UDTv4: Improvements in Performance and Usability

Yunhong Gu and Robert Grossman
National Center for Data Mining, University of Illinois at Chicago

ABSTRACT

This paper presents UDT version 4 (UDTv4), the fourth generation of the UDT high performance data transfer protocol. The focus of the paper is on the new features introduced in version 4 during the past two years to improve the performance and usability of the protocol.

UDTv4 introduces a new three-layer protocol architecture (connection-flow-multiplexer) for enhanced congestion control and resource management. The new design allows protocol parameters to be shared by parallel connections and to be reused by future connections. This improves the congestion control and reduces the connection setup time. Meanwhile, UDTv4 also provide better usability by supporting a broader variety of network environments and use scenarios.

1. INTRODUCTION

During the last decade there has been a marked boom in Internet applications, enabled by the rapid growth of raw network bandwidth. Examples of new applications include P2P file sharing, streaming multimedia, and grid/cloud computing. These applications vary greatly in traffic and connection characteristics. However, most of them still use TCP for data transfer. This is partly due to the fact that TCP is well established and contributes to the stability of the Internet.

TCP was designed as a general-purpose protocol and was first introduced three decades ago. It is not surprising that certain requirements from new applications cannot be perfectly addressed by TCP. Network researchers have proposed many changes to TCP to address those emerging problems and requirements (SACK, ECN, etc.) [6]. The new techniques are carefully studied and deployed, albeit slowly. For example, TCP's inefficiency problem in high bandwidth-delay product (BDP) networks was observed almost a decade ago yet it is only recently that several new high speed TCP variants were deployed: (CUBIC on Linux [13] and Compound TCP on Windows Vista [17]). Furthermore, because new TCP algorithms have to be compatible with the TCP standard, improvements to TCP are limited.

New transport protocols, DCCP [12] and SCTP [16], have also been proposed. However, it may take years for these new protocols to be widely deployed and used by applications (considering the example of IPv6). Moreover, both DCCP and SCTP are designed for specific groups of applications. New applications and requirements will continue to emerge and it is not a scalable solution to design a new transport layer protocol every few years. It is necessary to have a flexible protocol that provides basic functions and allows applications to define their own data processing. This is what UDP was designed for.

In fact, UDP has been used in many applications (e.g., Skype) but it is usually customized independently for each application. RTP [15] is a good example and it is a great success in supporting multimedia applications. However, there are few general-purpose UDP-based protocols that application developers can use directly or customize easily.

UDT, or UDP-based data transfer protocol, is an application level general-purpose transport protocol on top of UDP [8]. UDT address a large portion of the requirements from the new applications by seamlessly integrating many modern protocol design and implementation techniques at the application level.

The protocol was originally designed for transferring large scientific data over high-speed wide area networks and it has been successful in many research projects. For example, UDT has been used to distribute the 13TB SDSS astronomy data release to global astronomers [9].

UDT has been an open source project since 2001 and the first production release was made in 2004. While it was originally designed for big scientific data sets, the UDT library has been used in many other situations, either with its stock form or in a modified form. A great deal of user feedback has been received. The new version (UDTv4) released in 2007 introduces significant changes and supports better performance and usability.

- UDTv4 uses a three-layer architecture to enhance congestion control and reduce connection setup time by sharing control parameters among parallel connections and by using historical data.
- UDTv4 introduces new techniques in both protocol design and implementation to support better scalability, hence it can be used in a larger variety of use scenarios.

This paper describes these new features of UDTv4. Section 2 explains the protocol design. Section 3 describes several key implementation techniques. Section 5 presents the evaluation. Section 6 discusses the related work. Section 7 concludes the paper. Throughout the rest of the paper, we use UDT to refer the most recent version, UDTv4, unless otherwise explicitly stated.

2. PROTOCOL DESIGN

2.1 Protocol Overview

UDT is a connection-oriented, duplex, and unicast protocol. There are 3 logical layers in design: UDT connection, UDT flow, and UDP multiplexer (Figure 1).

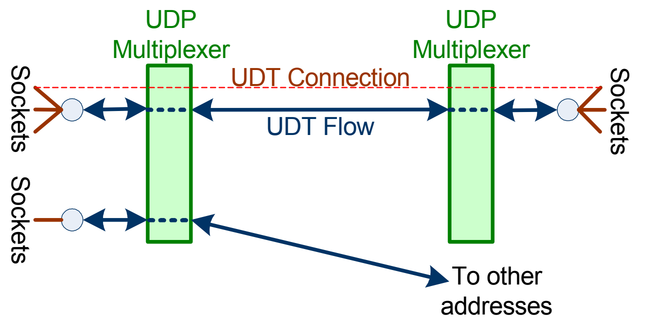

Figure 1. UDT Connection, Flow, and UDP Multiplexer.

A UDT connection is set up between a pair of UDT sockets as a distinct data transfer entity to applications. It can provide either reliable data streaming services or partial reliable messaging services, but not both for the same socket.

A UDT flow is a logical data transfer channel between two UDP addresses (IP and port) with a unique congestion control algorithm. That is, a UDT flow is composed of five elements (source IP, source UDP port, destination IP, destination UDP port, and congestion control algorithm). The UDT flow is transparent to applications.

One or more UDT connections are associated with one UDT flow, if the UDT connections share the same five elements described above. Every connection must be associated with one and only one flow. In other words, UDT connections sharing the same five elements are multiplexed over a single UDT flow.

A UDT flow provides reliability control as it multiplexes individual packets from UDT connections, while UDT connections provide data semantics (streaming or messaging) management. Different types of UDT connections (streaming or messaging) can be associated with the same UDT flow.

Congestion control is also applied to the UDT flow, rather than the connections. Therefore, all connections in one flow share the same congestion control process. Flow control, however, is applied to each connection.

Multiple UDT flows can share a single UDP socket/port and a UDP multiplexer is used to send and dispatch packets for different UDT flows. The UDP multiplexer is also transparent to applications.

2.2 UDP Multiplexing

Multiple UDT flows can bind to a single UDP port and each packet is differentiated by the destination (UDT) socket ID carried in the packet header. The UDP multiplexing method helps to traverse firewalls and alleviates the system limitation on the port number space. The number of TCP ports is limited to 65536. In contrast, UDT can support up to 2^{32} connections at the same time.

UDP multiplexing also helps firewall traversing. By opening one UDP port, a host can open virtually an unlimited number of UDT connections to the outside.

2.3 Flow Management

UDT multiplexes multiple connections into one single UDT flow, if the connections share the same attributes of source IP, source UDP port, destination IP, destination UDP port, and congestion control algorithm.

This single flow for multiple connections helps to reduce control traffic, but more importantly, it uses a single congestion control for all connections sharing the same end points. This removes the unfairness by using parallel flows and in most situations it improves throughput because connections in a single flow coordinate with each other rather than compete with each other.

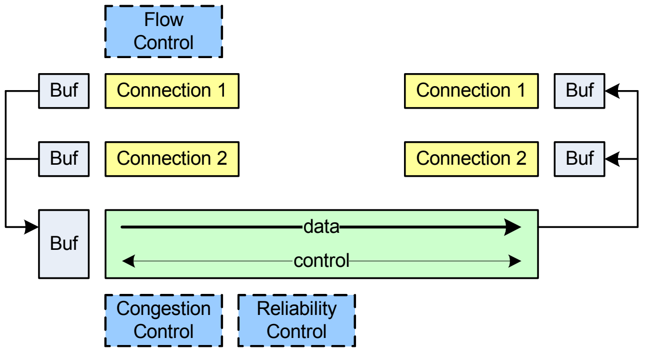

Figure 2. UDT Flow and Connection.

As shown in Figure 2, the flow maintains all activities required for a regular data transfer connection, whereas the UDT connection is only responsible for the application interface (connection maintenance and data semantics).

At the sender side, the UDT flow reads packets from each associated connection in a round robin manner, assigns each packet the flow sequence numbers and sends them out.

2.4 Connection Record Index/Cache

When a new connection is requested, UDT needs to look up whether there is already a flow existing between the same peers. A connection record index (Figure 3) is used for this purpose.

The index is sorted by the peer IP addresses. Each entry records the information between the local host and the peer address, including but not limited to RTT, path MTU, and estimated bandwidth. Each entry may contain multiple sub-entries by different ports, followed by multiple flows differentiated by congestion control (CC).

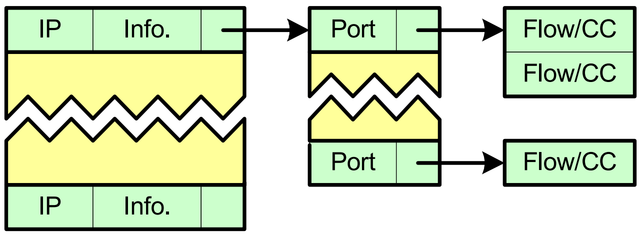

Figure 3. Connection Record Index.

The connection record index caches the IP information (RTT, MTU, estimated bandwidth, etc.) even if the connection and flow is closed, in which case there is no port associated with the IP entry. This information can be used when a new connection is set up. Its RTT value can be initialized with a previously recorded value; otherwise it would take several ACKs to get an accurate value for the RTT. If path MTU discovery is used, the MTU information can also be initialized with a historical value.

The index entry without an active flow will be removed when the maximum length of the index has been reached, and the oldest entry will be removed first.

Although the cache may be removed very quickly on a busy server (e.g., a web server), the client side may contain the same cache and pass the values to the server. For example, a client that frequently visits a web server may keep the link information between the client and the server, while the server may have already removed it.

2.5 Garbage Collection

When a UDT socket is closed (either by the application or because of a broken connection), it is not removed immediately. Instead, it is tagged as having closed status. A garbage collection thread will periodically scan the closed sockets and remove the sockets when no API is accessing the socket.

Without garbage collection, UDT would have needed stronger synchronization protection on its APIs, which increases implementation complexity and adds some slight overhead for the additional synchronization mechanism.

In addition, because of the delayed removal, a new socket can reuse a closed socket and the related UDP multiplexer when possible, thus it improves connection setup efficiency.

Garbage collection also checks the buffer usage and decreases the size of the system allocated buffer if necessary. If during the last 60 seconds, less that 50% of the buffer is used, the buffer will be reduced to half (a minimum size limit, 32 packets, is used so that the buffer size will not be decreased to a meaningless 1-byte).

3. IMPLEMENTATION

UDT is implemented as an open source project and is available for download from SourceForge.net. The UDT library has been used in both research projects and commercial products. So far 18,000 copies have been downloaded, excluding direct checkout from the CVS and redistribution from other websites.

The UDT implementation is available on both POSIX and Windows systems and it is thoroughly tested on Linux 2.4, 2.6, and Windows XP. The code is written in C++ with API wrappers for other languages available.

The latest stable version of the UDT library (version 4.2) consists of approximately 11,500 lines of C++ code, including about 4000 semicolons and about 20% of the code is comments.

3.1 Software Architecture

Figure 4 shows the software architecture of the UDT implementation. A global UDT API module dispatches requests from applications to a specific UDT socket. Data transfer for the UDT socket is managed by a UDT flow, while the UDT flow communicates via a UDP multiplexer. One UDP multiplexer can support multiple UDT flows, and one UDT flow can support multiple UDT sockets. Finally, both the buffer management module and the garbage collection module work at global space to support the resource management.

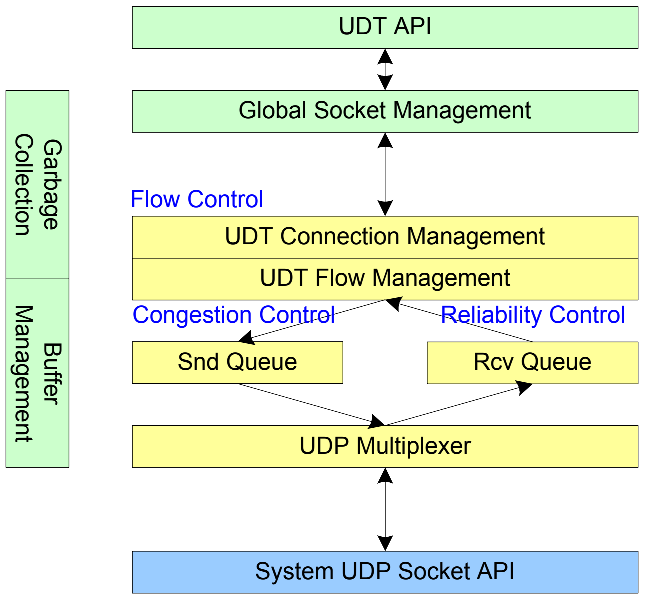

Figure 4. UDT Software Architecture.

Figure 5 shows the data flow in a single UDT connection. The UDT flow moves data packets from the socket buffer to its own sending buffer and sends the data out via the UDP multiplexer. The control information is exchanged on both directions of the data flow. At the sender side, the UDP multiplexer receives the control information (ACK, NAK, etc.) from the receiver and dispatches the control information to the corresponding UDT flow or connection. Lost lists are used at both sides to record the lost packets.

Lost lists work at flow level and only record flow sequence numbers. Flow control is applied to a UDT socket, while congestion control and reliability control are applied to the UDT flow.

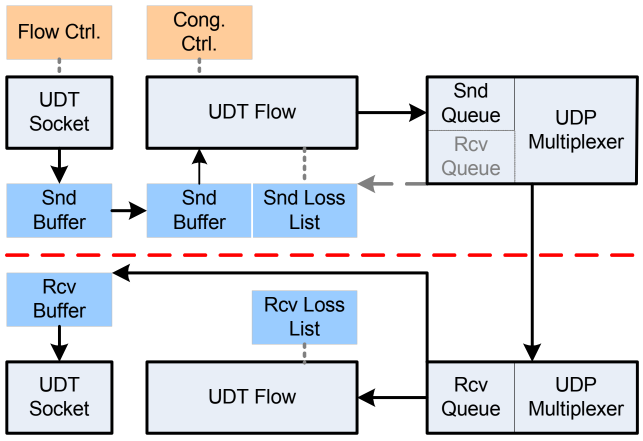

Figure 5. Data Flow over a Single UDT Connection.

3.2 UDP Multiplexer and Queue Management

The UDP multiplexer maintains a sending queue and a receiving queue. The queue manages a set of UDT flows to send or receive packets via the associated UDP port.

The sending queue contains a set of UDT flows that has data to send out. If rate based control is used, the flows are scheduled according to the next packet sending time; if pure window-based control is used, the flows are scheduled according to a round robin scheme.

The sending queue checks the system time and when it is time to send out the first packet, it removes the first flow on the queue and sends out its packet. If there are more packets to be sent for the particular flow, the flow will be inserted into the queue again according to the next packet sending time by rate/congestion/flow control.

The sending queue uses a heap structure to maintain the flows. With the heap structure, each *send* or *insert* action takes at most $\log_2(n)$ steps, where n is the total number of flows in the queue. The heap structure guarantees that the sender can find the flow instance with the smallest next scheduled packet sending time; however, it is not necessary to have all the flows sorted by the next scheduled time.

The job of the receiving queue is much simpler. It checks the timing events (retransmission timer, keep-alive, timer-based ACK, etc.) for each flow associated with the UDP multiplexer. Every fixed time interval (0.1 second), flows are checked in a round robin manner. However, if a packet arrived for a particular flow, the timers will be checked for the flow and the flow is moved to the end of the queue for the next round of check.

The receiving queue uses a double linked list to store the flows and each operation takes O(1) time.

The receiving side of the UDP multiplexer also maintains a hash table for the associated UDT connections, so that when a packet arrives, the multiplexer can quickly look up the corresponding connection to process the packet. Note that the flow processing handler can be looked up via the socket instance.

3.3 Connection and Flow Management

In the UDT implementation, a flow is a special connection that contains pointers to all connections within the same flow, including itself.

The first connection of the flow is set up by the normal 3-way handshake process. More connections are set up by a simplified 2-way handshake as it joins an existing flow. The first connection automatically becomes the flow and manages all the connections. If the current "flow" connection is closed or leaves (because of IP address change), another connection will become the flow and related flow information will be moved to the new flow from the old one.

The flow maintains a separate sending buffer in addition to the connections' sending buffers. In an ideal world, the flow should read packets from each connection in a round robin fashion. However, in this way the flow would either need to keep track of the source of each packet or copy the packet into its own buffer, because each ACK or NAK processing needs to locate the original packet.

In the current implementation, the socket sending buffer is organized as a link of multiple 32-packet blocks. The UDT flow reads one 32-packet block from each connection in round robin fashion, removes the block from the socket's sending buffer, and links the block to its own (flow) sending buffer. Note that there may be less than 32 packets in the block if there is not enough data to be sent for a particular connection.

Flow control is enforced at the socket level. The UDT *send* call will be blocked if either the sender buffer limit or the receiver buffer limit is full. This guarantees that data in the flow sending buffer is not limited by flow control.

By using this strategy, the flow simply applies ACKs and NAKs to its own buffer and avoids memory copies between flow and connections or a data structure to map flow sequence number to connection sequence number. In the latter case, UDT would also need to check every single packet being acknowledged, because they may belong to different connections and may not be continuous.

At the receiver side, all connections have their own receiver buffer for application data reading. However, only the flow maintains a loss list to recover packet losses.

Rendezvous connection setup. In addition to the regular client/server mode, UDT provides a method for rendezvous connection method. Both peers can connect to each other at (approximately) the same time, provided that they know the peer's address beforehand (e.g., via a 3rd known server).

3.4 Performance Considerations

Multi-core processing. The UDT implementation uses multiple threads to explore the multi-core ability of modern processors. Network bandwidth increases faster than CPU speed, and a single core of today's processors is barely enough to saturate 10Gb/s.

One single UDT connection can use 2 cores (sending and receiving) per data traffic direction on each side. Meanwhile, each UDP multiplexer has its own sending thread and receiving thread. Therefore, users can start more UDT threads by binding UDT sockets to different UDP ports, thus more UDP multiplexers will be started and each multiplexer will start their own packet processing threads.

New *select* API. UDT provides a new version of the *select* API, in which the result socket descriptor set is an independent output, rather than overwriting the input directly. The BSD style *select* API is inefficient for large numbers of sockets, because the input is modified and applications have to reinitialize the input each time. In addition, UDT provides a way to iterate the result set; in contrast, for the BSD socket API, applications have to test each socket against the result set.

New *sendfile*/*recvfile* API. UDT provides both *sendfile* and *recvfile* APIs to reduce one memory copy by exchanging data between the UDT buffer and application file directly. These two APIs also simplify application development in certain cases.

It is important to mention that file transfer can operate under both streaming mode and messaging mode. However, messaging mode is more efficient in this case, because *recvfile* does not require continuous data block receiving and therefore in messaging mode data blocks can be read into files out of order without the "head of line" blocking problem. This is especially useful when the packet loss rate is high.

Buffer auto-sizing. All UDT connections/flows share the same buffer space, which increases when necessary. The UDT socket buffer size is only an upper limit and it does not allocate the buffer until it has to.

UDT automatically increases the socket buffer size limit to 2*BDP, if the default or user-specified buffer size is less than this value. However, if the default or user-specified value is greater than this value, UDT will not decrease the buffer size. The bandwidth value (B in BDP) is estimated by the maximum packet arriving rate at the receiver side. The garbage collection thread may decrease the system buffers when it detects that only less than half of the buffers are used.

4. EVALUATION

This section evaluates UDT's scalability, performance, and usability. UDT provides superior usability over TCP and although it is at the application level, its implementation efficiency is comparable to the highly optimized Linux TCP implementation in kernel space. More importantly, UDT effectively addresses many application requirements and fills a blank left by transport layer protocols.

4.1 Performance Characteristics

This section summarizes the performance characteristics of UDT, in particular, its scalability.

Packet header size. UDT consumes 24 bytes (16-byte UDT + 8-byte UDP) for data packet headers. In contrast, TCP uses a 20-byte packet header, SCTP uses a 28-byte packet header, and DCCP uses 12 bytes without reliability.

Control traffic per flow. UDT sends one ACK per 0.01 second when there is data traffic. This can be overridden by a user-defined congestion control algorithm, if more ACKs are necessary. However, the user-defined ACKs will be lightweight ACKs and consumes less bandwidth and CPU [8]. ACK2 packet is generated occasionally, at a decreased frequency (up to 1 ACK2 per second). In contrast, TCP implementations usually send one ACK every one or two segments.

In addition, UDT may also send NAKs, message drop request, or keep-alive packets when necessary, but these packets are much less frequent than ACK and ACK2.

Limit on number of connections. The maximum number of flows and connections supported by UDT is virtually only limited by system resources (2^{32}).

Multi-threading. UDT starts 2 threads per UDP port, in addition to the application thread. Users can control the number of data processing threads by using a different number of UDP ports.

Summary of data structures. At the UDP multiplexer level, UDT maintains the sending queue and receiving queue. The sending queue costs $O(\log_2 n)$ time to insert or remove a flow, where n is the total number of flows. The receiving queue checks timers of each UDT flow every 0.1 second, but it is self clocked by the arrival of packets. Each check costs O(1) time. Finally, the hash table used for the UDP multiplexer to locate a socket costs O(1) look up time.

The UDT loss list is based on congestion events, and each scan time is proportional to the number of congestion events, rather than the number of lost packets [8].

4.2 Implementation Efficiency

UDT's implementation performance has been extensively tuned. This sub-section lists the CPU usage for one or more data flows between two local directly connected identical Linux servers. The server runs Debian Linux (kernel

2.6.18) on dual AMD Opteron Dual Core 3.0GHz processors, 4 GB memory, and 10GE MyriNet NIC. All system parameters are left as default except that the MTU is set to 9000 bytes. No TCP or UDP offload is enabled.

Figure 6 shows the CPU usage of a single TCP, UDP and UDT flow (with or without memory copy avoidance). The total CPU capacity is 400%, because there are 4 cores. Because each flow has a different throughput (varies between 5.4Gb/s TCP and 7.5Gb/s UDT with memory copy avoidance), the values listed in Figure 6 are CPU usage per Gb/s throughput.

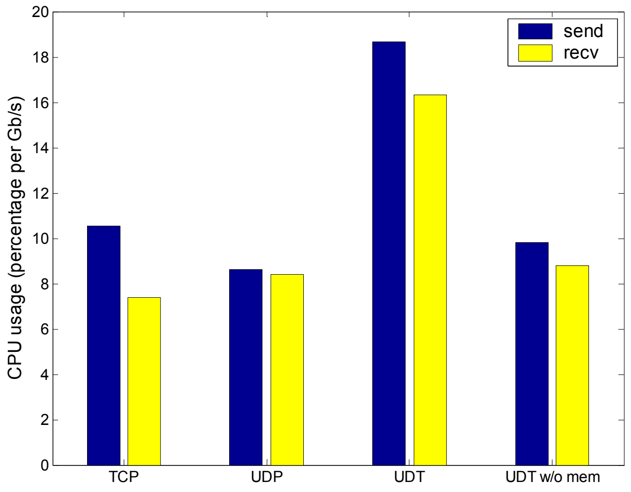

Figure 6. CPU Usage of Single Data Flow.

According to Figure 6, UDT with memory copy avoidance costs similar CPU as UDP and less CPU time than TCP. UDT without memory copy avoidance costs approximately double CPU time of that in the other three situations.

In the case of a single UDT flow without memory copy avoidance, at 7.4Gb/s, the CPU usage of the UDT thread and the application thread at the sender side cost 99% and 40%, respectively (per thread CPU time not shown in Figure 6); the UDT thread and the application thread at the receiver thread cost 90% and 36%, respectively. Although memory copy avoidance happens in the application thread, when it is used, it also reduces CPU usage on both the UDT sending and receiving threads because more memory bandwidth is available for the UDT threads and cache hit ratio is also higher.

Figure 7 shows the CPU usage (unit value is per Gb/s throughput, the same as in Figure 9) of multiple parallel connections of TCP and UDT. UDT memory copy avoidance is not enabled in these experiments because in the situation of multiple connections, the receiver side memory copy avoidance does not work well (see Section 3.6). The connection concurrency is 10, 100, and 500 respectively for each group (TCP, UDT with all connections sharing a single flow, and UDT with each connection having its own flow).

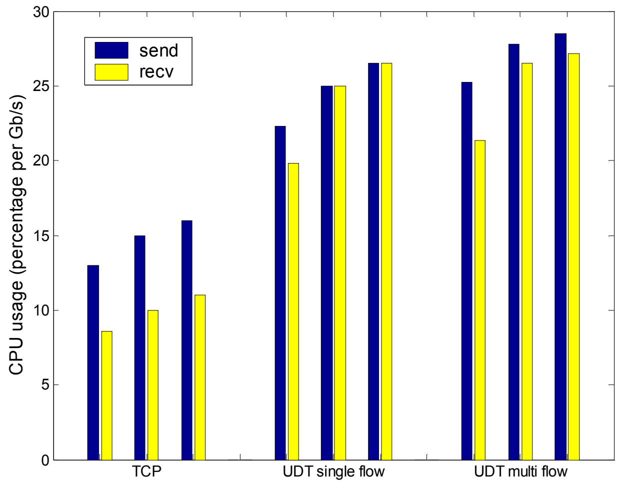

Figure 7. CPU Usage of Concurrent Data Flows.

According to Figure 7, CPU usage of UDT increases slowly as the number of parallel connections increases. The design and implementation is scalable to connection concurrency, and it is comparable to the kernel space TCP implementation.

Furthermore, the second group (all connections share one flow) costs slightly less CPU than the third group. In the case of multiple flows, the overhead of control packets for UDT increases proportionally to the number of flows, because each flow sends its own control packets.

4.3 Usability

UDT is designed to be a general purpose and versatile transport protocol. The stock form of UDT can be used in regular data transfer. Additionally, the messaging UDT socket can also be used in multimedia applications, RPC, file transfer, web services, etc. Currently UDT has been used in many real world applications, including data distribution and file transfer (especially scientific data), P2P applications (both data transfer and system messaging), remote visualization, and so on.

While TCP is mostly used for regular file and data transfer, UDT has the advantage of a richer set of data transfer semantics and congestion control algorithms. Proper congestion control algorithms can be used in special environments such as wireless networks.

UDT does not increase Internet congestion by allowing users to easily modify the congestion control algorithm. It has always been trivial to obtain unfair bandwidth share by using parallel TCP or constant bit rate UDP. In fact, UDT's connection/flow design improves the Internet congestion control by removing the unfairness and traffic oscillation caused by applications that start parallel data connections between the same pair of hosts.

The configurable congestion control feature of UDT can actually help network researchers to rapidly implement and experiment with control algorithms. To demonstrate this ability, six new TCP control algorithms (Scalable, HighSpeed, BiC, Westwood, Vegas, and FAST) are implemented in addition to the three predefined algorithms in the UDT release. Lines of code for the implementation of these control algorithms vary between 11 and 73 [8].

UDT can also be modified to implement other protocols at the application level. An example is to implement forward error correction (FEC) on UDT for low bandwidth high link error environments.

While UDT is not a completely modularized framework like CTP [4] due to performance considerations, it still provides high configurability (congestion control, user defined packets, user controllable ACK intervals, etc.).

It is also much easier to modify UDT than to modify a kernel space TCP implementation. Moreover, there are fewer limitations on deployment and protocol standardization.

Finally, UDT is also more supportive for firewall traversing (e.g., NAT punching) with UDP multiplexing and rendezvous connection setup.

5. RELATED WORK

While transport protocols have been an active research topic in computer networks for decades, there are actually few general purpose transport protocols running at the application level today. In this sense UDT fills a void left by the transport layer protocols where they cannot perfectly support all applications.

However, without considering its application level advantage, UDT can be broadly compared to several other transport protocols. (In fact, the UDT protocol could actually be implemented on top of IP, but the application level implementation was one of the major objectives in developing this protocol.)

UDT borrows the messaging and partial reliability semantics from SCTP. However, SCTP are specially designed for VoIP and telephony, but UDT targets general purpose data transfer. UDT unifies both messaging and streaming semantics in one protocol.

UDT's connection/flow design can also be compared to the multi-streaming feature in SCTP. SCTP creates an association (analogous to UDT flow) between two addresses and multiple independent streams (analogous to UDT connection) can be set up over the association. However, in SCTP, applications need to explicitly create the association and the number of streams is fixed at the beginning, while UDT implicitly joins the connections into the same UDT flow (applications only create independent connections). Furthermore, SCTP applies flow control at the association level. In contrast, UDT applies flow control at the connection level.

This layered design (connection/flow in UDT and stream/association in SCTP) can also be found in Structured Stream Transport (SST) [7]. SST creates channels (analogous to UDT flow) between a pair of hosts while starting multiple lightweight streams (analogous to UDT connection) atop the same channels. However, the rationales behind SST and UDT are fundamentally different.

In SST, the channel provides a secured (optional) virtual connection to support multiple independent application streams and to reduce stream setup time. This design particularly targets applications that require multiple data channels. In contrast, UDT flow automatically aggregates multiple independent connections to reduce control traffic and to provide better congestion control. Both protocols apply congestion control on the lower layer (channel and flow), but SST channel provides unreliable packet delivery only and the streams have to conduct reliability control independently. In contrast, UDT flow provides reliable packet delivery (unless a UDT connection requests a message drop). Beyond this 2-layer design, SST and UDT differ significantly on details of reliability control (ACK, etc.), congestion control, data transfer semantics, and API semantics.

UDT enforces congestion control on the flow level, which carries traffic from multiple UDT connections. This leads to a similar objective as that of congestion manager (CM) [2, 3]. UDT's flow/connection design makes it a natural way to share congestion control among connections between the same address pairs. This design is transparent to existing congestion control algorithms, because any congestion control algorithm originally designed for a single connection can still work on the UDT flow without any modification. In contrast, CM introduces its own congestion control that is specially designed for a group of connections. Furthermore, CM enforces congestion control at the system level and does not provide the flexibility for individual connections to have different control algorithms.

UDT allows applications to choose a predefined congestion control algorithm for each connection. A similar approach is taken in DCCP. However, UDT goes further by allowing users to redefine the control event handlers and write their own congestion control algorithm.

Some of the implementation techniques used in UDT are exchangeable with kernel space TCP implementations. Here are several examples. UDT's buffer management is similar to the Slab cache in Linux TCP [10]. UDT automatically changes socket buffer size to maximize throughput. Windows Vista provides socket buffer auto-sizing, while SOBAS [5] provides application level TCP buffer auto-tuning. UDT uses a congestion event based loss list that significantly reduces the scan time on the packet loss list. This problem occurred in the Linux SACK implementation (when a SACK packet arrives, Linux used to scan the complete list of in-flight packets, which could be very large for high BDP links) and was fixed later.

While there are so many similarities on the implementation issues, UDT's application level implementation is largely different from TCP's kernel space implementation. UDT cannot directly use kernel space thread, kernel timer, hardware interrupt, processor binding, and so on. It is more challenging to realize a high performance implementation at the applications level.

6. CONCLUSIONS

This paper has described the design and implementation of Version 4 of the UDT protocol and demonstrated its scalability, performance, and usability in layered protocol design (UDP multiplexer, UDT flow, and UDT connection), data transfer semantics, configurability, and efficient application level implementation.

As the end-to-end principle [14] indicates, the kernel space should provide the simplest possible protocol and the applications should handle application specific operations. From this point of view, the transport layer does provide UDP, while UDT can bridge the gap between transport layer and applications. This design rationale of UDT does not conflict with the existence of other transport layer protocols, as they provide direct support for large groups of applications with common requirements.

The UDT software is currently in production quality. At the application level, it is much easier to deploy than new TCP variants or new kernel space protocols (e.g., XCP [11]). This also provides a platform for rapidly prototyping and evaluating new ideas in transport protocols. Some of the UDT approaches can be implemented in kernel space if they are proven to be effective in real world settings. Furthermore, many UDT modifications are expected to be application or domain specific, thus they do not need to be compatible with any existing protocols.

Without the limitations of deployment and compatibility, even more innovative technologies on transport protocols will be encouraged and implemented than before, which is another ambitious objective of the UDT project.

7. REFERENCES

[1] M. Allman, V. Paxson, and W. Stevens: TCP congestion control, RFC 2581, April 1999.

[2] David G. Andersen, Deepak Bansal, Dorothy Curtis, Srinivasan Seshan, and Hari Balakrishnan: System Support for Bandwidth Management and Content Adaptation in Internet Applications, 4th USENIX OSDI Conf., San Diego, California, October 2000.

[3] Hari Balakrishnan, Hariharan Rahul, and Srinivasan Seshan: An Integrated Congestion Management Architecture for Internet Hosts, Proc. ACM SIGCOMM, Cambridge, MA, September 1999.

[4] Patrick G. Bridges, Matti A. Hiltunen, Richard D. Schlichting, and Gary T. Wong: A configurable and extensible transport protocol. ACM/IEEE Transactions on Networking, Vol. 15, No. 6, December 2007.

[5] Dovrolis, C., Prasad, R., Jain, M.: Socket Buffer Auto-Sizing for High-Performance Data Transfers. Journal of Grid Computing 1(4) (2004).

[6] M. Duke, R. Braden, W. Eddy, E. Blanton: A Roadmap for Transmission Control Protocol (TCP), RFC 4614, IETF, September 2006.

[7] Bryan Ford: Structured Streams: a New Transport Abstraction, ACM SIGCOMM 2007, August 27-31, 2007, Kyoto, Japan.

[8] Yunhong Gu, Robert L. Grossman, UDT: UDP-based Data Transfer for High-Speed Wide Area Networks. Computer Networks (Elsevier). Volume 51, Issue 7. May 2007.

[9] Yunhong Gu, Robert L. Grossman, Alex Szalay and Ani Thakar: Distributing the Sloan Digital Sky Survey Using UDT and Sector, Proceedings of e-Science 2006.

[10] Thomas Herbert: Linux TCP/IP Networking for Embedded Systems (Networking), Second Edition. Charles River Media, November 17, 2006.

[11] Dina Katabi, Mark Handley, and Charles Rohrs: Internet Congestion Control for High Bandwidth-Delay Product Networks, The ACM Special Interest Group on Data Communications (SIGCOMM 2002), Pittsburgh, PA, pp. 89-102, 2002.

[12] E. Kohler, M. Handley, and S. Floyd: Designing DCCP: Congestion Control Without Reliability, Proceedings of SIGCOMM, September 2006

[13] Injong Rhee and Lisong Xu: CUBIC: A New TCP-Friendly High-Speed TCP Variant, PFLDnet, Lyon, France, 2005.

[14] Jerome H. Saltzer, David P. Reed, and David D. Clark: End-to-end arguments in system design. ACM Transactions on Computer Systems 2, 4 (November 1984) pages 277-288.

[15] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson: RTP: A Transport Protocol for Real-Time Applications, RFC 3550, July 2003.

[16] R. Stewart (Editor), Stream Control Transmission Protocol, RFC 4960, Sep. 2007.

[17] Tan, K., Song, J., Zhang, Q., Sridharan, M. A: Compound TCP: Approach for High-Speed and Long Distance Networks, Proceedings of INFOCOM 2006. 25th IEEE International Conference on Computer Communications. April 2006 Page(s):1 – 12.