Increasing Performances of TCP Data Transfers Through Multiple Parallel Connections

Increasing Performances of TCP Data Transfers Through Multiple Parallel Connections

Andrea Baldini, Lorenzo De Carli, Fulvio Risso

Abstract— **Although Transmission Control Protocol (TCP) is a widely deployed and successful protocol, it shows some limitations in present-day environments. In particular, it is unable to exploit multiple (physical or logical) paths between two hosts. This paper presents PATTHEL, a session-layer solution designed for parallelizing stream data transfers. Parallelization is achieved by striping the data flow among multiple TCP channels. This solution does not require invasive changes to the networking stack and can be implemented entirely in user space. Moreover, it is flexible enough to suit several scenarios – e.g. it can be used to split a data transfer among multiple relays within a peer-to-peer overlay network.**

I. Introduction

Transmission Control Protocol (TCP) is the de-facto standard for stream-oriented communication over the Internet. It provides reliable and in-order delivery of a data stream, together with flow and congestion control. One of its characteristics, namely the capability to open only a single logical channel between two communicating host, appears to be a limitation in the present-day Internet. The maximum theoretical speed of a single TCP connection is bounded by the window size and the Round Trip Time, which often prevent the full utilization of high-speed links (unless in presence of specific settings). To overcome the problem, several solutions – such as XFTP ([1]) – open different parallel TCP connections in order to achieve higher data rates. However, synchronization and management of these concurrent channels has to be done by the application itself.

Another example comes from multi-homed hosts, which can only use a single network interface when exchanging data with a remote peer – in fact, TCP is not able to transparently use two different physical paths as a single logical channel. The availability of multiple (and concurrent) network connections in the same place (e.g. wireless Internet access, GPRS/UMTS, and more) may become quite common in the near future, thanks to the widespread diffusion of wireless Internet access. New technologies like Microsoft VirtualWiFi ([2]), whose aim is to allow a host to connect to multiple WiFi networks using a single interface, may also play an important role. These technologies could enable the concurrent use of different WiFi domestic networks, allowing the aggregation of the upstream bandwidth of different ADSL connections.

A third example relates to relay-based data transfers within a peer-to-peer network. When two P2P nodes cannot exchange data directly, communication is established through a relay, i.e. a machine that is reachable by both hosts and acts as a "pass-through" for the data flow. This practice is costly for relaying nodes, which use a portion of their bandwidth to transfer data they are not interested in. Consequently, communication through a relay usually uses a limited amount of bandwidth and is slow. Applying parallelization in this context would allow splitting a transfer among several relays, improving the transfer rate without putting excessive load on a single node.

In this paper, we present a generic technique to split a stream data transfer among multiple TCP connections. The rest of the paper is organized as follow: in Section II we present existing work in the field of parallelization of network transfers. Section III describes a new technique, called PATTHEL (*Parallel TCP Transfers Helper*), and discusses its most important features. Section IV presents experimental results and Section V gives some conclusive remarks.

II. Related Work

Several techniques for striping a data transfers over multiple paths – logical or physical – have been proposed in literature. The potential and the limitations of each solution are closely related to the ISO/OSI layer to which it belongs.

Layer-II techniques allow the concurrent use of multiple layer-II links. A well-known example is link aggregation ([3]), which works between a couple of nodes connected through multiple Ethernet link – for each link, a dedicated Ethernet card must be installed on both hosts. By striping data over the available interfaces, a channel is obtained, whose bandwidth is equal to the sum of the bandwidth of individual links. The drawback of this and other similar techniques is that they are strictly tied to a specific data-link technology. Other solutions are implemented at higher layers, to achieve a greater level of abstraction over the hardware.

Mobile IP ([4]) is a set of extensions to the basic IP protocol that allows a node to change its address while it is sending and receiving data. During the transition of a mobile host from a network to another, two paths can be used together to receive packets both from the old network and the new one. However, Mobile IP does not use multiple links in parallel to achieve higher throughput. A layer-III technique for streaming data across multiple IP links is presented in [5]. This technique has severe compatibility issues with TCP which may significantly

Fulvio Risso (corresponding author) is with Politecnico di Torino 10039, Italy (e-mail: fulvio.risso@polito.it, phone +39-011-090.7008).

Lorenzo De Carli is with the Department of Computer Science, University of Wisconsin-Madison, Madison, WI, USA (e-mail: lorenzo@cs.wisc.edu).

Andrea Baldini is with Cisco Systems, Inc., San Jose, CA, USA (e-mail: abaldini@cisco.com).

degrade aggregated performances. In fact, TCP flow control is optimized to work over a single path, and does not perform well when a connection is split across multiple links. The problem is likely to affect each layer-III striping solution that wants to retain compatibility with TCP. Therefore, it is unlikely that layer-III multipath solutions can achieve widespread adoption.

Other solutions avoid this issue by implementing multipath at layer IV. SCTP ([6]) is a transport protocol with support for multihoming and multipath. Its main limitation is that data transfers always use only one path, even if multiple ones are available. The remaining paths are only used for retransmitting packets when the main path fails. Therefore, SCTP supports multipath only for reliability purposes. Several recent papers (e.g. [7]) introduce SCTP variants implementing load sharing. A drawback of these approaches, as of SCTP in general, is that the protocol is relatively new and not widely used; spreading it would require a massive porting of current Internet applications. Moreover, compatibility problems with existing network devices – e.g. firewalls, NATs, etc. – are likely to arise. Other layer-IV solutions aim at adding load-sharing support to TCP. The study of TCP-like congestion control for multipath data transfers has been addressed in [8]. However, results are not easily applicable, as they require source routing and the static configuration of paths on routers. pTCP ([9]) is a derivative of TCP that supports striping and load balancing. The most serious drawback of pTCP is that it is not compatible with TCP, as it uses a modified version of the TCP header.

Layer-V and VII solutions implement multipath without affecting widely deployed protocols such as IP and TCP: in many situations, this is a decisive advantage. Munisocket ([10]) is a layer-V multipath solution that achieves parallelism by creating several TCP connections and striping data blocks on them. It targets nodes on the same LAN, typically in a computing grid; hence, its use is limited to large message transfers over homogeneous, high-speed local links. SEBAG ([11]) use a similar principle, but it is specialized for the case of mobile hosts with multiple radio interfaces. Moreover, the paper lacks details on SEBAG inner working, and on how it is integrated with applications.

Most of the presented solutions ([3], [5], [7], [9], [10], [11]) share a serious limitation: they are specifically designed for situations in which multiple network interfaces are available. On at least one side of the communication, each data pipe must end on a different network card. Such design cannot be adapted to the case of an overlay network, where transfers can be split among paths that terminate at the same endpoints.

Other techniques involve the creation of multiple logical paths over a single physical link. [12] aims at improving the performances of multimedia streaming over TCP. To mitigate the impact of packet losses on the throughput of a single transfer, media flows are striped among multiple TCP connections. [1] and [13] use a similar approach to overcome bandwidth limitations on links with high *bandwidth * delay* product. In particular, XFTP ([1]) consists of a modified version of the FTP protocol, with support for the creation of multiple sockets on a single link. The main problem of layer-VII solutions like XFTP is that they are usually developed in the context of a specific application; reusability is not trivial if possible at all. Also, it is important to point out that [1], [12] and [13] cannot exploit multiple physical paths, as their purpose is only to improve TCP performances on a single link.

The aim of our solution is to implement parallel transfer capabilities on top of the TCP protocol, therefore limiting the changes in both the network stack and in user applications, while introducing the idea of "logical channels". A logical channel is a data pipe that can either use a link exclusively, or share it with other logical channels; the way in which the available links are shared among logical channels depends on application needs. This concept enables both the parallelization of transfers through different physical paths (either in case of different endpoints, or through different intermediate relays), and the creation of multiple TCP pipes on the same link.

III. The Parallel TCP Transfer Helper

The Parallel TCP Transfers Helper (PATTHEL) described in this paper is a layer-V architecture designed to stripe a TCP data flow over multiple (physical or logical) channels. This solution has the advantage of being relatively simple to implement because it relies on TCP for the physical data transfer. Hence, it does not require the definition and the implementation of a new transport protocol, and it does not have a dramatic impact on the operating system. At the same time, it is extremely effective because the protocol can be easily leveraged by all the applications that currently use TCP.

PATTHEL main strengths are the capability to establish multiple communication channels transparently, a clever (and simple) scheduling algorithm for striping data over different channels, and a receiving module that limits the amount of memory copy operations in the receiver, avoiding – in many cases – the need for an intermediate receiver buffer. PATTHEL does not implement the (orthogonal) task of determining the set of available paths between two hosts, which can be delegated to mechanisms such as ALEX ([14]).

A. System architecture

Figure 1 depicts a high-level overview of the proposed architecture. Applications see a single input socket and a single output socket, but PATTHEL introduces more elements. Just after the input socket the data stream is split in chunks that are then distributed to a set of physical TCP channels. In a complementary way, transferred data are reassembled just before the output socket, in order to pass them to the application as a single stream. In addition, PATTHEL introduces a new TCP connection for controlling the transfer.

The main challenges faced by PATTHEL are (i) how to spread data across different channels, (ii) how to assign channels to the active interfaces, and (iii) how to compute the optimal number of channels. If PATTHEL is used between multihomed hosts, the number of channels can coincide with the number of network interfaces. However, PATTHEL can also be used to create multiple channels on the same path. In this case, the number of channels that maximizes throughput must be dynamically computed ([1], [13]). In this paper we concentrate on tasks (i) and (ii), and leave (iii) as future work.

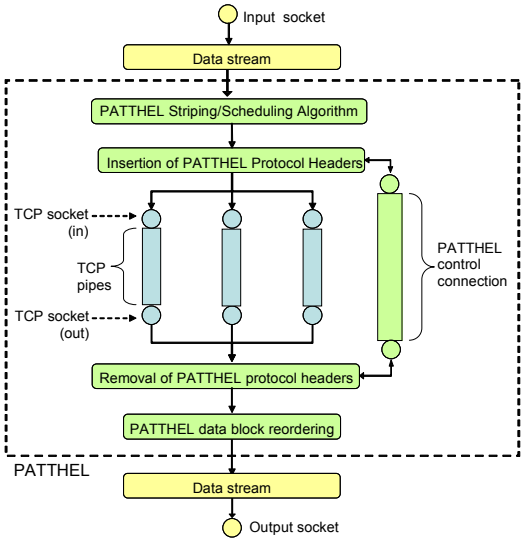

Figure 1. PATTHEL general architecture.

Chunks created by the scheduling algorithm are sent through a set of TCP channels, with the addition of a PATTHEL header to each chunk. The PATTHEL network protocol (presented in section III.D) defines the format of the chunks. It also specifies how to establish an initial channel, how to open additional channels and how to close the communication. At the destination, chunks are reordered using the information contained in the PATTHEL headers, restoring the original data stream.

B. *The PATTHEL Scheduler*

The PATTHEL scheduling algorithm receives a sequence of application-generated data blocks and outputs a set of chunks, each one assigned to a data channel. The size of chunks and the scheduling policy determine the efficiency of the algorithm in using the available bandwidth. Such efficiency is maximized if all channels are backlogged during the entire transfer. If D is the size (in bytes) of a data block passed to the scheduler at a given time, F_i is the fraction of the block assigned to the *i-th* channel (in bytes), and B_i is the bandwidth of the channel (in bytes/s), the channel will be busy transmitting data for F_i / B_i seconds. If the transmission on all channels starts at the same time, the maximum throughput is achieved when the following equalities hold:

$$\frac{F_1}{B_1} = \frac{F_2}{B_2} = \ldots = \frac{F_N}{B_N}$$

The fraction F_i of data assigned to the *i-th* channel can be written as $F_i = f_i * D$, where f_i is a coefficient between 0 and 1, $\sum f_i = 1$ and D is the total size of the data block. The optimal fraction of the block associated to the *i-th* channel is given by:

$$f_i = \frac{B_i}{\sum_{l=1}^{N} B_l} = \frac{B_i}{B_{total}} \qquad (1)$$

(see [5] for proof). Note that B_i is the bandwidth exhibited by a TCP channel, and not by a physical path. It depends on the bandwidths of the links through which the packets travel, but also on parameters such as TCP window size, RTT, and losses.

Equation 1 could be used directly to compute the coefficients f_i, but this approach has significant limitations. Estimating the bandwidth[1] of a TCP channel is difficult for a Layer-V solution such as PATTHEL, which cannot access information inside the TCP/IP driver. However, limited information can be inferred through the *select()* system call. In fact, *select()* can be used to determine whether the channel is full or it can accept new data.This information does not exactly represent the status of the channel, because it is influenced by the availability of space in driver and OS buffers, and it suffers from low temporal granularity. Despite these limitations, results show that its precision is enough for our application.

The PATTHEL scheduler is, in principle, quite simple: each data block generated by the application is split into chunks, whose size is configurable – the default behavior is to set the size of a chunk to the MTU of the interface on which the chunk is sent. Chunks are then assigned to the available channels in the following way: when a channel has free space in the send buffer, the PATTHEL subsystem is notified (through a *select()* call) and a new chunk is assigned to that channel. The process is repeated as soon as a new data block comes from the application: all the channels are always kept busy by never letting their input buffers become empty.

Consider a block that must be sent over N parallel channels. B_i is the rate (in bytes/s) at which the *i-th* channel can send data and B_{total} is the aggregate bandwidth (i.e. the sum of the bandwidth of the individual channels). As long as all the input queues are backlogged, each channel is always busy transmitting and is hence fully utilized. In this situation, every second the *i-th* channel sends B_i bytes of data. In the same time interval, the system sends a total of B_{total} bytes. Therefore, the fraction of data assigned to the *i-th* channel is, on average, the one given by Equation 1, although the scheduler is based on a different principle.

C. *The PATTHEL receiver*

The scheduling algorithm just described is simple and effective, and it has a low impact on CPU and resource usage. The pitfall is that it cannot guarantee that the chunks will arrive at the receiver in they same order in which they were sent. In fact, it does not take into account disparities between the RTTs of the channels, and delays caused by the operating system internal buffers. The problem is addressed on the receiver side, by a mechanism which is able to deal with out-of-order arrivals without introducing additional buffering.

To ease the receiver's task, the sender adds a header to each chunk. The header contains the index and the size of the block from which the chunk came, the size of the chunk, and the offset of the chunk from the beginning of the stream. By using this information, the PATTHEL receiver can place data from the TCP channels directly within the application buffer, without any further overhead due to copy operation.

The receiver accepts only chunks belonging to the block that is currently being received. For example, if a fragment belonging to the block N arrives on a channel while block $N-1$ is still being received on the other channels, PATTHEL will not read data from that channel until block $N-1$ has been

[1] Some scheduling algorithms, such as the one described in [15], also need an estimation of the RTT to minimize the out-of-order arrivals at the receiver. We deal with reordering in a different way, described in section III.C.

completely received. This behavior limits the amount of buffering, but it may introduce slowdowns if one or more packets containing the last bytes of a block are dropped. However, a more aggressive buffering technique would introduce additional overhead. The study of such a technique, which could be used in place of the standard one on loss-prone wireless links, is left as future work.

A simplified version of the receiver pseudocode is depicted in Figure 2. *WaitForPipes()* in line 4 is basically a *select()* which waits for data from the pipes. When new data arrive, the algorithm obtains the size of the block from the header of the first chunk (lines 7-10). *streamOffset* keeps track of the global amount of data received before the current block, while *header.streamOffset* indicates the position in the stream of the chunk that is being received. The offset of the chunk relative to the beginning of the application buffer is obtained by subtracting *streamOffset* to *header.streamOffset* (line 11). The *receiveChunk()* function (lines 12-13) reads *header.chunkSize* bytes from *pipe* and put them in the application buffer starting from position *chunkOffset*. The process is iterated until a whole block has been received.

```
01 sizeReceived = false ;
02 receivedData = 0;
03 do {
04   readyPipes = WaitForPipes();
05   foreach pipe in readyPipes {
06     header = GetHeader( pipe );
07     if ( !sizeReceived ) {
08       blockSize = header.blockSize ;
09       sizeReceived = true;
10     }
11     chunkOffset = header.streamOffset - streamOffset ;
12     receiveChunk( pipe , appBuffer[chunkOffset],
13                           header.chunkSize );
14     receivedData += header.chunkSize ;
15   }
16 } while ( receivedData < blockSize );
17 streamOffset += blockSize ;
```

Figure 2. Receiver pseudocode.

D. PATTHEL network protocol

The PATTHEL network protocol defines all the phases of a data session between two hosts. PATTHEL adopts a client/ server model and distinguishes between control information – which uses a dedicated TCP connection, the *control channe*l – and data, which use all the other active TCP connections (the *data channels*). Control operations require the exchange of *control blocks*, consisting of one or more TLV records.

When a client wants to start a new data session, it connects to the server. The first connection will be used as the control channel and will stay open for the entire communication. The use of a dedicated control connection makes implementation easier and avoids head-of-line blocking. In fact, it is desirable that commands that manage channels are transferred and processed as fast as possible, without having to deal with other in-flight data in TCP buffers.

After the control channel has been created, the client requests a first data channel. The server replies with a block containing a randomly generated token, an IP address, and a TCP port. At this point, the client opens a second connection towards the address/port just received. The first data sent by the client on the new channel must be the server-generated token, allowing the server to recognize the client and to associate the new data channel to the existing control channel.

Although the transfer can proceed using only the control channel and one data channel, during a PATTHEL session each host can request the creation of additional data channels – the procedure is the same as the creation of the first one. The number of channels to create is determined by a policy that can be fine-tuned acting on several parameters such as *desired bandwidth*, *channel spread* and *min improvement*. The current policy is to open multiple channels over the same interface till the number of channels reach *channel spread*, then moving to open channels on other interfaces (if any). The *channel spread* limit may not be reached in case the aggregated bandwidth is larger than the *desired bandwidth*, or the new channel does not lead to an improvement of the aggregated bandwidth of more than *min improvement*. Aggregated bandwidth is monitored for a pre-defined time interval (currently one second), which is used by PATTHEL to decide if a new channel has to be opened. In this way, PATTHEL avoid opening many channels for a short transfer. Note that the *desired bandwidth* parameter only allows to set a bandwidth requirement for received data. There is no parameter to set a target bandwidth for sent data – it is only possible to set an upper bound. Since the receiver uses the data, it is coherent that the receiver controls the rate at which data are provided. When a host issues a new channel request, the other host can either accept or refuse. The decision is based on other PATTHEL parameters, such as the *maximum send bandwidth*.

A consequence of this policy is that, even if all channels – being TCP pipes – are bidirectional, they are not always used in both directions. In particular, when a host requests and obtains a new channel, it can use the channel only for receiving data. The only exceptions to this rule – i.e. the only bidirectional channels – are the control channel and the first data channel (to guarantee a basic connectivity between the two hosts as long as the initial negotiation succeeds). This approach is coherent with the fact that, in many scenarios, the traffic between two hosts is not symmetric, i.e. the flow in one direction uses more bandwidth than the flow in the opposite direction. Hence, the fact that an additional channel is useful for traffic in one direction does not imply that also the traffic in the opposite direction needs to be parallelized.

The PATTHEL network protocol also defines the format of chunks transmitted on data channels. Each data chunk is accompanied by the header depicted in Figure 3. The *Chunk size* field (32 bits) informs the receiver of the size of the chunk payload, i.e. of how many bytes of data will follow the header. *Stream offset* (64 bits) indicates the position of the chunk within the stream, and it is used to copy the chunk in the right position of the application buffer. *Block size* (32 bits) allows the receiver to check if the block can fit in the application buffer and, if it is too large, to pre-allocate an additional temporary buffer. *Block index* (32 bits) is used to verify that the chunk belongs to the block currently being received.

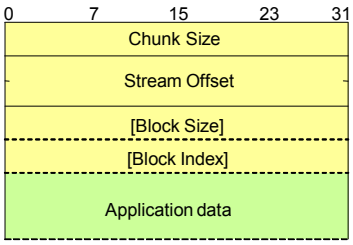

Figure 3. PATTHEL chunk header.

E. *Application interface*

The PATTHEL API is modeled over the Berkeley socket API. Primitives exported to applications mimic the behavior of well-known functions as *accept()*, *connect()*, *send()*, *receive()*, etc. Therefore, even if existing applications must be modified and re-compiled to make use of PATTHEL, in most cases the required changes are minimal.

Besides implementing the PATTHEL scheduler and receiver algorithms, the API functions transparently manage the routing policy configuration. A problem of application-level multipath is that, in modern operating systems (OS), the networking API does not allow user programs to bind a socket to a specific network interface. In fact, standard TCP applications leave to the OS the responsibility of choosing the best interface for sending data. To accomplish the task, the OS maintains a structure called routing table, accessed every time a packet is sent to a remote host. Using the routing table, the OS decides if the host is reachable and how to reach it. This situation is not compatible with the functioning of PATTHEL, because it does not allow the use of different network interfaces to reach the same destination. To circumvent the problem, PATTHEL takes care of adding a specific rule to the routing table each time a new channel is created. The rule forces the packets belonging to that specific TCP flow to go through the right interface.

Another issue is backward-compatibility with hosts that are not PATTHEL-aware. In the current PATTHEL version, a client application that wants to connect to a server can specify two different TCP ports, a *standard* one and a *fallback* one. First, the PATTHEL subsystem tries to open a connection to the *standard* port, and to negotiate a PATTHEL session. If the operation fails, another attempt is made toward the *fallback* port[2]. In the latter case, PATTHEL assumes that the other host is not PATTHEL-aware. Therefore, if the second connection attempt succeeds, PATTHEL wraps a standard TCP session.

The PATTHEL API is simple and immediately usable by developers already experienced in network-oriented programming, but lacks the power of expression to deal with features such as mobility support and channel failure recovery. Future PATTHEL developments should include an enrichment of the current API.

IV. Experimental evaluation

We implemented PATTHEL as a prototypal Dynamic Link Library for Windows. For testing, we created a simple file transfer suite – consisting of a client and a server – capable to use either PATTHEL connections or standard TCP socket. Three different testbeds (shown in Figure 4) were used.

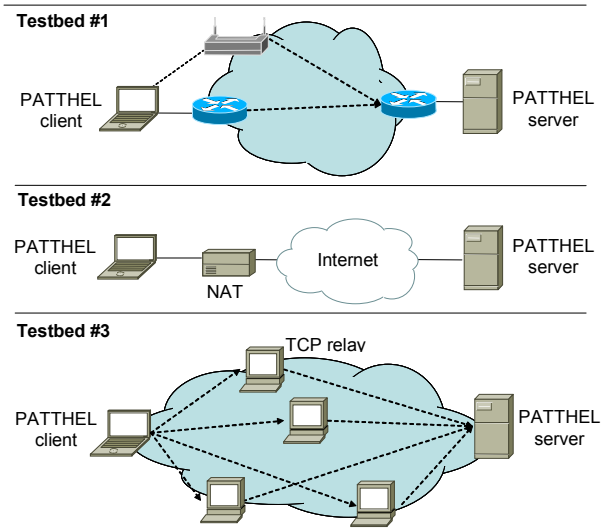

Figure 4. Testbeds.

In testbed #1, the client was connected through a Fast Ethernet link and a wireless one (54Mbps), and the server had a Fast Ethernet link. Both server and client were on the University network with public IP addresses, and bandwidth was artificially limited by configuring bandwidth shaping on intermediate routers (in order to create paths with reconfigurable bandwidths). In testbed #2, the client was connected to a single ADSL link (4Mbps download, 240Kbps upload) and had a private address, while the server configuration remained the same. In testbed #3, client and server were connected to the University network through a Fast Ethernet link, and both had public addresses.

The first test, conducted on testbed #1, aimed at evaluating the performance gain of PATTHEL compared to standard TCP in presence of multiple physical paths. We measured the time required to transfer a file with PATTHEL (using two channels) against a single TCP connection over the wired link. Results, reported in Table 1, show that the PATTHEL speedup is significant and close to the theoretical maximum (1.5 in the first two tests, 2 in the last ones).

The second test evaluated performances in the case of a single-channel transfer (over a single path), using only the wired links of testbed #1. Results (in Table 2) show that the overhead introduced by the protocol is negligible, with a penalty (in terms of increased duration of the transfer) of less than 1%. Another set of tests with the full link bandwidth showed that the maximum rate obtainable by PATTHEL was approximately 40 Mb/s – well below the nominal bandwidth of the link. By profiling the code, we determined that the reason was the *select()* system call. In fact, *select()* depends on the Operating System timer and reacts slowly when the pipe empties quickly – e.g. on high-speed LAN paths. However, this is a limit of our implementation and not of the PATTHEL mechanism.

The third test evaluated PATTHEL overhead in establishing new connections. PATTHEL first creates a TCP connection as the control channel, and then uses it to negotiate data

[2] Another solution would consist in using a TCP option; however this was cumbersome to do in our user-space library and it is left to a future work.

TABLE 1.PATTHEL TRANSFERS SPEEDUP.

Ethernet link bw	WiFi link bw	File size	PATTHEL Transf. time	TCP Transf. time	Gain
4 Mb/s	2 Mb/s	20 MB	27.33 s	40.60 s	1.49
16 Mb/s	8 Mb/s	100 MB	34.31s	50.99 s	1.49
4 Mb/s	4 Mb/s	20 MB	20.56 s	40.60 s	1.97
16 Mb/s	16 Mb/s	100 MB	26.73 s	50.99 s	1.91

TABLE 2. PATTHEL EFFICIENCY ON A SINGLE CHANNEL.

Channel bw	File size	Transfer time (PATTHEL)	Transfer time (TCP)	Penalty
4 Mb/s	20 MB	40.86 s	40.59 s	0.65 %
16 Mb/s	100 MB	51.32 s	50.99 s	0.65 %

connections. This requires the exchange of at least 9 packets before the data transfer can start. We transferred a large number of small (20 KB) files on a single channel, first using PATTHEL and then TCP. We used testbed #2 because, since the data had to travel through the Internet, we were able to observe much wider delays than on the LAN-based testbed #1.

Figure 5 shows the distribution of the setup time of PATTHEL and TCP, measured over 50 connections; the results show that PATTHEL setup time is higher than the corresponding time required to open a TCP connection, and it is definitely related to the RTT of the connection. This result confirms that PATTHEL, as expected, inserts a minimum bound for the setup time, which may become significant in case of short data transfers. However, PATTHEL was not designed for short transfers, e.g. the download of e-mail messages, which do not suffer much from bandwidth limitations. Note that results in Table 1 show that longer transfers are not affected by the increased setup time.

The fourth test analyzed the PATTHEL ability to quickly react to changes in the available bandwidth; it also evaluated its TCP friendliness. We used testbed #1, setting the bandwidth of the two links respectively to 4 Mb/s (Ethernet) and 3 Mb/s (Wireless). We started a TCP file transfer on the Ethernet link and, after about 15 seconds, a PATTHEL transfer using both links. The TCP transfer ended shortly before the PATTHEL one. Figure 6 depicts the bandwidth dynamics: TCP and PATTHEL were (as expected) able to share the bandwidth on the Ethernet link; moreover, PATTHEL was able to exploit all the Ethernet bandwidth as soon as it became available. Also note that the PATTHEL channel on the wireless link was not influenced by the behavior of the wired one. This result suggests that PATTHEL can manage bandwidth oscillations on a channel without degrading performances of the other active channels.

The next test aimed at verifying the effectiveness of PATTHEL in case of relay-based data transfers. Using testbed #3, we transferred files with different sizes through a different number of relays; the bandwidth of each channel was limited to 60000 bps. Results (in Figure 7) show that the advantage of using multiple relays is evident, particularly for large files. In fact, in this case the aggregated throughput increases linearly as more channels are added. Shorter transfers show an improvement as well, but because of the additional overhead imposed by the time required to open new connections, the

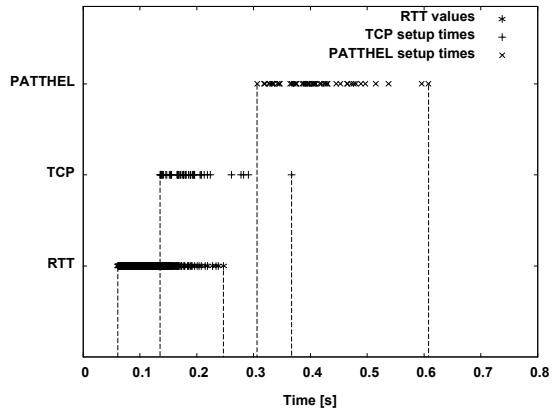

Figure 5. Connection setup time.

gain is smaller than the theoretical one (e.g., 471% improvement with 512KB file size and 6 relays). Additional measurements (omitted for brevity's sake) confirm that the overhead of the protocol is negligible (about 1.3% of exchanged bytes) and unrelated to the size of the file.

During the next series of tests, we simulated the peculiar situation in which, during a parallel transfer, a channel exhibits significant worse behavior – in terms of delay and RTT – than the others. The purpose was to estimate how much a faulty channel can influence the aggregate throughput of a transfer. We used testbed #1, splitting the transfer among three channels, all established through the wired link. On the path we installed a Linux machine, acting as a transparent bridge. Using the popular *netem* queue discipline ([16]), we configured the bridge to impose increasing delays and loss rates on one of the PATTHEL TCP flows (the other flows were not affected). In undisturbed conditions, the bandwidth of all the channels was limited at 60000 bps, with 100 ms RTT and no losses. The delay and the loss rate of channel #3 were then increasingly incremented[3], imposing the following conditions: a) 1% packet losses, 100 ms RTT; b) 5% packet losses, 200 ms RTT; c) 10% packet losses, 300 ms RTT; d) 15% packet losses, 400 ms RTT. For each different test, the time needed to transfer a 4 MB file was measured.

Figure 8 depicts the aggregate throughput for each test. The two horizontal lines are the throughput in undisturbed conditions using 3 channels and 2 channels. The former represents an upper bound for the aggregated throughput. The latter is the throughput beyond which the faulty channel should be closed, as it worsen the aggregated performances.

PATTHEL was able to obtain good performances with one channel exhibing light to medium losses and an RTT up to 3x larger than the RTTs of the other channels. However, when the third channel was heavily disturbed, the throughout dropped to less than a half of the throughput with only two channels. Future work on PATTHEL includes the study of a real-time channel profiling algorithm, to quickly locate and shut down channels which exhibit bad behavior.

[3] We run tests to decouple the effect of losses and RTT increases. However, we found that differences in RTT between channels have almost no effect without losses, and slightly worsen the performances only if coupled with high loss rates (>10%). We omitted those results for sake of brevity.

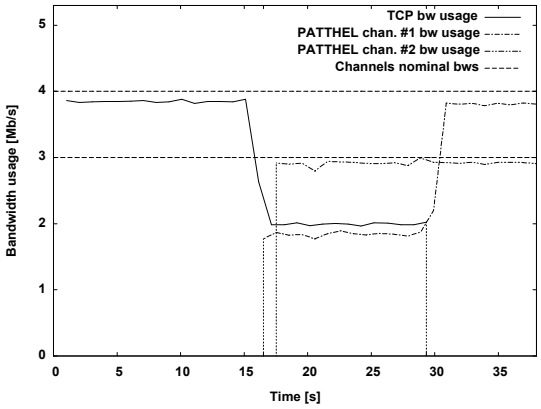

Figure 6. PATTHEL behavior with background traffic.

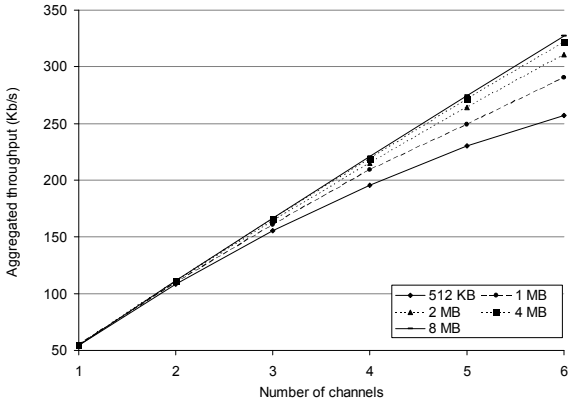

Figure 7. Transfer time with multiple relays and different file sizes.

V. CONCLUSIONS

This paper presents PATTHEL, a solution for data striping over multiple physical or logical paths. PATTHEL achieves parallelization by creating multiple TCP channels between two hosts. Its effectiveness has been demonstrated both in the case of multi-homed hosts and in the case of relay-based transfers.

Unlike many existing solutions, PATTHEL does not require invasive changes to the networking stack of hosts, as it is implemented on top of TCP and operates at layer V.

We created a PATTHEL software prototype and evaluated its performances. Test results are promising, and show that the proposed architecture is able to efficiently exploit multiple paths with a near-to-theoretical speedup, expecially for large data transfers. Moreover, the PATTHEL protocol causes a negligible increment in the overhead – compared to TCP – even when operating on a single channel. This is a significant advantage for applications, which do not have to switch to TCP primitives when they only need to use one channel.

A current limitation of PATTHEL relates to the policy that supervises the opening and the closing of new channels. Such policy depends on a set of parameters that may need to be fine-tuned on a case-by-case basis to achieve optimal performances. Future work will include an algorithm capable to auto-configure the policy through the profiling of the performances of active channels. This will allow PATTHEL to automatically detect the number of channels to open, and to decide how to distribute them among the available interfaces.

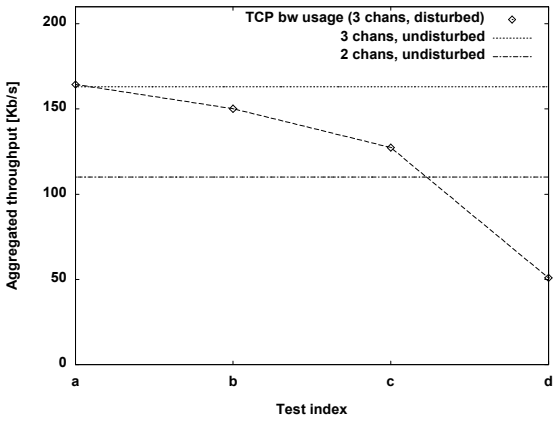

Figure 8. Performance loss caused by a faulty channel.

REFERENCES

[1] M. Allman, H. Kruse, S. Ostermann, "An Application-Level Solution to TCP's Satellite Inefficiencies", *Workshop on Satellite-based Information Services (WOSBIS)*, November 1996.

[2] Microsoft VirtualWiFi home page, *http://research.microsoft.com/netres/projects/virtualwifi/*.

[3] IEEE Standard 802.3: *CSMA/CD access method and physical layer specifications*, P802.3, 2005.

[4] C. E. Perkins, "Mobile networking through Mobile IP", in *IEEE Internet Computing*, Jan-Feb 1998, pp. 58-69, vol. 2, issue 1

[5] D. S. Phatak, T. Goff, "A Novel Mechanism for Data Streaming Across Multiple IP Links for Improving Throughput and Reliability in Mobile Environments", *IEEE INFOCOM*, New York, NY, June 2002, pp. 773-781, vol. 2.

[6] RFC 4960 (SCTP), *http://tools.ietf.org/html/rfc4960*.

[7] M. Fiore, C. Casetti, G. Galante, "Concurrent Multipath Communication for Real Time Traffic", *COMPUTER COMMUNICATIONS*, pp. 3307-3320, 2007, Vol. 30.

[8] H. Han, S. Shakkottai, C. V. Hollot, R. Srikant, D. Towsley, "Multi-Path TCP: A Joint Congestion Control and Routing Scheme to Exploit Path Diversity in the Internet", *IEEE/ACM Transactions on Networking*, Dec 2006, pp. 1260-1271, vol. 14, issue 6

[9] H. Hsieh, R. Sivakumar, "A transport Layer Approach for Achieving Aggregate Bandwidth on Multi-Homed Mobile Hosts", in *Proceedings of ACM MOBICOM*, Atlanta, GA, USA, Sept. 2002.

[10] N. Mohamed, J. Al-Jaroodi, H. Jiang, D. Swanson, "A Middleware-Level Parallel Transfer Technique over Multiple Network Interfaces", *ClusterWorld Conference and Expo*, San Jose, California, June 2003.

[11] M. Balakrishnan, R. Mishra, R. R. Rao, "On The Use of Bandwidth Aggregation Over Heterogeneous Last Miles", *2nd International Conference on Broadband Networks 2005*, 3-7 Oct 2005, pp. 1541-1547, vol. 2.

[12] T. Nguyen, Sen-Ching S. Cheung, "Multimedia streaming using multiple TCP connections", *24th IEEE International Performance, Computing, and Communications Conference*, 7-9 Apr 2005, pp. 215-223

[13] H. Sivakumar, S. Bailey, R. L. Grossman, "PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks", in *Proceedings of the IEEE/ACM SC2000 Conference*, 4-10 Nov 2000.

[14] Mario Baldi, Luca De Marco, Fulvio Risso, Livio Torrero, "Providing End-to-End Connectivity to SIP User Agents Behind NATs", *IEEE International Conference on Communications (ICC)*, May 2008.

[15] K. Chebrolu, R. Rao, "Communication using Multiple Wireless Interfaces", *IEEE Wireless Communications and Networking Conference*, 17-21 March 2002, pp. 327-331, vol.1.

[16] Netem configuration and usage, *http://www.linuxfoundation.org/en/Net:Netem*.