Achieving Efficient Parallelism in Transport Layer using P-TCP

Kamalakshi N[1], H Naganna[2]

Dept of Computer Science & Engg Sapthagiri College of Engg Bangalore, India

[1] kamalnags@yahoo.co.in [2]naganna_h@hotmail.com

***Abstract*— Existing transport layer protocols cannot seamlessly operate over multiple pipes. This paper proposes parallelism as the first fundamental principle that needs to be incorporated in transport layer design for supporting transparent host mobility. In particular, we discuss how TCP, the prevailing transport layer protocol designed for a single path, can be extended to be a parallel protocol called pTCP (parallel TCP) that can effectively operate over multiple paths. We first discuss the key tenets for designing pTCP the parallel protocol, and then present the detailed protocol operations including the state diagram and protocol handshake for such a parallel protocol**

Keywords*— congestion control*,* TCP-v, *p*TCP, flow control, *TCP- friendly

I. Introduction

TCP is designed for operating over a single path between the source and the destination, it captures the characteristics of the path it traverses such as bandwidth and latency in the form of TCB (Transmission Control Block) state variables [86]. TCB includes variables such as congestion window and round-trip time for TCP to determine the available data rate of the path. Since multiple pipes in a connection can exhibit a very high degree of heterogeneity in terms of the data rates, round-trip times, and loss rates, maintaining only one set of TCB variables (single state) can render the achieved throughput suboptimal. Therefore, the first element in designing a parallel transport protocol for operating over multiple pipes is to maintain multiple states in accordance with the number of pipes used in the connection. In the context of parallel TCP, multi-state design allows it to maintain one TCB state for each pipe that becomes active in the connection. Since each state is associated with only one pipe, the single-state design of TCP can be reused with minimal changes – without suffering from performance degradation

II. P-TCP

A. *Decoupling of Functionalities*

Application layer striping approaches suffer from considerable complexities and overheads due to repetitive implementations of functionalities across multiple sockets. Therefore, to incur minimum overheads resulting from the multi-state design, pTCP should decouple the transport layer functionalities associated with per-pipe characteristics from those pertaining to the aggregate connection. Toward this end, pTCP is designed as a *wrapper* around a slightly modified TCP that we refer to as *TCP-v* (TCP-virtual). The TCP-v opened for each pipe handles the per-pipe state (TCB), while the pTCP engine (which for simplicity we also refer to as pTCP) handles the aggregate connection. Specifically, 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.

B. *Delay Binding*

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 (BDP) of the pipe. While the TCP

congestion window *is* an approximation of the BDP, it is possible that it is an incorrect estimation (say, for example, due to deep buffers in the network). 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*.

C. Dynamic Reassignment

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 *pi* is *cwndi*. If *cwndi* worth of data is assigned to *pi*, and the window is cut down to *cwndi/2* due to bandwidth fluctuations, the *cwndi/2* worth of data that falls outside the congestion window of *pi* will be blocked from transmission till the *cwndi* 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 acknowledgments 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 can greatly improve the performance of pTCP under dynamic conditions.

D. Redundant Striping

While the strategy described above reassigns data that falls out of a pipe's congestion window,it does not deal with the one MSS (maximum segment size) 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.

III. PROTOCOL OPERATIONS

We now present the detailed operations of pTCP, including its software architecture, state diagram, protocol handshake, and protocol operations.

A. Architectural 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. pTCP creates and maintains one TCPv for each active pipe used by the application. 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`

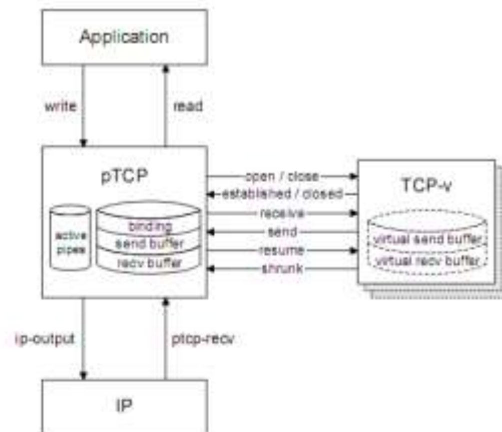

Fig.1 P-R2CP

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 `binding`, 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. 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

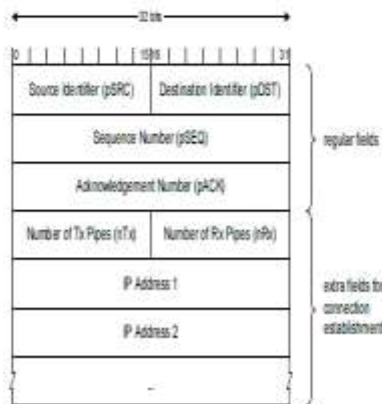

Fig.2 pTCP Header Format

the pTCP header and the data, enqueues the data in the `recv buffer`, and provides the appropriate TCP-v with only the skeletal 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.

B. 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 [2]. 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

C. 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 acknowledgment 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 acknowledgment similar to the TCP acknowledgment (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 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 .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 pipes to be desirably used (*nTx*), (ii) number of receiving pipes 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 pipes the source would ideally like to use for its transmissions (which in effect will require *nTx* pipes to be maintained at the receiving ends), and the *nRx* field is the maximum number of pipes on which the source is willing to serve the reverse path. Note that if

nTx or *nRx* equals one, the pTCP connection falls back to a regular TCP connection with single state.

D. Connection Management

We use the state machine of pTCP and the connection establishment handshake presented in Figure 10 and Figure 11 respectively for the following discussions on connection management. Note that the state machine for TCP-v is the same as that of default TCP, and the interface between pTCP and TCP- 2.

We assume the number of pipes 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

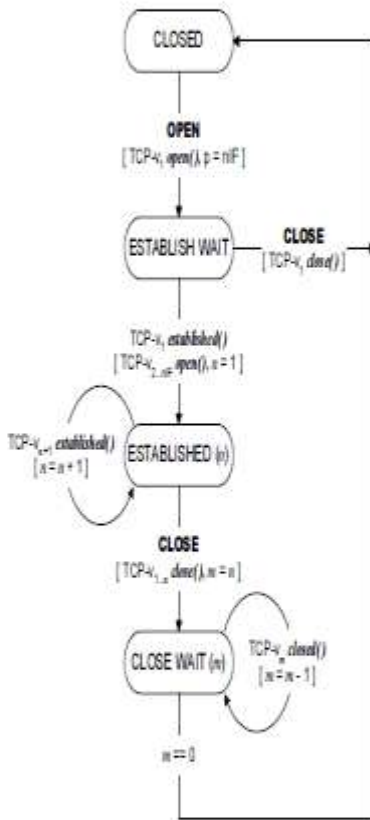

Fig 3 pTCP State Machine

earlier, 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 can support *nIF* TCP-v pipes.1 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 [2]. 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 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

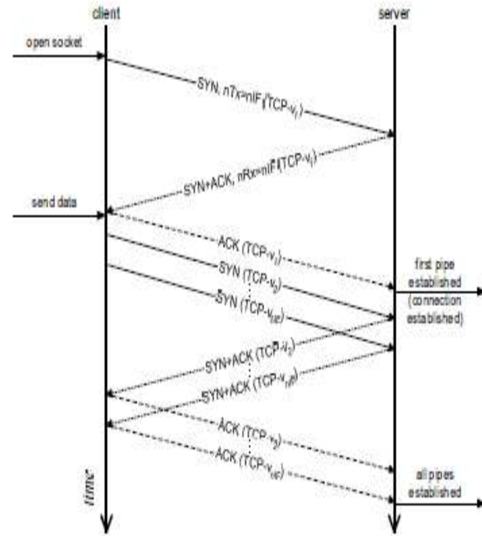

Fig 4: pTCP Connection Establishment Handshake

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 TCPv 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 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 10. Finally, when all the individual pipes enter the **ESTABLISHED** state, pTCP enters the **ESTABLISHED (nIF)** state.

1) *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 10 and confirms the close to the application layer.

E. 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 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

We assume the buffer space (both send buffer and receive buffer) available at the pTCP layer is equal to *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 *freeze* value returned for their *send*() calls since pTCP would be aware of the global situation.

F. Reliability

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. We now discuss how pTCP can still ensure reliability under these two conditions:

1) Dynamic Reassignment

Whenever pTCP performs dynamic reassignment for a particular TCP-v pipe, it unbinds a 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. When the old pipe "retransmits" the previously bound virtual segment, it will be reassigned a different data segment.

2) 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.

IV. CONCLUSION

This paper discusses how TCP, the prevailing transport layer protocol designed for a single path, can be extended to be a parallel protocol called pTCP (parallel TCP) that can effectively operate over multiple paths.

REFERENCES

[1] J. Postel, "Transmission control protocol," IETF RFC 793, Sept. 1981

[2] J. Nagle,"Congestion control in IP/TCP Internetworks," IETF RFC 896, Jan. 1984

[3]] D. Bansal and H. Balakrishnan, "Binomial Congestion Control

[4] V. Tsaoussidis and C. Zhang, "TCP-Real: Receiver-oriented congestion control," Computer Networks, vol. 40, no. 4, pp. 477–497, Nov. 2002

[5] A. S. Tanenbaum, Computer Networks. Upper Saddle River, NJ, USA: Prentice Hall, 1996