A Transport Protocol for Dedicated End-to-End Circuits
A. P. Mudambi, X. Zheng, M. Veeraraghavan
{apm9v, xz3y, mv}@virginia.edu
University of Virginia
Charlottesville, VA 22903

Abstract—Connection-oriented networks that offer guaranteed-rate circuits are being proposed as a solution to meet the high-end networking needs of distributed scientific research. This paper describes a transport protocol for data transfer across dedicated end-to-end circuits. We call this protocol Circuit-TCP (C-TCP). The key feature of C-TCP is to maintain a fixed sending rate, as closely matched to the circuit rate as possible, with the aim of achieving high circuit utilization. C-TCP is implemented by modifying TCP code in Linux. Experiments on a wide-area experimental circuit-switched testbed show that C-TCP is able to fully utilize circuit bandwidth and sustain a high circuit utilization. The challenge in this work lay in matching the variability inherent in general-purpose end hosts with the monotonic nature of circuits.

I. INTRODUCTION

The increase in computing power and maturing of grid computing technologies has opened up new possibilities for scientific research in fields such as earth sciences, high energy physics, astrophysics, molecular biology, and others [1]. Often such research involves collaboration between geographically distributed scientists accessing geographically distributed data sets and computational resources, requiring data transfers over a network. Traditional inter-networking protocols used in the connectionless sharing model of the Internet are inadequate in meeting the high-throughput, low-jitter and deterministic behavior needs of e-Science applications. Thus there is a need for new networking technologies to meet the demands of e-Science projects. Connection-oriented networks that allow the reservation of a dedicated circuit through the network have been proposed as a solution. There are a number of ongoing efforts to deploy testbeds for connection-oriented networks, such as UltraScience Net [2], Optical Metro Network Initiative (OMNINET) [3], Dynamic Resource Allocation via Generalized Multi-Protocol Label Switching Optical Networks (DRAGON) [4] and Circuit-switched High-speed End-to-End Transport Architecture (CHEETAH) [5].

A key consideration in using such networks is the design of a transport protocol for them. TCP, the most widely used transport protocol on the Internet, has been found wanting on high-capacity, long-delay packet-switched networks. A number of modifications and enhancements to TCP have been proposed. The nature of a dedicated circuit is very different from that of a path across a connectionless, packet-switched network, and hence we believe that this warrants a fresh look at the design of a transport protocol for connection-oriented networks. In this paper we propose a new transport protocol for such networks, and present results from our implementation. Our protocol is called Circuit-TCP (C-TCP), since we use many of TCP’s mechanisms in our implementation. This work, although general enough to be applicable to any connection-oriented network, has been carried out as part of the CHEETAH project and therefore a brief overview of the CHEETAH concept is presented next.

Ethernet and SONET are the predominant technologies used in LANs and WANs, respectively. At the same time, Multi Service Provisioning Platforms (MSPP), which map frames between the Ethernet and SONET domains, and support standard signaling protocols as defined for Generalized Multiprotocol Label Switching (GMPLS) networks, are being widely deployed. As part of the CHEETAH project we have leveraged these two developments to set up a testbed between North Carolina State University (NCSU), Raleigh, NC, and Oak Ridge National Laboratory (ORNL), Oak Ridge, TN via the MCNC point of presence in Research Triangle Park, NC and the Southern Crossroads/Southern LambdaRail (SOX/SLR) point of presence in Atlanta, GA. The testbed allows dedicated circuits, of up to 1 Gbps bandwidth, to be set up dynamically between CHEETAH-enabled end hosts. A CHEETAH-enabled end host is equipped with a secondary Network Interface Card (NIC) that is used exclusively for connecting to reserved circuits. We are also developing end host software to allow applications to make use of the CHEETAH network. The transport protocol

1. This work was carried out under the sponsorship of NSF grants, ITR-0312376 and AIN-0335190 at UVA.

implementation described in this paper is one of the components of the end host software.

The major difference in functionality between a transport protocol for a connectionless packet-switched network and a circuit-switched network is that the latter does not require congestion control in the data plane. Because bandwidth is reserved in circuit-switched networks, whether or not congestion occurs depends only on the behavior of the single entity that uses the reserved bandwidth. Thus, if the sender of data ensures that the transmission rate is less than or equal to the circuit rate there will be no congestion, unlike in a connectionless network where the behavior of other simultaneous flows can cause congestion. Further, to make the use of dedicated circuits economically feasible it is essential that the reserved bandwidth be used efficiently, making high circuit utilization an important objective. Ideally, to keep utilization high, data should be sent at the fixed circuit rate throughout the duration of the transfer.

In Section II we present related work and further motivation for this paper. Section III describes C-TCP protocol design and Section IV describes our implementation. Results of measurements taken over the CHEETAH testbed are presented in Section V. We conclude the paper in Section VI.

II. RELATED WORK

There has been significant activity in developing transport protocols suitable for high-bandwidth and/or high-delay networks. Even though very little of it is focussed explicitly towards dedicated circuits there is enough of an overlap in the problems to justify a closer examination. High-performance protocols can be classified as TCP enhancements, UDP-based and novel protocols. Ease of deployment and familiarity with the sockets API to the TCP and UDP stacks are reasons for the popularity of TCP and UDP-based solutions.

TCP is the most widely used reliable transport protocol on connectionless, packet-switched networks. In recent years a number of protocol extensions to TCP have been proposed and implemented to improve its performance on high-capacity networks. These include selective acknowledgements (SACK) [6][7], window-scaling and, the use of timestamps option for better round-trip time (RTT) estimation as well as protection against wrapped sequence numbers [8]. Standard (Reno) TCP has been found wanting in high-bandwidth, high-delay environments, mainly due to its congestion control algorithm. TCP’s Additive Increase Multiplicative Decrease (AIMD) algorithm is considered too slow in utilizing available capacity and too drastic in cutting back when network congestion is inferred. Modifications to the TCP congestion control algorithm have led to the development of HighSpeed TCP [9], Scalable TCP [10], FAST [11], and BIC-TCP [12]. TCP and its variants are geared towards finding out the fair share of a flow, but on a dedicated circuit the problem of determining how much capacity to allocate to a circuit is shifted to the control plane. Once a circuit is set up, the fair share is known and fixed. A transport protocol for such networks should take advantage of this extra information.

To overcome the shortcomings of TCP, many researchers have implemented protocols over UDP by adding required functionality, like reliability, in the user space. The most common model is to use UDP for the data transfer and a separate TCP or UDP channel for control traffic. SABUL [13], Tsunami, Hurricane [14], and RBUDP [15] use a TCP control channel and UDT [16] uses UDP for both data and control channels. The advantage of these solutions is that their user-space implementation makes deployment easy. We attempted to modify SABUL to implement a transport protocol for dedicated circuits [19]. This experience brought to light the shortcomings of a user-space implementation, namely, the difficulty of maintaining a steady sending rate and of implementing flow control.

Some novel protocols designed exclusively for high-performance data transfer have also been proposed. The eXPlicit Congestion Protocol (XCP) [17] has been proposed to solve TCP’s stability and efficiency problems. XCP’s requirement of multi-bit congestion signals from the network makes it harder to deploy. NETBLT [18] has been proposed for high-speed bulk data transfer. It provides reliability of the data transfer by sending blocks of data in a lock-step manner. This degrades bandwidth utilization while the sender awaits an acknowledgement (ACK) for the block.

III. CIRCUIT-TCP

In this section we present the design of C-TCP. We consider five functions of a transport protocol: connection establishment, congestion control, multiplexing, flow control and error control.
A. Connection Establishment

It is useful in the design of a transport protocol to think in terms of control and data planes. Control plane functions support the data plane. For instance, TCP’s three way handshake connection establishment is used to agree upon an initial sequence number to be used in the data transfer that follows. The TCP solution for connection establishment and release is used in C-TCP since the latter also needs sequence numbers for error control.

B. Congestion Control

Network congestion occurs when the demand for resources (e.g., bandwidth, switch buffers) exceeds the supply. Congestion control attempts to match the demand to the supply. In connectionless networks the supply changes during the data transfer. Therefore in TCP, congestion control has to be performed in the data plane. In connection-oriented networks the use of the data-plane can start only after the circuit has been set up. Therefore, in C-TCP the congestion control function of matching demand to supply is carried out in the control plane. C-TCP congestion control consists of initiating requests for circuits (e.g., by invoking signaling protocol clients at end hosts), receiving a connection setup success or failure indication, and correspondingly reacting to these indications. The reaction to a success indication is to initiate data transfer. A failure indication can be reacted to by waiting a random interval of time and initiating the request again. This is the main function in which C-TCP’s needs differ significantly from the services offered by TCP.

C. Multiplexing

As it is possible for an end host to have multiple processes making use of end-to-end dedicated circuits, C-TCP needs the capability to multiplex data from them. Multiplexing data from simultaneously running processes is supported in TCP by using port numbers. This solution works well for C-TCP too.

D. Flow Control

On a dedicated circuit, the sending rate should be matched to the reserved circuit bandwidth for optimal circuit utilization. So null flow control would be the ideal choice. To send and receive data at a fixed rate we need to reserve resources, such as processor cycles and buffer space, on the end hosts participating in the transfer. In practice, though, the end hosts run general-purpose operating systems, on which resource reservation is not possible without real-time support. Flow control mechanisms are required to match a sender’s demand for resources on the receiver with the available supply. Thus flow control is a required function in C-TCP.

There are three well-known flow control methods: ON/OFF, rate-based and window-based. The ON/OFF scheme is inefficient on a dedicated circuit because the circuit lies unused while a sender waits for an ON signal. In a rate-based scheme, the receiver sends signals to the sender to control the sending rate. The receiver needs to have a good estimate of its receiving capability during the whole transfer, which is very hard to implement. On the other hand, it is much simpler to monitor the status of the receive buffer during the data transfer and send back window size reports. TCP’s window-based flow control is thus a good match for C-TCP.

E. Error Control

Error control adds reliability to the data transfer by ensuring that all the data reaches the receiver in order and without duplicates. Sources of errors in today’s wired networks are typically buffer overflows at the packet switches and at the receiving end host. Dedicated circuits imply that resources are reserved for the data transfer and thus no losses can occur at network switches. The window-based flow control scheme prevents losses from occurring at the receiver. However, since link transmission errors are still possible, C-TCP needs to provide an end-to-end error detection/correction mechanism. We started out thinking that a simple negative acknowledgement (NAK) based scheme would be sufficient since circuits guarantee in-sequence delivery. Implementations showed that retransmission buffers have to be maintained in main memory at the sender, because of the prohibitive cost of retrieving lost blocks of data from the disk. This in turn meant that ACKs are required to free up buffer space held by data that has been successfully received. As TCP’s error control mechanism closely matches the above needs, here again we use it for C-TCP.

F. Protocol Format

Given that for four out of the five functions, TCP’s solution meets C-TCP’s needs, we use the TCP protocol format for C-TCP. In fact, the one function in which C-TCP differs from TCP, i.e., congestion control, does not have an impact on the protocol
header format.

IV. IMPLEMENTATION

In the previous section, we saw that many of TCP’s mechanisms for different functions are adopted for C-TCP. Thus, C-TCP is implemented by modifying TCP code in the Linux 2.6.11 kernel.

As noted in Section III.B, C-TCP initiates requests for circuits as part of its congestion control. If it requests a high-rate circuit, and the end hosts are not capable of maintaining data flow at that rate, circuit utilization will suffer. On the other hand, being over-cautious and requesting a low-rate circuit would lead to longer transfer delays than necessary. Thus, the circuit rate has to be selected properly by C-TCP. We discuss methods for selecting the circuit rate in IV.A.

Once the circuit is set up, the C-TCP implementation at the sender should maintain a data sending rate that matches the reserved circuit rate as closely as possible. Mechanisms for doing this are discussed in IV.B.

A. Selecting the circuit rate

As noted in Section III.D, variability can arise in the data sending and receiving rate at the end hosts. End host variability can occur due to a number of reasons, such as multitasking and disk access rate variability. We take a pragmatic approach to solve this problem:

1. To the extent possible, we require users of C-TCP to reduce the sources of variability, e.g., by not running other processes while the data transfer is in progress. We believe this is a reasonable requirement in the scientific computing domain.

2. Use an empirical method to estimate the average transfer rate that the end hosts can support. The disk write rate is the limiting factor, hence we have implemented a benchmark program to be run on the receiving host to estimate the average rate at which data can be written to disk. The circuit is set up at this rate.

If circuit utilization is not an important requirement, a user might choose to run other processes simultaneously with the data transfer. Even otherwise, the circuit rate is an empirically estimated average, and therefore flow control is required to avoid losses at the receiver.

B. Maintaining the sending rate

After much experimentation with mechanisms to maintain a fixed sending rate, such as busy waiting [19] and using system timers and signals, we found TCP’s self-clocking mechanism to be an efficient technique. The self-clocking mechanism of inserting new packets into the network when ACKs come back helps in maintaining a steady sending rate at a packet-level granularity.

Bandwidth reservation in the network means that the minimum amount of outstanding data that the network can sustain is fixed at the bandwidth delay product BDP (= circuit rate*RTT). If the receiver buffer space is not a limiting factor, then having less than BDP amount of data in the network is detrimental to circuit utilization. On the other hand, pushing more and more data into the network, filling up the network buffers, experiencing loss and then reducing the sending rate is also not desirable. Therefore in C-TCP, a fixed amount of outstanding data, greater than or equal to the BDP, is maintained. To implement C-TCP in Linux we use the Web100 [20] instrumented TCP stack. The Web100 instrumented stack provides an interface for user space programs to access many of TCP’s internal state variables. The interface also allows some fields (control parameters), in the internal data structure that Linux maintains for each TCP socket, to be set from the user space. We added 2 control parameters to the Web100 stack-

1. useckt to select whether a TCP socket is to be used as one end of a CHEETAH circuit and,
2. ncap (network capacity) to set the amount of data that is maintained in the network.

The useckt parameter is required so that a single TCP stack can be used to provide standard TCP service on Internet paths and C-TCP service when a dedicated circuit is being used.

We modified TCP sender code to ignore the congestion window cwnd, and instead maintain min(ncap, rwnd^3) amount of unacknowledged data in the network throughout the transfer when useckt is set.

Linux uses a slow start like scheme to update rwnd too. This makes rwnd a bottleneck during the initial part of the transfer and defeats the purpose of the changes made at the sender. Therefore, we modified the TCP receiver code to advertise the maximum

---

3.rwnd is the receiver advertised window that conveys information about how much free space is available in the receive buffer.
possible rwnd when the socket is being used over a CHEETAH circuit.

One consequence of implementing C-TCP as described above is that TCP's slow start is disabled. Thus, during the early part of the data transfer, C-TCP utilizes the reserved bandwidth better, since for long-delay networks, slow start can reduce the transfer throughput significantly. But slow start serves the purpose of starting up self-clocking without sending a large burst of packets. In a CHEETAH network, an end host’s 1 Gbps Ethernet NIC is connected to a dedicated SONET circuit. Disabling slow start is not an issue if the circuit rate and the NIC rate are matched, since the sender can not transmit a burst of packets. When the circuit rate is set to be less than 1 Gbps (which could happen if the receiver rate is lower than 1 Gbps, in which case, we would deliberately choose the lower rate for the wide-area circuit), C-TCP will send an initial burst of packets that could cause buffer overflows at the first downstream MSPP. For a NIC rate $N$, a circuit rate $C$ and an RTT of $T$, the buffer space $B$ needed to avoid overflow is:

$$B = T \times \left( C - \frac{C^2}{N} \right)$$

(1)

For a given $T$ and $N$, the maximum $B$ is required when $C = N/2$. For instance, with $N = 1$ Gbps and $T = 200$ ms, the maximum buffer space required at the first downstream MSPP is 6.25 MB. The MSPPs used in the CHEETAH testbed are capable of buffering this much data. In addition, data link layer flow control, like the use of PAUSE frames in full-duplex Ethernet [21], also prevents buffer overflow.

V. RESULTS

The Linux implementation of C-TCP described in the previous section has been tested on the CHEETAH testbed. The portion of the testbed relevant for our experiments is shown in Figure 1. The path of the reserved circuit is shown as a dotted line. The blocks marked zelda1 through zelda5 and wukong are end hosts with a primary Internet-connected NIC and a secondary NIC (shown shaded). For our experiments we loaded the modified Linux 2.6.11 kernel on hosts zelda4 at ORNL, zelda3 in Atlanta and wukong at MCNC. zelda3/4 are Dell PowerEdge 2850s, with dual 2.8 GHz Xeon processors and 2 GB of memory. Wukong is a Dell PowerEdge 1850 with a 2.8 GHz Xeon processor and 1 GB of memory. All three have an 800 MHz front side bus, 146 GB SCSI disks and a PERC4 RAID controller. The blocks marked as Sycamore SN16000 are the MSPPs.

To compare standard TCP with C-TCP, we use the iperf application (version 1.7.0) [22] for memory-to-memory transfers. A command-line option was added to iperf to select between TCP and C-TCP. To execute disk-to-disk transfers we implemented a simple file transfer application.

A. Utility of disabling slow start

A 1 Gbps circuit was set up between zelda4 and wukong. We ran memory-to-memory data transfers for various transfer sizes and gathered throughput and delay values. In Figure 2, the TCP and C-TCP transfer throughput and their relative delay (delay using TCP/delay using C-TCP) are plotted against transfer size. For data transfers of a few MB, slow start takes up a substantial portion of the total transfer time. The relative delay plot shows the utility of disabling slow start for such transfers. C-TCP completes the data transfer in less than half the time it takes for TCP. For very small transfers the three-way handshake connection establishment overhead dominates the transfer time. As expected, the returns of avoiding slow start diminish as transfer size increases.
and the throughputs achieved by TCP and C-TCP start converging. To show the difference in start-up behavior between C-TCP and TCP we captured packet traces for a 5 MB transfer, using tcpdump, and used tcptrace to generate the plot in Figure 3. The dark lines labelled ‘C-TCP data’, ‘TCP data’ show the sequence numbers (relative to the initial sequence number) of received data packets as a function of time (relative to the start of data transfer). When C-TCP finishes the data transfer at around 0.06s, TCP has only been able to transfer about a third of the total amount because of slow start. The dotted lines show the receiver advertised window. For example, at time 0.06s TCP has received data packets with relative sequence numbers up to 1.3 million and the advertised window is 3.8 million. This means the receiver is willing to accept 2.5 MB of data. The slow-start like growth of the receiver window in standard TCP and our modification to this behavior, which we mentioned in Section IV, can be seen. The gains from avoiding slow start will be even more pronounced when the RTT is higher.

B. Sustained data transfer

The next set of results show the utility of maintaining a fixed amount of outstanding data in the network. We set up a 500 Mbps circuit for this experiment, so there is a mismatch with the 1 Gbps NIC rate. A circuit with a rate smaller than the NIC rate might be set up for the reasons specified in Section IV.A, or because of lack of resources in the network. We used iperf to conduct sustained memory-to-memory transfers, for 600 seconds, from zelda4 to wukong. To get an estimate of the switch buffer status we simultaneously ran ping between the 2 end hosts and gathered the RTT values. The baseline RTT is 13.6 ms. Figure 4 shows the results for Reno-TCP. Notice the variability in achieved throughput and RTT. The average throughput over 600 seconds is 389 Mbps. The results for BIC-TCP\(^4\) are shown in Figure 5. BIC-TCP is designed for high BDP environments and is more aggressive in increasing cwnd. This is evident in the high RTT values. The average throughput achieved by BIC-TCP is 423 Mbps. In Figure 6, the results for C-TCP show that because C-TCP maintains a fixed amount of outstanding data in the network, it is able to achieve a steady throughput. C-TCP achieves an average throughput of 458 Mbps, and does so without stressing the switch buffer, as the RTT plot shows.

C. Disk-to-disk transfers

For disk-to-disk transfers the achieved throughput depends on factors such as file size, position of the file blocks on disk in addition to the protocol behavior. These experiments are conducted between zelda3 and zelda4 (see Figure 1). Figure 7 shows the achieved throughput in transferring a 1.6 GB file over a 1 Gbps

---

\(^4\)Linux 2.6.11 uses BIC-TCP’s congestion control by default. To use Reno in place of BIC set the sysctl variable net.ipv4.tcp_bic to 0.
Figure 6. Throughput and RTT using C-TCP

Figure 7. Throughput variability of disk to disk transfers circuit using TCP and C-TCP. The variability of the throughput evident in these 10 runs shows the difficulty in assigning a single number to the performance of disk-to-disk transfers.

As described in Section IV.A, the variability inherent in disk-to-disk transfers makes the choice of the circuit rate important. Intuitively, a very high-rate circuit’s bandwidth may not be well utilized because the throughput is constrained by factors such as the disk write rate. On the other hand a circuit with a lower rate will be well utilized but the data transfer time will be longer. For our next experiment we transferred a 1.6 GB file from zelda3 to zelda4, using C-TCP, over circuits with rates ranging from 400 Mbps to 1 Gbps. In Figure 8 the average (for 5 runs) data transfer delay and circuit utilization (throughput/circuit rate) are plotted against the circuit rate. Standard deviation around each point is shown\(^5\). The plot broadly confirms our intuition. The transfer delay can be halved with a circuit rate of 1 Gbps as compared to a 400 Mbps circuit, but at the cost of a 20% reduction in the bandwidth utilization (70% compared to 85%).

\(^5\) The large standard deviation in the utilization for circuit rate = 600 Mbps is due to a single anomalously low throughput transfer.

**VI. CONCLUSIONS**

Dedicated circuits have been proposed as a solution to meet the networking needs of large-scale scientific projects. We propose a transport protocol, Circuit-TCP, for data transfer between end hosts connected by a dedicated circuit. Unlike in transport protocols for connectionless networks, where congestion control is required in the data plane to vary the sending rate based on available bandwidth on the end-to-end path, C-TCP needs to maintain a constant sending rate matched to the circuit rate.

We presented an implementation of C-TCP for Linux that makes use of TCP code. We changed the congestion control code in TCP to always maintain a bandwidth delay product amount of outstanding data in the network. This disables TCP’s slow start and AIMD algorithms. Results on the CHEETAH testbed, an experimental circuit-switched network, were presented. The results showed the utility of disabling slow start for small data transfers and the suitability of C-TCP for long-lasting flows. Experimental results for disk-to-disk transfers showed the trade-off between circuit utilization and transfer delay.

**VII. REFERENCES**


[22] National Laboratory for Applied Network Research (NLANR), Distributed Applications Support Team (DAST), URL http://dast.nlanr.net/Projects/Iperf/