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

Hung-Yun Hsieh and Raghupathy Sivakumar
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA
{hyhsieh, siva}@ece.gatech.edu

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 the 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.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms

Algorithms, Design, Performance

Keywords

Multi-homed mobile host, Bandwidth aggregation, Striping

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 towards addressing this challenge, approaches have been proposed for performing *vertical handoffs* from one network to another as the mobile user migrates across coverage areas [19]. 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 2Mbps indoors (e.g. 3G systems), and a local-area wireless network with an effective data rate of 5Mbps (e.g. IEEE 802.11), 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 7Mbps 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 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, 17]. 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 [22].

Link layer striping schemes assuming stable link characteristics have been proposed earlier [6, 21] for bandwidth aggregation. However, such schemes are unfortunately inapplicable to wireless

*This work was sponsored by the National Science Foundation under award #0117840, and Yamacraw.

MOBICOM'02, September 23–26, 2002, Atlanta, Georgia, USA.

links. Even with an adaptive mechanism that monitors the quality of wireless links and stripes accordingly [18], optimal bandwidth aggregation still cannot 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, pTCP opens 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 and 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 *ns2* [15] 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 true session layer solution provided 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 approach in detail along with the pTCP state diagram, handshakes, and packet header formats. Section 5 provides simulation results of the pTCP approach. 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.

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. 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:

- *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 10Mbps and 2Mbps 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 (10Mbps) 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 (2Mbps) 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

[1]We refer to the individual connections as pipes from hereon to differentiate them from the aggregate end-to-end 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 [22].

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 $\frac{R_2}{R_1} * W$, 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 [22]. 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. We present results in Section 5 that illustrate this phenomenon.

- *Fluctuating Data Rates:*

  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 is 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 were to occur, 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 [16]. 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 overhead at the application layer is 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 worth of data during every write. If nagling is enabled [12], 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, [16] shows that snoop is inappropriate in WWANs due its key assumption that wireless link delays are insignificant when compared to the end-to-end delays. WTCP, proposed in [16] for WWANs, however will stand inappropriate in WLANs due 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 specif-

ically 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 the pTCP approach 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 presentation 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 approach is based on the following five key 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. However, 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 `NO_DATA`. 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 simple 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 $\frac{cwnd_i}{2}$ due to bandwidth fluctuations, the $\frac{cwnd_i}{2}$ 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

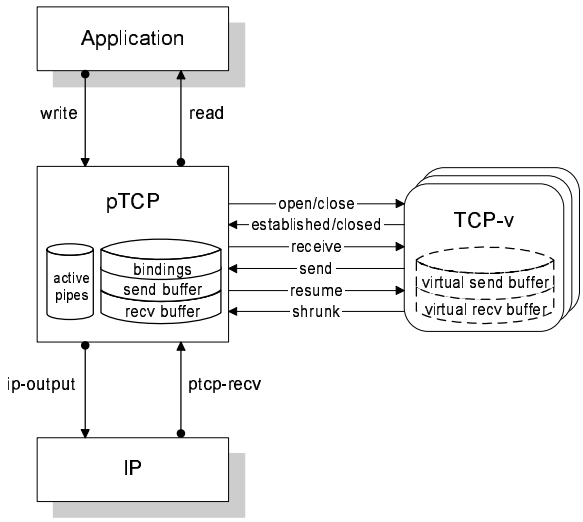


Figure 1: Overview of pTCP and Key Data Structures

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 [11], and more so in wireless environments to enable faster recovery from random channel errors [16], we believe that the recommendation for the use of SACK in pTCP is a reasonable one.

4. THE PTCP PROTOCOL

4.1 Overview

Figure 1 provides an architectural overview of the pTCP approach. 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 `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`, appends 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 `NO_DATA` 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 `NO_DATA` 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 an incoming data segment is received by pTCP, 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.

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

4.2 TCP-v Interface

As seen in Figure 1 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 [22]. The $send()$ call is used by TCP-v to send "virtual" segments to pTCP which will then 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 `NO_DATA` 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.

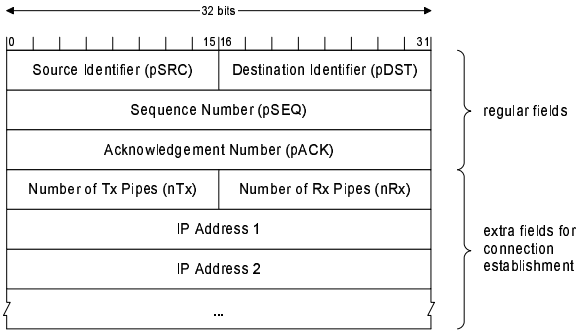


Figure 2: pTCP Header Format

4.3 Header Formats

Figure 2 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 field. Since 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), they cannot be reused by pTCP. Because pTCP is responsible for performing flow control (given that it controls the buffer), it requires 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 Figure 3 and Figure 4 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.

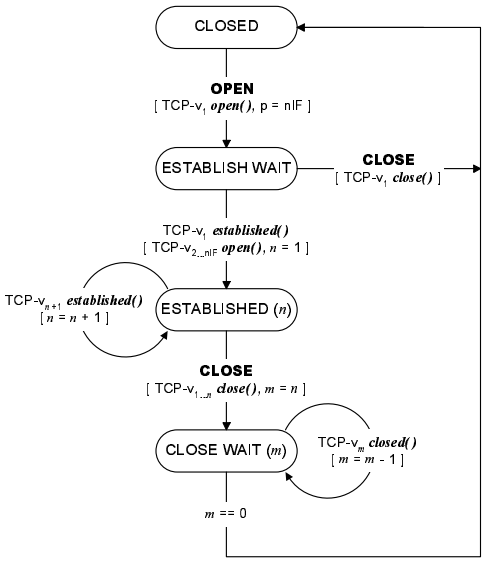


Figure 3: pTCP State Machine

- *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 [22]. 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 $ipRx$.

  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 SYNs 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

[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.

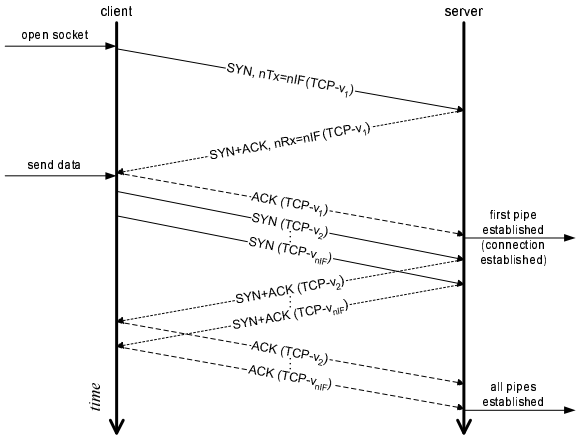


Figure 4: Connection Establishment Handshake

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 pipes, it creates the corresponding TCP-v TCBs and assigns the respective SYNs 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 3. 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 moves down the pTCP state machine. Upon successful completion of all TCP-v pipes, pTCP enters the **CLOSED** state of Figure 3 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 simple application layer approach, irrespective of the bandwidth-delay product (BDP) of the individual TCP pipes, each pipe would have a constant buffer (of 64KB 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 [14].

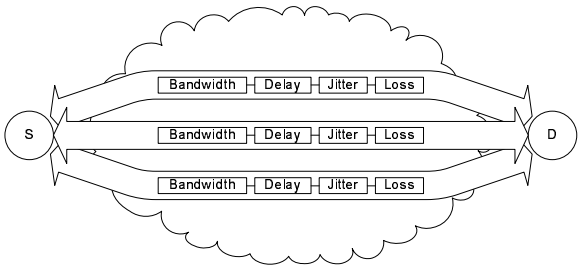


Figure 5: Network Topology

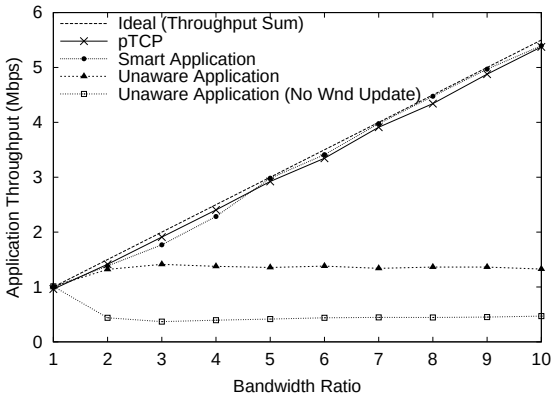
The buffer space (both send and receive) available at the pTCP layer is assumed to be $n * B$, where n is the number of TCP-v pipes, and B is the default TCP buffer size. Every segment that belongs to a pTCP connection always carries the available space in the pTCP receive buffer, irrespective of which 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 `NO_DATA` value returned for their $send()$ calls since pTCP would be aware of the global situation.

We have thus far described the key components of the pTCP protocol. In the next section, we present performance evaluation results for the pTCP protocol comparing it with the performance of simple and sophisticated application layer techniques.

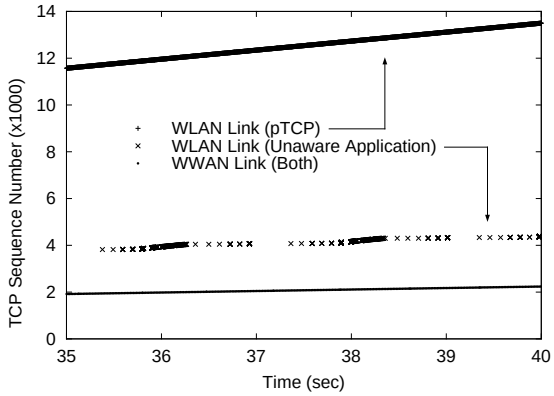
5. PERFORMANCE EVALUATION

5.1 Simulation Model

We use the *ns2* [15] network simulator and the generic topology shown in Figure 5 for all our simulations. We emulate the end-to-end path from the multi-homed mobile host (S) to the destination (D) through our custom bandwidth, delay, jitter, and loss modules added to the link object in *ns2*. Since we assume the bottleneck to be in the wireless link, we do not explicitly use a sophisticated backbone topology in the simulations. However, we introduce variations in the delay and jitter modules to capture the variations that connections would be expect to observe. We use three primary types of links in most of our simulations: (i) links with bandwidths ranging from 500Kbps to 5Mbps and a round-trip time of 100ms - representative of a connection through a WLAN, (ii) links with a bandwidth of 2Mbps and a round-trip time of 400ms - representative of a connection through a WWAN pico-cell, and (iii) links with a bandwidth of 500Kbps and a round-trip time of 400ms - representative of a connection through a WWAN macro-cell. Packet loss rates from 0.001% to 1% are used in the simulations, and we identify the specific link characteristics used in simulations as we present the different results. We use the TCP-SACK implementation for the transport layer by default. We use two application layer techniques in our comparisons: an "unaware" approach identical to



(a) Throughput vs. Bandwidth Ratio



(b) Sequence Number Progression (Ratio = 5)

Figure 6: Scalability with Rate Differential

the one described in Section 2, and a "smart" approach that uses a striping ratio based on the average bandwidths of the different links. pTCP is implemented as a wrapper around TCP-SACK as explained in Section 4. We also present the "ideal" performance of bandwidth aggregation by using multiple applications (one application and one socket for each link) and summing the respective throughputs. We explicitly introduce CBR traffic over UDP as the background traffic when necessary. All packet sizes are set to 1KB. Simulations are run for a period of 600 seconds, and are averaged over 10 samples when randomness is introduced.

We measure throughput (both aggregate and instantaneous) as the metric in our comparisons. We also present the TCP sequence number progression of a connection where appropriate. We present the following results in the rest of the section: (i) scalability with respect to rate differential, (ii) scalability with respect to the number of interfaces, (iii) resilience to rate fluctuations, (iv) resilience to blackouts, and (v) co-existence of different congestion control schemes.

5.2 Simulation Results

5.2.1 Rate Differential

In this section, we use the topology in Figure 5 with two active links between the source and the destination. We fix the bandwidth of one of the links to 500Kbps, and increase the bandwidth of the other link from 500Kbps to 5Mbps in increments of 500Kbps. The round-trip time is fixed as 400ms for the first link, and 100ms for

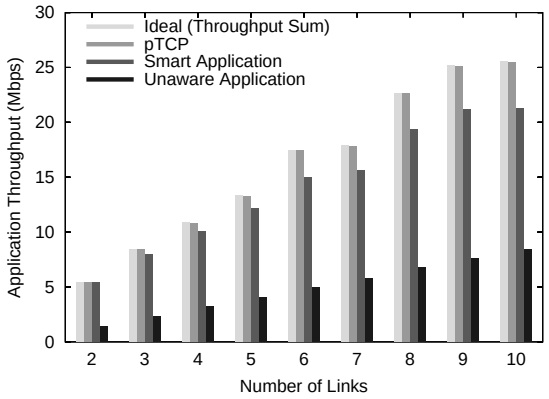


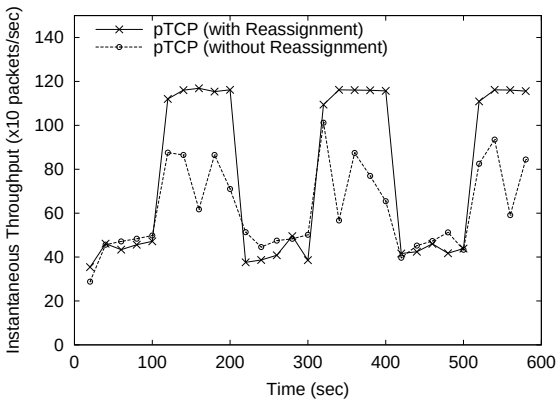
Figure 7: Scalability with Multiple Links

the second. While the first link is representative of a WWAN link, the second link is representative of a WLAN link. For each value of the bandwidth for the second link, we monitor the throughput performance of pTCP, smart application, and unaware application. For reference, we also plot the ideal aggregate throughput curve. We present the aggregate throughput results in Figure 6(a). The x-axis value represents the ratio of the bandwidth of the second link to that of the first. It can be observed that while pTCP and the smart application achieve near ideal performance, the unaware approach performs significantly worse and exhibits a non-increasing aggregate throughput beyond a ratio of 2. We also simulate the scenario when the window update from the TCP receiver is lost. The sending TCP of the faster pipe thus enters persist mode which in turns causes the slower pipe to do the same. As shown in the figure, the result is that the aggregate throughput is lower than that of the slowest pipe as explained in Section 2.

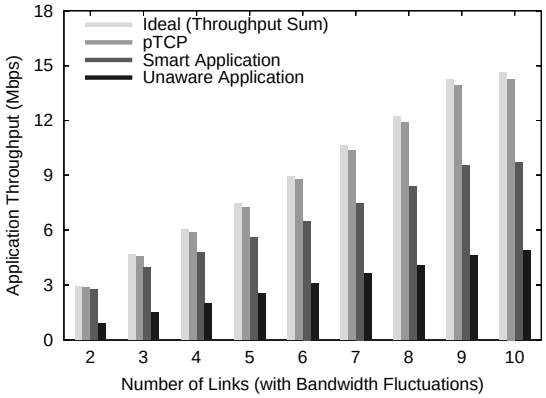
The non-performance of the unaware application, while explained in Section 2, is further illustrated by the results presented in Figure 6(b), where the sequence number progression is shown during a small time window for two pipes with bandwidth ratio of 5. For reference, we also present the case for pTCP. It can be observed that in the unaware application, the head-of-line blocking at the receiver due to the slower pipe stalls the faster pipe. The faster pipe in Figure 6(b) thus exhibits distinct idle periods (e.g. 36.5s to 38s), that results in the degraded performance of the unaware application. In contrast, the results for pTCP for the same scenario exhibits a smooth flow of transmissions through the faster pipe. This is attributed to the congestion window based assignment strategy adopted by pTCP.

5.2.2 Number of Links

For results in this section, we use the topology in Figure 5, and consider the performance of pTCP and the other approaches as the number of links is increased from 2 to 10. For a scenario with i links, we fix the bandwidth of the first link to 500Kbps, the bandwidth of the second link to 5Mbps, and the bandwidths of the remaining links to values between 500Kbps and 5Mbps randomly chosen. Figure 7 presents the aggregate throughput enjoyed by the application using different striping techniques as the number of links is increased. Since randomness is introduced in the form of the variable link bandwidth assignment, we use averages over 10 scenarios for this result. It can be observed that the performance of pTCP scales well with increasing number of links, while the performance of the unaware application does not. Note that the throughput degradation observed in the smart application as the number



(a) Effect of Dynamic Reassignment



(b) Throughput vs. Number of Links

Figure 8: Impact of Fluctuations

of links increases is in fact due to the particular implementation used. Since the smart application always stripes data across multiple pipes based on the bandwidth ratio of the underlying links (the minimum striping unit is one packet), any short-term unfairness exhibited in individual pipes will decrease the effectiveness of the algorithm. As the number of links increases, such short-term unfairness occur more frequently (the periodic TCP probing losses on each link contribute to the unfairness), thus degrading the aggregate throughput enjoyed by the smart application. While we acknowledge that a more sophisticated implementation of the smart application could conceivably be used to solve the problem, its performance will degrade further due to other reasons as shown in the next sections.

5.2.3 Rate Fluctuations

Since the characteristics of wireless links exhibit high variances in bandwidths and delays, in this section, we investigate the performance of pTCP in the presence of fluctuations in the available capacity of individual pipes. We show that pTCP's dynamic reassignment design element is effective in addressing the capacity fluctuations, and compare the performance of pTCP with other approaches.

We first consider the topology in Figure 5 with two links. One of the links is fixed at a bandwidth of 5Mbps and a round-trip time of 100ms, while the bandwidth of the other link is initially set to 2Mbps and a round-trip time of 400ms. A *square-wave* CBR flow with a period of t and an amplitude of 1.5Mbps is used as background traffic, to cause the fluctuation on the second link. Hence, the available bandwidth for the second link fluctuates between 0.5Mbps and 2Mbps, with a period of t seconds. While the fluctuations are caused by varying the CBR background load on the appropriate links, they are representative of both congestion based rate changes, and wireless channel condition based rate changes. The result shown in Figure 8(a) is obtained using $t = 200$ seconds. We use the number of packets delivered to the application (over a 20-second time-window) as the instantaneous throughput and compare the performance of pTCP with and without dynamic reassignment. It is clear from the figure that with dynamic reassignment pTCP is able to perform well under bandwidth fluctuations.

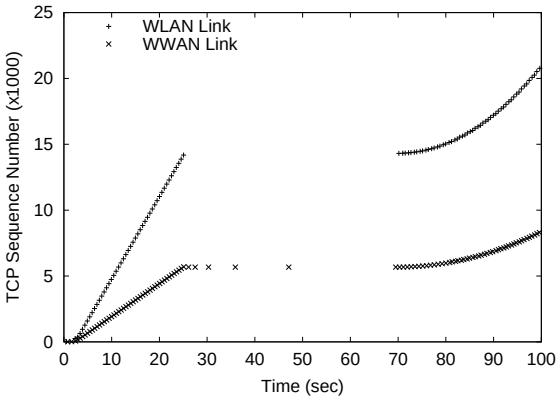
In Figure 8(b) we compare the performance of different striping techniques using the multi-link topology used in Section *5.2.2*. However, the bandwidth of each link now randomly fluctuates (every 1 second) between 20% and 100% of the normal value. For example, for a link with capacity of 5Mbps, the available bandwidth that the application can use randomly fluctuates between 1Mbps and 5Mbps throughout the simulation. The average bandwidth is thus 60% of the link capacity. We observe 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. The scenario is appropriate for demonstrating the inefficacy of even a smart application layer approach that does some rate estimation. Its performance will suffer as long as its rate estimation is coarser than the actual fluctuations. Although the smart application can eventually adapt to the changed rate, it cannot do anything about packets already in the pipe that has now slowed down.

5.2.4 Blackouts

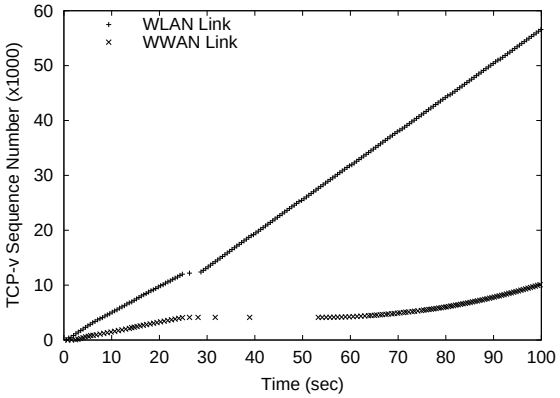
In this section, we show the impact of blackouts on the performance of pTCP and application striping techniques. We use the same two link topology used in the previous sections, but introduce a long (25 seconds) blackout between 25s and 50s when the available bandwidth on the WWAN link (2Mbps) decreases to zero. All packets transmitted through the WWAN link during the blackout period are dropped. Figure 9(a) shows the sequence number progression for the smart application on both the WLAN and WWAN pipes. While it is obvious that the WWAN link stops sending data during the blackout period, because of the head-of-line blocking problem described in Section 2, even the WLAN link stalls for most of the blackout period, resulting in the aggregate connection coming to a standstill. On the other hand, as seen in Figure 9(b), in pTCP, although the WWAN link stalls during the blackout period, the WLAN link 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 WWAN pipe to the WLAN pipe and prevents the latter from experiencing head-of-line blocking.

5.2.5 Different Congestion Control Schemes

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 two link topology again with bandwidths of 5Mbps (WLAN) and 2Mbps (WWAN), and round-trip times of 100ms and 400ms respectively. A loss module is inserted on the WWAN link. The module inserts losses at packet error rates ranging from 0.001% to 1%. We consider the performance of pTCP when the WLAN link uses regular TCP, and the WWAN link uses TCP-ELN [3]. TCP-ELN receives explicit loss notification from the underlying link layer when a packet is dropped due to random wireless loss, and does not react to such losses. Note



(a) Smart Application during Blackouts



(b) pTCP during Blackouts

Figure 9: Impact of Blackouts

that it is not the sophistication of the transport protocol used that is of key importance, but the ability of the pTCP approach to accommodate two different congestion control schemes within the same framework. Figure 10 presents the throughput performance results for pTCP, the smart application using either TCP or TCP-ELN on both links, and the ideal performance (sum of the throughputs of an independent TCP-ELN connection over the WWAN link and an independent TCP connection over the WLAN link). It can be seen that pTCP achieves almost the maximum achievable performance, illustrating the seamless nature in which pTCP allows the two congestion control schemes to co-exist.

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 ACKs 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 ACKs 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.

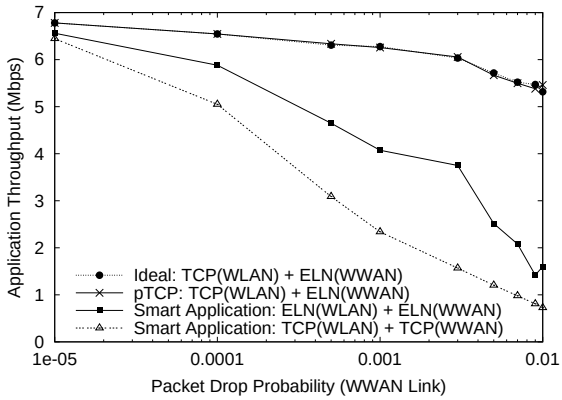


Figure 10: Multiple Congestion Control Schemes in pTCP

- *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 [13]. 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 reinsertion 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, 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 [17] 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 [9] 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 [20]. 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 [10]. 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.

- *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 dy-

namic 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 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 window 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.

9. REFERENCES

[1] H. Adiseshu, G. Parulkar, and G. Varghese. A reliable and scalable striping protocol. In *Proceedings of ACM SIGCOMM*, Palo Alto, CA USA, Aug. 1996.

[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 USA, Nov. 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):756–769, Dec. 1997.

[4] H. Balakrishnan, H. Rahul, and S. Seshan. An integrated congestion management architecture for Internet host. In *Proceedings of ACM SIGCOMM*, Boston, MA USA, Sept. 1999.

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

[6] J. Duncanson. Inverse multiplexing. *IEEE Communications Magazine*, 32(4):34–41, Apr. 1994.

[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 USA, Apr. 2002.

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

[9] 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, Sept. 2001.

[10] L. Magalhaes and R. Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. In *Proceedings of IEEE ICNP*, Riverside, CA USA, Nov. 2001.

[11] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options. IETF RFC 2018, Oct. 1996.

[12] J. Nagle. Congestion control in IP/TCP internetworks. In *IETF RFC 896*, Jan. 1984.

[13] 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 USA, June 2000.

[14] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP buffer tuning. In *Proceedings of ACM SIGCOMM*, Vancouver, Canada, Sept. 1998.

[15] The Network Simulator. *ns-2*. http://www.isi.edu/nsnam/ns.

[16] 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 USA, Aug. 1999.

[17] 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 USA, Nov. 2000.

[18] A. Snoeren. Adaptive inverse multiplexing for wide-area wireless networks. In *Proceedings of IEEE GLOBECOM*, Rio de Janeireo, Brazil, Dec. 1999.

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

[20] R. Stewart et al. Stream control transmission protocol. IETF RFC 2960, Oct. 2000.

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

[22] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume 2*. Addison-Wesley Publishing Company, Reading, MA USA, Oct. 1997.