TNC Transactions on Networks and Communications
SOCIETY FOR SCIENCE AND EDUCATION UNITED KINGDOM
VOLUME 3, ISSUE 1
ISSN: 2054 - 7420

Mechanism for reliable low latency communications in Computing Clusters

[1]Saibal K Ghosh and [2]Dharma P Agrawal
Department of Electrical Engineering and Computing Systems;
University of Cincinnati, Cincinnati, OH, United States;
[1]ghoshsl@mail.uc.edu; [2]agrawadp@ucmail.uc.edu

ABSTRACT

Recent increases in the demands for computing power have given rise to the prevalence of distributed computing. Computationally complex problems are broken down into smaller chunks and are distributed to computing nodes that perform the computation simultaneously. The nodes may exchange information as peers and their combined result is the final outcome of the computation. Multiple computers need a mechanism to communicate and exchange information in order to harness collective computing power. Traditionally, the Transmission Control Protocol (TCP) has been used to exchange information between computers. However, additional overhead, generally associated with TCP has been considered as a serious drawback for any rapid data exchange. The User Datagram Protocol (UDP) alleviates the overheads of TCP but provides no reliability for the data transfer. In this paper, we introduce an enhanced UDP mechanism to exchange data of limited size reliably and without any associated overhead of TCP.

Keywords: Reliable communications; Cluster computing; Transmission Control Protocol; User Datagram Protocol; Reliable User Datagram Protocol; Overhead in communication protocols.

1 Introduction

TCP protocol was designed to enable reliable communication and optimize the bandwidth between a sender and a receiver. It is a highly robust and versatile protocol, being able to account for varying availability of the network bandwidth between the computing nodes. It is able to adjust the transmission speed on the fly and allows for retransmissions thereby minimizing the data loss and ensuring reliability. TCP achieves this by setting up a communication channel between two communicating nodes, allocating buffers at the sender and the receiver and using sequence numbers to keep track of the packets already transmitted. Reliability is ensured through acknowledgement messages sent by receiver back to the sender [1] [2]. Traditional internet applications are built on the client-server paradigm and thus TCP is well suited for such applications. However, since TCP requires that a dedicated channel is setup between a server and a recipient it cannot provide any support for broadcasting messages. In a multiprocessor environment, broadcast is often required to notify all the participating nodes within a limited time. Setting up a dedicated channel for each node would be both time and resource intensive. The UDP protocol tries to solve these problems inherent in TCP. It does not require the setup of a dedicated channel between the sender and the intended recipient and has support for broadcast [3]. However, UDP does not provide support for acknowledgement messages and therefore the sender has no way of interpreting if the message was reliably transmitted to the recipient. Reliability of transmitted messages is important in large distributed systems since the outcome of a particular computational

DOI: 10.14738/tnc.31.992
Publication Date: 25th February 2015
URL: http://dx.doi.org/10.14738/tnc.31.992

task is the result of all the nodes working in tandem. Failures in a message transmission can stall the computation and erroneous data transmission may even result in unexpected outcomes. In this work, we propose a novel mechanism for fast transmission of data in a multiprocessor environment. Our mechanism ensures the reliability of the data being transmitted between nodes and allows for retransmissions in case of any failure. This mechanism has been tested on the 32 node computational cluster at the University of Cincinnati and could form the core of the networking architecture for fine grain parallelism in large scale distributed simulations.

2 Background

Our extensive experiments with network traffic in computing clusters have indicated that it is inherently highly intermittent in nature. Computing nodes only exchange traffic when they need to communicate with other nodes and/or fetch data from the master node. These exchanges are infrequent and only involve modest amount of data transfers. However, for the purposes of a successful computational task, we need to ensure that this data is transferred reliably with a bounded latency since the outcome of the computation depends on it. We normally use TCP to guarantee delivery of packets in a cluster. Our results illustrate that packet losses increase in direct proportion to the number of packets being exchanged at any given time in the cluster. Since packet drop causes TCP to retransmit, this exponentially increases the number of packets and the performance of the network degrades rapidly.

We used a simple setup in the cluster to simulate the behavior of a distributed computing system with fine grained parallelism. Such a system shares data between nodes on the level of object states across nodes and therefore has substantial number of data packets being exchanged when running. We simulated a lightly and a heavily loaded system on the cluster to examine the effects to traditional TCP under heavy intermittent traffic. For the lightly loaded system simulation, the nodes generated a packet every half second. The heavily loaded system simulation had nodes generating ten packets every second. For distributed systems involving hundreds of nodes, the packet transmission rates can approach the rates in our simulation. We exchanged 25,000 packets to simulate a sustained data transfer. The effect of using TCP for a lightly loaded and a heavily loaded system are shown in Figures 1 and 2 respectively. We used a timeout of 3 seconds for the lightly loaded system and 8 seconds for the heavily loaded one. We see from the figures that the heavily loaded system has substantially more packet losses than the lightly loaded one even with the higher timeout value. While most packets were received and acknowledged, the heavily loaded system shows a far higher proportion of lost packets. The white areas in the graph represent packets that were not received at the receiver required retransmission. From the graphs, we can infer that TCP is unable to deal with the requirements of intermittent network traffic even with a small payload. More precisely, our results indicate that since TCP is heavily geared towards stream based traffic, the overhead for setting up a TCP connection between a sender and a receiver far often exceeds the actual data that needs to be transferred over the established TCP channel. On the other hand while UDP allows fast data transmission and support for broadcasting, it does not provide support for acknowledgements and thus lacks in providing reliability. In this work, we have implemented the best of both transmission protocols by adding reliability on top of the UDP protocol.

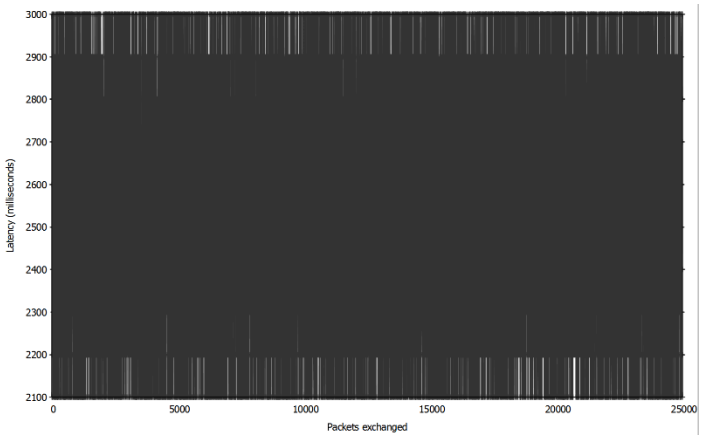

Figure 1: Latency and associated packet drops for our lightly loaded cluster.

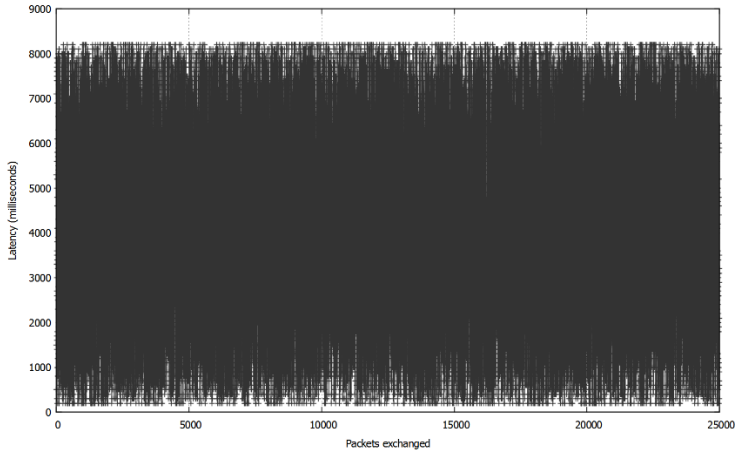

Figure 2: Latency and associated packet drops for our heavily loaded cluster.

3 Related Work

Studies have been performed in the past to improve and alleviate the limitations of the UDP protocol. Extensive analysis of the UDP protocol implementation has been performed in the UNIX kernel [4].Through protocol optimization, UDP has been shown to perform over 25-35% better in CISC and RISC systems [5]. Extensions have also been made to the protocol to support wireless transmissions and its usage in wireless multimedia sensor networks since wireless networks are more susceptible to packet losses due to the nature of the wireless medium [6] [7]. Efforts have also been made to improve the reliability of UDP. The most notable of these is the Reliable User Datagram Protocol (RUDP) designed at Bell Laboratories [8] [9]. RUDP attempts to achieve a middle ground between the simplicity of UDP and the overheads incurred by TCP. However, in order for RUDP to achieve a higher quality of service, it builds on the UDP by adding support for acknowledgement of received packets, mechanism for retransmissions of lost packets, flow control and adding support for buffers at the communicating endpoints [9]. Cisco uses an implementation of RUDP in its Signaling Link Terminals. RUDP Version 0 is used to backhaul the Signaling System 7 protocols over IP signaling control networks while Version 1 is used for ISDN networks [10] [11]. The low latency provided by UDP makes it suitable for IPTV and VoIP applications since they require very high throughputs and are somewhat tolerant to low levels of packet losses [12]. With the prevalence of VoIP applications in recent years, UDP currently forms a significant portion of the total internet traffic [13]. RUDP also forms the basis of the communications architecture in the MediaRoom IPTV platform originally developed by Microsoft which is now owned by Ericsson [14]. However, both Cisco and Microsoft/Ericsson implementations of RUDP are proprietary and are used in their own products, making them incompatible with existing network infrastructure. To the best of our

knowledge, this is an implementation of a general purpose low latency mechanism for fast transmission of packets with limited payload.

4 Our mechanism

4.1 Scope

Our mechanism as described in this work is specifically geared towards communication between nodes in a computing cluster. In order to make our mechanism compatible with existing networks, we have built it on the top of existing network paradigms. Specifically, it covers the Transport, Session and the Presentation layers of the communication stack. Computing nodes use IP addresses to identify and talk to each other. Our mechanism also uses the IP addressing scheme that is fully compatible with existing network architectures. We use UDP as the core of the protocol as it provides a better throughput as compared to TCP as we have shown earlier. Reinventing and implementing a completely new and separate transport layer protocol would have put in too much overhead and made our mechanism incompatible with current standards. Moreover, using UDP provides our mechanism with broadcasting support that is very useful in a cluster network. We have built on UDP's fast transmission capabilities and have implemented a mechanism that informs the sender when a packet has been received by an intended recipient. We use sender side buffer to store the information sent over the network until an acknowledgement is received from the recipient. The sender uses identification numbers on the data packets to achieve this. We understand that while the majority of packets that are expected to be exchanged using our mechanism would be small enough to fit inside a UDP datagram, certain scenarios may give rise to heavy traffic in a cluster. Therefore, we have incorporated functionality in our mechanism that is able to split a large payload into fragments that can be made to fit inside a conventional UDP datagram. This functionality ensures that our mechanism works under all traffic scenarios in a computing cluster.

4.2 Architecture

Our mechanism follows conventional network paradigms and defines a customized header for each message. The actual data is carried in the message payload and is fully described in the message header. The 25 byte header in our mechanism completely specifies the parameters for the sender and the receiver including the IP addresses and the port numbers. A schematic diagram of the message header is shown in Figure 3. As mentioned previously, we use sequence numbers to keep track of the messages sent. In order to keep our mechanism simple we choose the initial value of the sequence number as a pseudorandom number and keep incrementing the value with every packet that is sent. However, any monotonic function that varies with time can be used for this purpose. We identify each message uniquely by a combination of the sender and receiver and the message id.

Message Size (16 bits)	Message Sequence No. (16 bits)	Multipart message Field No. (32 bits)	Number of Messages (32 bits)	Message Id (32 bits)	Message Data Size (64 bits)	Bit Fields (8 bits)

Figure 3: Structure of the Header.

The header fields determine all the functionalities of our mechanism. However, the header overhead is minimal as we show in the results. We describe the usage of each field in the header in Table 1.

Table 1: Header fields and their usage

Field	Usage
Message Size	Stores the size of the complete message including the message header.
Message Sequence Number	Determines the order of messages originating from a particular sender. A pseudorandom number is used as the seed with a monotonic function for successive sequences. See Section 4.2.
Multipart Message Field Number	Determines the position of a particular message in a multipart message. Since messages can arrive out of order in a packet switched network, the receiver thread uses the information in this field to rearrange the message parts in the receiver buffer before handing over to the higher layers. Obfuscates message transmission details from higher network layers and enables compatibility with existing network protocols.
Number of Messages	Used only for multipart messages and is used to specify the total number of messages that are in a particular message. The receiver thread uses the information in this field to determine when all the parts of the multipart message have arrived successfully at the receiver. The Recombination thread starts once all the parts of the multipart message have been successfully received. See Section 4.3.4.
Message Id	Uniquely identifies a particular message in the list of all messages exchanged in the cluster.
Message Data Size	Stores the size of the payload in a particular message.
Bit Fields	Used to specify added functionality at the receiver. A combination of these fields can be used to signify the start or end of a session. Can also be used to specify no message confirmations based on a particular simulation scenario. See Section 4.3. Broadcast messages and cumulative acknowledgements for network traffic reduction can also be signified using these fields.

4.3 Implementation

We have implemented this mechanism as a multithreaded process running on the computing nodes. Since the nodes need to send and receive messages, the process is capable of running as a client and a server simultaneously. We would now give a detailed overview of the buffers that we mentioned earlier. We would then explain the process flow with respect to the workings of the buffers. All of these buffers are maintained by the process running on each node.

4.3.1 Session buffer

The session buffer is responsible for storing information related to all other nodes that a particular node needs to communicate with. It stores the value of the next sequence number and the timeout interval which is used to determine if a packet has been lost and need to be resend. If the sender does not receive an acknowledgement within the specified timeout, we assume that the packet has been lost and needs to be resent by the sender. We adaptively change the value of the timeout interval based on the current network conditions. We follow an exponential back off strategy for messages that need to be resent. This follows from conventional network protocols that have used this strategy to determine the network load and thereby prevent message collisions. For each new message that is sent or received, a new entry is made in the session buffer. This is to prevent ambiguous messages being received during a network storm. We also mark an entry with a broadcast flag if the message has been received as part of a broadcast.

4.3.2 Transmission buffer

The transmission buffer is responsible for arranging the messages that are to be sent. The memory allocated in the buffer is used to mark the headers and fill the payload section with the actual data that is to be transferred. The messages are removed from the transmission buffer once acknowledgements have been received from the receiver.

4.3.3 Reception buffer

The reception buffer is responsible for receiving the messages and forming an ordered list. This is essential in order to guarantee that the packets are received in the order in which they were sent. Due to the nature of the packet switched network, messages might arrive out of order at the receiver. A separate thread running on the receiver looks at these messages and arranges them in the order based on the sequence number of the messages. Once the messages have been arranged in the specified order, they are made available to be used by the higher layers of the network stack.

4.3.4 Multipart message buffer

The multipart message buffer is used for handle large payloads and to ensure compatibility with the existing network infrastructure. As mentioned previously, when the sender has a large message to send, it is split into a number of parts before transmitting to the receiver. The partial message buffer allows the messages to be gathered at the receiver while they are being sent and rearrange them by means of the multipart message field number as mentioned previously. Once the whole message has been reassembled at the receiver, the destination application is notified of the arrival. Providing this functionality frees the application layer from having to rearrange parts of a multipart message in order. This mechanism also provides compatibility with older application layer protocols that do not have functionality to assemble multipart messages

4.4 System Walkthrough

We have a custom Linux application written in C++ that spawns multiple threads on the computing node to implement the mechanism that we have described in this work. In this section we describe three major threads that are responsible for transferring the data from the application layer, sending acknowledgements and performing memory allocation and management of the buffers. The server thread forms the bridge between the processes running on the node and our mechanism. When an application is ready to send data, it notifies the server thread which then proceeds to add it to the transmission buffer. This thread also maintains the timeout counter. Once the counter expires and the server thread finds no acknowledgement for a particular message that had been sent earlier, it resends the message again to the intended recipient. The transmission buffer is cleared by the server thread once an acknowledgement is received from the recipient node.

Once a recipient node receives a message, it first verifies if the message is already present in the reception buffer. This situation can arise if the acknowledgement was lost and the originator node had resent the message. This functionality is performed by the receiver thread with the help of the message id. If the message is not present, the message is added to the reception buffer. However, if the receiver thread detects that the message is already present, it is an indication that the sender of the message has not yet received the confirmation. It then proceeds to resend the confirmation message to the sender. For multipart messages, the receiver thread has an added responsibility of storing the received messages in the multipart message buffer. As mentioned previously, messages in a multipart message may arrive out of order. It is also the responsibility of the receiver thread to arrange all the parts of the multipart message. Once the receiver thread receives all the parts of the

multipart message, it sends a confirmation to the sender. A cumulative confirmation for a multipart message suffices to inform the sender that all parts of the message have been received.

A buffer manager thread continuously runs in the background and is responsible for maintaining the buffers at an optimum load level. Too many packets in the buffer could mean that either the application layer is generating too many packets or there is some congestion in the network. Based on the values of the timeout counter, the buffer manager can determine the cause of the backlog. If the network is healthy, the buffer manager spawns new server and receiver threads to send and process the messages that are being generated. If it determines congestion in the network, the server manager thread informs the application that it needs to slow down or pause the generation of messages since most of the messages have to be retransmitted anyways. Informing the application is necessary since in a computing cluster, nodes work on data that is generated in a different node and thus they need to be synced periodically. If the node pauses or slows down computation, then the synchronization procedures can be performed without a large rollback penalty.

We use a publish/subscribe model to notify the application thread of the arrival of messages and the current health of the network. Each application can subscribe to be notified of arriving messages that match a predefined criterion. The receiver thread maintains a table of all possible matches for a particular node. Once a message satisfying a particular criterion arrives at the receiver, the particular application is notified of the arrival. Similarly, applications also subscribe to be notified of network events. Network congestion and severe packets loss scenarios are propagated to the application layer from our mechanism. We also provide error handlers for messages who have no subscribers and for common error codes. The publish/subscribe model allows us to decouple our mechanism from the application layer. We leave it to the particular application to decide on the course of action based on a particular error code or the prevailing network scenario.

A schematic diagram showing the flow of control for the mechanism described in this section is shown in Figure 4.

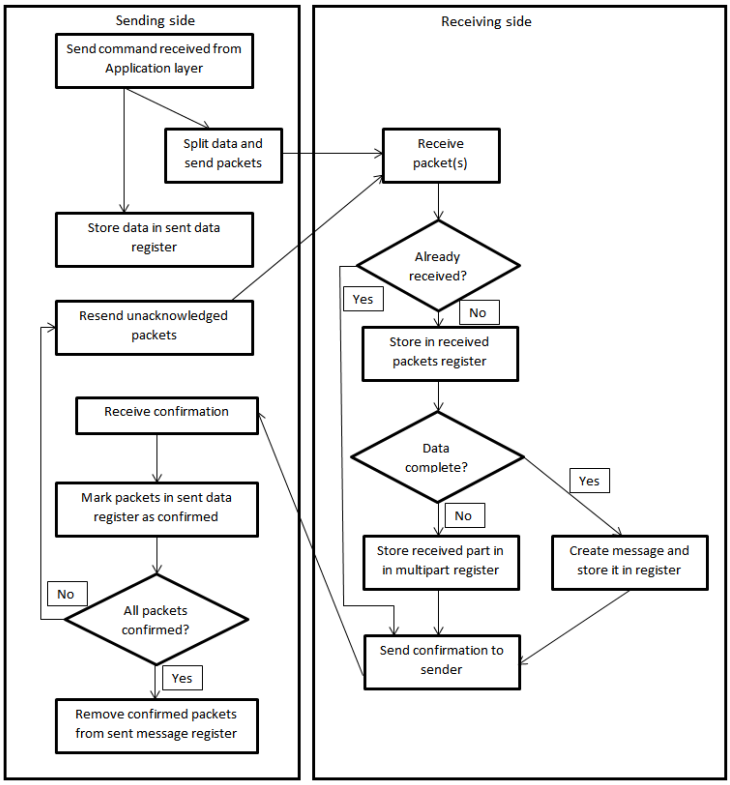

Figure 4: Control flow for our mechanism.

5 Results

We used the mechanism as described in this work at the University of Cincinnati computing cluster with 32 nodes. The nodes are synchronized from the same network time server. We exchanged messages ranging from a zero payload (only the header) to 5 Megabytes. Large message exchanges are also carried out in order to ensure that we could test the implementation of the multipart messaging scheme as described in Section 4.3. The nodes are programmed to exchange a million messages for the purposes of this study. In order to compare our mechanism we also measured the latencies using TCP and UDP as the communication protocol. The messages contained timestamps from the server and were used by the receiver to compare the latencies.

We understand that the network traffic encountered at any given time in the cluster would be a function of the number of nodes in the cluster, the current state of the computation and the number of signaling messages being exchanged at a particular instant. Generally, nodes exchange a lot of data during the start of a particular computation task thereby giving rise to high latencies. As the computation proceeds, the number of messages exchanged reduces since each node tends to work on their own set of data using their own memory. Towards the end of the computation, the number of messages exchanged increases since the nodes need to synchronize the results between themselves, weed out erroneous results and transmit all the computed work back to the master node. We used an average of the million latency measurements for each payload data size in order to eliminate any stray errors. We also simulated node failures and rollbacks with random bursts of heavy network traffic to temporarily increase the instantaneous latency and test how our mechanism holds out in such scenarios. Our results for 2, 3, 4 and 5 nodes are shown in Figures 5, 6, 7 and 8 respectively.

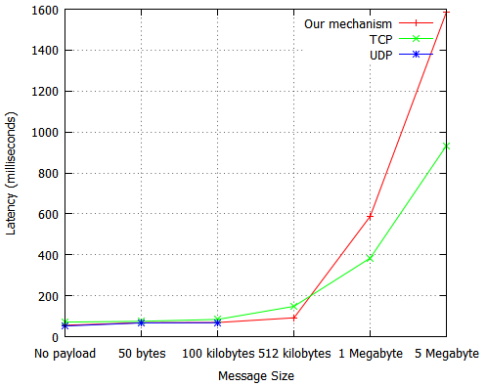

Figure 5: Latencies for messages exchanged between two nodes

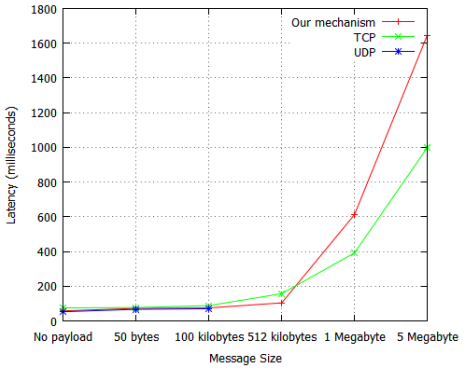

Figure 6: Latencies for messages exchanged between three nodes

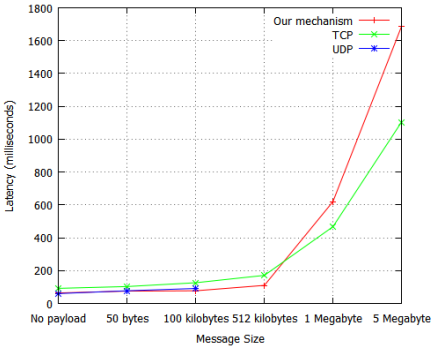

Figure 7: Latencies for messages exchanged between four nodes

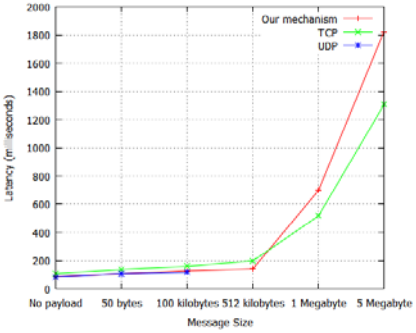

Figure 8: Latencies for messages exchanged between five nodes

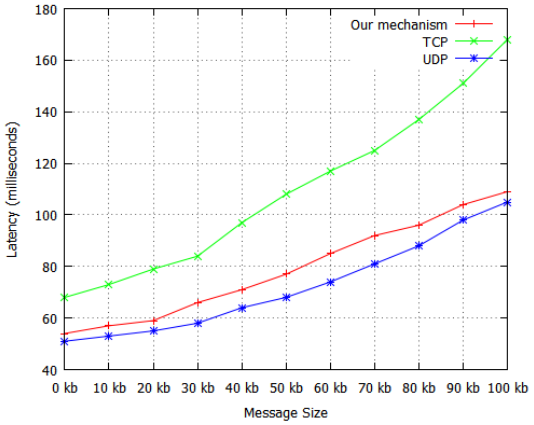

Figure 9: Latencies for small message payloads between two nodes

6 Conclusion

We found that for small payloads, the latencies for our mechanism matched very well with those of the UDP. However as discussed in this work, the overhead for our mechanism is almost negligible. Also for small payloads, our mechanism has almost half the latency of TCP. As we have mentioned earlier, almost all message exchanges in a computing cluster have an extremely small payload. Therefore, our mechanism takes the best of both the TCP and the UDP for most scenarios.

The results for our mechanism indicate that the latency for data transmissions carried out using our mechanism does not increase appreciably as the number of nodes in the cluster is increased. However, the latency for TCP increases exponentially. This is because, as the number of nodes increases, the number of channels that TCP needs to establish also increases, thereby degrading the

performance. However, since our mechanism does not need to setup channels, its overhead does not depend on the number of nodes in the network. Hence the performance would not drop drastically unlike TCP. Therefore, latencies experienced by our mechanism are not dependent on the number of nodes and thus could hold good for large clusters having hundreds or thousands of nodes.

For very small payloads, we observe that the graphs for our mechanism and those of TCP and UDP are almost overlapping. This is because of the large differences in the data sizes used in the experiment. For the purposes of clarity, Figure 9 is a detailed summary of the observed latencies for small payloads.

We also observe from the graph that TCP performs exceptionally well for large payloads. However, we can also see that our mechanism does not fail for large payloads even with the added complexity of sending, receiving and combining multipart messages and providing for reliability. Therefore, we can conclude that our mechanism can be used as a replacement for the generic networking mechanism in computational clusters involving a significant number of nodes.

REFERENCES

[1] Cerf, V., and R. Kahn, "A Protocol for Packet Network Intercommunication," IEEE Transactions on Communications, Vol. COM-22, No. 5, pp 637--648, May 1974.

[2] Postel, J., “DOD Standard transmission control protocol,” ACM SIGCOMM Computer Communication Review, Vol. 10, No. 4, pp 52-132, Oct 1980.

[3] Postel, J., User Datagram Protocol, RFC Editor, 1980

[4] Balasubramanian, S., Thanthry, N., Bhagavathula, R. and Pendse, R., "Performance analysis of UNIX user datagram protocol implementations," System Theory, 2004. Proceedings of the Thirty-Sixth Southeastern Symposium on , vol., no., pp.230,235, 2004

[5] Partridge, C. and Pink, S., "A faster UDP [user datagram protocol]," Networking, IEEE/ACM Transactions on , vol.1, no.4, pp.429,440, Aug 1993

[6] Wang, Y., Wu, C., Zhang, X., Li, B and, Zhang, Y., "An embedded wireless transmission system based on the extended user datagram protocol (EUDP)," Future Computer and Communication (ICFCC), 2010 2nd International Conference on , vol.3, no., pp.V3-690,V3-693, 21-24 May 2010

[7] Aghdam, S.M., Khansari, M., Rabiee, H.R. and Salehi, M., "UDDP: A User Datagram Dispatcher Protocol for Wireless Multimedia Sensor Networks," Consumer Communications and Networking Conference (CCNC), 2012 IEEE , vol., no., pp.765,770, 14-17 Jan. 2012

[8] RFC 908, The Reliable Data Protocol (RDP).

[9] RFC 1151, The Reliable Data Protocol (RDP) Version 2.

[10] Modarressi, A.R. and Skoog, R.A., "An overview of Signaling System No.7," Proceedings of the IEEE , vol.80, no.4, pp.590,606, Apr 1992

[11] Cisco Signaling Link Terminal http://www.cisco.com/en/US/products/hw/vcallcon/ps2152/products_data_sheet09186a0080091b58.html

[12] Saleh, S., Shah, Z. and Baig, A., "Capacity analysis of combined IPTV and VoIP over IEEE 802.11n," Local Computer Networks (LCN), 2013 IEEE 38th Conference on , vol., no., pp.785,792, 21-24 Oct. 2013

[13] Zhang, M., Dusi, M., John, W. and Chen, C., "Analysis of UDP Traffic Usage on Internet Backbone Links," Applications and the Internet, 2009. SAINT '09. Ninth Annual International Symposium on , vol., no., pp.280,281, 20-24 July 2009

[14] The MediaRoom IPTV Platform http://www.ericsson.com/ourportfolio/mediaroom-landingpage

