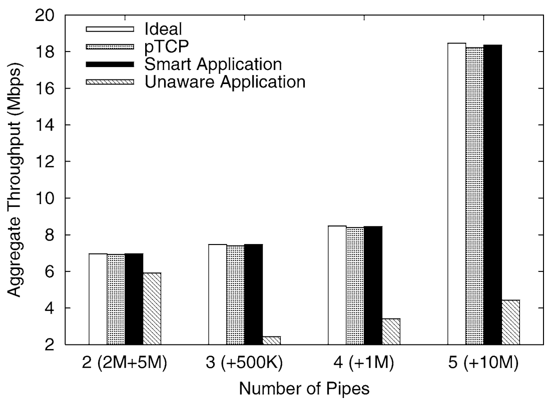


Figure 9. Scalability with number of pipes.

number progression of the faster pipe (the bandwidth ratio of the two pipes is set to 6) in the unaware application and pTCP during a small time window. We observe that in the unaware application, the head-of-line blocking at the receiver due to the slower pipe holding up packets stalls the faster pipe. The faster pipe thus exhibits distinct idle periods (e.g., 36 s, 38 s, and 40 s in figure 7(b)), resulting in the degraded performance of the unaware application. One way to avoid such head-of-line blocking is to perform bandwidth estimation at the application and stripe data according to the bandwidth ratio of the two pipes, as demonstrated by the performance of the smart application. In contrast, pTCP by virtue of its congestion window based striping, frees itself from re-implementing the bandwidth estimation already performed by TCP's congestion control, and achieves the same ideal performance.

Note that as we discuss in section 2, another way to improve the performance of the unaware application is to reduce the coupling due to head-of-line blocking between the faster and slower pipes by increasing the size of the resequencing buffer at the application. Equivalently, we can increase the size of the socket resequencing buffer in the faster pipe while keeping that of the slower pipe fixed. We show in figure 8 the effect of using a larger buffer size in the faster pipe (an x-axis value of 1 indicates the default buffer size) when the bandwidth ratio of the two pipes is 10. It can be observed that the unaware application will be able to achieve nearly the desired aggregate bandwidth only when buffer over-allocation ratio is at least the ratio of rate differential, which essentially limits its scalability.

5.3. Number of pipes

We now show that the performance observations made in the previous section still hold true when the number of pipes in the aggregate connection increases beyond 2. We use the same network topology as shown in figure 6, but increase the number of network interfaces at the mobile host from 2 to 5. Since the mobile host opens one pipe for each active network interface, it incrementally adds the following pipes to the aggregate connection:

(1) $pipe_1$ with 2 Mbps bandwidth and 400 ms round-trip time (via MH–R0 link),

(2) $pipe_2$ with 5 Mbps bandwidth and 100 ms round-trip time (via MH–R3 link),

(3) $pipe_3$ with 500 Kbps bandwidth and 300 ms round-trip time (via MH–R1 link),

(4) $pipe_4$ with 1 Mbps bandwidth and 200 ms round-trip time (via MH–R2 link), and

(5) $pipe_5$ with 10 Mbps bandwidth and 50 ms round-trip time (via MH–R4 link).

Figure 9 presents the aggregate throughput enjoyed by the application at the fixed host as the number of pipes increases. We compare the performance of ideal bandwidth aggregation and that of the 3 striping techniques, and conclude that the performance of pTCP scales well with increasing number of pipes. Note that although the smart application is also able to achieve the desired bandwidth aggregation, the overheads incurred at the application due to bandwidth estimation also increase with the number of pipes (bandwidth estimation is performed on a per-pipe basis). While it is clear the unaware application cannot achieve the desired performance, we observe that when the third pipe (500 Kbps) is added to the striped connection, the aggregate throughput of the unaware application even "degrades" to a value lower than that of using only 2 pipes! This observation further substantiates our argument in section 2 that the performance of the unaware application will be throttled by the slowest pipe of the aggregate connection. Hence it is not always advantageous for the unaware application to use as *many* pipes as possible to achieve the maximum aggregate throughput.

5.4. Rate fluctuations

Since the characteristics of wireless links exhibit large variances, in this section we investigate the performance of pTCP in the presence of fluctuations in the available data rate of individual pipes. First we emulate the bandwidth fluctuations of a wireless link by periodically changing its link bandwidth to a random value. Specifically, for a link k with a maximum capacity of C_k, its bandwidth will change every T_k seconds to a value randomly chosen from $[0.2C_k, C_k]$ and rounded to the

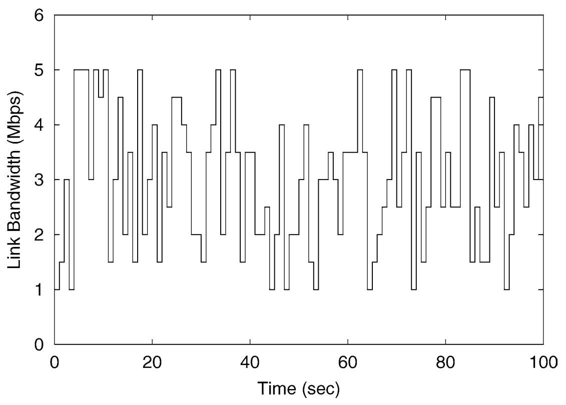
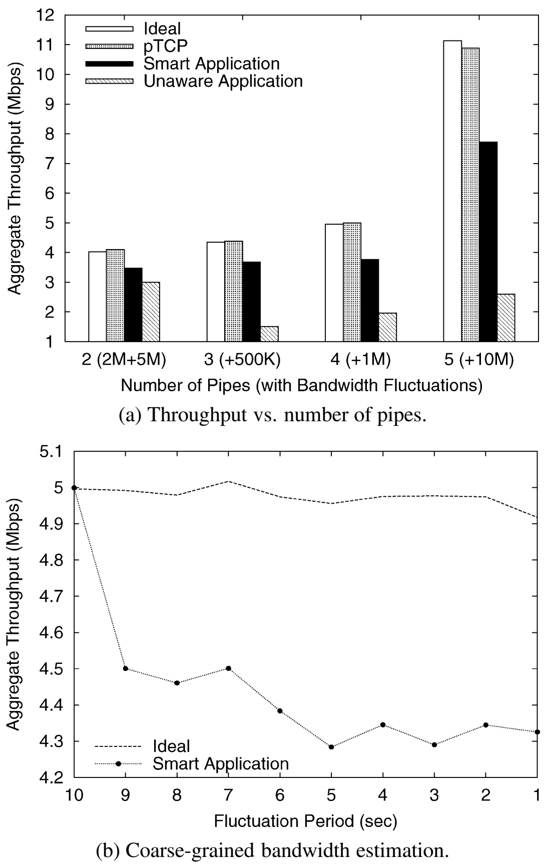


Figure 10. A sample trace of bandwidth fluctuations.

nearest tenth of C_k. Figure 10 shows a sample trace of bandwidth fluctuations for a wireless link with C_k set to 5 Mbps and T_k set to 1 second.

We now show the impact of rate fluctuations on the performance of pTCP, unaware application and smart application. We consider the same topology as the one used in section 5.3 where the mobile host is equipped with multiple network interfaces ranging from 2 to 5. Each pipe has the same round-trip time as before, but now with random data rate fluctuations. The fluctuation period of each pipe is to 3, 10, 6, 20, and 5 seconds, respectively. Figure 11(a) shows the ideal performance of bandwidth aggregation and the 3 striping techniques as the number of pipes with rate fluctuations increases. We note that figure 11(a) mirrors figure 9 while the ideal aggregate throughput scales down to 60% due to bandwidth fluctuations (the average bandwidth for a link with capacity C_k is $0.6C_k$). It is clear that even under such a dynamic environment, the performance of pTCP still closely follows that of the ideal performance. However, this is not the case for the smart and unaware applications. Specifically, compared with figure 9 we find that the performance of the smart application drops significantly, demonstrating the inefficiency when its bandwidth estimation mechanism cannot accurately track the actual data rate fluctuations.

We use figure 11(b) to further substantiate this argument, where we show the performance of the ideal bandwidth aggregation and smart application in a two-pipe scenario. The first pipe has 2 Mbps bandwidth and 400 ms round-trip time, while the second pipe has 5 Mbps bandwidth and 100 ms round-trip time. We fix the bandwidth of the first pipe, but introduce bandwidth fluctuations in the second one. We assume the smart application performs a coarse-grained bandwidth estimation such that it is able to obtain correct bandwidth estimates of the second pipe every 10 seconds. Figure 11(b) shows that as the fluctuation period of the second pipe decreases from 10 seconds to 1 second, the performance of the smart application degrades drastically, even when there is a slight mismatch between the bandwidth estimation interval and the actual bandwidth fluctuation period (say 9 seconds).



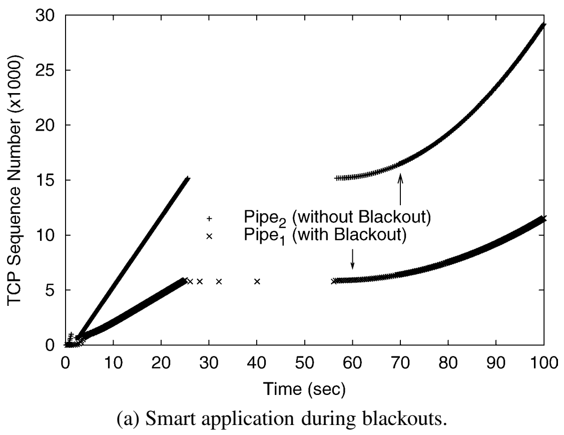
(a) Throughput vs. number of pipes.

(b) Coarse-grained bandwidth estimation.

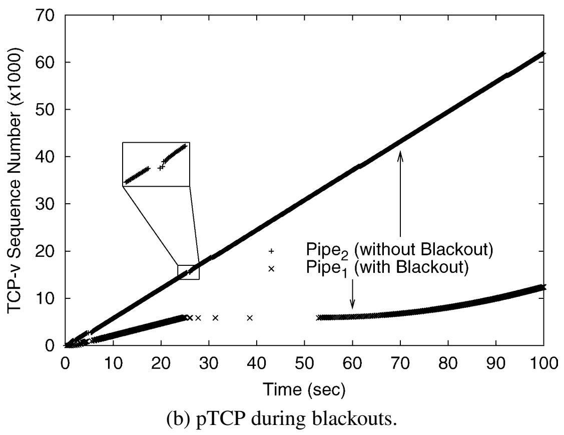
Figure 11. Impact of rate fluctuations.

5.5. *Blackouts*

In this section, we show the impact of blackouts on the performance of pTCP and application striping techniques. We use a two-pipe scenario, where the first pipe has 2 Mbps bandwidth and 400 ms round-trip time, and the second one has 5 Mbps bandwidth and 100 ms round-trip time. We introduce a long blackout in the first pipe between 25 s and 50 s when the available bandwidth decreases to zero, and all packets transmitted during such period are dropped. However, the second pipe does not experience any blackout. Figure 12(a) shows the TCP sequence number progression (at the mobile host) for both pipes in the smart application. While it is obvious that during the blackout period, the first pipe stops sending data after repeated failures, and starts bandwidth probing using exponential backoff, because of the head-of-line blocking problem described in section 2, even the second pipe (which does not experience blackouts) stalls almost for the entire duration of the blackout period, resulting in the aggregate connection coming to a standstill. On the other hand, as seen in figure 12(b), in pTCP, although the first pipe stalls during the blackout period as in the smart application, the second pipe continues to progress after a minor stall. This is possible because of the redundant striping policy in pTCP that reassigns even the first segment within the congestion window of the first pipe to the second pipe and prevents the latter from experiencing head-of-line blocking. pTCP thus makes a case for using the aggregate connection to improve reliability in the



(a) Smart application during blackouts.



(b) pTCP during blackouts.

Figure 12. Impact of blackouts.

face of link failures, which is not observed in the smart application. While the smart application can potentially use some mechanisms to handle blackouts, this will impose added complexity at the application. Not only will the application need to detect the presence of blackouts through appropriate feedback mechanisms, but will also have to "pull" back segments that are already in the pipe experiencing a blackout, and restripe them over the other pipe.

5.6. Different congestion control schemes

So far we have only considered a pTCP connection when its TCP-v pipes use the same congestion control scheme. In this section, we demonstrate the ability of the pTCP protocol to use two different congestion control schemes within the same aggregate connection. We consider the same two-pipe scenario as the one used in section 5.5. We introduce random losses to the first pipe by inserting a loss module in the MH–R0 wireless link (refer to figure 6). The module inserts losses at packet error rates ranging from 0.01% to 1%. We consider the performance of pTCP when the first pipe uses TCP–ELN [3] and the second pipe uses regular TCP (we do not explicitly introduce losses in the second pipe). TCP–ELN receives explicit loss notification from the underlying link layer when a packet is dropped due to random wireless losses, and does not react to such losses. Note that we use TCP–ELN to emulate an intelligent congestion control scheme that can differentiate the wireless random losses from congestion losses. It is not the sophistication of the congestion control scheme used that is of key importance, but the ability of the pTCP protocol to accommodate two different congestion control schemes within the same framework. Figure 13 shows the performance of pTCP when the loss rate in the first pipe increases from 0.01% to 1%. pTCP is able to achieve the ideal aggregate bandwidth (sum of the throughputs of an independent TCP–ELN connection over the first pipe and an independent TCP connection over the second pipe) for all cases, illustrating the seamless nature in which pTCP allows the two congestion control schemes to co-exist. For comparison we also show the performance of the smart application for the same scenario. We observe that when the packet loss rate is low, the smart application is able to achieve the ideal performance. However, as the loss rate increases, the fact that TCP–ELN becomes ineffective in recovering from the random losses in the first pipe (the ideal curve goes down as loss rate increases) introduces head-of-line blocking at the receiver, and stalls the second pipe in the smart application. (For example, the aggregate throughput of the smart application when the loss rate is 1% is lower than the bandwidth of the second pipe.)

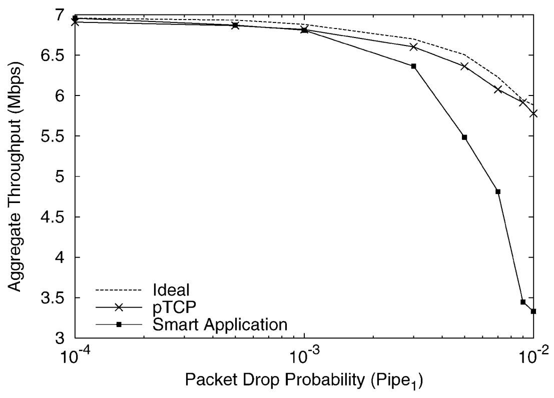


Figure 13. Multiple congestion control schemes in pTCP.

6. Issues and discussion

In this section we discuss some issues with the pTCP design.

Congestion window and bandwidth-delay product

One of the key assumptions made by pTCP when it performs congestion window based striping is that the congestion window is a tight approximation of the available bandwidth-delay product. However, this might not always be true. For example, deep buffering in the network can artificially inflate the congestion window to much larger than the true bandwidth-delay product of a connection. One plausible solution to this problem is to complement the basic congestion control scheme in TCP-v with mechanisms that help estimate the BDP more accurately. For example, if the rate of incoming *ACK*s for a particular pipe is monitored to keep track of the available rate, once the sending rate of the pipe exceeds the BDP, the rate of incoming *ACK*s will hit a plateau. pTCP thus can use this as an indication to cap the amount of data assigned to that pipe. Note that since pTCP is implemented as a transport layer approach, it has easier access to the states of TCP than a higher layer approach does. Our current work involves studying this problem more closely.

TCP friendliness

An important consideration when using multiple interfaces and bandwidth aggregation is how the aggregation plays a role in the TCP friendliness of the connection. Can it happen that in performing the aggregation the end-to-end connection loses its TCP friendly nature? If the assumption made earlier in the paper that the bottlenecks are solely in the wireless domain were true, this would not be an issue. However, if the bottlenecks happen to be in the wired domain, it is possible that multiple TCP-v pipes share a single bottleneck in the wired domain and make the aggregate pTCP connection more aggressive than a single regular TCP connection. There are two possible solutions to this problem: (i) Recent works have proposed schemes to heuristically determine if connections share the same bottleneck by monitoring the packet loss patterns and inter-arrival times [16]. If it can thus be inferred that two TCP-v pipes are sharing the same bottleneck link, approaches like [4] can then be employed to take care that the aggregate behavior of the two TCP-v pipes mimics that of a single regular TCP connection. (ii) Another approach that we have extensively explored in a different context is to use a variant of TCP–Vegas. TCP–Vegas by itself has been shown to be *subservient* to TCP–Reno and TCP–NewReno flows. A more subservient variant of TCP–Vegas (where the congestion window is reset to one upon detection of consistent increase in the history of round-trip times maintained) can be made to use only bandwidth given up by other Reno or NewReno flows. Therefore, in pTCP while one TCP-v pipe would use a Reno or a NewReno congestion control scheme, the other pipes would use a variant of the TCP–Vegas congestion control algorithm, thus lessening the TCP unfriendly effect when multiple TCP-v pipes share the same bottleneck.

Backward compatibility

In this paper, we have made an assumption that both the sender and the receiver are pTCP aware. We believe this assumption to be reasonable under scenarios where mobile users predominantly communicate with proxies that are already mobile-host aware. However, when the mobile hosts communicate with static hosts that can be potentially unaware, the problem can be handled just like it is handled in other protocols such as TCP–SACK or when the timestamp option is used in TCP. When the SYN packet is sent to start the connection, the option is enabled. If the other end replies with the same option, “awareness” is inferred and the respective protocol is used. However, if the other end replies without the option, normal TCP operation resumes. Since pTCP headers and connection establishment handshakes can be implemented through TCP options in the first place, using such “pTCP PERMITTED” techniques seems to be a realistic solution to ensure correct operations when a pTCP aware host communicates with a pTCP unaware host.

Handoffs

While we do not consider handoffs explicitly in the paper, note that the handoff experienced by an individual TCP-v connection can be handled in the default manner it would have been handled if it were the only pipe used by the application. In fact, the use of pTCP ensures that even if stalls are caused in one pipe due to handoffs, the other pipes remain unaffected. Moreover, soft handoff can be easily achieved (where the mobile host is connected to multiple access points in the intersection of their coverage areas) using pTCP over the two pipes established to the two access points during handoffs.

Complexity

There are two sources of complexity in pTCP that can cause potential problems: (i) The creation and maintenance of multiple TCB states can be a drain on the end-host’s resources. pTCP’s connection establishment phase addresses this problem by allowing the end-host to accept requests for an aggregate connection by specifying a limit on the number of pipes that it can support. (ii) The overheads incurred by the buffer management at pTCP and individual TCP-v pipes. However, in pTCP the manipulation of the socket buffer incurs the same overhead as the buffer management mechanisms in a regular TCP socket. The only additional overheads occur when packets are unbound following a congestion window reduction. In this case, the unbound packets need to be re-inserted into the unsent list sorted according to sequence numbers. We are currently investigating efficient data structures that can reduce the overheads incurred when the re-insertion is performed.

7. Related work

We classify related work based on whether the proposed approaches to achieve bandwidth aggregation are performed at the application layer, transport layer, network layer, or link layer.

Application layer techniques

Several approaches have been proposed to use multiple TCP connections in parallel to provide higher throughput to the application. For example, in [19] the authors develop the PSockets library used to stripe data over multiple TCP sockets for a better utilization of the network bandwidth, while avoiding the time-consuming process of manually tuning the TCP buffer size. In [2] the authors develop a new application called XFTP that uses multiple TCP connections to overcome the limitation of TCP window size in long-fat links such as satellite links. Similarly, in [10] an extension of the FTP protocol called GridFTP is developed for bulk data transfer where parallel TCP connections are used to increase the throughput in a bottleneck link. In [7], the authors characterize and substantiate the performance improvement of an aggregate connection that uses parallel TCP connections over the same path. Note that this class of related work deals with using multiple TCP connections over the same path. We discuss in section 2 the pitfalls of such approaches when used in the context of multi-homed mobile hosts.

Transport layer techniques

The Stream Control Transmission Protocol (SCTP) is a reliable transport protocol that was designed for the transport of message-based signaling information across IP-based networks [23]. One salient feature of SCTP is the support for multi-streaming and multi-homing. A SCTP connection can consist of multiple data streams across one or multiple interfaces. SCTP provides reliable in-sequence delivery within each data stream, *but it does not provide a total ordering across the data streams.* Although the head-of-line blocking among different data streams is thus avoided in SCTP, it cannot provide bandwidth aggregation as pTCP does. If a SCTP user is to stripe data across multiple data streams, it must handle packet resequencing itself. Another distinct difference between SCTP and pTCP lies in the congestion control mechanism. Multiple data streams within the same SCTP connection are subject to one common flow and congestion control mechanism. In the context of multi-homed mobile hosts where different data streams traverse through vastly differing wireless links and heterogeneous access networks, such design unnecessary leads to bandwidth underutilization for the aggregate connection. The Reliable Multiplexing Transport Protocol (RMTP) is a rate-based transport layer approach that is specifically designed to aggregate bandwidths on multi-homed mobile hosts [11]. Although RMTP targets the same scenario as pTCP does, it differs from pTCP in several ways: (i) RMTP performs explicit bandwidth based striping. The available bandwidth of the underlying pipe is estimated by periodically sending packet-pair probes. The effectiveness of RMTP thus greatly depends on the accuracy of the bandwidth estimation. However, the bandwidth probing rate limits how fast it can detect and adapt to bandwidth fluctuations. When bandwidth fluctuations occur at a time-scale smaller than the bandwidth probing period, RMTP will exhibit the same problem as the smart application does shown in section 5. (ii) The RMTP design does not explicitly address the interaction between component pipes of the aggregate connection as pTCP does through delayed binding, dynamic reassignment and redundant striping. We show in earlier sections that these design components play an important role in achieving effective bandwidth aggregation on multi-homed mobile hosts. (iii) Finally, RMTP does not provide interfaces allowing the flexible inclusion of different congestion control mechanisms optimized for different wireless links.

Network layer techniques

In [15], the authors propose a mechanism for data streaming across multiple wireless links on a multi-homed mobile host, where the traffic splitting and aggregation are done at the network (IP) layer with the purpose of being transparent to the transport layer. Since the aggregate connection opens only one TCP socket and thus is associated with only one pair of IP addresses,[4] packets to and from different network interfaces (hence carrying different IP addresses) need to be tunneled using the primary IP addresses to reach the desired socket. Although such IP-in-IP encapsulation addresses the routing problem for the aggregate connection, the authors show that the scalability of the proposed mechanism in terms of tolerating disparity between different wireless links will be limited by TCP's unawareness of the striping operation. Specifically, the out-of-order delivery due to bandwidth and delay mismatch will trigger TCP's "fast retransmit" mechanism, making TCP unnecessarily reduce its congestion window and thus sending rate. Furthermore, out-of-order delivery may even cause TCP's retransmission timer to timeout and make TCP crawl in slow start. For a typical RTO value that incorporates 4 times RTT deviation, the authors show that, to avoid unnecessary timeouts the bandwidth ratio of the two wireless links to be aggregated cannot be more than 7 (given the packet size used on different links is the same and RTT is dominated by the transmission delay). The authors thus suggest appropriately sizing the packet, employing out-of-order transmission, or increasing the RTO value to improve the scalability. However, such mechanisms still cannot prevent out-of-order delivery from impacting TCP's performance when there are bandwidth fluctuations. Moreover, the authors consider only the characteristics of the wireless links without taking into account the path in the backbone network. Any packet reordering introduced in the backbone network [9] will further limit the performance of such "TCP-unaware" approach.

Link layer techniques

As mentioned in section 1, conventional link layer striping techniques do not perform well in the context of multi-homed mobile hosts, where the multiple interfaces are more likely to belong to different network domains altogether. An ideal striping algorithm not only has to deal with a highly dynamic and vastly differing set of wireless links, but also has to address fluctuations in capacity caused by the end-to-end multi-hop nature of the paths. In [1], the authors propose a "channel" striping algorithm where the channel is defined as a logical FIFO path at any protocol layer. The authors show that the striping (load sharing) algorithm is, in fact, the reverse of the fair queueing algorithm. By reversing the direction of packet flow in the fair queueing algorithm, the sender can achieve optimal load sharing across channels of different capacities. If the receiver is running the same fair queueing algorithm used at the sender and no

[4] A socket is uniquely identified by the (source IP address, source port, destination IP address, destination port) tuple.

packets are lost, in-sequence delivery can be achieved. Although this algorithm is presented as a generic approach for striping over any logical channel including the transport layer, due to the nature of the fair queueing algorithm *the capacity of each channel must be known a priori at both ends*. Moreover, the states of the algorithms at both ends must be synchronized to ensure in-sequence delivery. However, packet losses cause loss of synchronization between the sender and the receiver. Hence the algorithm has to periodically insert "marker" packets into the channels to achieve resynchronization. Nonetheless, in wireless environments with high loss rates, even the marker packets may get lost, resulting in degrading the performance of the striping algorithm and potentially delivering packets out-of-order to the application. Fluctuations in channel capacity further limit the applicability of this approach in the targeted environment.

8. Conclusions

In this paper, we consider the problem of using multiple interfaces on a mobile host to provide aggregate bandwidths to applications. Since the multiple interfaces can potentially and will most likely belong to different wireless network domains, link layer striping schemes cannot be use to achieve bandwidth aggregation. At the same time, we show that application layer techniques using default TCP sockets do not scale well when the link characteristics are different and fluctuating. In this context, we propose a transport layer approach called *pTCP* that achieves bandwidth aggregation using a combination of mechanisms including: (i) decoupled congestion control and reliability, (ii) congestion window based striping, (iii) dynamic reassignment, (iv) redundant striping to handle blackouts, and (v) support for different congestion control schemes to co-exist within a single transport layer framework. We show through simulations that pTCP achieves bandwidth aggregation efficiently under a variety of network conditions.

References

[1] H. Adiseshu, G. Parulkar and G. Varghese, A reliable and scalable striping protocol, in: *Proceedings of ACM SIGCOMM*, Palo Alto, CA (August 1996) pp. 131–141.

[2] M. Allman, H. Kruse and S. Ostermann, An application-level solution to TCP's satellite inefficiencies, in: *Proceedings of Workshop on Satellite-Based Information Services (WOSBIS)*, Rye, NY (November 1996).

[3] H. Balakrishnan, V. Padmanabhan, S. Seshana and R. Katz, A comparison of mechanisms for improving TCP performance over wireless links, IEEE/ACM Transactions on Networking 5(6) (1997) 756–769.

[4] H. Balakrishnan, H. Rahul and S. Seshan, An integrated congestion management architecture for Internet host, in: *Proceedings of ACM SIGCOMM*, Boston, MA (September 1999) pp. 175–187.

[5] H. Balakrishnan, S. Seshan and R. Katz, Improving reliable transport and handoff performance in cellular wireless networks, Wireless Networks 1(4) (1995) 469–481.

[6] J. Duncanson, Inverse multiplexing, IEEE Communications Magazine 3(4) (1994) 34–41.

[7] T. Hacker and B. Athey, The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network, in: *Proceedings of IEEE IPDPS*, Fort Lauderdale, FL (April 2002) pp. 434–443.

[8] T. Henderson and R. Katz, Transport protocols for Internet-compatible satellite networks, IEEE Journal on Selected Areas in Communications 17(2) (1999) 345–359.

[9] M. Laor and L. Gendel, The effect of packet reordering in a backbone link on application throughput, IEEE Network Magazine 5(16) (2002) 28–36.

[10] J. Lee, D. Gunter, B. Tierney, B. Allcock, J. Bester, J. Bresnahan and S. Tuecke, Applied techniques for high bandwidth data transfers across wide area networks, in: *Proceedings of Computers in High Energy Physics (CHEP)*, Beijing, China (September 2001).

[11] L. Magalhaes and R. Kravets, Transport level mechanisms for bandwidth aggregation on mobile hosts, in: *Proceedings of IEEE ICNP*, Riverside, CA (November 2001) pp. 165–171.

[12] M. Mathis, J. Mahdavi, S. Floyd and A. Romanow, TCP selective acknowledgement options, IETF RFC 2018 (October 1996).

[13] J. Nagle, Congestion control in IP/TCP Internetworks, IETF RFC 896 (January 1984).

[14] K. Pahlavan, P. Krishnamurthy, A. Hatami, M. Ylianttila, J.-P. Makela, R. Pichna and J. Vallstrom, Handoff in hybrid mobile data networks, IEEE Personal Communications Magazine 7(2) (2000) 34–47.

[15] D. Phatak and T. Goff, A novel mechanism for data streaming across multiple IP links for improving throughput and reliability in mobile environments, in: *Proceedings of IEEE INFOCOM*, New York (June 2002) pp. 773–781.

[16] D. Rubenstein, J. Kurose and D. Towsley, Detecting shared congestion of flows via end-to-end measurement, in: *Proceedings of ACM SIGMETRICS*, Santa Clara, CA (June 2000) pp. 145–155.

[17] J. Semke, J. Mahdavi and M. Mathis, Automatic TCP buffer tuning, in: *Proceedings of ACM SIGCOMM*, Vancouver, Canada (September 1998) pp. 315–323.

[18] P. Sinha, N. Venkitaraman, R. Sivakumar and V. Bharghavan, WTCP: A reliable transport protocol for wireless wide-area networks, in: *Proceedings of ACM MOBICOM*, Seattle, WA (August 1999) pp. 231–241.

[19] H. Sivakumar, S. Bailey and R. Grossman, PSockets: The case for application-level network striping for data intensive applications using high speed wide area networks, in: *Proceedings of IEEE Supercomputing (SC)*, Dallas, TX (November 2000).

[20] K. Sklower, B. Lloyd, G. McGregor, D. Carr and T. Coradetti, The PPP multilink protocol, IETF RFC 1990 (August 1996).

[21] A. Snoeren, Adaptive inverse multiplexing for wide-area wireless networks, in: *Proceedings of IEEE GLOBECOM*, Rio de Janeiro, Brazil (December 1999) pp. 1665–1672.

[22] M. Stemm and R. Katz, Vertical handoffs in wireless overlay networks, Mobile Networks and Applications 3(4) (1998) 335–350.

[23] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang and V. Paxson, Stream control transmission protocol, IETF RFC 2960 (October 2000).

[24] The Network Simulator, ns-2 (2001) `http://www.isi.edu/nsnam/ns`

[25] C.B. Traw and J. Smith, Striping within the network subsystem, IEEE Network Magazine 9(4) (1995) 22–32.

[26] G.R. Wright and W.R. Stevens, *TCP/IP Illustrated*, Vol. 2 (Addison-Wesley, Reading, MA, 1997).

Hung-Yun Hsieh received the B.S. and M.S. degrees in electrical engineering from the National Taiwan University, Taipei, Taiwan, R.O.C. and the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, USA. He joined the Department of Electrical Engineering and the Graduate Institute of Communication Engineering at the National Taiwan University as an Assistant Professor in August 2004. His research interests include wireless networks, mobile computing, and communications systems.
E-mail: hungyun@ntu.edu.tw

Raghupathy Sivakumar received his Masters and Doctoral degrees in computer science from the University of Illinois at Urbana-Champaign in 1998 and 2000, respectively. He joined the School of Electrical and Computer Engineering at Georgia Institute of Technology as an Assistant Professor in August 2000. His research interests are in wireless network protocols, mobile computing, and network quality of service.
E-mail: siva@ece.gatech.edu