ISSN 2047-3338

Exploring Configurable Congestion Control Feature of UDT Protocol

S. Kishore[1], R. Chandra[2] and D. Ganesh[3]
[1]Wipro Technologies, Bangalore, India
[2]St. Mary's College of Engineering and Technology, India
[3]Department of Information Technology, Sree Vidyanikethan Engineering College, India
[1]kishore.sirasala@hotmail.com, [2]r.chandra8000@gmail.com, [3]dgani05@gmail.com

***Abstract*– TCP becomes inefficient when the Bandwidth of the network and delay increases, because slow loss-recovery, a RTT bias inherent in its AIMD congestion-control algorithm, and the bursting data flow caused by its window control. To overcome this problem an application-level is built on top of UDP, called UDP based data Transfer Protocol, or UDT. UDT protocol contains the protocol design and its own congestion control algorithm which is also a framework for configuring new congestion control algorithm. UDT is a high performance data transport protocol, which is mainly, designed for bulk data transfer over high speed wide area networks. It is extended with Configurable Congestion Control to new enhanced versions called UDT/CCC, which supports a wide variety of control algorithms, including TCP algorithms. It is a configurable or reusable user space network stack on which a new congestion control algorithm can be easily implemented, deployed and evaluated. In this paper the potential of the UDT/CCC framework is analyzed by implementing the new control algorithms which is inherited from the CCC. Two types of control algorithm are implemented which is of type Rate based and Window based, after implementing the performances are to be obtained.**

***Index Terms*– TCP, RTT Bias, UDT, High Speed Wide Area Networks and UDT/CCC**

I. INTRODUCTION

THE prevalent and significant development of advanced high speed networks has created opportunities for new technology to prosper. Recent developments in network research introduced an enhanced version of UDT [1], considered to be one of the next generation of high performance data transfer protocols. UDT introduces a new three-layer protocol architecture that is composed of a connection flow multiplexer, enhanced congestion control, and resource management. The new design allows protocol to be shared by parallel connections and to be used by future connections. It improves congestion control and reduces connection set-up time. UDT provides better usability by supporting a variety of network environments and application scenarios. It addresses TCP's limitations by reducing the overhead required to send and receive streams of data [2].

There is a need to create a configurable or reusable user space network stack on which a new congestion control algorithm can be easily implemented, deployed, and evaluated. This stack can be useful in three ways. 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. To serve these needs UDT protocol was designed and the problem is it is very vulnerable for the attackers because it is not having the security and the configurable congestion control feature of UDT/CCC is not analyzed.

In this paper we have covered the Overview of UDT protocol, the application interface of UDT, UDT application socket interface, the design of configurable congestion control algorithm, deploying congestion control algorithms, results, Conclusion and future work, acknowledgements and authors.

II. OVERVIEW OF UDT

UDT is a UDP-based approach and is considered to be the only UDP-based protocol that employs a congestion control algorithm targeting shared networks. It is a new application level protocol with support for user configurable control algorithms and more powerful APIs.

A. Packet Structures

UDT is designed to have two packet structures: first, the data packets and second, the control packets. They are distinguished by the first bit (flag bit) of the packet header. The data packet header starts with 0, while the control packet starts with 1 (Fig. 1).

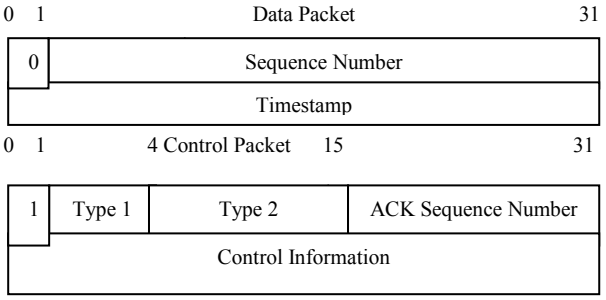

Fig. 1: UDT Packet Header Structure

The first bit of the packet header is a flag indicating if this is a data or control packet. Data packets contain a 31-bit sequence number, 29 bit message number, and a 32 bit time stamp. On the other hand, control packet header, 1-15 bit is the packet type information and 16 -31 can be used for user defined types. The detailed control information depends on the packet type [1]. The packet sequence number uses 31 bits after the flag bit. It uses packet based sequencing, which means the sequence number is increased by 1 for each sent data packet in the order of packet sending. The Sequence number is wrapped after it is increased to the maximum number (2^{31} - 1).

As in other protocols such as DCCP, the sequence number is used to arrange packets into sequence, to detect loss and network duplicates, and to protect against attackers, half-open connections, and delivery of very old packets. Every packet carries a Sequence Number; most packet types include an Acknowledgment Number, which is carried in a control packet - the second packet structure of UDT. The control packet is parsed according to the structure if the flag bit of a UDT packet is 1.

Meanwhile, UDT is a connection-oriented duplex protocol, which supports data streaming and partial reliable messaging. It also uses rate-based congestion control (rate control) and window-based flow control to regulate outgoing traffic. This was designed such that rate control updates the packet sending period for every constant interval, whereas flow control updates the flow window size each time an acknowledgment packet is received. It was expanded to satisfy more requirements for both network research and applications development. This expansion is called Composable UDT and designed to complement the kernel space network stacks. However this feature is intended for:

- Implementation and deployment of new control algorithms. Data transfer through the private links can be implemented using Composable UDT.
- Composable UDT supports application aware algorithms.
- Ease of testing new algorithms for kernel space when using Composable UDT compared to modifying an OS kernel.

The Composable UDT library implements a standard TCP Congestion Control Algorithm (CTCP). CTCP can be redefined to implement more TCP variants, such as TCP (low-based) and TCP (delay-based). The designers emphasized that Composable UDT library does not implement the same mechanisms as in the TCP specification. TCP [3] uses byte-based sequencing, whereas UDT uses packet-based sequencing. It was stressed that this does not prevent CTCP from simulating TCP's congestion avoidance behavior. Nevertheless, UDT was designed with the Configurable Congestion Control (CCC) interface which composed of four categories 1) control event handler call backs, 2) protocol behavior configuration, 3) packet extension, and 4) performance monitoring. Its services/features can be used for bulk data transfer and streaming data processing, unlike TCP which cannot be used for this type of processing because it has two problems.

Firstly, in TCP, the link must be clean (little packet loss) for it to fully utilize the bandwidth. Secondly, when two TCP streams start at the same time, the stream with longer RTT will be starved due to the RTT bias problem, thus, the data analysis process will have to wait for the slower data stream.

UDT, moreover, can cater for streaming video to many clients. It can also provide selective streaming for each client when required, while TCP cannot send data at a fixed rate, and in UDP most of the data reliability control work has to be handled by the application.

III. UDT APPLICATION SOCKET INTERFACE

UDT was developed to adapt itself into the layered network protocol architecture (Figure 2) [1]. It uses UDP through the socket interface provided by operating systems. It provides a UDT socket interface to applications. Applications can call the UDT socket API in the same way they call the system socket API. Since UDT is a duplex transport protocol, each UDT entity has two logical parts: the sender and the receiver. The sender sends (and retransmits) application data according to the flow control and rate control. The receiver receives both data packets and control packets, and sends out control packets according to the received packets as well.

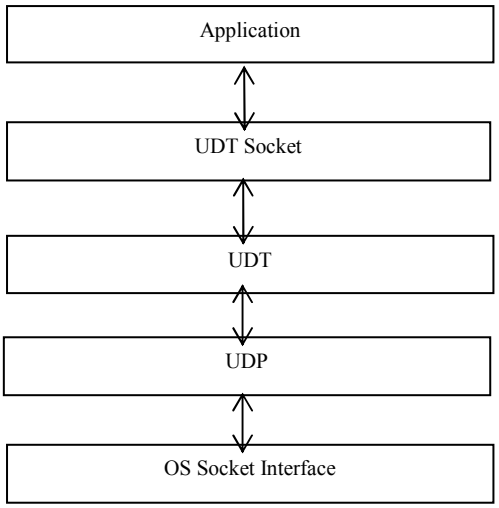

Fig. 2: Layered architecture of UDT

A. Implementation

According to Gu [1], the special difficulty in processing Gb/s speed data transfer was noticed a decade ago. Gu contended that although the need for additional processor and hardware overhead no longer required today, the implementation of an application level transport protocol is still sensitive to its performance. Overheads of memory copies and context switches bring more difficulty for application level implementations. This section will discuss those implementation issues from the software point of view and give practical solutions.

B. Software Architecture

Fig. 3 depicts the UDT 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.

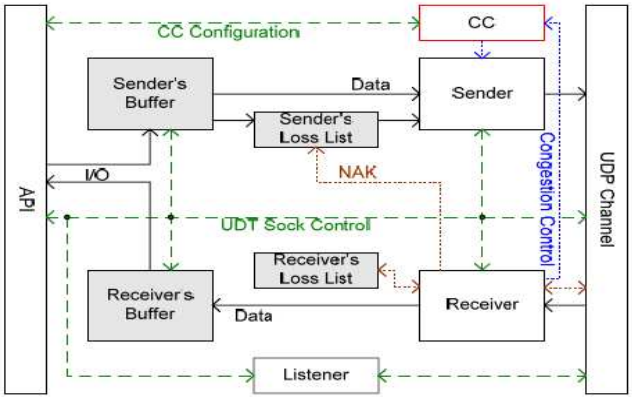

Fig. 3: UDT framework

In the above figure 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, 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 [4] (UDP socket option), the listener (backlog), and CC (the congestion control algorithm, which is only used in Composable UDT). Options can also be read from these modules and provided to applications by the API module.

C. User Interface

The API (application programming interface) is an important consideration when implementing a transport protocol. Generally, it is a good practice to comply with the socket semantics. However, due to the special requirements and use scenarios in high performance applications, additional modifications to the original API are necessary according to Gu and Bernardo [1].

In the past several years, network programmers have welcomed the new *sendfile* method. It is also an important method in data intensive applications, as these are often involved with disk-network IO. In addition to *sendfile*, a new *recvfile* method is also added, to receive data directly onto disk. The *sendfile/recvfile* interfaces and *send/recv* interfaces are orthogonal.

UDT also implements overlapped IO at both the sender and the receiver sides. Related functions and parameters are added into the API. Some lower level APIs should be exposed to applications by an upper level protocol. For example, if the transport layer knows whether a packet loss is due to congestion or link error from the network layer, it will be very helpful for congestion control on links with high bit error rates. UDT exposes many UDP interfaces to give applications the most flexibility for configuring their transport facilities.

An application can make use of the UDT library in a few ways according to Gu. 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. 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, the 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.

D. Protocol Configuration

To accommodate certain control algorithms according to Gu 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 acknowledgment is sent, in elapsed time and the 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 acknowledgment (SACK) and robust reordering (RR) [5], cannot be configured by CCC users, although some of the features can be configured through UDT interfaces.

IV. CONFIGURABLE CONGESTION CONTROL (CCC) DESIGN

UDT/CCC supports a wide variety of control algorithms, including but not limited to, TCP algorithms (e.g., New Reno, Vegas, FAST, Westwood, High-speed, BiC, and Scalable), bulk data transfer algorithms (e.g., SABUL, RBUDP, Lambda Stream, CHEETAH, and Hurricane), and group transport control algorithms (e.g., CM and GTP) [6].

The following are the 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.

Fig. 4 shows how the new CCC feature is inserted into UDT's layered architecture. An application can provide a congestion control class instance (CC in Fig. 4) for UDT to process the control events, or use the default congestion control algorithm provided by UDT. The CC instance includes a set of necessary user-defined callback functions (control event handlers) to process certain control events.

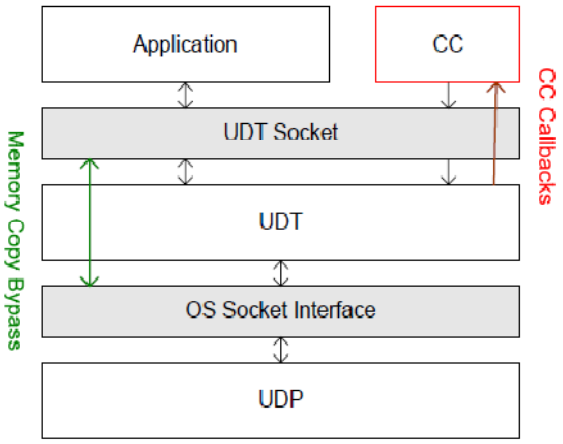

Fig. 4: UDT/CCC Architecture

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

A. 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.
- *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.
- *onPktSent and onPktReceived* are the two most powerful event handlers, because they allow users to check every single data packet. For example, *onPktReceived* can be redefined to compute the loss rate in TFRC. Due to the

same reason, these two callbacks can also allow users to trace the microscopic behavior of a protocol.

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

B. The 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 [7].

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 and the number of unacknowledged packets does not exceed the congestion window size, 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.

Fig. 5: Sending Algorithm

C. The Receiving Algorithm

It describes the abstract UDT/CCC 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) Process the data packet.
6) 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).
7) **onPktReceived()**; Go to Step 1.

Fig. 6: Receiving Algorithm

V. DEPLOYING CONGESTION CONTROL ALGORITHMS

In this section, details about the implementation of congestion control algorithms of 2 types will be described. They are rate based and window based including both loss-based and delay-based algorithms [8]. UDT/CCC uses an object-oriented design. It provides a base C++ class (CCC) that contains all the functions and event handlers that helps in the creation of new congestion control algorithm. 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 packet-sending period (*m_dPacketPeriod*), both of which are CCC class member variables.

A. Rate-based Congestion Control Algorithm

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.

In the algorithm, CUDPBlast inherits from the base class CCC. In the constructor, it sets the congestion window size to a large value so that it will not affect the packet sending. (This is pure rate based method to blast UDP packets.) [4] The method SetRate() can be used to set a fixed packet sending rate at any time.

The application can use setsockopt/getsockopt to assign this control class to a UDT instance, and/or set its parameters.

```
UDT::setsockopt(usock,0, UDT_CC, new
        CCCFactory<CUDPBlast>,
    sizeof(CCCFactory<CUDPBlast> ));
```

The above code assigns the CUDPBlast control algorthm to a UDT socket usock. Note that CCCFactory is using the Abstract Factory design pattern.

To set a specific data sending rate, the application needs to obtain a handle to the concrete CCC class instance used by the UDT socket usock.

```
CUDPBlast* cchandle = NULL;
int temp;
UDT::getsockopt(usock, 0, UDT_CC, &cchandle, &temp);
```

The application can then call the method of setRate() to set a 500Mbps data rate.

```
if (NULL != cchandle)
cchandle->setRate(500);
```

The UDT/CCC can be used to implement most control mechanims, including but not limited to rate-based approaches, TCP variants (e.g., TCP, [9] Scalable, HighSpeed, BiC, Vegas, FAST), and group-based approaches [10] (e.g., GTP, CM).

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.

Then after cresting this pure Rate Based UDPBlast congestion control algorithm, the client and server uses the newly created control algorithm to transfer the data by making use of UDT protocol and the performance is captured.

B. *Window-based Congestion Control Algorithm*

A Window-based TCP Library is also used to transfer bulk data through private links. It doesn't uses the rate instead of rate here it uses the window size (the number of packets sent at a time from the sender to receiver) which updates itself on seeing the acknowledgements from the client. If there exists any negative acknowledgements [6] then the window size will be reduced and if all the packets are received properly then the window size will be slightly increased.

In the algorithm CTCP class is inherited from the CCC class and redefined control event handlers like onACK() and onTimeout(). It also contains the steps that are to be taken when the proper ACK is came from the receiver and also the required steps when duplicate acknowledgements comes [2].

VI. RESULTS

In this section results are obtained from three types of congestion control algorithms one is the UDT's default congestion control Algorithm, second is the rate based (UDP) congestion control algorithm and third is the window based (TCP) congestion control algorithm.

A. *Performance using its own Algorithm (CCC)*

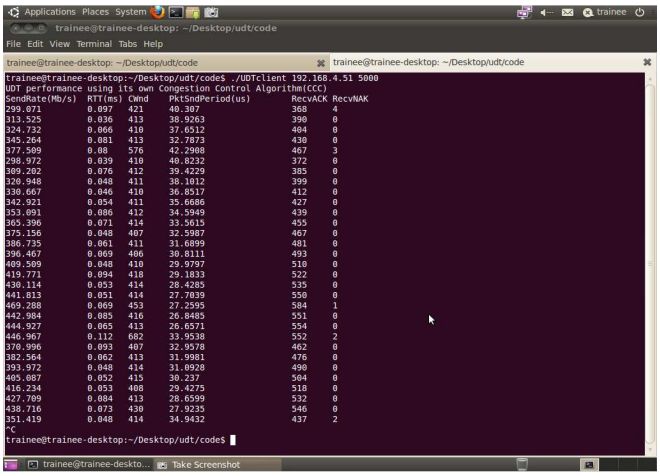

Fig. 7: UDT performance using its own algorithm

Here RecvNAK field shows the number of negative acknowledgements which means that the number of packets lost in the middle. Here the performance is measured based on the number of NAKs. By using its own control algorithm there are less number of NAKs.

B. *Performance using Rate Based Algorithm*

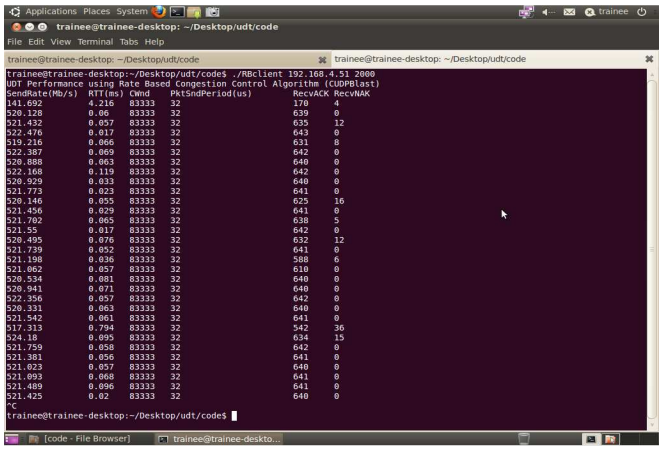

Fig. 8: UDT performance using rate based algorithm

Here there are more number of NAKs when we compared with the NAKs obtained while using its own congestion control algorithm. So that it can be understood that the own algorithm is better than our user defined rate based control algorithm.

C. *Performance using Rate Based Algorithm*

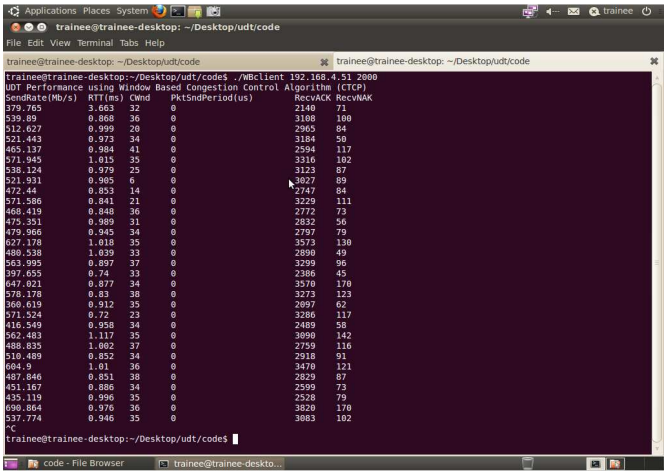

Fig. 9: UDT performance using window based algorithm

In this context there are most number of NAKs are observed which means that there are many packet losses in the transmission so that it can be concluded that the own control algorithm is the efficient one from these three algorithms. In the same manner some other control algorithms can also be implemented, deployed and can be evaluated using the configurable framework of UDT protocol. So that large amount of data can be transferred through the effective control algorithm.

VII. CONCLUSION AND FUTUREWORK

UDT contains configurable congestion control framework, by making use of this framework new congestion control algorithms can be created, evaluated and deployed. Through the configurable feature of UDT, the protocol can use newly generated effective congestion control algorithms. So the emerging efficient control algorithms can also be used with the UDT protocol. In this paper two types of congestion control algorithms which are of type rate-based and window-based are created, deployed and used in the data transfer by making use of UDT protocol from the sender to the receiver.

When the Rate-Based congestion control algorithm is used in the UDT protocol data transfer there are so many negative acknowledgements that means there are so many packet losses. On the other hand when the window-based congestion control algorithm is used, in this here also there exist some negative acknowledgements which are lesser when compared to the rate- based and greater than the NAKs generated while UDT using its default congestion control algorithm. So it can be concluded that the new control algorithms can be deployed using feature of configurable congestion control feature of UDT. So it can be useful to control the congestion the future by the emerging control algorithms.

ACKNOWLEDGEMENT

This work was carried out at the Centre for Development of Advanced Computing Hyderabad and supported by the project engineers and Research and Development Coordinator of CDAC.

REFERENCES

[1] Y. Gu, R. Grossman, UDT: UDP-based Data Transfer for High-Speed Wide Area Networks. Computer Networks (Elsevier). Vol. 51, Issue 7, 2007.

[2] Sumitha Bhandarkar, Saurabh Jain, and A. L. Narasimha Reddy: Improving TCP performance in high bandwidth high RTT links using layered congestion control. Proc. PFLDNet 2005 Workshop, February 2005.

[3] A. Aggarwal, S. Savage, and T. Anderson "Understanding the performance of TCP pacing". IEEE Infocom '00, Tel Aviv,Israel, Mar. 26-30, 2000.

[4] J. Postel: User datagram protocol. RFC 768, IETF, 1980

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

[6] V. Gorodetsky, V. Skormin, and L. Popyack (Eds.), Information Assurance in Computer Networks: Methods, Models, and Architecture for Network Security, St. Petersburg, Springer, 2001.

[7] A. Leon-Garcia, I. Widjaja, Communication Networks, McGraw Hill, 2000.

[8] Cosimo Anglano, Massimo Canonico: A comparative evaluation of high-performance files transfer systems for dataintensive grid applications. WETICE 2004, pp. 283-288.

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

[10] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow: TCP selective acknowledgment options. IETF RFC 2018, April 1996.

Kishore Sirisala has received his B.Tech degree in Information Technology from SK University, Anantapur in 2009 and M.Tech degree in Software Engineering from JNT University, Anantapur in 2011.Currently working as a Project Engineer in WIPRO Technologies Bangalore, India.

R. Chandra has received his B.Tech Degree in Computer science in Information Technology (CSIT) from G. Pulla Reddy Engineering College Affiliated to Sri Krishna Devaraya University, Anantapur and currently studying M.Tech in Computer Science Engineering at St. Mary's College of Engineering and Technology, Affiliated to JNTU, Hyderabad, A.P, India.

D. Ganesh received his B.Tech degree in Information Technology from JNT University, Hyderabad in 2006 and M.Tech degree in Computer Science and Engineering from Acharya Nagarjuna University in 2010.During the period 2006-07 he worked as Assistant Professor in Information Technology department at AITS, Rajampet, India. Since 2007, he is working as Assistant Professor in IT Department at Sree Vidyanikethan Engineering College, Tirupati, India. He has Published 9 papers in national and International conferences. His current research interests are computer networks,wireless networks, ad mobile ad-hoc networks. He is a member of ISTE, CSI.