Supporting Configurable Congestion Control in Data Transport Services

Yunhong Gu[1] and Robert L. Grossman[1, 2]

1. University of Illinois at Chicago
2. Open Data Partners

{ygu3, grossman}@uic.edu

ABSTRACT

As wide area high-speed networks rapidly increase, new applications emerge and require new control mechanisms in data transport services to support them. Network researchers have proposed numerous congestion control algorithms in the past few years. However, a difficult problem is how to evaluate, implement, and deploy these new algorithms in a practical way.

In this paper, we present UDT/CCC (UDP-based Data Transport Library with Configurable Congestion Control), a data transport library that allows users to make use of a new control algorithm through simple configurations. We aim to provide a tool for fast implementation and deployment, as well as easy evaluation, of new congestion control algorithms. UDT/CCC uses an objected oriented design. It collects a set of congestion control event handlers and parameters in a base C++ class, so a new control mechanism can redefine or modify them through C++ class inheritance. We show that our UDT/CCC library can be used to implement a large variety of control algorithms and can easily simulate the behavior of their native implementations. The UDT/CCC library is at the application level and it does not need root privilege to get installed. Meanwhile, it was specially developed to require very few changes to the existing applications. This paper describes its design, implementation, and evaluation.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Protocol Architecture

General Terms

Management, Performance, Design, Experimentation

Keywords

Transport Protocols, Congestion Control, UDT, CCC

1. INTRODUCTION

The rapid increase of network bandwidth has enabled numerous new distributed data intensive applications. These new applications vary from bulk data transfer (e.g., SDSS [34] and e-VLBI [40]) to high throughput interactive systems (e.g., GeoWall [35]). Different applications have specific requirements for data transfer services. For example, a GeoWall application may prefer a smooth image/video data transfer rate, whereas it is desirable to move SDSS data at the highest possible speed in private networks.

However, the current Internet is designed to provide general service to support as many different applications as possible. This design philosophy has a major impact on the transport protocols. The majority of traffic on the Internet is contributed by TCP flows; but there still are applications that TCP cannot support well. In the context of high performance computing, TCP is well known for its poor efficiency and fairness in high bandwidth delay product networks [1, 2].

In the past few years, network researchers have proposed numerous new congestion control algorithms for both general and specific data transfers. However, most of them only end with simulations and limited laboratory experiments; few get tested and deployed in real networks. On the one hand, modifications of the kernel network stack (e.g., new TCP variants) usually take years for standardization, implementation, and widespread deployment. On the other hand, although user space stacks are much easier to get deployed, it is often a painstaking and time-consuming job to implement each of them from scratch. In fact, ever since the emergence of TCP about three decades ago, only four versions have been widely deployed, namely Tahoe, Reno, NewReno, and SACK.

It is very desirable to create a configurable or reusable user space network stack on which a new congestion control algorithm can be easily implemented, deployed, and evaluated. First, a user space stack is much easier to get deployed, and so is the congestion control algorithms built in it. Second, this stack is useful to support application aware control approaches. An application may prefer to use different congestion control strategies in different situations. Third, this stack can save significant time for network researchers and developers because they can focus on the control algorithm itself rather than the whole protocol implementation. As a sequence, as there are more and more users, this stack can provide good software quality to support application development.

However, this stack is not a replacement for, but a complement to the kernel space network stacks. General protocols like UDP, TCP, DCCP [8], and SCTP [7] should still exist inside the kernel space of operating systems, but OS vendors may be reluctant to support too many protocols and algorithms, especially those application specific or network specific ones.

To address the above requirement, we developed the UDT/CCC library, or UDP-base Data Transfer library with Configurable Congestion Control. This work is based on our UDT library [1], which is a user space data transport library developed for high performance data intensive applications. CCC is one of the features included in the current UDT release.

UDT/CCC is written in C++ and it provides a set of control events handlers and parameters in a base C++ class. A new control algorithm inherits this base class, redefines the proper control event handlers, and modifies certain control parameters when necessary.

UDT/CCC supports a wide variety of control algorithms, including but not limited to, TCP algorithms (e.g., NewReno, Vegas [25], FAST [26], Westwood [24], HighSpeed [27], BiC [29], and Scalable [28]), bulk data transfer algorithms (e.g., SABUL [30], RBUDP [31], LambdaStream [32], CHEETAH [33], and Hurricane [41]), and group transport control algorithms (e.g., CM [5] and GTP [36]).

The UDT library provides a socket-like API so that applications can change their data transport service between TCP and UDT/CCC with ease. In certain situations, an existing TCP-based application can make use of UDT without any change to the source code.

We envision the following use scenarios for UDT/CCC:

- Implementation and deployment of new control algorithms. Certain control algorithms may not be appropriate to be deployed in kernel space, e.g., a bulk data transfer mechanism used only in private links. These algorithms can be implemented using UDT/CCC.
- Application awareness support and dynamic configuration. An application may choose different congestion control strategies under different networks, different users, and even different time slots. UDT/CCC supports these application aware algorithms.
- Evaluation of new control algorithms. Even if a control algorithm is to be deployed in kernel space, it needs to be tested thoroughly before OS vendors distribute the new version. It is much easier to test the new algorithms using UDT/CCC than modifying an OS kernel.

In this paper, we present the design, implementation, and evaluation of the UDT/CCC library. The rest of the paper is organized as follows. We begin with the CCC design and architecture in Section 2, followed by the key implementation details in Section 3. We then evaluate the UDT/CCC library in the following two sections. In Section 4, we demonstrate the expressiveness and simplicity of using UDT/CCC to develop new control algorithms. In Section 5, we use experimental studies to examine the performance characteristics. Finally, we give a brief review of related work in Section 6 and conclude the paper in Section 7.

2. ARCHITECTURE AND DESIGN

The UDT/CCC library can be regarded as a middleware layer between applications and the network transport layer. The library is completely implemented at the application level above UDP. In this section, we focus on how the congestion control interface is provided by the library and how it is implemented inside the UDT layer.

2.1 Overview

The UDT framework has a layered architecture (Figure 1). UDT uses UDP through the socket interface provided by operating systems. Meanwhile, it provides a UDT socket interface to applications.

Applications can call the UDT sockets API in the same way they call the system sockets API. They can provide a congestion control class instance (CC in Figure 1) for UDT to process the control events, or use the default congestion control algorithm [1] provided by UDT. The CC instance includes a set of necessary user-defined callback functions (control event handlers) to process certain control events.

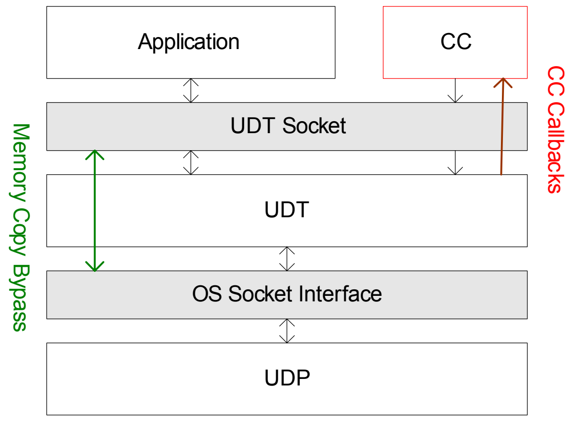

Figure 1. UDT/CCC Architecture. *In this layered architecture, the UDT layer is completely in user space above the network transport layer of UDP, whereas the UDT layer itself provides transport for applications. Meanwhile, applications provide optional control handlers to UDT as callbacks***.**

2.2 The CCC Interface

We identify four categories of configuration features to support configurable congestion control mechanisms. They are 1) control event handler callbacks, 2) protocol behavior configuration, 3) packet extension and 4) performance monitoring.

2.2.1 Control Event Callbacks

Seven basic callback functions are defined in the base CCC class. They are called by UDT when a control event is triggered.

init and **close**: These two methods are called when a UDT connection is set up and when it is torn down. They can be used to initialize necessary data structures and release them later.

onACK: This handler is called when an ACK (acknowledgment) is received at the sender side. The sequence number of the acknowledged packet can be learned from the parameters of this method.

onLoss: This handler is called when the sender detects a packet loss event. The explicit loss information is given to users as the *onLoss* interface parameters. Note that this method may be redundant for most TCP algorithms that use only duplicate ACKs to detect packet loss.

onTimeout: A timeout event can trigger the action defined by this handler. The timeout value can be assigned by users, otherwise it uses the default value according to the TCP RTO calculation described in RFC 2988 [11].

onPktSent: This is called right before a data packet is sent. The packet information (sequence number, timestamp, size, etc.) is available through the parameters of this method.

onPktReceived: This is called right after a data packet is received. Similar to *onPktSent*, the entire packet information can be accessed by users through the function parameters.

processCustomMsg: This method is used for UDT to process user-defined control messages.

2.2.2 *Protocol Configuration*

To accommodate certain control algorithms, some of the protocol behavior has to be customized. For example, a control algorithm may be sensitive to the way that data packets are acknowledged. UDT/CCC provides necessary protocol configuration APIs for these purposes.

It allows users to define how to acknowledge received packets at the receiver side. The functions of *setACKTimer* and *setACKInterval* determine how often an acknowledgement is sent, in elapsed time and number of arrived packets, respectively.

The method of *sendCustomMsg* sends out a user-defined control packet to the peer side of a UDT connection, where it is processed by callback functions *processCustomMsg*.

Finally, UDT/CCC also allows users to modify the values of RTT and RTO. A new congestion control class can choose to use either the RTT value provided by UDT, or its own calculated value. Similarly, the RTO value can also be redefined.

There are other features of the UDT protocol that are either not related to congestion control or are helpful to most control algorithms. These features, such as selective acknowledgement (SACK) [37] and robust reordering (RR) [38], cannot be configured by CCC users, although some of the features can be configured through UDT interfaces.

2.2.3 *Packet Extension*

It is necessary to allow user-defined control packets for a configurable protocol stack.

Because our UDT/CCC library is mainly focused on congestion control algorithms, we only give limited customization ability to the control packets. Data packet processing contributes to a large portion of CPU utilization and customized data packets may hurt the performance.

A UDT data packet contains a packet-based sequence number and a relative timestamp (it starts counting since the connection is set up) in the resolution of microseconds (Figure 2), in addition to the UDP header information. We believe that this information is sufficient for most control algorithms.

Users can define their own control packets using the Type 2 information in the UDT control packet header (Figure 2). The detailed control information carried by these packets varies depending on the packet types. At the receiver side, users need to override *processCustomMsg* to tell UDT/CCC how to process these new types of packets.

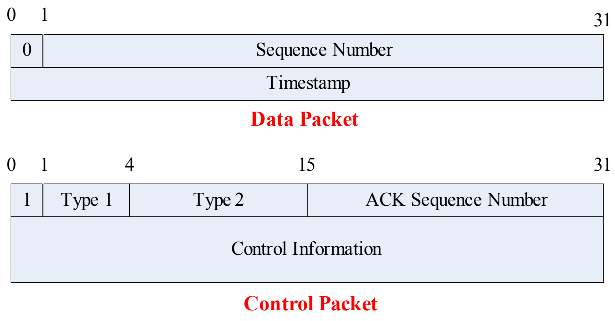

Figure 2. UDT Packet Header Structures. *The first bit of the packet header is a flag to indicate if this is a data packet (0) or a control packet (1). Data packets contain a 31-bit sequence number and a 32-bit timestamp. In the control packet header, bit 1- 4 is the packet type (type 1) information. Type 0 – 6 are used by UDT, whereas type 7 is used for user defined types, whose detail type information is put in bit 5 – 15 (type 2). The detailed control information depends on the packet type.*

Note that UDT's packet-based sequencing with the packet size information provided by UDP is equivalent to TCP's byte-based sequencing and can also support data streaming.

2.2.4 *Performance Monitoring*

Protocol performance information supports the decisions and diagnosis of a control algorithm. For example, algorithms like TFRC [9] need to learn the loss rate to tune the packet sending rate. Meanwhile, when testing new algorithms, performance statistics and internal protocol parameters are needed.

The performance monitor provides information including the duration time since the connection was started, RTT, sending rate, receiving rate, loss rate, packet sending period, congestion window size, flow window size, number of ACKs, and number of NAKs. UDT records these traces whenever the values are changed.

These performance traces can be read in three categories (when applicable): the aggregate values since the connection started, the local values since the last time the trace is queried, and the instant values when the query is made.

2.3 The UDT Protocol

UDT is a unicast, connection-oriented, duplex data stream protocol.

Inside the UDT layer, there are two basic logical parts: the sender and the receiver. The sender is responsible for sending data packets according to congestion and flow control, whereas the receiver is responsible for receiving and processing packets, as well as sending control packets when necessary.

Figure 3 describes the relationship between the UDT sender and receiver. In Figure 3, the UDT entity A sends application data to the UDT entity B. The data is sent from A's sender to B's receiver, whereas the control flow is exchanged between the receivers.

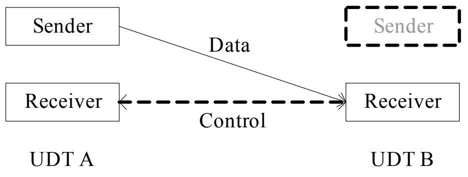

Figure 3. Relationship between UDT Sender and Receiver. *All UDT entities have the same architectures, each having both a sender and receiver. This figure demonstrates the situation when a UDT entity A sends data to another UDT entity B. Data is transferred from A's sender to B's receiver, whereas control information is exchanged between the receivers.*

The detailed UDT protocol specification can be found in [42]. In the rest of the section, we will introduce the abstract UDT sending and receiving algorithms and how the control event handlers are processed in these algorithms. The detailed implementation will be introduced in next section.

2.3.1 The Sending Algorithm

Figure 4 describes the abstract sending algorithm. In this algorithm, a sender's loss list is a data structure that records the lost data packets when informed of them by loss reports from the receiver or by sender side timeouts. ACK and NAK are the abbreviations of acknowledgment and loss report (negative acknowledgment), respectively.

```
1) If there is no application data to send, sleep until it is activated
   by the application.
2) Packet sending:
   a) If the sender's loss list is not empty, remove the first lost
      sequence number from the list and pack the corresponding
      packet.
   b) Otherwise, if the number of unacknowledged packets does
      not exceed the congestion and flow window sizes, pack a
      new packet.
   c) Otherwise, wait here until an ACK or NAK is received, or
      timeout occurs. Go to Step 1.
3) onPktSent().
4) Send the packed packet out.
5) Wait until the next packet sending time. Go to Step 1.
```

Figure 4. UDT Sending Algorithm.

Step 2.b is the window/flow control, which limits the number of unacknowledged packets. The limit is equal to either the congestion window size or the flow window size, whichever one is smallest.

Step 2.c implements self-clocking. This step could be removed to realize a pure rate-based control. However, due to the lack of high precision timing in general operating systems, self-clocking is useful in reducing CPU overhead (by avoiding busy waiting). Note that the timeout in this step is used to break the deadlock when there is no feedback. It is different from the packet sending timeout in the o*nTimeout* method.

Step 4 is the rate control, which suspends the data sending until the next sending time.

2.3.2 The Receiving Algorithm

Figure 5 describes the receiving algorithm. In this algorithm, the receiver's loss list is a data structure to store the sequence numbers of the lost packets. EXP is the abbreviation for timeout (expiration).

```
1) Query the timers
   a) If ACK timer is expired and there are new packets to
      acknowledge, send back an ACK report; otherwise, if the
      user-defined ACK interval is reached, send back a
      lightweight ACK report.
   b) If NAK timer is expired and the receiver's loss list is not
      empty, send back a NAK report;
   c) If EXP timer is expired and there are sent but
      unacknowledged packets, execute onTimeOut(), and put
      the sequence numbers of these packets into the sender's
      loss list;
   d) Reset the expired timers.
2) Start time bounded UDP receiving. If nothing is received before
   the UDP timer expires, go to Step 1.
3) If there is no unacknowledged packet, reset the EXP timer.
4) If the received packet is a control packet, process it, and reset
   EXP timer if it is an ACK or NAK; According to the packet type,
   one of the following callback functions may be executed:
   onACK(); onLoss(); processCustomMsg();
   Go to Step 1.
5) Check packet loss. If there are packet losses, insert the sequence
   numbers of the lost packets into the receiver's loss list and
   generate a loss report (NAK).
6) onPktReceived(); Go to Step 1.
```

Figure 5. UDT Receiving Algorithm.

The receiver uses self-clocked timers to trigger acknowledgment, loss reports, and timeout events (step 1 and 2). This timing mechanism takes advantage of time-bounded UDP receiving using the SO_RCVTIMEO option. On systems where SO_RCVTIMEO is unavailable, *select* can be used.

As soon as a packet is received, the receiver processes the new packet according to its type (step 4 and 5).

If there is no unacknowledged packet, the receiver simply resets the EXP timer (step 3); otherwise, the EXP timer only resets when it is an ACK or NAK packet (step 4).

Note that UDT has its own ACK and NAK processing mechanism in addition to the user-defined event handler for data reliability purposes.

Step 5 is to check packet loss. Various loss detection techniques such as robust reordering [38] can be used here. UDT uses packet based sequencing and the packet size can be determined from the UDP interface.

3. IMPLEMENTATION

One particular motivation of UDT/CCC is to support high performance data transfer in high-speed networks. Efficiency was always a major guideline when we implemented the UDT library.

The efficiency of a protocol mainly depends on two factors: the congestion/flow control algorithm and the protocol design/implementation. UDT/CCC inherits much of its design and implementation from the UDT transport protocol [1].

Another important implementation objective is to provide good support for applications. User space libraries can be much easier to get deployed, but it is difficult for them to be transparent to applications. Therefore, existing applications may need to change their source codes. Our implementation makes a great effort to reduce the application developers' work.

3.1 Software Architecture

Figure 7 depicts the UDT/CCC software architecture. The UDT layer has five function components: the API module, the sender, the receiver, the listener, and the UDP channel, as well as four data components: sender's protocol buffer, receiver's protocol buffer, sender's loss list, and receiver's loss list.

Because UDT is bi-directional, all UDT entities have the same structure. The sender and receiver in Figure 6 have the same relationship as that in Figure 2.

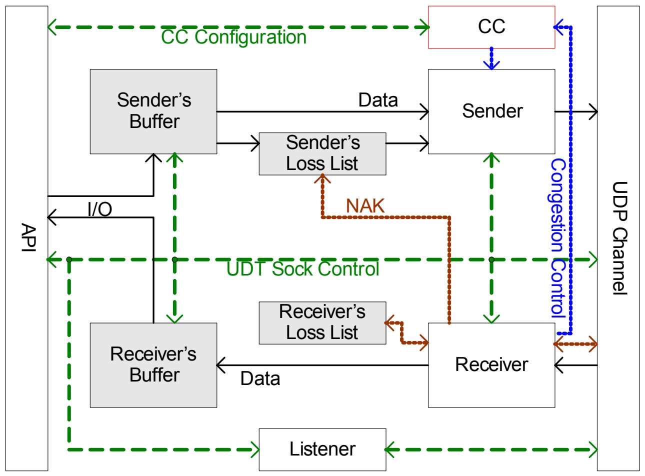

Figure 6. UDT/CCC Implementation. *The solid line represents the data flow, and the dashed line represents the control flow. The shading blocks (buffers and loss lists) are the four data components, whereas the blank blocks (API, UDP channel, sender, receiver, and listener) are function components.*

The API module is responsible for interacting with applications. The data to be sent is passed to the sender's buffer and sent out by the sender into the UDP channel. At the other side of the connection (not shown in this figure but it has the same architecture), the receiver reads data from the UDP channel into the receiver's buffer, reorders the data, and checks packet losses. Applications can read the received data from the receiver's buffer.

The receiver also processes received control information. It will update the sender's loss list (when NAK is received) and the receiver's loss list (when loss is detected). Certain control events will trigger the receiver to update the congestion control module, which is in charge of the sender's packet sending.

The UDT socket options are passed to the sender/receiver (synchronization mode), the buffer management modules (buffer size), the UDP channel (UDP socket option), the listener (backlog), and CC (the congestion control algorithm). Options can also be read from these modules and provided to applications by the API module.

3.2 Optimizations

Implementation efficiency is critical to UDT/CCC for two major reasons: 1) the primary goal of the library is to support the emerging high performance applications; 2) we do not want to limit the performance of the control algorithm by a poor implementation.

We have described the optimizations of the UDT library in previous work [1]. In this section, we only give a very brief summary of these optimizations, plus new problems arising from the introduction of CCC.

The first optimization category is focused on the avoidance of memory copy. Ideally, application data should be exchanged directly with the UDP channel. This optimization needs support from the API and protocol buffer management modules.

The second optimization category is designed to reduce the frequency of control events and the overhead of event handling. The majority of control events come from ACK/NAK triggering and processing. Other events, such as API call and timeout, are much less frequent. The UDT library has introduced an optimized loss processing algorithm to handle NAK processing.

The UDT protocol uses timer-based acknowledgement so there are only a small portion of ACKs. However, when introducing CCC, a new algorithm may require much more frequent ACKs. To handle this problem, UDT/CCC still only modify the protocol buffers at certain time intervals; for all other ACKs (namely light ACKs), only the necessary sequence number related data are updated.

Finally, since high-end workstations usually have multiple processors, UDT use a multi-threading implementation. The sender, the receiver, and the listener are concurrent threads, but they are only started when necessary (lazy start). It is also necessary to provide fine granularity of the threading implementation.

3.3 Application Interface

An application can make use of the UDT/CCC library in four ways. The library provides a set of C++ API that is very similar to the system socket API. Network programmers can learn it easily and use it in a similar way as using TCP sockets. In particular, applications can use the *setsockopt/getsockopt* method to set and configure a specific congestion control algorithm at run time.

When used in applications written by languages other than C/C++, an API wrapper can be used. So far, both Java and Python UDT API wrappers have been developed.

Certain applications have a data transport middleware to make use of multiple transport protocols. In this situation, a new UDT driver can be added to this middleware, and then used by the applications transparently. For example, a UDT XIO driver has been developed so that the library can be used in Globus applications.

Finally, our library also provides a set of C API that has exactly the same semantics as the system socket API. An existing application can be re-compiled and linked against the UDT/CCC C library. In this way, the applications use our library transparently without any changes to the source codes. There is one limitation, though. UDT does not support multi-process models (e.g., using *fork* system call) due to efficiency considerations, so this method does not work if the existing application uses the same sockets in multiple processes.

4. EXPRESSIVENESS

To evaluate the expressiveness of UDT/CCC, we implement a set of representative control algorithms using the library. Any algorithms belonging to a similar set can be implemented in a similar way. Meanwhile, we show that the implementation is simple and easy to learn.

In this section, we describe in detail how to implement control algorithms of rate based UDP, TCP variants, including both loss-based and delay-based algorithms, and group transport protocols as well.

UDT/CCC uses an object-oriented design. It provides a base C++ class (CCC) that contains all the functions and event handlers described in Section 2.2. A new control algorithm can inherit from this class and redefine certain control event handlers.

The implementation of any control algorithm is to update at least one of the two control parameters: the congestion window size (*m_dCWndSize*) and the inter-packet time (*m_dPacketPeriod*), both of which are CCC class member variables.

4.1 Rate-based UDP

A rate-based reliable UDP library (CUDPBlast) is often used to transfer bulk data over private links. To implement this control mechanism, CUDPBlast initializes the congestion window with a very large value so that the window size will not limit the packet sending. The rest is to provide a method to assign a data transfer rate to a specific CUDPBlast instance. A piece of pseudo code is shown below:

```
class CUDPBlast: public CCC
{
public:
  CUDPBlast() {m_dCWndSize = 83333.0;}

  void setRate(int mbps)
  {
    m_dPktSndPeriod = (SMSS * 8.0) / mbps;
  }
}
```

By using *setsockopt* an application can assign CUDPBlast to a UDT socket and by using *getsockopt* the application can obtain a pointer to the instance of CUDPBlast being used by the UDT socket. The application can then call the *setRate* method of this instance to set or modify a fixed sending rate at any time.

4.2 Standard TCP (TCP NewReno)

As a more complex example, we further show how to use the UDT/CCC library to implement the standard TCP congestion control algorithm (CTCP). Because a large portion of new proposed congestion control algorithms are TCP-based, this CTCP class can be further inherited and redefined to implement more TCP variants, which we will describe in the next two subsections.

TCP is a pure window-based control protocol. Therefore, during initialization, the inter-packet time is set to zero. In addition, TCP need data packets to be acknowledged frequently, usually every one or two packets[1]. This is also configured in the initialization.

TCP does not need explicit loss notification, but uses three duplicate ACKs to indicate packet loss. Therefore, for congestion control, CTCP only redefined two event handlers: *onACK* and *onTimeout*. In *onACK*, CTCP detects duplicate ACKs and takes proper actions. Here is the pseudo code of the fast retransmit and fast recovery algorithm in RFC 2581:

```
virtual void onACK(const int& ack)
{
  if (three duplicate ACK detected)
  {
    // ssthresh = max{flight_size / 2, 3}
    // cwnd = ssthresh + 3 * SMSS
  }
  else if (further duplicate ACK detected)
  {
    // cwnd = cwnd + SMSS
  }
  else if (end fast recovery)
  {
    // cwnd = ssthresh
  }
  else
  {
    // cwnd = cwnd + 1/cwnd
  }
}
```

The CTCP implementation can provide more TCP event handlers such as *DupACKAction and ACKAction*, which will further reduce the work of implementing new TCP variants.

Note that here we are only implementing TCP's congestion control algorithm, but NOT the whole TCP protocol. The UDT/CCC library does not implement exactly the same protocol mechanisms as in the TCP specification but it does provide similar functionality. For example, TCP uses byte-based sequencing whereas UDT uses packet-based sequencing, but this should not prevent CTCP from simulating TCP's congestion avoidance behavior. Certain TCP mechanisms that can affect congestion control, such as SACK and RR, have their equivalents implemented in the UDT library. We believe these mechanisms will benefit most congestion control algorithms.

4.3 New TCP Algorithms (Loss-based)

New TCP variants that use loss-based approaches usually redefine the increase and decrease formulas of the congestion window size. Implementations of these protocols can simply inherit from CTCP and redefine proper TCP event handlers.

For example, to implement Scalable TCP, we can simply derive a new class from CTCP, and override the actions of increasing (by

[1] Although TCP uses accumulative acknowledgements, a TCP implementation usually acknowledges at the boundary of a data segment. This is equivalent to acknowledging a UDT data packet in CTCP.

0.1 instead of 1/*cwnd*) and decreasing (by 1/8 instead of 1/2) the congestion window size.

Similarly, we have also implemented HighSpeed TCP (CHS), BiC TCP (CBiC), and TCP Westwood (CWestwood).

4.4 New TCP Algorithms (Delay-based)

Delay-based algorithms usually need accurate timing information for each packet. For efficiency, UDT does not calculate RTT for each data packet because it is unnecessary for most control algorithms. However, this can be done by overriding *onPktSent* and *onACK* event handlers, where the time of packet sending and the arrival of its acknowledgement can be recorded. For algorithms preferring one-way delay (OWD) information, each UDT packets contains the sending time in its packet header, and a new algorithm can override *onPktReceived* to calculate OWD.

Using the strategy described above, we implement the TCP Vegas (CVegas) control algorithm. CVegas uses its own data structure to record packet departure timestamps and ACK arrival timestamps, and then calculates accurate RTT values. With simple modifications to the control formulas, we further implement FAST TCP (CFAST).

4.5 Group Transport Control

While we have demonstrated that UDT/CCC can be used to implement end-to-end unicast congestion control algorithms, we now show that it can also be used to implement group-based control mechanisms, such as CM and GTP.

To support this feature, the new algorithm class simply needs to implement a central manager to control a group of connections. The control parameters are calculated by the central manager and then fed back to the control class instance of each individual connection.

We implemented GTP (CGTP) as an example of group-based control mechanisms. The GTP protocol controls a group of flows with the same destination. CGTP tunes the packet sending rate at the receiver side periodically and feeds back the parameters using UDT/CCC's *sendCustomMsg* method.

4.6 Summary

We have implemented nine example algorithms using UDT/CCC, including rate-based reliable UDP, TCP and its variants, and group-based protocols. We demonstrated that our UDT/CCC library can support a large variety of congestion control algorithms, which are supported by only 8 event handlers, 4 protocol control functions, and 1 performance monitoring function.

The concise UDT/CCC API is easy to learn. In fact, it takes a small piece of code to implement most the algorithms described above. Table 1 lists the lines of code (LOC) of implementations of TCP algorithms using UDT/CCC, as well as the LOC of those native implementations (Linux kernel patches). The LOC value is estimated by the number of semicolons in the corresponding C/C++ code segment.

As a reference point, the UDT library has 3134 lines of effective code (i.e., excluding comments, blank lines, etc.), SABUL has 2670 lines of code, and the RBUDP library has approximately 2330 lines of code. While these numbers are not enough to reflect the complexity of implementing a transport protocol, the much smaller number of LOC values of UDT/CCC based implementation can indicate the simplicity of using UDT/CCC.

Table 1. Lines of Code (LOC) of implementations of TCP algorithms. *This table lists LOC of different TCP algorithms implemented using UDT/CCC and their respective Linux kernel patches (native implementations). The LOC of Linux patches include both added lines and removed lines.*

Protocol	UDT/CCC	Native
TCP	28	-
Scalable TCP	11	370
HighSpeed TCP	8	40
BiC TCP	38	430
TCP Westwood	27	182
TCP Vegas	37	247
FAST TCP	31	481

The class inheritance relationship of these UDT/CCC implemented algorithms can be found in Figure 7. Code reuse by class inheritance also contributes to the small LOC values of those TCP-based algorithms.

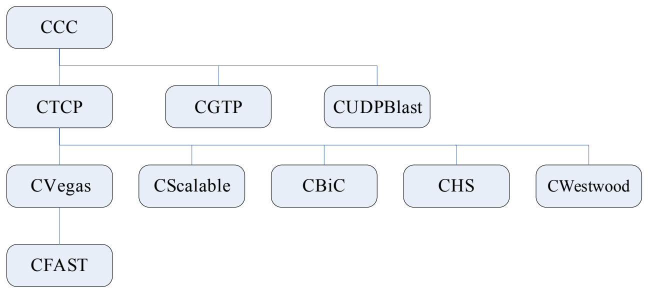

Figure 7. UDT/CCC based protocols. *This figure shows the class inheritance relationship among the control algorithms we implemented. Note that this is only for the purpose of code reuse, and it does NOT imply any other relationship among these algorithms.*

5. PERFORMANCE

In this section, we examine the performance characteristics of UDT/CCC. We focus on two important evaluations: 1) can UDT/CCC based implementations simulate the behaviors of their native implementation counterparts and 2) what is the additional CPU overhead introduced by this application implementation?

5.1 Similarity

It is a fundamental goal for UDT/CCC to simulate the performance of a control algorithm's native implementation or realize the algorithm's theoretical performance.

In most cases, congestion/flow control algorithms are the most significant factor that determines a protocol's performance-related behavior (throughput, fairness, and stability). Less significant factors include other protocol control mechanisms, such as RTT calculation, timeout calculation, acknowledgment interval, etc.

We have made most of these control mechanisms configurable through the CCC interface and the UDT protocol control interface. In this subsection we will show that a UDT/CCC based implementation demonstrates similar performance to a native implementation.

Since TCP is probably the most representative control protocol, we compared an application level TCP implementation using our UDT/CCC library (CTCP) against the standard TCP implementation provided by Linux kernel 2.4.18.

The experiment was performed between two Linux boxes between Chicago and Amsterdam. The link is 1 Gb/s with 110ms RTT and was reserved for our experiment only in order to eliminate cross traffic noises. Each Linux box has dual Xeon 2.4GHz processors and was installed with Linux kernel 2.4.18. We started multiple TCP and CTCP flows in separate runs, each of which was kept running for at least 60 minutes. The total TCP buffer size was set to at least the size of BDP (bandwidth delay product). Both TCP and CTCP experiments used the same testing program (except the connections were TCP and CTCP, respectively) with same configuration (buffer size, etc.).

We recorded the aggregate throughput (value between 0 and 1000 Mbps), fairness index (value between 0 and 1), and stability index (equal to or greater than 0) in Table 2. The definitions of the fairness index and stability index can be found in [1, 26]. The fairness index represents how fairly the bandwidth is shared by concurrent flows and larger values are better. The stability index describes the oscillations of the flows and smaller values mean less oscillation. These three measurements summarize most of the performance characteristics of a congestion control algorithm.

Table 2. Performance characteristics of TCP and CTCP with various parallel flows. *The table lists the aggregate throughput (in Mb/s), fairness index, and stability index of concurrent TCP and CTCP flows. Each row records an independent run with a different number of parallel flows.*

Flow	**Throughput**		**Fairness**		**Stability**	
#	TCP	CTCP	TCP	CTCP	TCP	CTCP
1	112	122	1	1	0.517	0.415
2	191	208	0.997	0.999	0.476	0.426
4	322	323	0.949	0.999	0.484	0.492
8	378	422	0.971	0.999	0.633	0.550
16	672	642	0.958	0.985	0.502	0.482
32	877	799	0.988	0.997	0.491	0.470
64	921	716	0.994	0.996	0.569	0.529

From Table 2, we find that TCP and CTCP have pretty similar throughput for small numbers of parallel flows. However, as the number of parallelism increases, CTCP stops increasing its throughput first and thus has a significantly smaller throughput than TCP when there are 64 parallel flows[2]. Further analysis indicates that the reason for this is that CTCP costs more CPU than kernel implemented TCP and with 64 flows the CPU time has been used up. To verify this assertion, we started another experiment using machines with dual AMD 64-bit Opteron processors and this time CTCP reaches more than 900Mbps at 64 parallel flows. The CPU usage problem will be further analyzed in Section 5.2.

[2] TCP throughput will also start to decrease as the number of parallel flows increases [12].

In spite of the CPU utilization limitation, both of the implementations have similar performance on fairness and stability. They both realize good fairness with near-one fairness indexes, as the AIMD algorithm indicates. The stability indexes are around 0.5 for all runs.

In addition to the experiments above, we have also tested several reliable UDP-based protocols such as UDP Blast (CUDPBlast) to examine if the UDT/CCC based implementation conforms to the protocol's theoretical performance. We also examined the performance of UDT/CCC in a real streaming merge application, in which the receiver (where data is merged) requests an explicit sending rate to the data sources. This service is provided by a specific control mechanism implemented using UDT/CCC. The results of these experiments were positive and expected performance was reached.

5.2 CPU Usage Overhead

While we have addressed the expressiveness and similarity issues, there is one last major concern in using the UDT/CCC library: how much is the overhead brought in by UDT/CCC?

Before we go into the details, we show the CPU usages (percentage of CPU time used by TCP and CTCP) of the experiments in Table 2, Section 5.1. The result is listed in Table 3. Because there are two processors in each of our testing machine, the percentage varies between 0% and 200%.

Table 3. CPU usage of TCP and CTCP with various parallel flows. *The table lists the CPU usage percentage of both the sender and receiver sides. Each row records the data for the two runs of TCP and CTCP, respectively. The maximum CPU usage percentage is 200%.*

Flow	**Sender**		**Receiver**	
#	**TCP**	**CTCP**	**TCP**	**CTCP**
1	9.5	16.7	10.1	13.0
2	18.6	33.9	19.8	24.4
4	31.2	58.2	20.9	26.5
8	35.0	71.3	38.6	44.0
16	62.5	122.8	69.6	73.1
32	88.1	198.0	90.5	105.7
64	93.2	198.0	91.9	94.2

Table 3 shows that at the receiver side, CTCP has very good CPU usage compared to the Linux TCP implementation. However, at the sender side CTCP has a much higher CPU usage; this is why CTCP stops increasing its throughput after 32 parallel flows in Table 2.

The major performance overhead added by an application level implementation comes from additional memory copy between application buffer and protocol buffer and additional context switches by packet processing and threading.

UDT has a best-effort method to reduce the additional memory copy and in the best case, additional memory copy will be completely eliminated.

However, UDT can do little for context switches caused by packet processing. The number of packets (for both data and control information) is decided by how much data applications need to transfer, whereas the number of control packets also depends on the specific protocol. For example, most reliable UDP protocols (SABUL, RBUDP, etc) only feed back an acknowledgement at the end of a large data block, whereas TCP needs to acknowledge arrived packets more frequently, at about every 1 or 2 packets.

Our more detailed experiments discovered two major overheads added by CTCP compared to TCP. One is from acknowledging, including the subsequent overheads of context switches, buffer and sequence number updating, and thread synchronizations. The other is from application level threading.

In the following experiments, we increased the acknowledgement interval of our CTCP implementation, and, accordingly, the increases per ACK at the sender side. We run the experiments with the same setup as Table 2 in Section 5.1. The CPU usages are recorded in Table 4.

In Table 4 we use MHz/Mbps to describe CPU utilization. Because we cannot force TCP or CTCP to output a predictable throughput, we need to consider the data throughput when comparing CPU utilizations. The measurement of MHz/Mbps equals *CPU percentage * CPU frequency (MHz) / throughput (Mbps)*. Note that both CPU percentage and MHz/Mbps are NOT generic measurements. That is, these values are only comparable against those values obtained on the same system, or at least systems with the same configuration.

Table 4. CPU utilization of CTCP against number of parallel flows and ACK intervals. *This table lists the CPU utilization (MHz/Mbps) of CTCP with different numbers of parallel flows and different numbers of ACK intervals. Note that the first column (ACK interval = 2) is the result from Table 3 and it is listed here for comparison*

Flow	ACK Intervals						
#	*2*	*4*	*8*	*16*	*32*	*64*	*128*
1	3.28	3.15	3.20	3.43	2.57	2.59	2.07
2	3.91	3.77	3.95	3.59	3.52	3.35	3.51
4	4.32	4.36	1.45	3.08	3.54	3.44	3.27
8	4.05	4.87	4.32	3.84	3.91	3.63	3.63
16	4.59	5.07	5.60	4.41	4.41	4.17	3.12
32	5.41	5.31	5.27	4.99	5.15	4.53	4.01
64	6.63	6.58	6.15	5.89	5.35	5.08	4.51

Table 4 shows that as the number acknowledgments decreases, the CPU usage drops sharply. For the same reason, less frequent acknowledging is recommended when designing an application level protocol for purpose of efficiency.

Meanwhile, the MHz/Mbps measurement increases as the number of flows increases, but this is insignificant for TCP experiments. The situation, however, is caused by the user level threads used by CTCP. CTCP needs to start at least one thread for each connection, whereas TCP realizes the multiplexing in the kernel and does not have this overhead. This situation is worse for the sender side because each CTCP sender has to start two threads (UDT sender and receiver, see Section 2.3 and 3.1) and to deal with the thread synchronization caused by packet acknowledging.

In fact, through profiling analysis we found that UDP IO, threading synchronization, and control event handling contributes more than 90% of CPU utilization of CTCP. The fourth most significant CPU consumer is timing, which takes about 5% of overall CPU utilization on stamping each packet and triggering timer related events.

Although the CPU overhead of UDT/CCC may limit its usage in certain scenarios, we argue that the library is still useful in many other situations. First, in high performance computing, there are usually only a small number of flows sharing the high bandwidth. In this case, the threading overhead is low. Second, the overhead can be overcome by more powerful processors and more machines. Third, the overhead of UDT/CCC only exists when compared to kernel space implementations; it is insignificant when compared to other user space implementations. Finally, as we have shown in this section, control mechanisms with less frequent acknowledging would result in much less overhead.

We are, however, quite aware of the importance of CPU efficiency. Our current work is focused on code optimization.

We have also performed the same experiments on similarity and performance on other systems with different operating systems (Linux, BSD, OS X, Windows XP), hardware (Intel, AMD, and PowerPC processors), and networks. Although specific systems have more or less impact on the results, all the experiments conform to the similarity and performance trends we obtained from the experiments described in this section.

6. RELATED WORK

There are few user level protocol stacks that provide a programming interface for user-defined congestion control algorithms as UDT/CCC does.

The Globus XIO [39] library has somewhat similar objectives, but the approach is quite different. XIO implements a set of primitive protocol components and APIs for fast creation or prototyping new protocols. XIO can support more protocols than UDT/ CCC, however, UDT/CCC is much simpler to learn and use.

Less similar user level libraries include several user level TCP implementations [15, 16, 17, 18, 19]. One particular implementation is the Alpine [15] library. Alpine is an attempt to move the entire kernel protocol stack into the user space, and provides (almost) transparent application interfaces at the same time. None of these libraries provide programmable interfaces.

In kernel space, the most similar work to UDT/CCC is probably the icTCP [4] library. It exposes key TCP parameters and provides controls to these parameters to allow new TCP algorithms deployed in user space. Despite the different nature of kernel and user space implementations, icTCP limits the update on TCP controls only, whereas UDT/CCC supports a broader set of protocols. Similar work as icTCP also includes Web100/Net100 [6] and CM.

Another work, STP [3], has more radical changes but also has more powerful expression ability. The STP's approach is to provide a set of protocol implementation APIs in a sandbox. Meanwhile, STP itself is a protocol that supports run time code upgrading; thus, new protocols or algorithms can be deployed implicitly. To address the security problem arising from untrusted code, STP involves a complex security mechanism.

Yet another more complex library is the *x*-kernel [13]. *x*-kernel is an OS kernel designed to support data transport protocol implementations. The support mechanism of *x*-kernel is a modular based system and it is more decomposed than STP. Besides the support of protocol implementation, *x*-kernel has many optimizations inside the OS kernel for data communications.

Other modularized approaches include Horus [14], CTP [22] and its high performance successor [23].

While some of these in-kernel libraries may have performance and transparency advantages, their goals of fast deployment of new protocols/algorithms are compromised by the difficulty of getting themselves deployed. For example, *x*-kernel has been proposed for more than a decade and it still remains a research tool. In contrast, UDT/CCC library provides a very practical solution for the time being.

In addition, kernel space approaches need to protect their host systems and the network from security problems and they have to limit users' privileges to control the protocol behavior. For example, both STP and icTCP prevent new algorithms from utilizing more bandwidth than standard TCP. Such limitations are improper to the new control algorithms for high-speed networks such as Scalable, HighSpeed, BiC, and FAST. The security problem is much less serious for UDT/CCC because it is at user space and it is only installed as needed (in contrast, those libraries such as icTCP and STP will be accessible to every user if they are accepted by OS vendors).

Finally, there is another category of related work that attempts to provide a protocol language for easier, faster, or more readable protocol implementation. Such work includes Prolac [20] and FoxNet [21].

7. CONCLUSION

The maturity of high-speed wide area networks encouraged the emergence of numerous new applications, and new control mechanisms supporting these applications as well. It has often been the case that implementing these new control algorithms in the kernel is not practical or proper. On the one hand, the wide deployment of new protocols or algorithms usually suffers long time lag. On the other hand, OS vendors may only choose a very small number of protocols to implement in kernel.

We have presented a user level transport protocol stack named UDT/CCC, which allows user defined congestion control algorithms to be easily implemented. Our UDT/CCC library enables easy implementations of a large variety of control algorithms while these implementations can still match the performance characteristics of those native implementations.

However, UDT/CCC is not meant to replace kernel protocol stacks or proposed as a mean to implement any new protocols. Instead, it provides a practical alternative when a kernel space approach is difficult to implement, evaluate, and deploy. These use scenarios include the implementation of a new application or network specific congestion mechanism and the evaluation of new congestion control algorithms.

Finally, we are aware of the CPU utilization limitations of the current UDT/CCC implementation. On the one hand, there are unavoidable efficiency side effects for application level protocol implementations because of the additional memory copy and context switches. On the other hand, we will continue to optimize the implementation to minimize these negative impacts.

The UDT project, including the feature described in this paper, is meant for productivity use, rather than just for research or prototyping. The UDT library is open source software and can be obtained from http://udt.sf.net.

8. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant ANI 9977868 and the Department of Energy under grant DE-FG02-04ER25639.

9. REFERENCES

[1] Yunhong Gu, Xinwei Hong, and Robert Grossman: *Experiences in Design and Implementation of a High Performance Transport Protocol*, SC 2004, Nov 6 - 12, Pittsburgh, PA, USA.

[2] W. Feng and P. Tinnakornsrisuphap: *The Failure of TCP in High-Performance Computational Grids*, SC 2000, Dallas, TX, Nov. 00.

[3] Parveen Patel, Andrew Whitaker, David Wetherall, Jay Lepreau, and Tim Stack: *Upgrading Transport Protocols using Untrusted Mobile Code*, in Proceedings of the 19th ACM Symposium on Operating System Principles*, October* 19-22, 2003.

[4] Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau: *Deploying Safe User-Level Network Services with icTCP*, OSDI 2004.

[5] David G. Andersen, Deepak Bansal, Dorothy Curtis, Srinivasan Seshan, and Hari Balakrishnan: *System Support for Bandwidth Management and Content Adaptation in Internet Applications*, Proc. 4th USENIX Conference on Operating Systems Design and Implementation (OSDI 2000), San Diego, CA, October 2000.

[6] M. Mathis, J. Heffner, and R. Reddy: *Web100: Extended TCP Instrumentation for Research, Education and Diagnosis*, ACM Computer Communications Review, Vol 33, Num 3, July 2003.

[7] R. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson: *Stream control transmission protocol*. RFC 2960, Oct. 2000.

[8] Eddie Kohler, Mark Handley, Sally Floyd, Jitendra Padhye: *Datagram Congestion Control Protocol (DCCP)*, http://www.icir.org/kohler/dcp/. Jan. 2005.

[9] Sally Floyd, Mark Handley, Jitendra Padhye, and Joerg Widmer: *Equation-Based Congestion Control for Unicast Applications*, ACM SIGCOMM 2000, Stockholm, Aug. 2000.

[10] M. Allman, V. Paxson, W. Stevens, *TCP Congestion Control*, IETF, RFC 2581, April 1999.

[11] V. Paxson and M. Allman: *Computing TCP's Retransmission Timer*, RFC 2988, IETF, Nov. 2000.

[12] Harimath Sivakumar, Stuart Bailey, Robert L. Grossman. *PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks*, SC 2000, Dallas, TX, Nov. 2000.

[13] N. C. Hutchinson and L. L. Peterson: *The x-Kernel: An architecture for implementing network protocols*, IEEE

Transactions on Software Engineering, 17(1): 64-76, Jan. 1991.

[14] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr: *A framework for protocol composition in Horus*, in Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, pages 80-89, Ottawa, Ontario, Canada, 2-23 Aug. 1995.

[15] David Ely, Stefan Savage, and David Wetherall: *Alpine: A user-level infrastructure for network protocol development*, in Proc. 3rd USENIX Symposium on Internet Technologies and Systems (USITS 2001), pages 171-183, March 2001.

[16] Thekkath, C. A., Nguyen, T. D., Moy, E., and Lazowska, E. D: *Implementing network protocols at user level*, IEEE/ACM Transactions on Networking, 1(5): 554--565, October 1993.

[17] A. Edwards and S. Muir: *Experiences Implementing A High-Performance TCP In User-Space*, in Proc. ACM SIGCOMM 1995, Cambridge, MA, pages 196 - 205.

[18] Prashant Pradhan, Srikanth Kandula, Wen Xu, Anees Shaikh, Erich Nahum: *Daytona: A User-Level TCP Stack*, http://nms.lcs.mit.edu/~kandula/data/daytona.pdf.

[19] Kieran Mansley: *Engineering a user-level TCP for the CLAN network*, in SIGCOMM 2003 workshop on Network-I/O convergence: experience, lessons, implications.

[20] E. Kohler, M. F. Kaashoek, and D. R. Montgomery: *A Readable TCP in the Prolac Protocol Language*, in Proceedings of SIGCOMM '99, pages 3–13, Cambridge, Massachusetts, Aug. 1999.

[21] Edoardo Biagioni: *A structured TCP in Standard ML*, in Proceedings of the ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications, pages 36-45, London, England, 1994.

[22] Gary T. Wong, Matti A. Hiltunen, and Richard D. Schlichting: *A configurable and extensible transport protocol*, IEEE Infocom 2001, April 22-26, 2001. Anchorage, Alaska, April 2001.

[23] Xinran Wu, Andrew A. Chien, Matti A. Hiltunen, Richard D. Schlichting and Subhabrata Sen, *High Performance Configurable Transport Protocol for Grid Computing*, in Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005).

[24] M. Gerla, M. Y. Sanadidi, R. Wang, A. Zanella, C. Casetti, and S. Mascolo. *TCP Westwood: Congestion Window Control Using Bandwidth Estimation*. IEEE Globecom 2001, Volume: 3, pp 1698-1702.

[25] L. Brakmo and L. Peterson. *TCP Vegas: End-to-End Congestion Avoidance on a Global Internet*. IEEE Journal on Selected Areas in Communication, Vol 13, No. 8 (October 1995) pages 1465-1480.

[26] C. Jin, D. X. Wei, and S. H. Low. *FAST TCP: motivation, architecture, algorithms, performance*. IEEE Infocom '04, Hongkong, China, Mar. 2004.

[27] S. Floyd. *HighSpeed TCP for Large Congestion Windows*. RFC 3649, Experimental Standard, Dec. 2003.

[28] T. Kelly. *Scalable TCP: Improving Performance in Highspeed Wide Area Networks*. ACM Computer Communication Review, Apr. 2003.

[29] L. Xu, K. Harfoush, and I. Rhee. *Binary Increase Congestion Control for Fast Long-Distance Networks*. IEEE Infocom '04, Hongkong, China, Mar. 2004.

[30] Y. Gu and R. L. Grossman: *SABUL: A Transport Protocol for Grid Computing*. Journal of Grid Computing. 2003, Volume 1, Issue 4, pp. 377-386.

[31] E. He, J. Leigh, O. Yu, T. A. DeFanti: *Reliable Blast UDP: Predictable High Performance Bulk Data Transfer*, IEEE Cluster Computing 2002, Chicago, IL 09/01/2002.

[32] C. Xiong, Leigh, J., He, E., Vishwanath, V., Murata, T., Renambot, L., DeFanti, T.: *LambdaStream - a Data Transport Protocol for Streaming Network-intensive Applications over Photonic Networks*, Third International Workshop on Protocols for Long-Distance Networks (PFLDnet 2005), Lyon, France, Feb. 2005.

[33] M. Veeraraghavan, X. Zheng, H. Lee, M. Gardner, W. Feng, *CHEETAH: Circuit-switched High-speed End-to-End Transport ArcHitecture*, Proc. of Opticomm 2003, Oct. 13-17, 2003. Dallas, TX.

[34] A. Szalay, J. Gray, A. Thakar, P. Kuntz, T. Malik, J. Raddick, C. Stoughton. J. Vandenberg: *The SDSS SkyServer - Public Access to the Sloan Digital Sky Server Data*, ACM SIGMOD 2002.

[35] Naveen Krishnaprasad, Venkatram Vishwanath, Shalini Venkataraman, Arun G.Rao, Luc Renambot, Jason Leigh, Andrew E.Johnson: *JuxtaView - A Tool for Interactive Visualization of Large Imagery on Scalable Tiled Displays*, in the proceedings of IEEE Cluster 2004, San Diego, CA, September 20-23, 2004.

[36] Ryan Wu and Andrew Chien: *GTP: Group Transport Protocol for Lambda-Grids*, in Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2004), Chicago, Illinois, April 2004.

[37] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A., *TCP Selective Acknowledgement Options*. RFC 2018, April 1996.

[38] Zhang, M., Karp, B., Floyd, S., and Peterson, L., *RR-TCP: A Reordering-Robust TCP with DSACK*, in Proceedings of the Eleventh IEEE International Conference on Networking Protocols (ICNP 2003), Atlanta, GA, November, 2003.

[39] Globus XIO: http://www-unix.globus.org/toolkit/docs/3.2/xio/index.html. Retrieved on Apr. 3, 2005.

[40] E-VLBI, http://web.haystack.edu/e-vlbi/evlbi.html. Retrieved on Apr. 6, 2005.

[41] Qishi Wu, Nageswara S. V. Rao, *Protocol for High-Speed Data Transport Over Dedicated Channels*, Third International Workshop on Protocols for Long-Distance Networks (PFLDnet 2005), Lyon, France, Feb. 2005.

[42] Yunhong Gu and Robert L. Grossman, *UDT: A Transport Protocol for Data Intensive Applications*. Internet Draft. Work in progress.