

A Cross-Layer Architecture and Protocols for Reliable File-Stream Distribution

A Thesis

Presented to
the faculty of the School of Engineering and Applied Science
University of Virginia

in partial fulfillment
of the requirements for the degree

Master of Science

by

Shuoshuo Chen

May

2016

APPROVAL SHEET

The thesis
is submitted in partial fulfillment of the requirements
for the degree of
Master of Science


AUTHOR

The thesis has been read and approved by the examining committee:

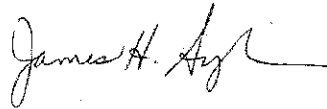
Malathi Veeraraghavan

Advisor

Joanne Bechta Dugan

Alfred Weaver

Accepted for the School of Engineering and Applied Science:



Dean, School of Engineering and Applied Science

May
2016

Abstract

The growing deployment of OpenFlow/SDN networks makes it increasingly possible to leverage network multicast services. This work proposes a novel cross-layer Multicast-Push Unicast-Pull (MPUP) architecture that includes functionality in the application, transport and link layers to offer users a reliable file-stream distribution service to multiple subscribers. In addition, for the transport layer, a reliable multicast protocol named File Multicast Transport Protocol (FMTP) was designed and implemented.

A prototype implementation of the MPUP architecture, which includes FMTP, was realized in a new version of Local Data Manager (LDM), LDM7, a software program that has been in use since 1994 for real-time meteorology data distribution. LDM6, the currently deployed version, uses application-layer multicast (ALM).

Experiments were run on the GENI infrastructure to compare LDM7 and LDM6. The two main findings are (i) LDM7 can be run at a higher sending rate than LDM6 allowing for improved performance (lower file-delivery latency), and (ii) to achieve the same performance, LDM7 uses significantly lower bandwidth and compute capacity. A three-fold improvement in performance improvement was possible with LDM7, and a bandwidth reduction from 350 Mbps to 21.4 Mbps was observed with 24 receivers.

Acknowledgements

I would first like to thank my advisor, Professor Malathi Veeraraghavan of the School of Engineering and Applied Science at University of Virginia. During the two years of my study and research, she has provided me great support and valuable advice on how to do excellent research. She is also a great mentor in teaching me lessons for life. Her curiosity in science, critical and logical thinking and passion towards discovery all inspires me so much. I have been amazingly fortunate to have this experience and study with her.

I would also like to thank my mentor and collaborator, Steve Emmerson of the University Corporation for Atmospheric Research. He gives me much insightful advice for the NSF CC-NIE project. As a rookie in developing large software, he teaches me a lot of the best practices in thread-safe and maintainable programming. His deep understanding in OS and C/C++ significantly shortens our developing duration and benefits my future career.

I appreciate my fellow students in the high speed networking research group for their enlightenment in our group meetings and their help on my thesis writing.

I am also grateful to Dr. Yufeng Xin and Dr. Ilya Baldin. Their support on the GENI testbed made this work possible.

I thank Professor Joanne Bechta Dugan and Professor Alfred Weaver for serving on my defense committee and providing informative and helpful feedbacks.

I would also like to thank my family and friends for their support and help towards my success.

This work is supported by NSF grants CNS-1116081, OCI-1127340, ACI-1340910, CNS-1405171, and CNS-1531065, and U.S. DOE grant DE-SC0011358.

Contents

Contents	e
List of Tables	g
List of Figures	h
1 Introduction	1
1.1 Problem statement	1
1.2 Proposed solution and contributions	2
1.3 Thesis layout	3
2 A Cross-Layer Multicast-Push Unicast-Pull (MPUP) Architecture for Reliable File-Stream Distribution	4
2.1 Introduction	4
2.2 Cross-Layer MPUP architecture	5
2.3 System model and metrics	9
2.3.1 System model	9
2.3.2 Metrics	10
2.4 Prototype implementation	12
2.5 Evaluation and analysis	14
2.5.1 Experimental setup	14
2.5.2 Experiment execution	15
2.5.3 Experiment 1: Determine file-set size for throughput metric	18
2.5.4 Experiment 2: FMTP File Delivery Ratio (FFDR)	19
2.5.5 Experiment 3: Throughput measurement	20
2.5.6 Experiment 4: Resource requirements comparison	22
2.6 Related work	25
2.7 Conclusions	26
3 Enhanced MPUP Architecture	28
3.1 Introduction	28
3.2 New maximum retransmission period design	28
3.2.1 Problem statement	29
3.2.2 Design alternatives	30
3.2.3 FFDR measurement	34
3.2.4 Throughput measurement	36
3.2.5 Resource requirements	36
3.3 Sender <code>tc</code> -layer buffer size revisit	39

3.3.1	Experiments to determine minimum <code>tc</code> -layer buffer size for LDM7/FMTP/UDP	39
3.3.2	Comparison of TCP, CTCP and UDP behavior in response to a full <code>tc</code> buffer using a common application <code>iperf3</code>	41
3.3.3	Explanation for the <code>tc</code> -layer buffer size required for LDM7/FMTP/UDP	46
3.4	Impact of sender multicast rate on LDM7 throughput	48
3.4.1	Impact of RTT on LDM7 throughput	48
3.4.2	Impact of the sender <code>tc</code> buffering delay on throughput	49
3.4.3	Impact of other delay components on throughput	51
3.5	New metrics	53
3.5.1	File latency	54
3.5.2	Block retransmission ratio	56
3.6	Conclusions	61
4	File Multicast Transport Protocol (FMTP)	62
4.1	Introduction	62
4.2	Protocol overview	63
4.3	Packet structure and message formats	65
4.3.1	FMTP packet structure	65
4.3.2	FMTP message formats	67
4.4	Protocol operation	71
4.4.1	Underlying protocol layers	72
4.4.2	FMTP sender initialization	73
4.4.3	FMTP receiver initialization	75
4.4.4	File multicast and retransmissions	76
4.4.5	Transmission and retransmission timeout mechanisms	80
4.5	Implementation	81
4.6	Related work	84
4.7	Conclusions	86
5	Conclusions and Future Work	87
5.1	Conclusions	87
5.2	Future work	88
	Appendices	90
	A Linux kernel implementation of the network stack	91
	Bibliography	93

List of Tables

2.1	Model Parameters	9
2.2	Values for input parameters	15
3.1	Evidence of sender <code>tc</code> buffer buildup; $r = 20$ Mbps, $p = 0\%$, $RTT = 0.2$ ms	50
3.2	Example latencies for five products; $r = 8$ Gbps, $p = 0\%$, $RTT = 0.2$ ms . .	52
3.3	Experiment A to F for BRR study	57

List of Figures

2.1	Multicast Push Unicast Pull (MPUP) Architecture	5
2.2	Impact of Aggregate File-Set (Group) Size G on Throughput	19
2.3	FMTP File Delivery Ratio (FFDR) for LDM7	20
2.4	Impact of tc rate limiting on throughput under lossless/lossy conditions; m : number of receivers	21
2.5	Experimental settings: $m = 8$ receivers, lossless setting	23
2.6	Resource utilization; $r = 20$ Mbps; (LDM version, loss rate) are shown for each plot	24
3.1	Deliver-time distribution for all files in the 1-hour NGRID trace; $m = 16$, $r = 20$ Mbps;	35
3.2	FFDR with new $\tau_{snd}(n)$ design	35
3.3	Impact of tc rate limiting on throughput under lossless/lossy conditions; m : number of receivers	37
3.4	Resource utilization; $r = 20$ Mbps; (LDM version, loss rate) are shown for each plot	38
3.5	Minimum loss-free tc buffer size required for various multicast rates	41
3.6	Experiment 1 with TCP Reno	42
3.7	Experiment 2 with CTCP and no retransmission	43
3.8	Experiment 3 with CTCP and retransmission	44
3.9	Experiment 4 with UDP	45
3.10	Throughput for $r = 20$ and $r = 500$ Mbps	49
3.11	Throughput under different r_{mc} ; $p = 0\%$, RTT=0.2 ms	51
3.12	Ideal latency vs. actual latency for individual files; r_{mc} is 500 Mbps or 8 Gbps; $p = 0\%$, RTT=0.2 ms	53
3.13	The pipelining effect of packetization; small propagation delay assumed (e.g., datacenter networks	54
3.14	Latency distribution: $m = 16$, $r = 20$ Mbps	55
3.15	Latency distribution: $m = 16$, $r = 60$ Mbps	55
3.16	Block retransmission ratio per aggregate group; different colors used	57
3.17	A study to determine the impact of background traffic on the choice of f_{rcv}	58
3.18	Block retransmission ratio with the patched log parser; different colors used	58
4.1	FMTP packet structure	66
4.2	FMTP message types	67
4.3	FMTP_BOP message format	68

List of Abbreviations

ACK	Acknowledgment
ALC	Asynchronous Layered Coding
ALM	Application-Layer Multicast
BDP	Bandwidth-Delay Product
BOF	Begin-of-File
BOP	Begin-of-Product
BRR	Block Retransmission Ratio
CM	Control Module
CTCP	Circuit TCP
CWR	Congestion Window Reduced
DDS	Data Distribution Service
EOF	End-of-File
EOP	End-of-Product
EWMA	Exponential Weighted Moving Average
FEC	Forward Error Correction
FFDR	FMTP File Delivery Ratio
FMTP	File Multicast Transport Protocol
HTB	Hierarchical Token Bucket
IDD	Internet Data Distribution
IQR	Inter-Quartile Range
IoT	Internet of Things
LDM	Local Data Manager
M2M	Machine to Machine
MPLS	MultiProtocol Label Switching
MPM	Multicast-Push Module
MPUP	Multicast-Push Unicast-Pull
NIC	Network Interface Card
NORM	NACK-Oriented Reliable Multicast
NTP	Network Time Protocol
OFM	OpenFlow Multicast
P2P	Point to Point
PQ	Product Queue
QoS	Quality of Service
RCRM	Rateless Code based Reliable Multicast
RTT	Round Trip Time

SDN	Software Defined Network
SRM	Scalable Reliable Multicast
TBF	Token Bucket Filter
TTL	Time to Live
UCAR	University Corporation for Atmospheric Research
UPM	Unicast-Pull Module
VLAN	Virtual Local Area Network
VM	Virtual Machine
WAN	Wide Area Network

Chapter 1

Introduction

1.1 Problem statement

There is a need in various domains to distribute file streams reliably to multiple receivers. A file stream is defined as a series of files of potentially varying sizes that arrive at random time intervals. Application-Layer Multicast (ALM) is commonly used for distributing data reliably to multiple receivers. For example, an application called Local Data Manager (LDM) [1], used to distribute near real-time meteorological data to multiple organizations, is an ALM solution.

While ALM is easier to deploy than network multicast solutions such as IP multicast, ALM consumes more network bandwidth and compute capacity at the sending hosts. For example, the University Corporation for Atmospheric Research (UCAR) uses LDM in a project called Internet Data Distribution (IDD) [2] to distribute 30 different types of file-streams, e.g., radar data and satellite data. UCAR receives 20 GB/hr from various input sources but transmits 1 TB/hr from its sending compute cluster because each file in each of multiple file-streams is repeated as many times as the number of subscribers. As the data volume and the number of subscribers in such data distribution projects grow, there is an increasing need to find solutions that scale the required resources (bandwidth and CPU capacity) more gradually.

Network multicast solutions have the advantage of requiring lower bandwidth and compute capacity because a sending host can transmit a single copy of a file in the form of packets,

while a switch/router somewhere within the network can make multiple copies of the packets, and transmit these copied packets on to multiple ports. IP multicast has been the only network multicast solution available on Wide-Area Networks (WANs). However, distributed routing protocols, e.g., MSDP [3], which are used to spread reachability information for IP-multicast Class-D addresses, are complex [4], and have been difficult to deploy.

What has changed recently is the introduction of a new networking paradigm in the form of OpenFlow and Software Defined Networks (SDN) [5]. This paradigm promotes a more centralized approach in which a single SDN controller engages in control-plane communications with network switches using protocols such as OpenFlow. Inter-domain control protocols are being developed for SDN controllers in two different domains (autonomous systems) to communicate and jointly configure inter-domain paths [6]. This new paradigm could potentially be leveraged to create control-plane mechanisms for configuring flow-table entries within switches to realize multicast trees. User data can then be transferred from one sender to multiple receivers via such network multicast trees. The term OpenFlow Multicast (OFM) [7] has been used to describe this new network multicast solution.

This work addresses the problem of what functionality is required at the transport and application layers to leverage network multicast solutions such as OFM for reliable file-stream distribution to multiple receivers.

1.2 Proposed solution and contributions

We propose a cross-layer Multicast-Push Unicast-Pull (MPUP) architecture that defines functions at three layers: (i) link layer, (ii) transport layer, and (iii) application layer. It combines (i) a multicast-push function at the transport layer, (ii) a unicast-pull function at the transport and application layers, and (iii) rate-guaranteed link-layer network multicast.

The proposed work uses a cross-layer design to offer a high-performance solution for reliable file-stream distribution to multiple receivers by leveraging new network services on OpenFlow/SDN networks. This solution has the ability to offer customers a solution in which bandwidth and compute capacity can be lowered while still achieving the same performance. Alternatively, for the same bandwidth and compute capacity, higher file-delivery performance

(lower latency) is possible with the MPUP solution. For receivers who do not have access to the new OpenFlow/SDN network services, the sender can continue using the ALM solution for file-stream delivery. Therefore, the MPUP solution can be slowly expanded to cover an increasing number of receivers as the OpenFlow/SDN network services spread in availability. Key contributions of this proposed work are as follows:

1. A File Multicast Transport Protocol (FMTP) that provides reliable multicast file delivery.
2. A cross-layer MPUP architecture for reliable file-stream distribution to multiple receivers.
3. A prototype implementation of the MPUP architecture called LDM7.
4. An experimental comparison of the performance and resource requirements of LDM6, which is an ALM implementation, with LDM7.

This work was reported in one publication [8].

1.3 Thesis layout

This thesis is organized into 5 chapters. Chapter 2 describes a cross-layer Multicast-Push Unicast-Pull (MPUP) architecture as a solution to a file-stream distribution problem. The proposed solution is compared experimentally with the existing solution. Results show that our solution offers better scalability, but requires the deployment of SDNs that support multipoint service. Chapter 3 describes a few enhancements to the MPUP architecture and additional evaluation metrics. Chapter 4 describes the File Multicast Transport Protocol (FMTP) in detail. Chapter 5 summarizes this work and lists future work items.

Chapter 2

A Cross-Layer Multicast-Push Unicast-Pull (MPUP) Architecture for Reliable File-Stream Distribution

2.1 Introduction

This chapter describes a cross-layer multicast-push unicast-pull (MPUP) solution for reliable file-stream distribution. This cross-layer solution includes the application layer, transport layer and link layer. It makes use of an emergent technology, OpenFlow Multicast (OFM), to save bandwidth and compute resources at the file-stream sender. A system model for MPUP defines parameters for each layer, and output metrics for evaluation. A prototype of the MPUP solution was implemented and named as LDM7, which was then compared with UCAR's current Application Layer Multicast (ALM) solution, LDM6. Experimental results showed that LDM7 can save significant sender bandwidth because it requires a constant bandwidth that is independent of the number of receivers, while the bandwidth required by LDM6 is proportional to the number of receivers. LDM7 also saves CPU resources on the sender when compared with LDM6. As the number of receivers increases, LDM6 will

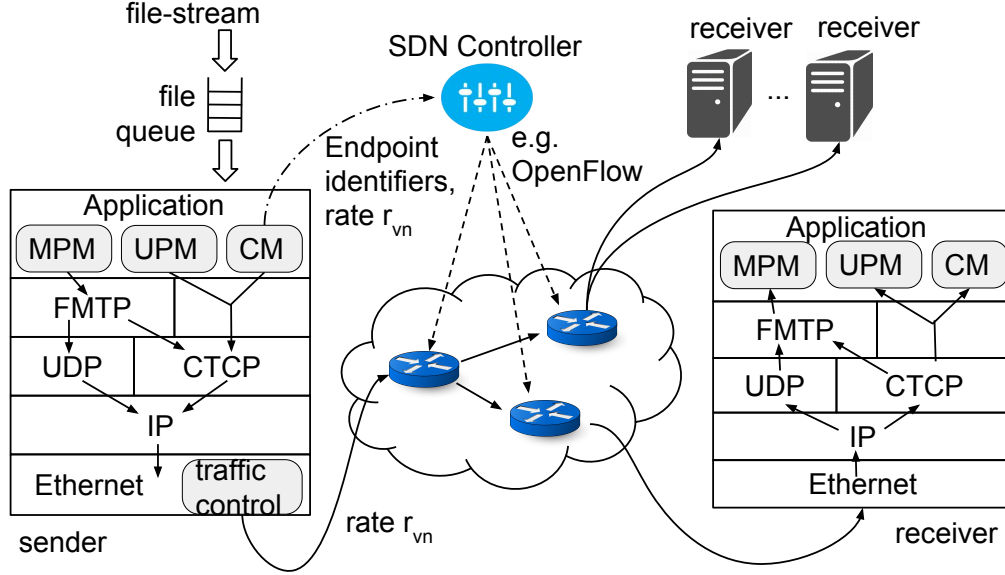


Figure 2.1: Multicast Push Unicast Pull (MPUP) Architecture

be limited by the bandwidth of the sender access link. However, LDM7 can send data at a higher rate as it feeds a single multipoint VLAN. With a higher sending rate, LDM7 can deliver file-streams with a higher throughput, or correspondingly lower latency. The work presented in this chapter was published in an IEEE conference [8] for all relevant papers.

Section 2.2 describes the cross-layer MPUP architecture. Section 2.3 describes a system model for the MPUP architecture, and defines metrics for performance evaluation of MPUP implementations. Section 2.4 describes our prototype implementation of LDM7. Section 2.5 describes our experiments for a comparative evaluation of LDM6 (as a representative ALM solution) with LDM7 (a representative MPUP solution). Related work is reviewed in Section 2.6. Finally, key conclusions are provided in Section 2.7.

2.2 Cross-Layer MPUP architecture

Fig. 2.1 illustrates the cross-layer Multicast-Push Unicast-Pull (MPUP) architecture. The main systems in this architecture are the sender, multiple receivers, network switches, and an SDN controller. This solution assumes the availability of a rate-guaranteed multicast network service supported by the network switches and SDN controller. As shown in Fig. 2.1, a `file queue` is used to receive a stream of files from outside. A sender application reads

and distributes files from the file queue to multiple receivers. The sender application has three modules: *Control Module (CM)*, *Multicast-Push Module (MPM)*, and a *Unicast-Pull Module (UPM)*.

The sender *CM* collects subscription requests for a file-stream from receiver CMs. The sender CM has knowledge of the traffic characteristics of the file-stream, based on which, it can select an appropriate rate r_{vn} for the multipoint virtual network. Endpoint identifiers and the computed rate r_{vn} are sent in a request message by the sender CM to the SDN controller as shown in Fig. 2.1. The SDN controller computes the multicast tree topology based on the specified endpoints, and sends control-plane messages (e.g., OpenFlow) to the switches to configure the multipoint virtual network. The SDN controller also sends the rate parameter r_{vn} to the switches.

Switches can be configured to offer multicast service on different fields of the packet header. One example is the Ethernet IEEE 802.1Q VLAN ID. A switch can replicate frames incoming on a particular port with a particular VLAN ID, translate the VLAN ID in each frame replica to a potentially different value for each outgoing port (as per the forwarding-table entry), and then forward the corresponding frame replica to each outgoing port. Other header fields such as Destination IP address and MultiProtocol Label Switching (MPLS) label field could also be used to realize the multipoint virtual network. The rate parameter is used to configure QoS mechanisms such as traffic policing on ingress ports, and scheduling/shaping on egress ports.

The sender application's *Multicast-Push Module (MPM)* uses File Multicast Transport Protocol (FMTP) [9] to provide reliable multicast service over the multipoint virtual network. FMTP is a protocol in which a file is divided into blocks that are sent over a multicast network tree from the sender to the receivers. FMTP uses the services of UDP and Circuit TCP (CTCP) as shown in Fig. 2.1. FMTP uses UDP datagrams to send its data blocks to an IP-multicast Class-D address, which is configured at the sender and all receivers. Even if the multipoint virtual network is realized at Layer-2, the IP layer is used at the endpoints for ease-of-programming with sockets. The rate of the virtual network could be lower than the sender Network Interface Card (NIC) rate, in which case traffic control is required in the Ethernet layer to limit sending rate as shown in Fig. 2.1. The FMTP blocks carried

within UDP/IP datagrams are rate limited by the traffic control module so as not to exceed the virtual network rate r_{vn} .

Even though the multipoint virtual network is rate-guaranteed, which means packets should not be dropped in switch buffers, bit errors and receive-buffer overflows can occur leading to dropped packets. Dropped FMTP blocks are identified from the Block Sequence Number field carried in FMTP headers since the multipoint virtual network is assumed to guarantee in-sequence delivery. A receiving FMTP can thus send a retransmission request for an errored/dropped FMTP block upon receiving an out-of-sequence block. A CTCP connection is used for sending these requests, and receiving retransmissions of errored/dropped FMTP blocks. CTCP is a variant of TCP in which congestion control is disabled as it is designed for use over rate-guaranteed circuits [10]. Thus, using unicast CTCP connections, an FMTP sender offers FMTP receivers the opportunity to request and receive retransmissions of individual blocks within a file.

For performance reasons, FMTP limits the duration for which it serves retransmissions of lost/errored packets to individual receivers. Without such a limit, an FMTP sender could expend CPU cycles serving a few receivers with high packet-loss rates, while adversely affecting the multicast delivery of new files. Given that this MPUP solution is designed for file-streams, the tradeoff between file-delivery latency and successful file-delivery ratio needs to be considered in the design. To handle this tradeoff, FMTP uses a sender timeout factor f_{snd} to set a maximum retransmission period $\tau_{snd}(n)$ for each file n , given by

$$\tau_{snd}(n) = \max(f_{snd} * S_n / r_{mc}, \max_{1 \leq i \leq m} RTT_i) \quad (2.1)$$

where S_n is the size of file n , r_{mc} is the rate used for multicasting FMTP blocks in UDP/IP datagrams, RTT_i is the Round-Trip Time of receiver i , and m is the number of receivers. A round-trip time is needed for the last data block to reach all receivers and for a receiver that missed this data block to send a block retransmission request to the sender. In wide-area networks, RTT could be higher (e.g., tens of ms) than the first term in (2.1) if files are small and sending rates are high. A receiver's requests for blocks of a file with an expired $\tau_{snd}(n)$ will be rejected by the sender.

The role of the *Unicast Pull Module (UPM)* (see Fig. 2.1) is to handle cases when an FMTP receiver is unable to deliver a whole file to the application. Upon receiving a rejection for retransmission requests for blocks of a file, the FMTP receiver will drop all successfully received blocks of the file and notify the application of a dropped file. The UPM at the receiver will send a “pull” request for the dropped file to the UPM at the sender. The latter sends the dropped file over a separate (unicast) CTCP connection to the receiver.

One final aspect of FMTP is that a receive-side timer was added as part of this work. It was not present in the original FMTP specification [9]. This timer is required because retransmission requests will not be generated fast enough if all blocks at the end of a file are dropped, and there is a large silence period to the next file. FMTP uses a Begin-of-File (BOF) block and End-of-File (EOF) block, both of which are also multicast. Upon receiving a BOF, a receiver timeout value $\tau_{rcv}(n)$ is computed for file n using a factor f_{rcv} as follows:

$$\tau_{rcv}(n) = f_{rcv} \times S_n / r_{mc} \quad (2.2)$$

If EOF for file n is not received within $\tau_{rcv}(n)$ of its BOF arrival, the FMTP receiver will generate retransmission requests for all missing data blocks. The presence of a File Identifier (File ID) in each FMTP block header, which is incremented by 1 for each file in a file stream, allows FMTP receivers to detect complete loss of a file, and to notify the application UPM. RTT does not appear in (2.2) because the timeout interval is between the reception of BOF and reception of EOF. The receiver timeout factor f_{rcv} should be an integer value greater than 1 so that if individual packets are caught in switch buffers behind other packets, the receiver avoids sending premature retransmission requests. Furthermore, f_{snd} should be set to a value larger than f_{rcv} so that the receiver’s block retransmission requests for a file reach the sender before the sender timer corresponding to the file times out.

In *summary*, MPUP requires a rate-guaranteed multipoint virtual network interconnecting the sender and all receivers, a multicast FMTP/UDP/IP session running over this multipoint virtual network, and $2m$ CTCP connections. The Multicast Push Module (MPM) of the application uses the multicast FMTP/UDP/IP session. The first set of m CTCP connections are used by FMTP to handle dropped blocks within files, while the second set of m CTCP

Table 2.1: Model Parameters

Parameter	Symbol
File-stream	F
No. of receivers	m
Round Trip Time	RTT_i
Packet loss rate	p_i
Base rate	r
Virtual network rate	r_{vn}
Sender multicast (mc) rate	r_{mc}
Sender unicast (uc) rate	r_{uc}
Sender traffic-control mc buffer size	b_{mc}
Sender traffic-control uc buffer size	b_{uc}
Receiver UDP buffer size	b_{rcv}
Sender CTCP congestion window	f_{cwnd}
FMTP sender timeout factor	f_{snd}
FMTP receiver timeout factor	f_{rcv}
Sender file queue size	q_{snd}
Sender limit on # files in file queue	n_{snd}

connections are shared by the Unicast Pull Module (UPM) and Control Module (CM) of the application. Pull requests for files and retransmissions of complete files are sent between UPMs, while file-stream subscription requests are sent between CMs.

2.3 System model and metrics

Section 2.3.1 describes a model for MPUP systems. Section 2.3.2 describes metrics used to characterize the performance of an MPUP system.

2.3.1 System model

Table 2.1 lists the parameters of an MPUP system model. The file-stream arrival process **F** is represented by the notation (t_n, S_n) , where the file inter-arrival times t_n could be unevenly spaced [11], and file sizes S_n could vary.

The number of receivers m is 2 or more since MPUP is used for multicast. The network path from a sender to a receiver i is characterized by three properties: RTT, packet loss rate and virtual-network rate, denoted by $\{RTT_i, p_i, r_{vn}\}, 1 \leq i \leq m$. Since the virtual-network rate r_{vn} is the same for all receivers, it is not indexed by i . Before explaining r_{vn} , we explain

the term *base rate* r used in Table 2.1. The rates used for LDM7 and LDM6 experiments are multiplicative factors of this base rate.

The traffic-control module within the sender Ethernet layer will be configured to rate limit multicast packets to a rate r_{mc} , and unicast packets to a rate, r_{uc} , such that $r_{uc} \leq r_{vn} - r_{mc}$. Since the application could send bursts of packets at rates higher than r_{mc} and r_{uc} , buffers of size b_{mc} and b_{uc} should be allocated in the traffic-control module to absorb the bursts.

The receiver UDP buffer size b_{rcv} should be made large enough to hold packets if the rate at which the application removes packets from the UDP buffer is lower than the packet arrival rate.

The next parameter listed in Table 2.1, the sender CTCP congestion window $fcwnd$ is held at a fixed value that is slightly larger than Bandwidth-Delay Product (BDP) on the path with the highest RTT so that the sender can keep sending packets without waiting for acknowledgments. The next two parameters, FMTP sender timeout factor f_{snd} and FMTP receiver timeout factor f_{rcv} , were explained in Section 2.2.

At the application layer, the sender file-queue size q_{snd} should be large enough to absorb bursts in the file-stream given the fixed sending rate at the lowest layer. An application could also limit the number of files n_{snd} in the file queue.

2.3.2 Metrics

Four metrics are defined: throughput, FMTP file delivery ratio (FFDR), sender NIC bandwidth usage and sender CPU utilization.

Throughput A seemingly simple measure to evaluate our multicast service is latency, which is the time taken for a receiver to fully receive a file, if successful. This latency measurement would include the time for the original multicast and for FMTP block retransmissions if any were needed. However, latency depends on file size.

Therefore, a better measure is per-file throughput, which is defined as file size divided by latency. But averaging per-file throughput values across a set of files effectively gives an equal weight for the throughput values of all files in the set irrespective of their sizes. This could result in scenarios in which the average throughput is misleading.

A better representative metric is to compute file-set throughput at each receiver by summing file sizes over a set of file indices and summing corresponding latencies, and dividing these two sums. Then an averaging operation is performed to compute the average file-set throughput values across all receivers. This average metric is referred to as *throughput*, and defined as follows:

$$\Gamma_{(I_1, I_2)} = \frac{1}{m} \sum_{i=1}^{i=m} \frac{\sum_{j \in \mathbf{Z}_i} S_j}{\sum_{j \in \mathbf{Z}_i} D_{ij}} \quad (2.3)$$

where indices I_1 and I_2 are chosen such that $\sum_{k=I_1}^{k=(I_2-1)} S_k < G$ and $\sum_{k=I_1}^{k=I_2} S_k \geq G$, where G is the aggregate file-set (group) size, \mathbf{Z}_i is the subset of files within file-set (I_1, I_2) that were successfully received at receiver i , and D_{ij} is the latency incurred for the delivery of file j at receiver i . Latency should be measured from the time instant when the application provides a file to the FMTP layer at the sender to the time instant when the receiver sends a final acknowledgment of the file to the sender confirming that the whole file was successfully received. Since these two time instants are logged at different hosts, e.g., Network Time Protocol (NTP), is required to determine latency.

File indices are used to characterize (average file-set) throughput on a rolling basis since the file-arrival process \mathbf{F} is a time series.

A fixed time interval is not used as is commonly done for time series because there could be variability in the file arrival process from one time interval to another if the selected interval is too small. We also considered using a fixed number of files in each file-set. But for the reasons cited earlier, the interpretation of mean throughput could be wrong if the total size of files varies from one file-set to the next. Therefore, we chose to use aggregate file-set size as the defining parameter for computing rolling throughput values while allowing the time intervals and number of files to vary between file-sets.

FMTP File Delivery Ratio (FFDR) The metric FFDR is a measure of the success of file delivery by FMTP from a single sender to multiple receivers. A file j is said to have been delivered successfully to receiver i by FMTP if all blocks of file j were received by receiver i either via multicast or via the CTCP connection between the FMTP layers at the

sender and receiver. The presence of the application-layer UPM ensures successful delivery of all files to all receivers as long as receivers request files within the specified duration for which files are served by the UPM. However, this FFDR metric captures the extent to which FMTP is successful in delivering files without the application-layer UPM. FFDR is defined as follows:

$$\Lambda_{(I_1, I_2)} = \frac{1}{m} \sum_{i=1}^{i=m} \frac{|\mathbf{Z}_i|}{N_{(I_1, I_2)}} \quad (2.4)$$

where $N_{(I_1, I_2)}$ is the number of files sent by the multicast sender with indices in the range (I_1, I_2) . Thus FFDR is an average metric, with the averaging done across the ratios computed for all receivers.

Sender NIC bandwidth usage This metric offers a measure of how much traffic is generated by the sender to support file multicasts and retransmissions. It is defined as follows:

$$\Theta(t_1, t_2) = \frac{\mathbb{B}(t_2) - \mathbb{B}(t_1)}{(t_2 - t_1)} \quad (2.5)$$

where t_1, t_2 are time instants, and \mathbb{B} is a counter that tracks the number of bytes transmitted out by the sender NIC. The difference in the counter values at t_2 and at t_1 yields the number of bytes sent in the interval (t_1, t_2) .

Sender CPU utilization The ratio of the aggregate share of CPU time used by the sender-side processes (MPM, UFM and CM, as described in Section 2.2) within a time interval (t_1, t_2) to the total CPU time available in the interval (t_1, t_2) , expressed as a percentage, is sender CPU utilization, $\Phi(t_1, t_2)$.

2.4 Prototype implementation

This section describes an implementation of the cross-layer MPUP architecture. Specifically, a new version of the Local Data Manager (LDM) application used for real-time meteorology data distribution was implemented. The new version is LDM7. The currently used version

LDM6 uses unicast TCP connections from the sender to each receiver. In other words, LDM6 is an ALM solution.

Three terms used in LDM are introduced here for usage in the rest of the paper: (i) “product” is synonymous with “file,” (ii) “Product-Queue (PQ)” describes the file queue of the MPUP architecture (see Fig. 2.1), and (iii) “feedtype” is used to describe a file-stream.

To implement LDM7, the following modifications were made to LDM6: (i) added the interface to FMTP for reliable multicast (LDM7 is the application layer of the MPUP architecture of Fig. 2.1), (ii) modified the PQ component to support access to the product-queue from multiple threads, and (iii) added the UPM module of the MPUP architecture to allow a receiver to request a single product from the sender.

The FMTP design and implementation described in our prior work [9] was the first version of FMTP, FMTPv1. Modifications were required in FMTPv1 to support LDM7. Therefore, a new version FMTPv2 was implemented. It includes the following changes: (i) the addition of a receive-side timer, whose purpose was described in Section 2.2, (ii) modification of the FMTP Application Programming Interface (API) at the sender to allow for FMTP to serve block retransmissions for a product directly from the PQ without creating its own local copy of the product, (iii) elimination of one user-space copy within LDM at the receiver, and (iv) improved support for delivery of file-streams. Further details on the FMTPv2 implementation are provided in Chapter 4.

In addition to coding FMTPv2 and LDM7, for testing purposes, a utility called `pq.insert` was created to emulate the real-world generation of data-products, both in terms of their creation-times and their sizes. First the `notifyme` utility was executed on a local host to collect IDD feedtype metadata (product creation-times and sizes). This utility connects to a UCAR LDM server and obtains the creation times (time instant when a product was injected into the IDD system) and product size for live feedtypes. The received metadata was stored in log files at the local host. The `pq.insert` utility reads these existing LDM log files, and uses the metadata size for each product to create a file filled with random bits (dummy data), which is then added to the PQ at the creation time for the product.

The `pq.insert` utility emulates LDM data-product ingesters, which receive data from radar sites or other such installations. These ingesters then create data products and insert

these products into the PQ. The PQ is a memory-mapped structure that is shared by all processes of an LDM process group. Linux signals such as `SIGCONT` are used by the ingesters to notify the LDM process group when a new data product has been inserted into the PQ. The upstream LDM7 server receives the signal and then reads the product from the PQ and calls the `SendProduct` library function of FMTP to initiate multicast.

2.5 Evaluation and analysis

Section 2.5.1 describes the experimental testbed. Section 2.5.2 describes the input parameter values used in our experiments and the experimental workflow. Section 2.5.3 describes an experiment to determine an appropriate value to use for aggregate file-set size G , which is required for throughput computation (see Section 2.3.2). The second metric, FMTP File Delivery Ratio (FFDR) (see Section 2.3.2) is the focus of the experiment described in Section 2.5.4. Section 2.5.5 compares LDM6 and LDM7 throughput and finds an operating point at which LDM6 and LDM7 achieve the same average throughput for use in the next experiment, which compares resource requirements between LDM6 and LDM7. This comparison is described in Section 2.5.6.

2.5.1 Experimental setup

GENI [12], an NSF supported network testbed, was used to run all the experiments. Users are offered an interface to request slices, each of which consists of one or more virtual machines (VMs) that are interconnected by rate-specified VLANs. For our experiments, we created a slice consisting of VMs at five different racks located at (i) Wayne State University (WSU), Detroit, MI, (ii) StarLight (SL), Chicago, IL, (iii) Oakland Scientific Facility (OSF), Berkeley, CA, (iv) University of Massachusetts (UMass), Amherst, MA, and (v) University of Houston (UH), Houston, TX. The number of VMs at each rack was varied in our experiments to change the number of receivers in our multicast experiments. While the machines had different characteristics at the various racks, we provide an example of the type of resources used in these experiments. One of our VMs was assigned 2 cores (out of the 20 cores of an Intel Xeon E5-2660v2 processor in the physical machine), 6 GB

RAM (out of 96 GB RAM in the physical machine), and 50 GB disk space (out of a large disk array on the physical machine). The physical machine had a 40 Gbps network interface card. A multipoint VLAN was stitched between the top-of-rack switches at these five racks, and the rate of the VLAN was set to 1 Gbps. In other words, r_{vn} was 1 Gbps.

The software used in our experiments consists of: (i) LDM6, (ii) LDM7, (iii) Linux traffic-control (`tc`) utility to control sending rate, (iv) Linux `iptables` to inject artificial packet losses, (v) Linux `sar` utility of the `sysstat` package to measure bandwidth usage, (vi) Linux `ps` to measure CPU utilization, (v) Python scripts to parse LDM log files for throughput and FFDR, and to parse log files created by the `sar` and `ps` programs for bandwidth usage and CPU utilization, respectively, and (vi) R programs to create graphs.

2.5.2 Experiment execution

Experiments were run to measure the performance and resource requirements of (i) LDM7: MPUP implementation, and (ii) LDM6: ALM implementation. We first explain how values were chosen for the input parameters, and then explain the experimental workflow.

Table 2.2 shows values used for the parameters in our experiment. The reader is referred to Table 2.1 for interpretations of the symbols.

Table 2.2: Values for input parameters

Symbol	Value
F	NGRID 06/15/15 00:00-01:00
m	$\{4, \dots, 24\}$
RTT_i	$\{36, 41, 50, 90\}$ ms
p_i	$\{0, 1\}$ %
r	$\{10, \dots, 60\}$ Mbps
r_{vn}	$\geq r_{mc} + r_{uc}$ (LDM7) and $\geq m \times r$ (LDM6)
r_{mc}	r
r_{uc}	r
b_{mc}	600 MB
b_{uc}	600 MB
b_{rcv}	b_{mc}
f_{cwnd}	$1.2 \times r_{uc} \times (\max_{1 \leq i \leq m} RTT_i)$
f_{snd}	5000
f_{rcv}	20
q_{snd}	5 GB
n_{snd}	35000

A data analysis of five IDD feedtypes showed that both file inter-arrival times and file sizes have long-tailed right-skewed distributions [13]. Of the analyzed feedtypes, we chose NGRID as a representative file-stream with which to compare LDM7 and LDM6 performance and resource requirements. Specifically, we collected metadata for 7 hours (12 AM to 7 AM) of the NGRID feedtype on June 15, 2015. As described in Section 2.4, the real metadata collected for the NGRID feedtype was used as input to the `pq.insert` program to create dummy data products with the corresponding creation times and sizes.

The number of receivers was varied from 4 to 24 in steps of 4. One VM on a UMass server was used as the upstream LDM server (sender) and the VMs at the remaining four sites were used for the downstream LDM servers (receivers). If the total number of receivers m used in an experiment was four, one VM was used in each rack. Correspondingly for $m = 24$, six VMs were used in each rack. The RTT values from a UMass VM to VMs in WSU, SL, UH, and OSF, were, as indicated in Table 2.2, 36, 41, 50, and 90 ms, respectively. To compare the performance of LDM6 and LDM7 under lossy conditions, random artificial packet drops were injected at all receivers using Linux `iptables` using the loss rates specified in Table 2.2.

The Linux `tc` utility was used for the traffic control module shown in Fig. 2.1. A combination Hierarchical Token Bucket (HTB) and Bytes First In First Out (BFIFO) queueing disciplines of `tc` were used for LDM7, while Token Bucket Filter (TBF) was used for LDM6. For LDM7, two queues were needed for multicast and retransmissions, and hence HTB was used. The UDP datagrams were directed to one queue, while packets from all $2m$ CTCP connections were directed to the second queue. Borrowing of bandwidth between the queues was disabled because FMTP does not implement flow control, and therefore a high sending rate could overwhelm the UDP buffer at the receivers.

Analysis of the NGRID feedtype metadata in our prior work [13] showed that a sending rate of 10 Mbps and a sending buffer size of 300 MB was sufficient to meet a specified throughput value. Therefore, we used 10 Mbps as a starting value for r_{mc} , but doubled the buffer size b_{mc} to 600 MB to ensure that no packets were dropped by the `tc` buffer at the sender. A dropped packet at the sender will require retransmissions for all the receivers, and is hence avoided. The rate r_{mc} was set as the HTB `rate` and `ceil` parameters for both

queues, while the 600 MB buffer size was set in the BFIFO parameter. The same values were used for rate r_{uc} and buffer size b_{uc} with a rough estimation that if loss rate was 1% and there were 100 receivers, on average, all blocks of all files would be retransmitted once.

For LDM6, the TBF parameters were set as follows: $\text{rate} = m \times r$, $\text{burst} = 50 \text{ KB}$, and $\text{limit} = m \times 2 \times r \times (\max_{1 \leq i \leq m} RTT_i)$ [14]. All $2m$ CTCP connections from the sender to the m receivers were directed to the same queue, whose size is given by the `limit` TBF parameter minus the `burst` (token-bucket) size. Even for the high r rate of 60 Mbps, when $m = 24$, the TBF `limit` value is only 32.4 MB, since the maximum RTT is 90 ms. Compare this 32.4 MB value with the 600 MB value used for the `tc` queues for LDM7. The difference is because UDP and TCP react differently to a full Ethernet-layer `tc` queue. Experiments showed that in the case of UDP, if the `tc` queue is full, packets are simply dropped, whereas TCP will block and hold the packets in its own buffer. Furthermore, if FMTP tries writing to a TCP socket with a full TCP buffer, FMTP will be blocked. Therefore, in LDM6, queueing delays occur only in the PQ, but in LDM7, queueing delays also occur in the `tc` queue. Through experimentation, we found that 600 MB was sufficient to ensure 0 dropped packets by the sender `tc` module for the NGRID 1-hour feedtype used.

The UDP buffer size b_{rcv} at the receiver was also set to 600 MB for LDM7. This large value was chosen to limit packet losses due to flow-control problems at the receiver. With this choice, there were no packet drops at any of the receivers in the experiments in which no artificial packet losses were injected.

The fixed congestion window (f_{cwnd}) parameter of CTCP was set to 20% more than the maximum bandwidth-delay product to ensure continuous sending of segments without waiting for an acknowledgment. The same f_{cwnd} value is applied to all (system-wide) CTCP sockets since this value is set in Linux `sysctl`. The Linux `setsockopt` function can be used to set (possibly different) per-socket values for f_{cwnd} , but this requires modification of the application code.

Values for the two FMTP time-out factors, f_{snd} and f_{rcv} , were selected as follows. The receiver factor f_{rcv} was set to 20 with the expectation that even if many packets from other flows become interspersed between two packets of a given product, a multiplicative factor of 20 applied to the product transmission delay is sufficient for delivery of all blocks of a

product. The sender time-out factor f_{snd} was selected after some experimentation. To avoid packet loss at the sender `tc` queue, f_{snd} was chosen to be the large value of 5000.

The LDM PQ has two parameters: q_{snd} and n_{snd} , which represent the maximum size of the PQ in bytes, and the maximum number of files that can be stored in the PQ. If a newly arriving file causes either of these limits to be exceeded, one or more of the oldest files will be deleted to make space for the new file. We selected the values for these parameters to hold approximately 1-hour of the NGRID file-stream.

The *experimental workflow* consists of four steps: (i) upload LDM6 and LDM7 software, and configuration files (for `tc`, FMTP, CTCP, and LDM), to the remote GENI VMs from a local host, (ii) run the software and monitoring tools on the GENI VMs, (iii) download collected logs from the GENI VMs to a local host, and (iv) run the log parsers to extract performance measures. A script was used for automated execution of this workflow. The monitoring tools, `sar` for bandwidth usage, and `ps` for CPU utilization, run as a Linux `cron` job that is executed every min while the LDM processes are running. The log files include bandwidth logs, CPU logs, and LDM logs. The log parsers extract the four metrics: throughput, FFDR, bandwidth usage, and CPU utilization.

2.5.3 Experiment 1: Determine file-set size for throughput metric

Section 2.3.2 defined a metric called throughput on file-sets rather than single files. Per-receiver file-set throughput was defined as the effective rate at which files within a file-set of size G were received. The purpose of Experiment 1 is to determine an appropriate size for G .

An LDM7 run was executed to send products whose sizes and creation times were extracted from 7 hours of the NGRID feedtype¹. A single receiver was used in this run. Specifically, the data was sent by LDM7 from a UMass host to an SL host. No artificial losses were injected. The rate r_{mc} was set to 100 Mbps.

The throughput metric was computed using (2.3) for different values of G , starting from 1 MB. The side-by-side boxplots in Fig. 2.2 shows the throughput variability across file-sets

¹For only this experiment, we used a 7-hour clip of the NGRID feedtype, specifically 00:00-07:00 on June 15, 2015, while for all other experiments only the first hour data was used as listed in Table 2.2.

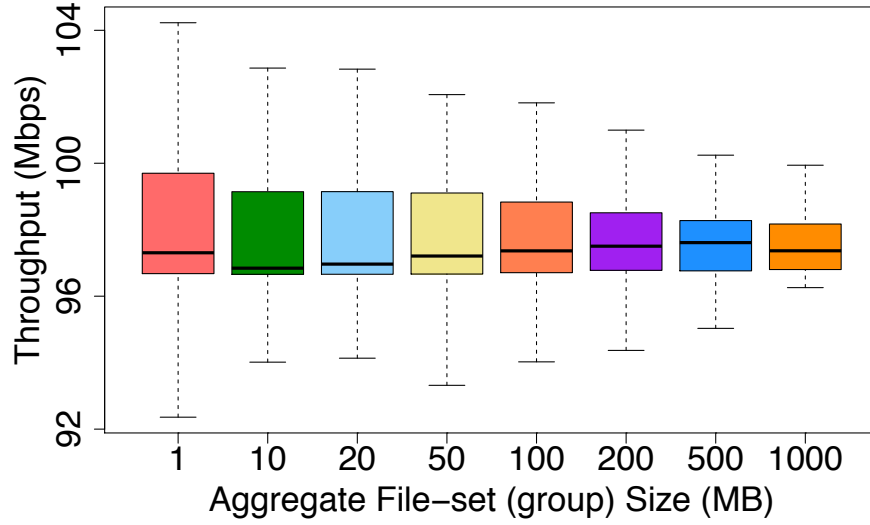


Figure 2.2: Impact of Aggregate File-Set (Group) Size G on Throughput

(groups) for each setting of G . When G was 1 MB, there were 17631 groups within the 7-hour input file-stream. For larger values of G , the total number of groups decreases because the aggregate size of all files in the 7-hour file-stream is fixed (31.5 GB). When $G = 200$ MB, there were only 150 groups. While the median throughput did not change much for different values of G , the Inter-Quartile Range (IQR) decreased as G increased. For example, IQR was 3.02 Mbps when G was 1 MB, while it dropped to 1.73 Mbps for a G value of 200 MB.

Ideally, the selected group size should have a small IQR; however, the number of groups drops as G increases, which then limits the number of computed throughput values within a single run of the experiment. With a G value of 200 MB, the number of groups in the 1-hour file-stream used in the remaining experiments, was 27, which is still large enough for averaging operations. Therefore, we chose $G = 200$ MB as a good compromise between these two opposing factors.

2.5.4 Experiment 2: FMTP File Delivery Ratio (FFDR)

FFDR is a metric that only applies to LDM7. In this experiment, LDM7 was executed with 8 and 16 receivers, using a base rate r of 20 Mbps. With the default setting for f_{snd} of 5000, when no artificial packet drops were injected at the receivers, FFDR was 100% in all

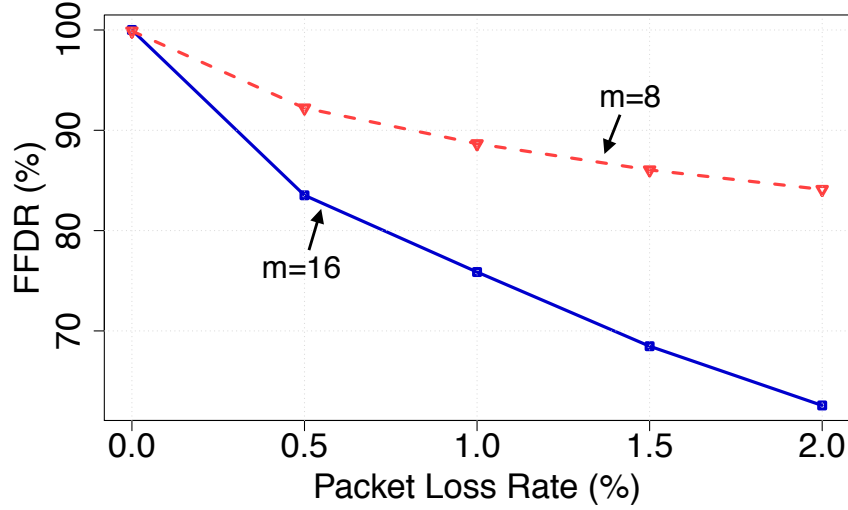


Figure 2.3: FMTP File Delivery Ratio (FFDR) for LDM7

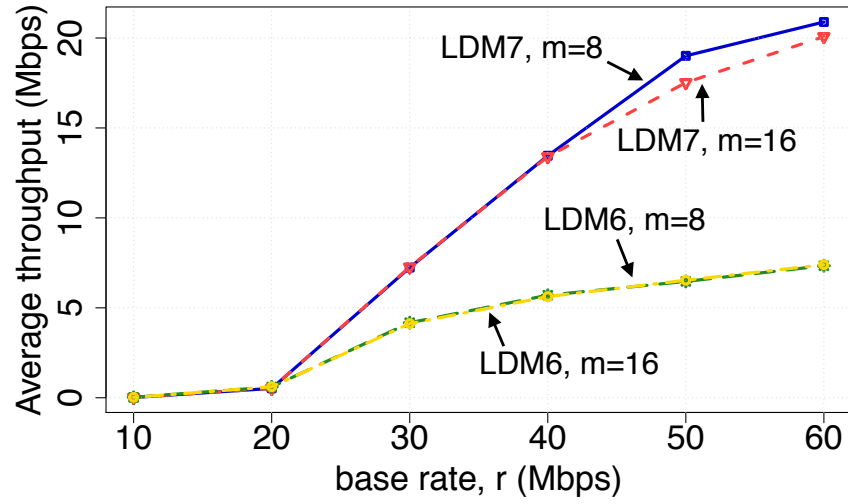
our experiments. However, when artificial packet drops were injected at receivers, FFDR dropped below 100%, which means receivers will need to seek retransmissions of files from the UPM in the sending application.

Fig. 2.3 shows when packet loss rate was set to 1%, FFDR was 88.7% and 75.9% when the number of receivers was 8 and 16, respectively.

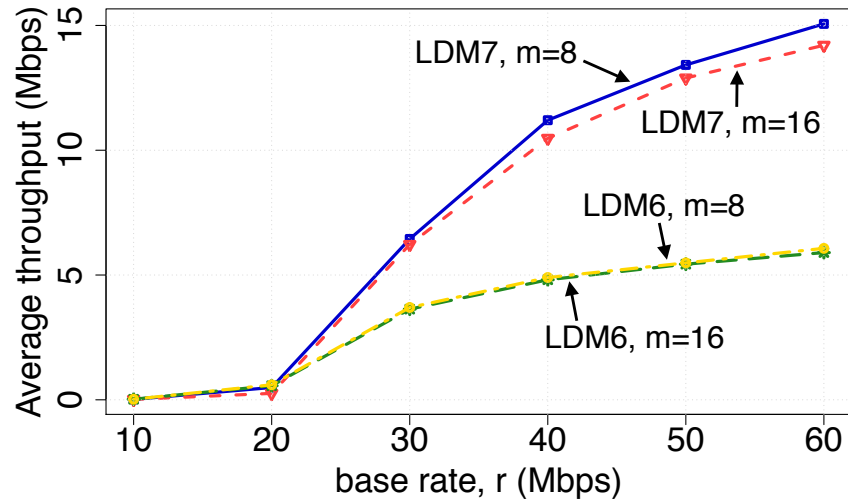
2.5.5 Experiment 3: Throughput measurement

The goals of this experiment were two-fold: (i) compare throughput of LDM6 and LDM7, (ii) find a suitable value for the base rate r to achieve the same throughput with LDM6 and LDM7 so as to enable a fair comparison of resource requirements. The main parameter that was varied in this experiment was the base rate r .

This experiment used four base configurations: (i) LDM7 with 8 receivers, (ii) LDM7 with 16 receivers, (iii) LDM6 with 8 receivers, and (iv) LDM6 with 16 receivers, with two variants: lossless and lossy, for a total of 16 configurations. In the lossless variant, no artificial packet drops were injected in these runs, while in the lossy variant, 1% random packet loss was injected at all receivers. For each of these 8 configurations, 6 runs were executed corresponding to base rate settings from 10 Mbps to 60 Mbps, for a total of 48



(a) No artificial packet loss injections ("lossless")



(b) Packets dropped randomly at 1% at each receiver ("lossy")

Figure 2.4: Impact of tc rate limiting on throughput under lossless/lossy conditions; m : number of receivers

runs. For each of these 48 runs, throughput values were obtained for each of the 27 file-sets of size 200 MB in the 1-hour file-stream. The average throughput across the 27 file-sets was computed for each setting and plotted against the base rate r in Fig. 2.4.

Towards meeting our first goal of comparing LDM6 and LDM7 throughput, Fig. 2.4 shows that LDM7 is able to take better advantage of higher rates than LDM6. This is

because LDM6 needs to create copies of each product and send each copy individually to each receiver, while LDM7 sends out a single file-stream to all receivers.

When the base rate is 60 Mbps, and the number of receivers is 16, the total TBF τ_c rate ($m \times r$; see Section 2.5.2) is 960 Mbps for LDM6. Recall the 1 Gbps limit for r_{vn} in our GENI experimental slice as mentioned in Section 2.5.1. On the other hand, with LDM7, the sender multicast rate r_{mc} , which is equal to the base rate r , could be increased to values above 60 Mbps. But already at the base rate setting of 60 Mbps, average throughput is three-fold better for LDM7 than LDM6 as see in Fig. 2.4a. Increasing the base rate further yielded an average throughput of 75 Mbps, which is a ten-fold increase. This observation becomes important in applications that require a low file-delivery latency. ALM solutions can only support relatively low rates per receiver because of file-stream replication. The higher the number of receivers, the smaller the bandwidth available for any single receiver, which in turn adversely affects file-delivery latency. On the other hand, LDM7 can use a significantly higher-rate virtual network to deliver files with lower latency.

Our second goal was to find an appropriate setting of parameters at which LDM6 and LDM7 achieve the same average throughput for use in our resource-requirements comparison experiment. Fig. 2.4 shows that the 20 Mbps base-rate setting is a good choice as it yields the same throughput for LDM6 and LDM7 under both lossless and lossy conditions.

2.5.6 Experiment 4: Resource requirements comparison

LDM6 and LDM7 were executed with several combinations of parameter settings. The number of receivers was varied from 4 to 24 as described in Section 2.5.2. The same lossless and lossy settings described in Section 2.5.5 were used in these experiments.

Three sets of results are shown: (i) per-min comparison of bandwidth usage by LDM6 and LDM7, (ii) time-averaged bandwidth comparison between LDM6 and LDM7 as a function of the number of receivers, and (iii) time-averaged CPU utilization as a function of the number of receivers.

Fig. 2.5 shows the per-min bandwidth usage on the sender NIC as a function of time, and the aggregate size of all products created in each min. LDM6 uses more bandwidth than LDM7 as seen in the top graph of Fig. 2.5. This is simply because LDM6 is sending eight

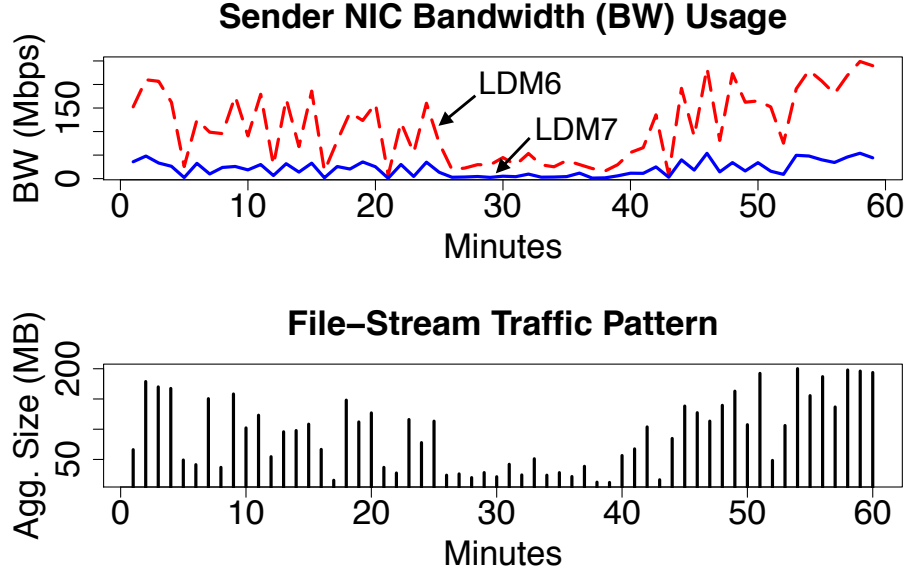
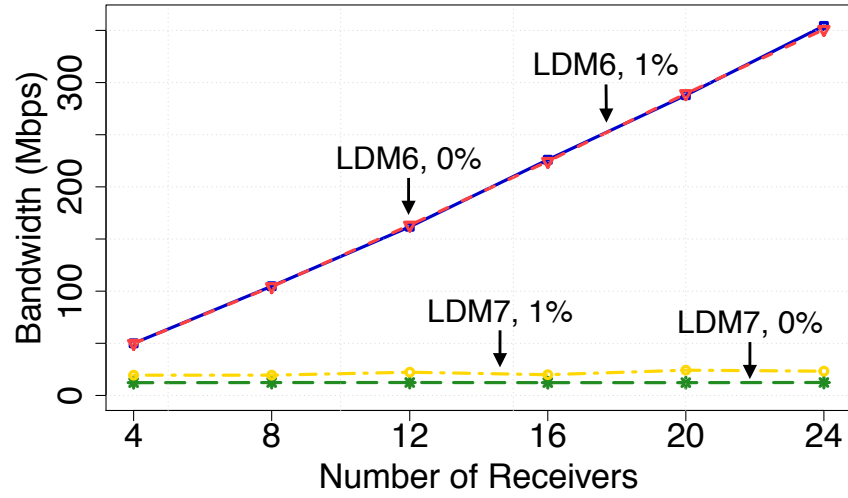


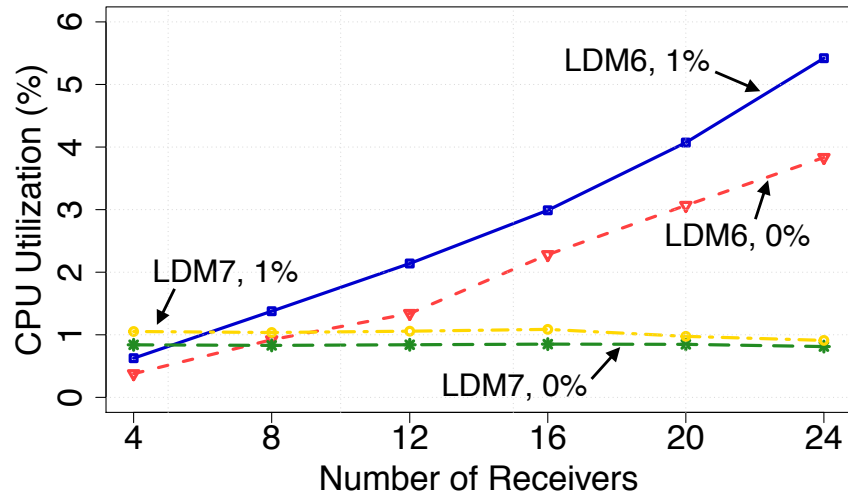
Figure 2.5: Experimental settings: $m = 8$ receivers, lossless setting

copies of each product, while LDM7 is sending only a single copy of each product along with a few block retransmissions. A second observation is that bandwidth usage for both LDM6 and LDM7 drops between mins 25 and 40. This drop is explained in lower graph of Fig. 2.5, which shows that the NGRID traffic pattern had smaller aggregate per-min size of products between these mins than in other time intervals. Having shown this per-min variation in bandwidth usage, we can now use the time-averaged bandwidth in the next figure.

Fig. 2.6a shows a time-averaged bandwidth comparison between LDM6 and LDM7 as a function of the number of receivers. Since LDM6 needs to send m copies of all products, it consumes more bandwidth than LDM7, and further the difference in bandwidth requirement between LDM6 and LDM7 increases with the number of receivers. For LDM7, bandwidth usage is almost independent of the number of receivers and is about 12.4 Mbps and 21.4 Mbps for the 0% and 1% cases settings, respectively. While the multicast rate r_{mc} is 20 Mbps, because of silence periods between products, and smaller product sizes in some mins as seen in Fig. 2.5, the time-averaged bandwidth usage is less than r_{mc} . Data analysis of the 1-hour NGRID file-stream shows that while the third-quartile for file inter-arrival time is 34 ms, the 90% is 297 ms, and 1% of files arrive more than 1.8 sec after their predecessors. In other words, there are gaps between file arrivals, which explains why the NIC was not



(a) Time-averaged sender NIC bandwidth usage



(b) Time-averaged sender CPU utilization

Figure 2.6: Resource utilization; $r = 20$ Mbps; (LDM version, loss rate) are shown for each plot

used all the time even at the low rate setting of 20 Mbps. Finally, we observe that LDM6 bandwidth usage increases linearly, reaching 350 Mbps for 24 receivers. NGRID is just one of 30 feedtypes distributed by the IDD project, and the number of subscribers, which is currently 240, is growing. Therefore the total bandwidth required on the UCAR WAN access link for the IDD project is already high, and growing. Use of LDM7 will offer a

significant reduction in the access-link bandwidth and cluster needed to support real-time data distribution.

Figs. 2.6b plots time-averaged sender CPU utilization for LDM6 and LDM7 as a function of number of receivers. Under both lossless and lossy settings, LDM6 enjoys lower sender CPU utilization when the number of receivers, m , is 4. This is because FMTP runs in user space, which incurs more CPU cycles than CTCP used by LDM6, which runs in the kernel. However, with larger numbers of receivers, CPU utilization increases rapidly for LDM6, while it stays almost flat for LDM7. This is because there is a separate upstream LDM process corresponding to each receiver in LDM6. While LDM7 also requires one sender process per receiver, these sender processes just handle retransmission requests and hence do not consume much CPU time. Thus, the total CPU utilization for LDM6 is an almost linear function of m , while for LDM7, CPU utilization is almost independent of m . In the lossy setting, higher CPU time is needed for both LDM6 and LDM7. Since the random packet drops were injected at all receivers, the additional CPU time needed at the sender increased linearly with the number of receivers.

In *summary*, it is clear that LDM7 requires fewer bandwidth and CPU resources than LDM6, with the resource savings increasing almost linearly with the number of receivers.

2.6 Related work

A Software Defined Network Aware Pub/Sub (SAPS) solution that uses a hybrid approach with both Application Layer Multicast (ALM) and OpenFlow based multicast (OFM) was proposed for IoT/M2M and other applications [7]. The messages sent to OFM use UDP, while ALM messages use TCP. Message latency and the total number of messages exchanged were the parameters of interest. Our work addresses reliability aspects in FMTP and in the UPM module of the application to guarantee delivery of files.

Another recent solution called Rateless Code based Reliable Multicast (RCRM) [15] protocol builds on Data Distribution Service (DDS), which uses the Publish-Subscribe model. RCRM removes the use of heartbeats in DDS, and instead uses ACKs that are sent after many received messages are decoded. It relies on a form of FEC coding for reliability. In

contrast, FMTP does not use coding as it is designed for smaller numbers of receivers (in the hundreds); instead it uses negative acknowledgments to obtain retransmissions on unicast connections.

There are two IETF reliable multicast solutions: NACK-Oriented Reliable Multicast (NORM) [16] and Asynchronous Layered Coding (ALC) [17]. NORM is an NACK-based reliable multicast protocol that uses multicast in the first transmission attempt, and then either multicast or unicast for retransmitting lost packets. NORM is designed for IP-multicast, and since IP networks do not offer rate guarantees, NORM includes a rate-based congestion control mechanism. FMTP avoids data-plane congestion control by using SDN controllers to configure a rate-guaranteed multipoint VLAN.

Asynchronous Layered Coding (ALC) [17] is a massively scalable reliable content delivery protocol. The data is sent on multiple channels at different rates, and encoded with FEC for reliability. Receivers can obtain packets from multiple channels. There are no positive or negative ACKs from receivers. Hence this solution scales to a million receivers. FLUTE [18] is a protocol for unidirectional delivery of files to multiple receivers. It is built on ALC and is therefore massively scalable. FMTP is designed for scenarios with hundreds of receivers, not millions of receivers, and hence it avoids the overhead of sending to multiple channels and FEC.

We already compared LDM7, an implementation of our MPUP architecture to an ALM implementation, LDM6. However, there are other ALM approaches such as P2P, an example of which is Bit Torrent [19]. For file-streams carrying new data, P2P solutions are not ideal as they require multiple downloads before all receivers can receive all blocks. If latency is a consideration, a network multicast solution such as our MPUP architecture is more suitable.

2.7 Conclusions

This paper proposed and evaluated a cross-layer multicast-push unicast-pull (MPUP) architecture for reliable file-stream distribution. The architecture combines a rate-guaranteed multipoint virtual network service, a reliable File Multicast Transport Protocol (FMTP)

that uses a multicast-push with UDP and unicast-pull with Circuit TCP (CTCP), and an application-layer unicast-pull module. A new version of the Local Data Manager (LDM) application, LDM7, was implemented based on the MPUP architecture. LDM6, the current version, uses application-layer multicast to send near real-time file-streams of meteorology data to 240 institutional subscribers. LDM7 and LDM6 performance and resource requirements were compared in an experimental evaluation. First, LDM7 achieves higher throughput (lower latency) in file delivery when compared to LDM6 since the sending rate can be higher for LDM7. Second, for a given sending rate, at which both LDM7 and LDM6 yield the same throughput performance, the sender NIC bandwidth usage and sender CPU utilization for LDM6 increase linearly with the number of receivers in the multicast group, while both metrics stay almost constant for LDM7. For example, with 24 receivers, to serve a particular filestream, LDM6 needed 350 Mbps, while LDM7 needed only 12.4 Mbps in a lossless setting and 21.4 Mbps in a lossy setting.

Chapter 3

Enhanced MPUP Architecture

3.1 Introduction

Three enhancements were made to the MPUP solution described in Chapter 2. First, the maximum retransmission period ($\tau_{snd}(n)$), as defined in Equation (2.1), used by the FMTP sender was re-defined. Second, the value setting of the sender traffic-control multicast buffer size, b_{mc} (see Table 2.1), was re-visited. Third, we studied the impact of increasing the sender multicast rate r_{mc} (see Table 2.1) on LDM7 throughput towards selecting appropriate value. These three enhancements are described in sections 3.2, 3.3 and 3.4. Section 3.2 also describes experimental results presenting the three metrics (throughput, FFDR and resource utilization) defined in Chapter 2. Section 3.5 defines new metrics and presents results of the new metrics. Section 3.6 concludes the chapter.

3.2 New maximum retransmission period design

In this section, after describing our reasons for revisiting the maximum retransmission period definition, we propose a new method for selecting this period. To evaluate the new design, we ran experiments using the same 1-hour NGRID traffic that was used in the experiments presented in Chapter 2. Our experimental evaluation shows that the FFDR is better than under the previous design, and throughput and resource utilization were largely unaffected.

3.2.1 Problem statement

In Chapter 2, section 2.2, we defined the FMTP parameter maximum retransmission period $\tau_{snd}(n)$ for file n . This parameter limits the duration for which an FMTP sender serves retransmission requests for file n . With this limit, an FMTP sender prevents receivers with high packet loss rates from adversely affecting the multicast throughput of other receivers. The specific definition of $\tau_{snd}(n)$ in equation (2.1) was formulated on an assumption that file transmission delay will dominate the total delay. Instead we found that propagation delay and sending-host queueing delay were the dominant factors. The impact of this incorrect assumption is described below.

Fig. 2.3 shows that even if f_{snd} is large, e.g., 5000, FFDR is not 100% when packet loss rate is non-zero. LDM7 log files showed that a majority of unsuccessful files, i.e., files that were not fully delivered because of sender-side retransmission period timeout, were small-sized. For example, if $f_{snd} = 5000$ and $r_{mc} = 20$ Mbps, the retransmission period for the smallest observed file, which had a size of 0.06 KB [13], would have been set to 0.12 second. This retransmission period is so small that the sender-side timer could have expired even when the file was still queued in the sender `tc` buffer because RTT could be high (e.g., maximum RTT was 90 ms in our 5-rack GENI slice). If this happens, none of the receivers would have received this file, which would result in a significant number of retransmissions.

On the other hand, $\tau_{snd}(n)$ for the largest file, which had a size of 23.7 MB [13], would have been set to 13.17 hours. However, LDM7 guarantees to hold each file in its product queue for only one hour. This could cause a problem when $\tau_{snd}(n)$ is greater than one hour because FMTP does not hold a copy of the file in its memory space; rather it serves retransmission requests by reading blocks of files directly from the product queue. If $\tau_{snd}(n)$ is larger than one hour for any file, the FMTP sender could be performing book keeping operations (waiting for retransmission requests) for files that have already been removed by LDM7 from the product queue. Therefore, application constraints should be considered when deciding the FMTP maximum retransmission period.

In summary, the above example illustrates our motivation for re-designing the method for computing the FMTP-sender maximum retransmission period.

3.2.2 Design alternatives

Two alternatives were considered: file-dependent solution, and file-independent solution. In both solutions, an exponentially weighted moving average (EWMA) scheme is used to dynamically adjust the value of the timer. As the file-independent solution is simpler of the two alternatives, following Occam's razor, we chose this solution for implementation.

File-dependent solution: An analysis on the time incurred for file delivery showed that a significant component of the delay was buffering delay at the sending host. This delay is incurred because the product arrival process is bursty, and there are some time intervals in which a sudden burst of products could cause LDM7/FTMP to deliver data at a rate higher than the \mathbf{tc} rate. In such instances, FMTP packets will get held up in the \mathbf{tc} -layer buffer awaiting transmission. This queueing delay should be considered in setting the maximum retransmission period, $\tau_{snd}(n)$.

Considering the several protocol layers shown in Fig. 2.1, the file delivery-time can be characterized as:

$$d_{total}(n) = d_{stack_s} + d_{tc}(n) + d_{trans}(n) + \max_{1 \leq i \leq m} (2 \times prop_i + queue_i(n)) + d_{stack_r} + \max_{1 \leq i \leq m} retx_i(n) \quad (3.1)$$

where d_{stack_s} and d_{stack_r} represent the delays of the network stack in the sender and the receiver, respectively, $d_{tc}(n)$ is the \mathbf{tc} -layer buffering delay incurred by file n , $d_{trans}(n)$ is the transmission delay of file n , $prop_i$ is the propagation delay to receiver i , $queue_i(n)$ is the cumulative queueing delays incurred by all the packets of the file on the path from the sender to receiver i , and m is the number of receivers, and $retx_i(n)$ is the total delay incurred in serving retransmissions, if any, to receiver i for file n . The reason the propagation delay $prop_i$ is multiplied by 2 is because the acknowledgment of successful file reception needs to arrive at the sender before the maximum retransmission period timer expires. Other delays such as transmission delay are neglected as acknowledgment packets are small. Since processor clock speeds are typically in the GHz range, if the \mathbf{tc} sending rate is set in the Mbps range, the network-stack processing delays, d_{stack_s} and d_{stack_r} can be neglected. The

delivery-time model simplifies to:

$$\begin{aligned} d_{total}(n) &= d_{tc}(n) + d_{trans}(n) + \max_{1 \leq i \leq m} (2 \times prop_i + queue_i) + retx_i(n) \\ &= \frac{S_n}{r_{mc}} + d_{variant}(n) \end{aligned} \quad (3.2)$$

since the transmission delay $d_{trans}(n)$ can be computed for each file by the sender from the file size S_n , and the sender multicast rate r_{mc} , and $d_{variant}(n)$ represents the variable portion of the total delay. While $prop_i$ will not change, it is difficult to isolate this value, and is hence included along with all other variable components in $d_{variant}(n)$.

To set the maximum retransmission period, the sender needs to estimate $d_{variant}(n)$, but herein lies a challenge. The EWMA scheme strives to store minimal data about each file, and therefore typically stores just an average value, which is updated based on each new measurement. But in this context, each measurement provides a $d_{total}(n)$, which is a function of file size S_n . As the FMTP sender knows the starting time point ($t_{start}(n)$) at which file n was transmitted, and the ending time point ($t_{end}(n)$) at which the sender receives the last confirmation of successful reception of all blocks of file n . The time interval $t_{end}(n) - t_{start}(n)$ offers the FMTP sender a measurement for $d_{total}(n)$. Next, the FMTP sender can subtract out the transmission delay S_n/r_{mc} for file n to obtain $d_{variant}(n)$. But this $d_{variant}(n)$ cannot be used directly in the EWMA average estimation as it depends upon file size. Therefore, we propose to divide this delay $d_{variant}(n)$ by the file size expressed in packets (assuming maximum sized packets) to normalize this measure before applying EWMA.

An average value of per-packet variable delay is denoted $EstDelay$. Upon the successful acknowledgement of a file, $EstDelay$ is updated as follows:

$$EstDelay = 0.875 \times EstDelay' + 0.125 \times \frac{d_{variant}}{S_n/P} \quad (3.3)$$

where $EstDelay'$ is the current average value, and P is packet size. Further, we define $DevDelay$ as:

$$DevDelay = 0.75 \times DevDelay' + 0.25 \times \left| \frac{d_{variant}}{S_n/P} - EstDelay \right| \quad (3.4)$$

where $DevDelay'$ is an estimate of the current deviation. In the above formulations, the 0.875/0.125 and 0.75/0.25 weights are recommended based on the successful use of these values for retransmission timeout computation by a TCP sender [20].

The maximum retransmission period for file i by:

$$\tau_{snd}(n) = d_{total}(n) = EstDelay \times (S_n/P) + 4 \times DevDelay \times (S_n/P) + S_n/r_{mc}, \quad (3.5)$$

per (3.2). In other words, the EWMA approach is applied on the variable portion of the delay normalized to file size expressed in packets, and the transmission delay of the file is added to this variable portion.

File-independent solution: Typically, each protocol layer sets its own parameters to achieve its objectives. For example, in order to achieve reliable data delivery, TCP selects its own retransmission timeout value for its segments. However, in our FMTP design, as described in Chapter 2, section 2.4, for performance reasons, the FMTP sender does not create its own local copy of a product. Instead, it serves blocks of data, for the original multicast as well as for retransmission, directly from the product queue held by the LDM7 application. Given this design choice, the time duration for which LDM7 guarantees storage for a product in its product queue becomes a critical determinant of the FMTP maximum retransmission period. Therefore, in this file-independent solution, the application, e.g., LDM7, provides a single value for the maximum retransmission period for all files:

$$\tau_{snd}(n) = c \quad (3.6)$$

While the application, e.g., LDM7, necessarily needs to dictate a maximum value for this FMTP timer (given the use of this approach whereby FMTP serves files directly from

a memory-mapped structure held in application space), layers below FMTP and path characteristics dictate a minimum timer value. For example, LDM7 guarantees that files will be held in its product queue for 1 hour, which sets the maximum value for the timer. On the other hand, RTT, queueing delays, \mathbf{tc} -layer buffering delays, packet loss rates (which determine the time for retransmissions) should be considered to determine the minimum value for the timer as shown in (3.1). As the lower-layers-dictated timer value could be larger than the application-dictated timer value, a wholistic approach is required to setting this timer value. Such a solution is proposed below.

The solution is to deploy an external application management system that collects LDM7 log files from the sender, and parses these log files at fixed intervals (e.g., one hour) to determine what would have been an ideal value for the file-independent single value c shown in (3.6). The ideal value is simply the maximum $d_{total}(n)$ across all files sent in the past interval. In other words, a post-facto analysis is proposed to determine the ideal for already transmitted files in a given environment (which includes the file-arrival process, set of receivers and the network paths). This computed ideal value is then used in an EWMA scheme to update a running average value that is then used as a prediction for the ideal value for the files that are to be transmitted in the next interval. This predicted ideal value is then set as c for the FMTP sender to use for all files transmitted in the next interval. This method is represented as follows

$$c = \frac{7}{8} \times c' + \frac{1}{8} \times c_{ideal}, \quad (3.7)$$

where c' is the c value used in the past interval, c_{ideal} is the ideal value that should have been used in the past interval, and c is the new value computed for use in the next interval. This process is repeated at the end of each time interval.

As an example, we parsed the sender-side log file created when the 1-hour NGRID feedtype traffic trace was sent to multiple receivers on the 5-rack GENI slice, and c_{ideal} was found to be 2 mins. In other words, every file was delivered to all receivers in less than 2 mins. Had the FMTP sender maximum retransmission period been set to 2 min, no receiver would have received a rejection to its request for block retransmissions for any file

in the 1-hour trace. A smaller value for the maximum retransmission timer period is better than the default 1-hour value required by the LDM7 application because the FMTP layer needs to maintain its per-file per-receiver state information for the duration of the maximum retransmission timer.

Given our argument in favor of simplicity, we chose to implement the file-independent solution. This solution is evaluated on the same 5-rack GENI slice using the same 1-hour NGRID traffic trace as used in the experiments described in Chapter 2. The next three sub-sections present the same three output metrics used in Chapter 2, FFDR, throughput, and resource requirements.

3.2.3 FFDR measurement

An experiment was executed with $m = 16$, $r_{mc} = 20$ Mbps and $p = 1\%$. The LDM7 sender logs the start timestamp $t_{start}(n)$ for each file n (time at which the file was inserted into the PQ) and the end timestamp $t_{end}(n)$ for file n (time at which the sender receives the last confirmation of successful reception of all blocks of file n). From these timestamps, $d_{total}(n)$ was computed as $t_{end}(n) - t_{start}(n)$ for each file n . Analysis of the measured values of $d_{total}(n)$ for all files in the 1-hour trace revealed that c_{ideal} was 2 mins. for this experimental setting and this traffic trace. We refer to $d_{total}(n)$ as the *delivery-time* for file n .

In order to study the impact of loss rate, we conducted another run with $m = 16$, $r = 20$ Mbps and $p = 2\%$. Fig. 3.1 shows a histogram of $d_{total}(n)$ for the whole trace for both settings of the artificially injected packet loss rates 1% and 2%.

Most of the files required a delivery-time of a few seconds while some of them needed over 100 seconds. The maximum delivery-times in the $p = 1\%$ run and $p = 2\%$ run were 113.3 seconds and 113.5 seconds, respectively. We can draw two conclusions from Fig. 3.1. First, the impact of loss rate on delivery-time is not significant even when packet loss rate was 2%. Second, the FMTP-sender maximum retransmission period timer c could have been set to 2 minutes for this environment and traffic trace.

With conclusion 2, we set the maximum retransmission period to 2 minutes, and executed the experiment without changing any of the other parameters except the packet loss rate p , which was varied from 0% to 2% in steps of 0.5%. Fig. 3.2 shows that our new solution for

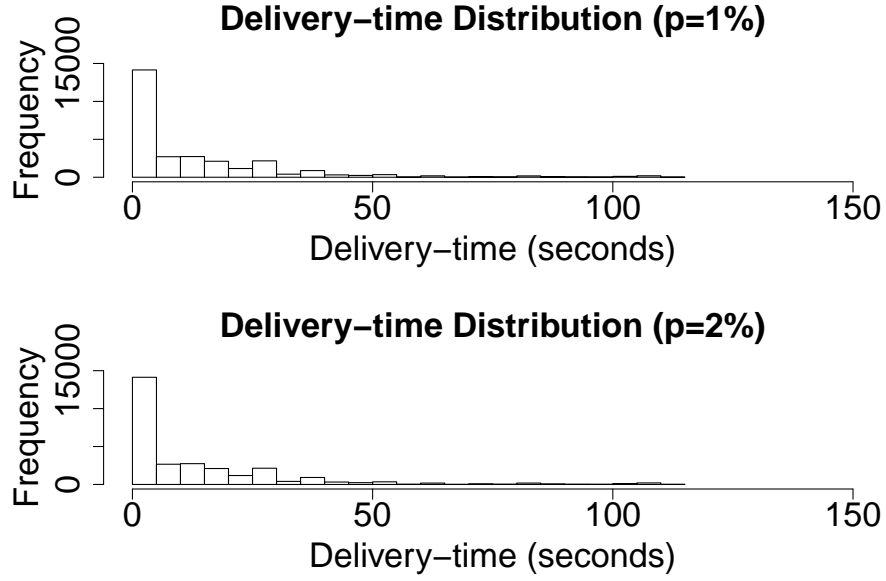


Figure 3.1: Deliver-time distribution for all files in the 1-hour NGRID trace; $m = 16$, $r = 20$ Mbps;

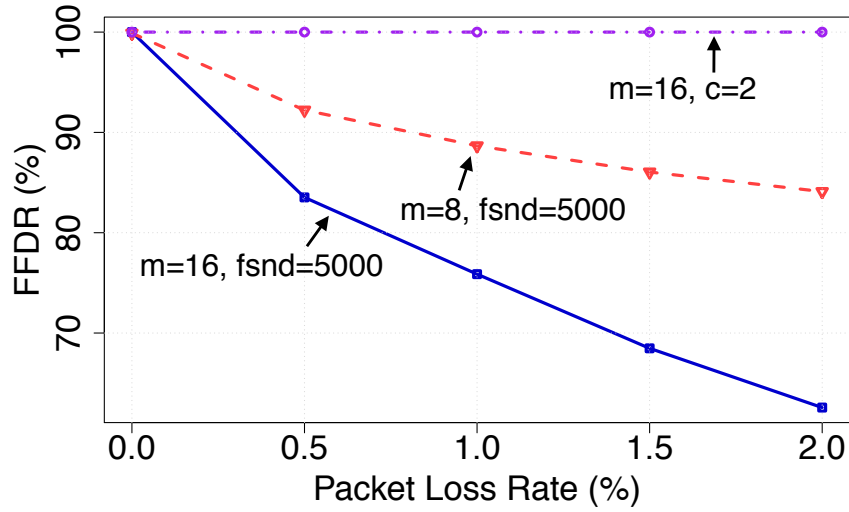


Figure 3.2: FFDR with new $\tau_{snd}(n)$ design

setting the maximum retransmission period could achieve 100% FFDR, if the environment does not suffer from significant variations. In other words, if the new c value computed in each interval is close to the c_{ideal} value, even a small value of the maximum retransmission period such as 2 mins is sufficient. Recall that in the previous design, the file-dependent

timer $\tau_{snd}(n)$ would have been 13.17 hours for the largest sized file given the other parameter settings. For all the following experiments, we use the 2-min timer value as the environment and traffic trace was unchanged.

3.2.4 Throughput measurement

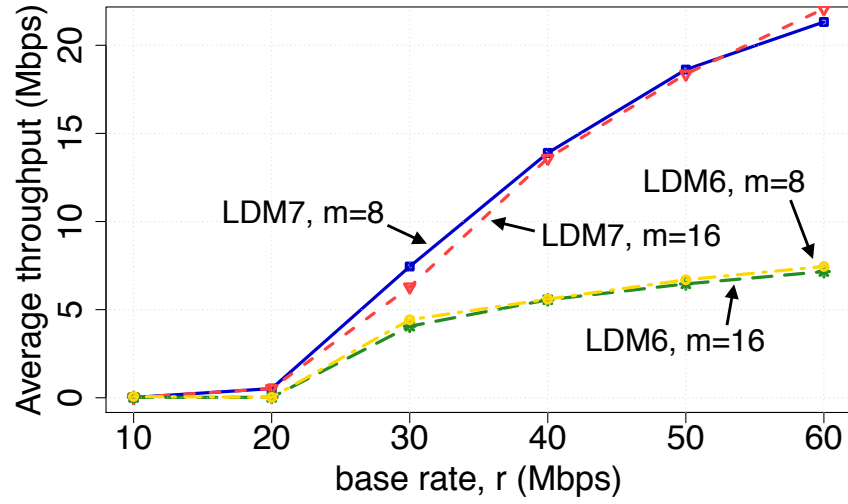
A set of experiments were executed to: (i) compare the throughput of LDM6 and LDM7, (ii) find a suitable value for the base rate r to achieve a close throughput for fair comparison. In this set of experiments, throughput was measured for both lossless and lossy conditions. In the lossy case, the artificially injected packet loss rate was set to 1%. The base rate r was varied from 10 Mbps to 60 Mbps, and for each setting of r , two values, 8 and 16 were used for the number of receivers m . Each run used the same 1-hour NGRID feedtype.

Throughput results for LDM7 with the new design for the maximum retransmission period are similar to the results presented in Chapter 2 under the old design. Fig. 3.3 shows the average throughput for LDM6 and LDM7 in the lossless and lossy settings. Regardless of the number of receivers, LDM7 achieved better throughput than LDM6. In the lossless setting, when base rate r was set to 60 Mbps, LDM7 achieved an average throughput of 22 Mbps while LDM6 achieved only 7 Mbps, which is roughly one-third of the LDM7 value.

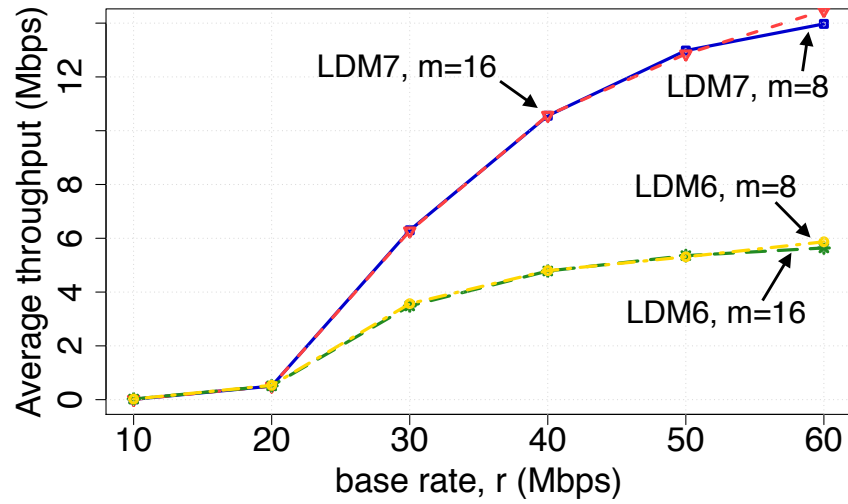
In addition, Fig. 3.3 shows that $r = 20$ Mbps is still a fair choice for comparing the resource requirements of LDM6 and LDM7 as the throughput is roughly equal.

3.2.5 Resource requirements

LDM6 and LDM7 were executed with several combinations of parameter settings. The number of receivers m was varied from 4 to 24. The same lossless and lossy settings (0% and 1%) were used in these experiments. Fig. 3.4a shows a time-averaged bandwidth comparison between LDM6 and LDM7 as a function of the number of receivers. These results also do not show a significant difference when compared to the results presented in Chapter 2 in spite of the change in the maximum retransmission period. Since LDM6 needs to send m copies of all products, it consumes more bandwidth than LDM7, and further the difference in bandwidth requirement between LDM6 and LDM7 increases with the number of receivers. For LDM7, bandwidth usage is almost independent of the number of receivers, and is 12 and



(a) No artificial packet loss injections ("lossless")

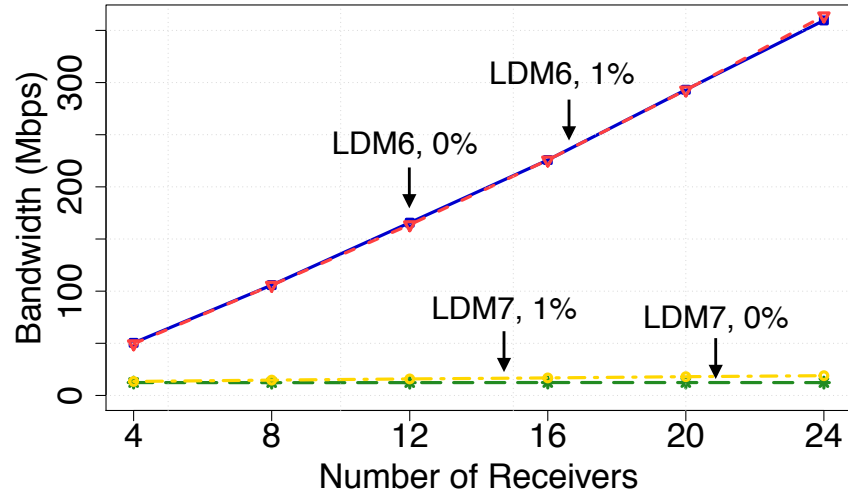


(b) Packets dropped randomly at 1% at each receiver ("lossy")

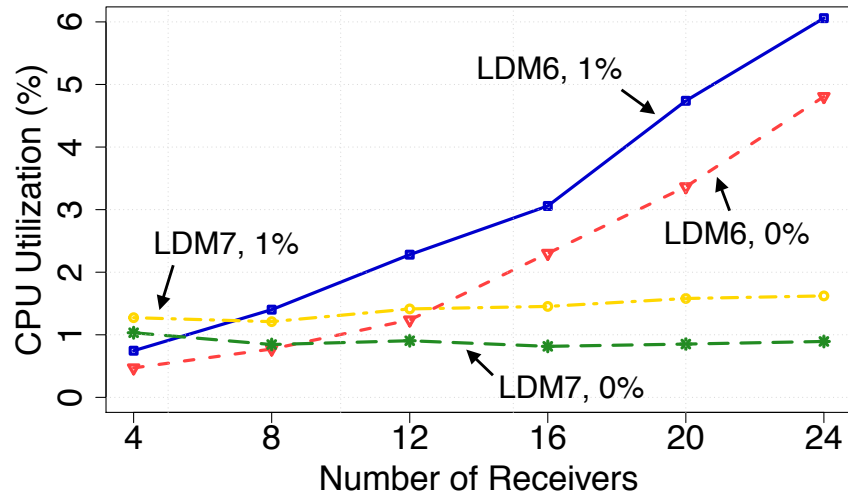
Figure 3.3: Impact of tc rate limiting on throughput under lossless/lossy conditions; m : number of receivers

18 Mbps for the 0% and 1% cases settings, respectively. LDM6 bandwidth usage increases linearly, reaching 364 Mbps for 24 receivers. Thus, LDM7 offers a significant reduction in the access-link bandwidth needed to support real-time data distribution.

Figs. 3.4b plots time-averaged sender CPU utilization for LDM6 and LDM7 as a function of number of receivers. These results also do not show a significant difference when compared



(a) Time-averaged sender NIC bandwidth usage



(b) Time-averaged sender CPU utilization

Figure 3.4: Resource utilization; $r = 20$ Mbps; (LDM version, loss rate) are shown for each plot

to the results presented in Chapter 2. The explanation for the graphs is the same as that provided for Fig. 2.6, Chapter 2.

Overall, the new sender maximum retransmission period design does not impact the advantages of LDM7 over LDM6 in bandwidth and CPU requirements.

3.3 Sender `tc`-layer buffer size revisit

In Chapter 2, Section 2.5.2, we noted that there was a significant difference in the size of the `tc`-layer buffer size needed for LDM6/CTCP (only 32.4 MB) vs. the size needed for LDM7/FMTP/UDP (600 MB). In this section, we describe our in-depth study of this difference to determine whether a smaller-sized buffer could be used for LDM7/FMTP/UDP.

Section 3.3.1 describes a set of experiments to compare the `tc`-layer buffer size needed by LDM6/CTCP and by LDM7/FMTP/UDP. Also, this section describes a set of experiments to determine the minimum `tc`-layer buffer size needed for LDM7/FMTP/UDP as a function of the sender multicast rate. The results of the experiments presented in Section 3.3.1 show that larger sized buffers are needed for LDM7/FMTP/UDP when compared to LDM6/CTCP, but the reasons for this difference are unknown. Section 3.3.2 describes an experimental study conducted to understand these differences. To remove application-layer differences (i.e., differences between LDM6 and LDM7 could influence results), we used a common application `iperf3` to study differences in three transport-layer protocols, TCP, CTCP and UDP. Finally, Section 3.3.3 translates the findings from Section 3.3.2 into answers to our question of why LDM7/FMTP/UDP needs a larger `tc`-layer buffer than LDM6/CTCP.

3.3.1 Experiments to determine minimum `tc`-layer buffer size for LDM7/FMTP/UDP

We first conducted a set of four experiments to study the impact of `tc`-layer buffer size before undertaking a systematic experimental study to determine the minimum `tc`-layer buffer size needed for LDM7/FMTP/UDP for different values of the sender multicast rate, r_{mc} . For all experiments, we used the same 5-rack GENI slice used in the experiments described in Chapter 2, Section 2.5.1, and the same 1-hour NGRID data was used to create the input file-stream. The number of receivers, m , was 16, the sender multicast rate, r_{mc} , was 20 Mbps, and no artificial packet losses were injected. The maximum RTT, as shown in Table 2.2 of Chapter 2, Section 2.5.2, was 90ms.

In *experiment 1*, we ran LDM6/CTCP with the TBF `tc` rate set to $r \times m = 20 \times 16 = 320$ Mbps, and the `limit` parameter set to $2 \times BDP \times m = (2 \times 20 \times 0.09 \times 16)/8 = 7.2$ MB.

The `-s` (or `-stats` or `-statistics`) option can be used with `tc` to obtain statistics on the number of packets dropped by the `tc` layer. With these settings of the TBF and CTCP parameters, no packets were dropped. In *experiment 2*, LDM7/FMTP/UDP was executed with the multicast HTB class rate set to 20 Mbps, and the corresponding BFIFO buffer size set to 7.2 MB as in experiment 1. In this run, `tc` statistics reported that 623019 packets were dropped. In *experiment 3*, LDM7/FMTP/UDP was executed again, but with the BFIFO buffer size corresponding to the multicast HTB class increased to 300 MB, as was done in the experiments described in Chapter 2, Section 2.5.2. In this run, `tc` statistics showed zero dropped packets. Finally, in *experiment 4*, the the BFIFO buffer size was decreased to 200 MB, and LDM7/FMTP/UDP was executed. Packet drops were again observed at the `tc`-layer (`tc` statistics showed 48354 dropped packets). In *summary*, these four experiments confirmed that the `tc`-layer buffer size used with LDM6/CTCP was insufficient for LDM7/FMTP/UDP. It appears that for this particular experimental setting, a 300-MB `tc`-layer buffer is needed for LDM7, which is consistent with a number that was estimated for the NGRID feedtype, using an analytical approach, in our prior work [13].

Next, we conducted a systematical experimental study of LDM7/FMTP/UDP/`tc` behavior by varying the sender multicast rate, r_{mc} , and lowering the `tc`-layer buffer size for each setting of r_{mc} until the dropped-packet rate reported in `tc` statistics reached 0. Fig. 3.5 shows that the required buffer size dropped from 300 MB to 20 MB as the sender multicast rate, r_{mc} , was increased from 20 Mbps to 500 Mbps.

The next section addresses the question of why LDM7/FMTP/UDP needs a larger buffer than LDM6/CTCP. But before answering this question, we point out that the `tc`-layer buffer size needed for LDM6/CTCP increases linearly with the number of receivers m , which is not the case for the `tc`-layer buffer needed for the multicast HTB class for LDM7/FMTP/UDP. The `tc`-layer buffer size needed for the retransmission traffic HTB class for LDM7/FMTP/UDP grows with m but at a much lower rate since packet loss rate is typically small, e.g., if there are 100 receivers and the packet loss rate is 1%, the `tc`-layer buffer size for the retransmission traffic HTB class is only twice BDP.

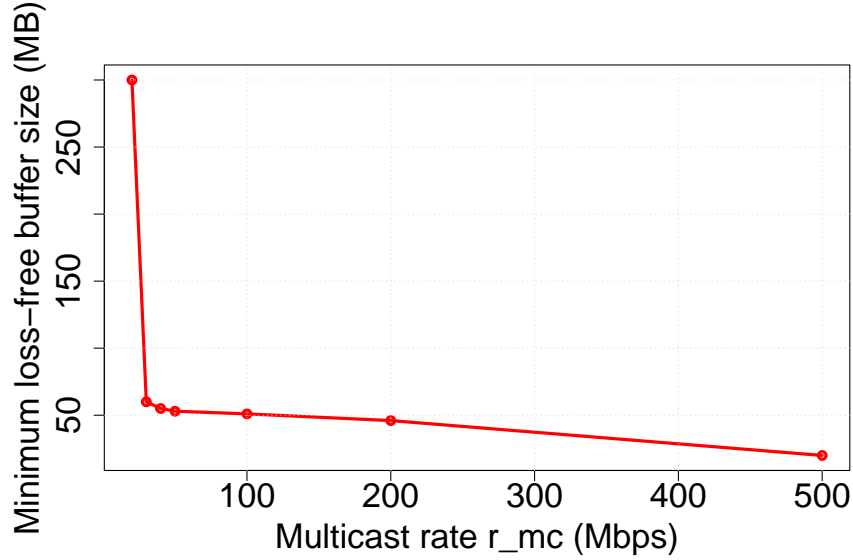


Figure 3.5: Minimum loss-free `tc` buffer size required for various multicast rates

3.3.2 Comparison of TCP, CTCP and UDP behavior in response to a full `tc` buffer using a common application `iperf3`

Appendix A describes the part of the Linux implementation of the network stack, which relates to how TCP, CTCP and UDP behave when the operation to enqueue a packet to the `tc` buffer fails due to a lack of space. The function that calls `tc`-layer `enqueue` function is an IP-layer function, but this IP-layer function merely passes through the return status from the `enqueue` function to the calling transport-layer function. The differences in behavior of the three transport-layer protocols lie in the manner in which the calling transport-layer function reacts to a failed enqueue attempt.

Irrespective of the calling transport-layer function, if there is no space in the `tc`-layer buffer, the function responsible for enqueueing the packet will simply drop the packet, and send back a failed response to the calling function. A TCP sender will react to a failed-enqueue response by entering the `CWR` state, and halving the congestion window (`cwnd`). A UDP sender will not react to the failed-response status because UDP does not offer reliable transport service. With CTCP, the sender also ignores the failed-response status. This is because CTCP assumes that the underlying network offers a rate-guaranteed path, and therefore packet losses occur only due to bit errors, not buffer overflows. CTCP makes no

```
[sc7cq@node-0 ~]$ iperf3 -c 10.10.1.2 -t 30 -b 80M
Connecting to host 10.10.1.2, port 5201
[ 4] local 10.10.1.1 port 40270 connected to 10.10.1.2 port 5201
```

[ID]	Interval		Transfer	Bandwidth	Retr	Cwnd
[4]	0.00-1.00	sec	2.60 MBytes	21.8 Mbits/sec	1	120 KBytes
[4]	1.00-2.00	sec	2.25 MBytes	18.9 Mbits/sec	0	133 KBytes
[4]	2.00-3.00	sec	2.25 MBytes	18.9 Mbits/sec	0	146 KBytes
[4]	3.00-4.00	sec	2.25 MBytes	18.9 Mbits/sec	0	156 KBytes
[4]	4.00-5.00	sec	2.25 MBytes	18.9 Mbits/sec	0	167 KBytes
[4]	5.00-6.00	sec	2.25 MBytes	18.9 Mbits/sec	0	175 KBytes
[4]	6.00-7.00	sec	2.38 MBytes	19.9 Mbits/sec	0	185 KBytes
[4]	7.00-8.00	sec	2.25 MBytes	18.9 Mbits/sec	1	97.6 KBytes
[4]	8.00-9.00	sec	2.25 MBytes	18.9 Mbits/sec	0	113 KBytes
[4]	9.00-10.00	sec	2.25 MBytes	18.9 Mbits/sec	0	127 KBytes
[4]	10.00-11.00	sec	2.25 MBytes	18.9 Mbits/sec	0	140 KBytes
[4]	11.00-12.00	sec	2.25 MBytes	18.9 Mbits/sec	0	150 KBytes
[4]	12.00-13.00	sec	2.38 MBytes	19.9 Mbits/sec	0	161 KBytes
[4]	13.00-14.00	sec	2.25 MBytes	18.9 Mbits/sec	0	171 KBytes
[4]	14.00-15.00	sec	2.25 MBytes	18.9 Mbits/sec	0	180 KBytes
[4]	15.00-16.00	sec	2.25 MBytes	18.9 Mbits/sec	0	189 KBytes
[4]	16.00-17.00	sec	2.25 MBytes	18.9 Mbits/sec	1	106 KBytes
[4]	17.00-18.00	sec	2.25 MBytes	18.9 Mbits/sec	0	120 KBytes
[4]	18.00-19.00	sec	2.38 MBytes	19.9 Mbits/sec	0	133 KBytes
[4]	19.00-20.00	sec	2.25 MBytes	18.9 Mbits/sec	0	144 KBytes
[4]	20.00-21.00	sec	2.25 MBytes	18.9 Mbits/sec	0	156 KBytes
[4]	21.00-22.00	sec	2.25 MBytes	18.9 Mbits/sec	0	165 KBytes
[4]	22.00-23.00	sec	2.25 MBytes	18.9 Mbits/sec	0	175 KBytes
[4]	23.00-24.00	sec	2.25 MBytes	18.9 Mbits/sec	0	184 KBytes
[4]	24.00-25.00	sec	2.25 MBytes	18.9 Mbits/sec	1	122 KBytes
[4]	25.00-26.00	sec	2.38 MBytes	19.9 Mbits/sec	0	112 KBytes
[4]	26.00-27.00	sec	2.25 MBytes	18.9 Mbits/sec	0	126 KBytes
[4]	27.00-28.00	sec	2.25 MBytes	18.9 Mbits/sec	0	139 KBytes
[4]	28.00-29.00	sec	2.25 MBytes	18.9 Mbits/sec	0	150 KBytes
[4]	29.00-30.00	sec	2.25 MBytes	18.9 Mbits/sec	0	160 KBytes

[ID]	Interval		Transfer	Bandwidth	Retr
[4]	0.00-30.00	sec	68.4 MBytes	19.1 Mbits/sec	4

Figure 3.6: Experiment 1 with TCP Reno

attempt to reduce its effective sending rate (e.g., by dropping its `cwnd`) when a packet is dropped by the `tc` layer. Therefore, with CTCP, it is important to ensure that the `tc` buffer size is at least as large as its `cwnd`.

The above description of how TCP, CTCP and UDP work is based on an analysis of the source code. Next, we verify the accuracy of this description through four experiments. In all four experiments, `iperf3` was executed on a two-node GENI slice with different transport protocols and TBF was used as the `tc`-layer queueing discipline. The transport protocols used were as follows: Experiment 1: TCP Reno, Experiments 2 and 3: CTCP, Experiment 4: UDP. In all experiments, the `iperf3` flow was executed for 30 seconds, with a sending rate of 80 Mbps. The `tc` TBF `rate` was set at 20 Mbps and the `burst` parameter was set to 50 KB. The RTT between the `iperf3` sender and receiver was 1 ms.

In Experiment 1, the TBF `limit` parameter was set to 200 KB. Fig. 3.6 shows the

```
[sc7cq@node-0 ~]$ iperf3 -c 10.10.1.2 -t 30 -b 80M
Connecting to host 10.10.1.2, port 5201
[ 4] local 10.10.1.1 port 40229 connected to 10.10.1.2 port 5201
[ ID] Interval            Transfer          Bandwidth        Retr  Cwnd
[ 4]  0.00-1.00    sec   3.38 MBytes    28.3 Mbits/sec     0   707 KBytes
[ 4]  1.00-2.00    sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4]  2.00-3.00    sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4]  3.00-4.00    sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4]  4.00-5.00    sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4]  5.00-6.00    sec   2.38 MBytes    19.9 Mbits/sec     0   707 KBytes
[ 4]  6.00-7.00    sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4]  7.00-8.00    sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4]  8.00-9.00    sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4]  9.00-10.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 10.00-11.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 11.00-12.00   sec   2.38 MBytes    19.9 Mbits/sec     0   707 KBytes
[ 4] 12.00-13.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 13.00-14.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 14.00-15.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 15.00-16.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 16.00-17.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 17.00-18.00   sec   2.38 MBytes    19.9 Mbits/sec     0   707 KBytes
[ 4] 18.00-19.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 19.00-20.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 20.00-21.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 21.00-22.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 22.00-23.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 23.00-24.00   sec   2.38 MBytes    19.9 Mbits/sec     0   707 KBytes
[ 4] 24.00-25.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 25.00-26.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 26.00-27.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 27.00-28.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 28.00-29.00   sec   2.25 MBytes    18.9 Mbits/sec     0   707 KBytes
[ 4] 29.00-30.00   sec   2.38 MBytes    19.9 Mbits/sec     0   707 KBytes
-----
[ ID] Interval            Transfer          Bandwidth        Retr
[ 4]  0.00-30.00    sec  69.2 MBytes    19.4 Mbits/sec     0
```

Figure 3.7: Experiment 2 with CTCP and no retransmission

`iperf3` sender log. The growth in `cwnd` can be observed in this log. Further, the log shows that 4 packets were dropped. Other than the packet drop in the first second, the remaining three packet drops occurred just as `cwnd` value reached the `tc`-layer buffer size (as set by `TBF limit` parameter) of 200 KB. All 4 packets were dropped by the `tc` layer at the sender, as verified by `tc` statistics. After each packet drop, `cwnd` was halved. This experiment validates our two key observations from an analysis of the source code that if the `tc`-layer enqueue function fails due to a lack of buffer space, (i) the packet will be dropped, and (ii) TCP sender will enter the CWR state, and reduce `cwnd` by a factor of 2. Further, the experiment shows that since the TCP sender sets its size of outstanding bytes to a maximum of two parameters: `cwnd` and receiver's advertised flow window, as long as this size is smaller than the `TBF limit` parameter, no packets are dropped by the `tc` layer. But if TCP is configured correctly for high-BDP paths, its `cwnd` could grow to values larger than the

```

-bash-4.1# iperf3 -c 10.10.1.7 -t 30 -b 80M
Connecting to host 10.10.1.7, port 5201
[ 4] local 10.10.1.1 port 60940 connected to 10.10.1.7 port 5201
[ ID] Interval            Transfer          Bandwidth        Retr  Cwnd
[ 4]  0.00-1.00      sec  2.56 MBytes      21.5 Mbits/sec    73    331 KBytes
[ 4]  1.00-2.00      sec  2.00 MBytes      16.8 Mbits/sec   230    288 KBytes
[ 4]  2.00-3.00      sec  2.50 MBytes      21.0 Mbits/sec   204    329 KBytes
[ 4]  3.00-4.00      sec  2.25 MBytes      18.9 Mbits/sec   218    310 KBytes
[ 4]  4.00-5.00      sec  2.25 MBytes      18.9 Mbits/sec   208    327 KBytes
[ 4]  5.00-6.00      sec  2.25 MBytes      18.9 Mbits/sec   238    320 KBytes
[ 4]  6.00-7.00      sec  2.38 MBytes      19.9 Mbits/sec   174    403 KBytes
[ 4]  7.00-8.00      sec  2.25 MBytes      18.9 Mbits/sec   231    310 KBytes
[ 4]  8.00-9.00      sec  2.12 MBytes      17.8 Mbits/sec   187    298 KBytes
[ 4]  9.00-10.00     sec  2.50 MBytes      21.0 Mbits/sec   197    322 KBytes
[ 4] 10.00-11.00     sec  2.12 MBytes      17.8 Mbits/sec   248    373 KBytes
[ 4] 11.00-12.00     sec  2.25 MBytes      18.9 Mbits/sec   224    403 KBytes
[ 4] 12.00-13.00     sec  2.38 MBytes      19.9 Mbits/sec   262    356 KBytes
[ 4] 13.00-14.00     sec  2.00 MBytes      16.8 Mbits/sec   174    287 KBytes
[ 4] 14.00-15.00     sec  2.50 MBytes      21.0 Mbits/sec   200    423 KBytes
[ 4] 15.00-16.00     sec  2.12 MBytes      17.8 Mbits/sec   230    328 KBytes
[ 4] 16.00-17.00     sec  2.50 MBytes      21.0 Mbits/sec   205    304 KBytes
[ 4] 17.00-18.00     sec  2.12 MBytes      17.8 Mbits/sec   265    332 KBytes
[ 4] 18.00-19.00     sec  2.25 MBytes      18.9 Mbits/sec   204    332 KBytes
[ 4] 19.00-20.00     sec  2.38 MBytes      19.9 Mbits/sec   215    310 KBytes
[ 4] 20.00-21.00     sec  2.25 MBytes      18.9 Mbits/sec   229    337 KBytes
[ 4] 21.00-22.00     sec  2.25 MBytes      18.9 Mbits/sec   184    320 KBytes
[ 4] 22.00-23.00     sec  2.25 MBytes      18.9 Mbits/sec   261    308 KBytes
[ 4] 23.00-24.00     sec  2.25 MBytes      18.9 Mbits/sec   228    325 KBytes
[ 4] 24.00-25.00     sec  2.25 MBytes      18.9 Mbits/sec   187    344 KBytes
[ 4] 25.00-26.00     sec  2.12 MBytes      17.8 Mbits/sec   229    288 KBytes
[ 4] 26.00-27.00     sec  2.50 MBytes      21.0 Mbits/sec   211    325 KBytes
[ 4] 27.00-28.00     sec  2.25 MBytes      18.9 Mbits/sec   244    410 KBytes
[ 4] 28.00-29.00     sec  2.12 MBytes      17.8 Mbits/sec   256    310 KBytes
[ 4] 29.00-30.00     sec  2.38 MBytes      19.9 Mbits/sec   208    421 KBytes
-----
[ ID] Interval            Transfer          Bandwidth        Retr
[ 4]  0.00-30.00     sec  68.3 MBytes      19.1 Mbits/sec   6424

```

Figure 3.8: Experiment 3 with CTCP and retransmission

`tc`-layer buffer size causing packet drops at the sender itself.

In *Experiment 2*, the `TBF limit` parameter was set to 600 MB to avoid packet drops. CTCP was initialized with its bandwidth parameter set to 20 Mbps to match the `TBF rate` setting, and its `fcwnd` parameter was set to 400 packets. Fig. 3.7 shows that `cwnd` in CTCP was fixed at 707 KB as expected. Since the `tc`-layer buffer size, at 600 MB, is larger than the total amount of data sent, 69.2 MB, there was no chance for packet drops at the `tc`-layer. The `tc` statistics verified that no packets were dropped by the `tc`-layer.

In *Experiment 3*, `TBF limit` parameter was lowered to 200 KB for the purpose of observing packet drops. The CTCP bandwidth parameter remained unchanged but the `fcwnd` parameter was set to 300 packets. With Ethernet’s Maximum Transmission Unit (MTU) size of 1500 B, the `fcwnd` parameter was effectively 450 KB. As this size is larger than the 200-KB `tc`-layer buffer size (as set by the `TBF limit` parameter), packet drops were

```

Accepted connection from 10.10.1.1, port 32808
[ 5] local 10.10.1.7 port 5201 connected to 10.10.1.1 port 47971
[ ID] Interval      Transfer      Bandwidth      Jitter      Lost/Total Datagrams
[ 5]  0.00-1.00    sec  1.91 MBytes  16.0 Mbits/sec  5.886 ms    745/989 (75%)
[ 5]  1.00-2.00    sec  2.27 MBytes  19.0 Mbits/sec  5.901 ms    931/1221 (76%)
[ 5]  2.00-3.00    sec  2.27 MBytes  19.0 Mbits/sec  5.910 ms    930/1220 (76%)
[ 5]  3.00-4.00    sec  2.27 MBytes  19.0 Mbits/sec  5.916 ms    931/1221 (76%)
[ 5]  4.00-5.00    sec  2.27 MBytes  19.0 Mbits/sec  5.891 ms    931/1221 (76%)
[ 5]  5.00-6.00    sec  2.27 MBytes  19.0 Mbits/sec  5.920 ms    931/1221 (76%)
[ 5]  6.00-7.00    sec  2.27 MBytes  19.0 Mbits/sec  5.879 ms    930/1220 (76%)
[ 5]  7.00-8.00    sec  2.27 MBytes  19.0 Mbits/sec  5.946 ms    931/1221 (76%)
[ 5]  8.00-9.00    sec  2.27 MBytes  19.0 Mbits/sec  5.890 ms    931/1221 (76%)
[ 5]  9.00-10.00   sec  2.27 MBytes  19.0 Mbits/sec  5.957 ms    930/1220 (76%)
[ 5] 10.00-11.00   sec  2.27 MBytes  19.1 Mbits/sec  5.733 ms    931/1222 (76%)
[ 5] 11.00-12.00   sec  2.26 MBytes  18.9 Mbits/sec  5.906 ms    931/1220 (76%)
[ 5] 12.00-13.00   sec  2.27 MBytes  19.0 Mbits/sec  5.904 ms    931/1221 (76%)
[ 5] 13.00-14.00   sec  2.27 MBytes  19.0 Mbits/sec  5.898 ms    930/1220 (76%)
[ 5] 14.00-15.00   sec  2.27 MBytes  19.0 Mbits/sec  5.916 ms    931/1221 (76%)
[ 5] 15.00-16.00   sec  2.27 MBytes  19.1 Mbits/sec  5.732 ms    930/1221 (76%)
[ 5] 16.00-17.00   sec  2.26 MBytes  18.9 Mbits/sec  5.898 ms    931/1220 (76%)
[ 5] 17.00-18.00   sec  2.27 MBytes  19.0 Mbits/sec  5.910 ms    931/1221 (76%)
[ 5] 18.00-19.00   sec  2.27 MBytes  19.0 Mbits/sec  5.892 ms    931/1221 (76%)
[ 5] 19.00-20.00   sec  2.27 MBytes  19.0 Mbits/sec  5.878 ms    930/1220 (76%)
[ 5] 20.00-21.00   sec  2.27 MBytes  19.0 Mbits/sec  5.877 ms    931/1221 (76%)
[ 5] 21.00-22.00   sec  2.27 MBytes  19.1 Mbits/sec  5.752 ms    931/1222 (76%)
[ 5] 22.00-23.00   sec  2.27 MBytes  19.0 Mbits/sec  5.723 ms    931/1221 (76%)
[ 5] 23.00-24.00   sec  2.27 MBytes  19.0 Mbits/sec  5.749 ms    930/1220 (76%)
[ 5] 24.00-25.00   sec  2.27 MBytes  19.0 Mbits/sec  5.728 ms    931/1221 (76%)
[ 5] 25.00-26.00   sec  2.27 MBytes  19.0 Mbits/sec  5.752 ms    930/1220 (76%)
[ 5] 26.00-27.00   sec  2.27 MBytes  19.0 Mbits/sec  5.749 ms    931/1221 (76%)
[ 5] 27.00-28.00   sec  2.27 MBytes  19.0 Mbits/sec  5.736 ms    931/1221 (76%)
[ 5] 28.00-29.00   sec  2.27 MBytes  19.0 Mbits/sec  5.744 ms    931/1221 (76%)
[ 5] 29.00-30.00   sec  2.27 MBytes  19.0 Mbits/sec  5.748 ms    930/1220 (76%)
[ 5] 30.00-30.08   sec   144 KBytes  14.8 Mbits/sec  4.101 ms     0/18 (0%)
-----
[ ID] Interval      Transfer      Bandwidth      Jitter      Lost/Total Datagrams
[ 5]  0.00-30.08   sec  285 MBytes  79.5 Mbits/sec  4.101 ms   27735/36408 (76%)

```

Figure 3.9: Experiment 4 with UDP

expected. Fig. 3.8 shows that retransmissions occurred throughout the run. This finding is consistent with our source-code analysis. Recall CTCP does not make any alterations to its `cwnd` when it receives a failed-response status from the `tc-layer` function (when the latter is unable to enqueue a packet due to lack of buffer space). Therefore, the CTCP function keeps sending packets, through the IP layer, down to the `tc-layer`, which in turn keeps dropping packets, and causing retransmissions. In total, there were 6424 retransmissions as seen in Fig. 3.8. The conclusion is that with CTCP it is critical to make the TBF `limit` parameter as large as the CTCP `fcwnd` parameter.

Finally, UDP behavior was tested in Experiment 4. The TBF `limit` parameter was still set at 200 KB. While the 80-Mbps sending rate specified as a run-time argument in the `iperf3` send command was not relevant in the first three experiments because TCP/CTCP sending rate is largely controlled by `cwnd`, the `iperf3` sending rate is relevant for UDP. Given the mismatch between `iperf3` passing data down to the UDP layer at 80 Mbps, and the `tc-layer` transmitting packets at only 20 Mbps, packet drops are inevitable. For

a UDP flow, packet losses can only be observed in the `iperf3` receiver log, not the sender log, because UDP does not support retransmissions. Therefore, unlike for the first three experiments in which the `iperf3` sender log was shown, for experiment 4, the receiver log is shown in Fig. 3.9. The per-sec number of dropped packets was roughly constant across the whole flow. As `iperf3` was sending data constantly at 80 Mbps to the UDP layer, which was then passing it down through the IP layer to the `tc` layer, the drop rate is almost constant. This experiment verifies our source-code findings that the UDP transmit function does not react to failed-response status from the lower layers, but instead keeps sending packets at the rate dictated by the application even if packets are being dropped by the `tc` layer.

In summary, the source-code analysis and four experiments revealed how `tc` interacts with TCP Reno, CTCP and UDP. These experiments helped us better understand `tc` queueing, TCP CWR state, and the Linux kernel network stack.

3.3.3 Explanation for the `tc`-layer buffer size required for LDM7/FMTP/UDP

This section translates the findings from the `iperf3` experiments described in Section 3.3.2 into answers to our question of why LDM7/FMTP/UDP needs a larger `tc`-layer buffer than LDM6/CTCP. As learned from the `iperf3` experiments, in CTCP, `cwnd` limits the rate at which packets are sent, through the IP-layer, to the `tc`-layer for enqueueing before transmission. If the `tc`-layer buffer size, which is controlled by the `TBF limit` parameter, is set to $2 \times BDP \times m$, and the CTCP `fcwnd` parameter is set to $1.2 \times BDP$, as indicated in Table 2.2 of Chapter 2, Section 2.5.2, where BDP is computed using the largest RTT among all receivers, then the total size of enqueued packets from all m CTCP connections (one connection per receiver) will not exceed the `tc`-layer buffer size. Therefore, no packets will be dropped by the `tc` layer.

To verify this relationship between the CTCP `fcwnd` parameter and the `TBF limit` parameter, we conducted two experiments with LDM6/CTCP. In **Experiment 1**, LDM6/CTCP was run with one sender and one receiver. The `TBF rate` parameter was set to 20 Mbps and the `limit` parameter was set to 450 KB. The CTCP parameter `fcwnd` was set to 120

packets (which is equivalent to 180 KB). The `tc` statistics verified that no packets were dropped by the `tc` layer.

In **Experiment 2**, LDM6/CTCP was run with one sending host and $m = 16$ receivers. In other words, there were 16 CTCP connections, one to each receiver, from the single sending host. The TBF queueing discipline was used at the `tc` layer, with its `rate` parameter set to 320 Mbps and `limit` parameter set to 7.2 MB. Packets from all 16 CTCP connections were sent to the same TBF queue. The CTCP parameter `fcwnd` was set to 400 packets (which is equivalent to 720 KB). Since the CTCP parameters apply to each of the 16 connections, the total size of packets sent down to the `tc` layer is 16×720 KB, which is 11.52 MB. Since this number is larger than the `tc`-layer buffer size (as the `limit` parameter was set to 7.2 MB), packets were dropped by the `tc`-layer. The `tc` statistics showed that 8147175 packets were dropped. These results are consistent with our findings from the `iperf3` experiments that `cwnd` is the key determinant of `tc`-layer buffer size. In summary, for LDM6/CTCP, as long as the CTCP `fcwnd` parameter is set to $1.2 \times BDP$, and the `tc`-layer buffer size is set to $2 \times BDP \times m$ via the TBF `limit` parameter, with the delay component of BDP corresponding to the highest RTT from among all receivers, there will be no packet drops at the `tc` layer.

On the other hand, UDP has no parameter comparable to `cwnd` to limit the number of packets passed down to the `tc` layer. The only way to avoid packet drops at the `tc` layer is to compute the minimum loss-free buffer size required to hold packets from data products passed down by LDM7/FMTP to the UDP layer for each setting of the sender multicast rate (multicast HTB class rate r_{mc}). As an example, for a 1-hour segment of the NGRID feedtype, the required minimum loss-free `tc`-layer buffer size, b_{mc} , is plotted as a function of r_{mc} in Fig. 3.5. This required buffer size will depend upon the feedtype characteristics, as demonstrated in prior work [13]. Since the product inter-arrival times and product sizes do change from day-to-day, an exponentially weighted moving average (EWMA) algorithm was proposed to modify r_{mc} and b_{mc} on a day-by-day basis [13]. Therefore while packet drops at the `tc` layer are not completely avoidable, the drop rate can be kept low by using this algorithm to size the `tc`-layer buffer.

3.4 Impact of sender multicast rate on LDM7 throughput

In section 2.5.5, Chapter 2, we compared the throughput measurement of LDM6 and LDM7. Fig. 2.4 shows that the throughput plots of LDM7 flattened out as the base rate r increased. Section 2.5.5 also recommended the use of a high base rate r in order to leverage the advantages of LDM7 and achieve low latency. An extra run with $r = 800$ Mbps executed on the 5-rack slice yielded an average throughput of only 75 Mbps, even without artificially injected packet losses. Our expectation was that multicast packets would be sent unhindered at the VLAN rate of 800 Mbps, and hence we expected a throughput close to this rate. Therefore, the purpose of the work described in this section is to explain this interesting finding.

We found that there are three factors that impact throughput: RTT, `tc` buffering delays, and processing delays. When the VLAN rate r_{mc} is high, e.g., 8 Gbps, transmission delays drop, as do `tc` buffering delays. On low-RTT paths, processing delays become the dominant factor. The impact of RTT is explored in Sections 3.4.1, the impact of `tc` buffering delays is presented in Section 3.4.2, and the impact of processing delays is presented in Section 3.4.3.

3.4.1 Impact of RTT on LDM7 throughput

The experiments described in Chapter 2 were executed on a 5-rack GENI slice, in which the RTTs from hosts on the sender rack to the four receiver racks varied (the RTT values were 36, 41, 50, and 90ms, to the four receiver racks). Hence, we hypothesized that the throughput experienced by files delivered to the highest-RTT receivers could be pulling down the average throughput values.

For better control of RTT, we ran a new experiment with only one sender and one receiver, both located in the same rack. Different RTT values were emulated on the path by artificially adding a delay to each packet with the `tc netem` module.

Two sets of experiments were executed with the base rate r set to 20 Mbps and 500 Mbps. In each set of experiments, five representative RTT values were used, 1 ms, 10 ms, 20 ms, 50 ms, and 100 ms. The sender `tc` multicast buffer (b_{mc}) size was set to 600 MB. The new maximum retransmission period design was applied with c set to 2 minutes.

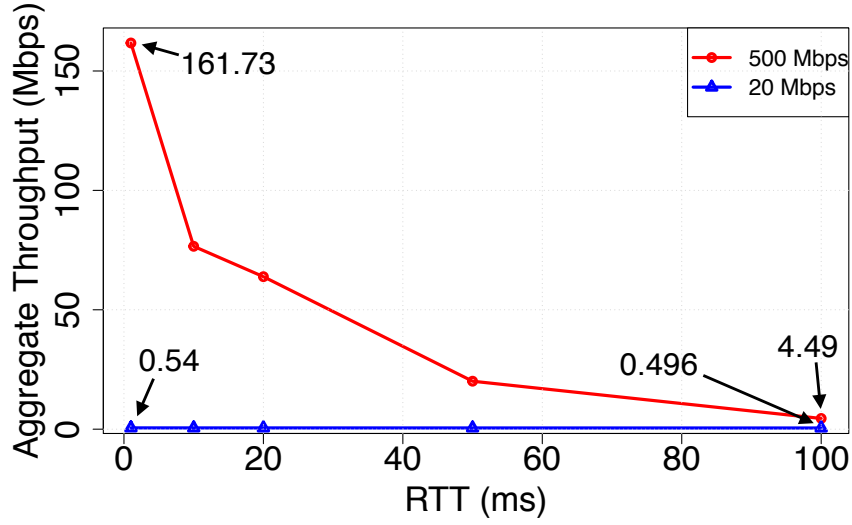


Figure 3.10: Throughput for $r = 20$ and $r = 500$ Mbps

From the log files generated in the experiments, the average throughput across all aggregates (groups of files) were computed and plotted against RTT as shown in Fig. 3.10. When $RTT = 1$ ms, the $r = 20$ Mbps plot shows a throughput value of 0.54 Mbps. When RTT was increased to 100 ms, this throughput dropped to 0.496 Mbps. The $r = 500$ Mbps plot shows a more significant drop from 161.73 Mbps to 4.49 Mbps. But even operating when RTT is just 1 ms, which means RTT should not have a major impact, with a base rate r of 500 Mbps, LDM7 could only achieve a throughput of 161.73 Mbps.

Two conclusions can be drawn from these results. First, when the base rate r (which is the rate of the multipoint VLAN) is low, there is some other dominant factor determining throughput, not RTT. But when the base rate is higher, RTT plays a significant role in lowering throughput. Second, RTT alone does not explain why throughput is not equal to, or even close to, the VLAN rate. There is some other factor.

3.4.2 Impact of the sender t_c buffering delay on throughput

We developed a script to parse the LDM7 receiver log file and extracted the elapsed latency for each file, which is the time taken to deliver a file completely from the sender to the receiver. For each data product (file), the LDM7 receiver log file shows the time at which the

Table 3.1: Evidence of sender `tc` buffer buildup; $r = 20$ Mbps, $p = 0\%$, $RTT = 0.2$ ms

File (No.)	creation-time (HHMMSS.ms)	reception-completion-time (HHMMSS.ms)	size (B)	latency (ms)	throughput (Mbps)
0	230119.375	230119.408	90654	32.98	21.97
1	230119.577	230119.610	91495	33.53	21.83
2	230119.591	230119.631	50421	39.62	10.18

product was completely received by FMTP and logged by one of the LDM7 processes at the receiver (which we call *reception-completion-time*), as well as the time at which the product was created, which means the time instant when it was inserted into the product queue at the upstream LDM7 server (which we call *creation-time*). The product creation-time is sent as part of the metadata associated with each data product [21]. Thus, this sender-side timestamp is obtained for each product by the receiver, and saved in the log file. The difference between these two time values in each log entry is the latency. The Network Time Protocol (NTP) was used to synchronize clocks of all hosts running LDM7.

To check whether there is `tc` buffering delay, we analyzed the LDM7 receiver logs. Table 3.1 shows the creation-time, reception-completion-time and size for three selected products, which were obtained from the receiver log. The latency value was computed by taking the difference between the reception-completion-time and creation-time. The throughput value was obtained by dividing the product size by latency.

The throughput values computed for files 0 and 1 are 21.97 and 21.83 Mbps, respectively, which are slightly higher than the VLAN rate of 20 Mbps. Since the `tc`-rate shaping at the sender is accurate, the files would have been multicast only at 20 Mbps, and therefore, theoretically, the throughput cannot be higher than 20 Mbps. Our best explanation for these higher throughput values is a lack of perfect clock synchronization. But more relevant to the focus of this sub-section, we can assert that files 0 and 1 did not experience `tc` buffering delays.

Next, consider whether file-2 packets experienced `tc` buffering delays. To determine when the time instant at which the sender can transmit file 2, consider when file 1 transmission ended. The instant when file transmission ended is $230119.577 + 91495 \times 8/20000000 = 230119.613$. File 2 creation-time, as reported in Table 3.1, is 230119.591. Therefore, file

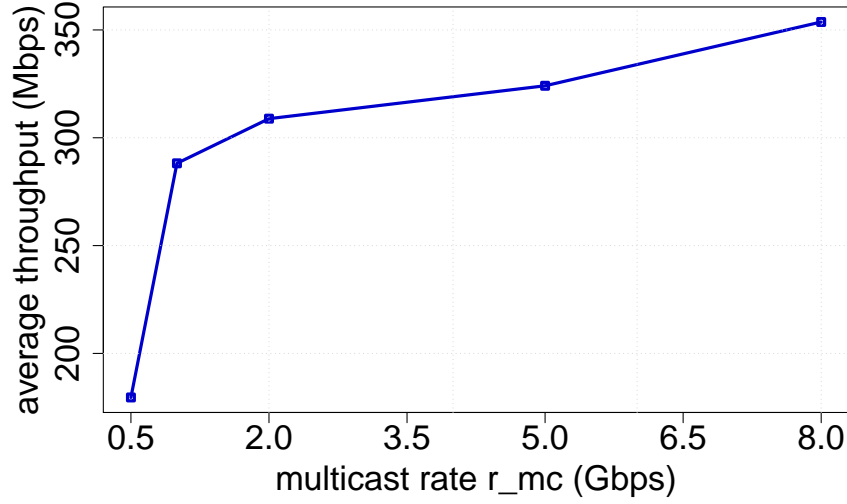


Figure 3.11: Throughput under different r_{mc} ; $p = 0\%$, RTT=0.2 ms

1 would have been still under transmission, when file 2 was inserted into the product queue. File 2 transmission could only have begun at 230119.613, which is the time instant when file 1 transmission ended. File 2 transmission would have ended at $230119.613 + 50421 \times 8 / 20000000 = 230119.633$. The difference between file-2 reception-completion-time 230119.631, and 230119.633 is only 2 ms, which could be attributed to processing delays and/or lack of perfect clock synchronization. This analysis shows that file-2 latency shown in Table 3.1 consists primarily of transmission delay (20.2 ms) and τ_c buffering delay (19.45 ms). The buffering delay was computed by subtracting the transmission delay, 20.2 ms, from the latency, 39.62 ms. In summary, file-2 packets did experience τ_c buffering delays causing the throughput to drop from the ideal value of 20 Mbps (VLAN rate) to 10.18 Mbps.

3.4.3 Impact of other delay components on throughput

On a 0.2-ms RTT path, we wanted to test the hypothesis that increasing the VLAN rate could reduce the τ_c buffering delay to close to 0, and consequently, allow for throughput values to be close to the VLAN rate. For this experiment, a slice was created between two hosts in one ExoGENI rack with 10 Gbps NICs. The RTT was 0.2 ms. Fig. 3.12 shows throughput vs. VLAN rate varied from 500 Mbps to 8 Gbps. To our surprise, the average

Table 3.2: Example latencies for five products; $r = 8$ Gbps, $p = 0\%$, $RTT = 0.2$ ms

File (No.)	creation-time (HHMMSS. μ s)	reception-completion-time (HHMMSS. μ s)	size (KB)	latency (ms)	throughput (Mbps)
0	181005.809918	181005.812516	90.654	2.598	279.15
1	181006.011918	181006.014427	91.495	2.509	291.73
2	181006.025918	181006.027633	50.421	1.715	235.20

throughput was still low, reaching only 350 Mbps when the VLAN rate was 8 Gbps.

To understand this result, we consider the three products shown in Table 3.2. The transmission delay of file 0 is 90.7μ s. The one-way propagation delay is 100μ s (since RTT is 0.2 ms). Therefore, we expected the latency to be just 190.7μ s, but the actual latency was 2.598 ms. As the packets of this file could not have experienced \mathbf{tc} -buffering delays, this extra delay of 2.407 ms could be due to processing delays in the networking protocol layers, FTMP, UDP, IP, and \mathbf{tc} , or due to lack of perfect clock synchronization.

A similar analysis of file 1 latency is as follows. The transmission delay of file 1 is 91.5μ s, and hence the ideal latency should be 191.5μ s, but the actual latency is 2.509 ms, which explains the low throughput of 291.73 Mbps even though the VLAN rate was set to 8 Gbps. File 1 ended its transmission at $181006.011918 + 91.495 \times 8/8000000 = 181006.012009$, which was earlier than the time when File 2 was created. This computation shows that file-2 packets would not have experienced any \mathbf{tc} buffering delay (file-1 packets did not suffer from \mathbf{tc} buffering delays either because file-0 was fully transmitted before file-1 was inserted into the product queue).

Finally, we computed ideal latency, which is defined to be transmission delay plus one-way propagation delay, which is $S_n/r_{mc} + RTT/2$ for file n . The real latency is computed as described above. Fig. 3.12 shows these delays for the first 200 files of the 1-hour NGRID trace under two settings of the VLAN rate 500 Mbps and 8 Gbps. The difference between the real latency and the ideal latency was higher with the 500 -Mbps VLAN-rate setting than with the 8 -Gbps setting. This could be due to processing delays, or due to \mathbf{tc} -buffering delays in the 500 -Mbps setting. Further in-depth studies, which measure processing times of networking software, are required to determine whether processing delays could become the dominant factor in high-speed networks.

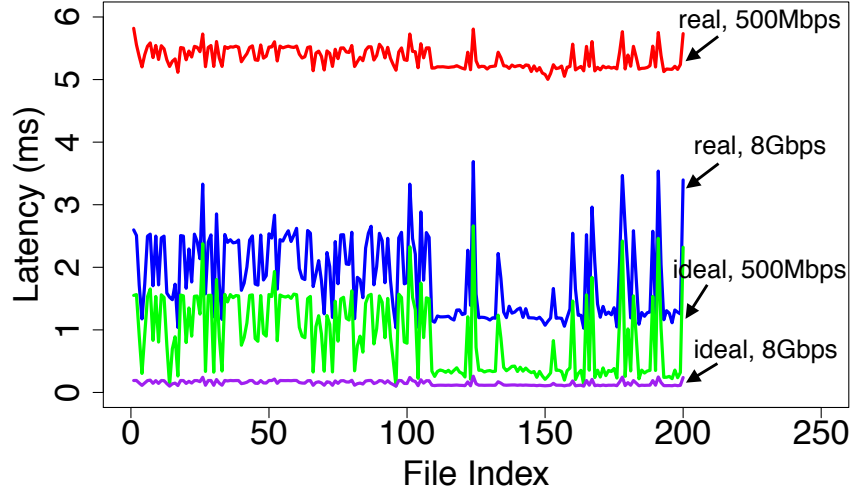


Figure 3.12: Ideal latency vs. actual latency for individual files; r_{mc} is 500 Mbps or 8 Gbps; $p = 0\%$, $RTT=0.2$ ms

Fig. 3.13 illustrates the pipelining effect between the various components of latency. On a low-rate VLAN, we expect transmission delay to dominate on paths with small propagation delays, e.g., in datacenter networks. The packet processing delays are effectively hidden behind the transmission delays. This model explains the latency results described in Section 3.4.2. For example, the latencies of file 0 and file 1 in the 20-Mbps VLAN-rate setting, shown in Table 3.1, were completely determined by transmission delays. On the other hand, the file-0 latency in the 8-Gbps VLAN-rate setting, shown in Table 3.2 was determined by processing delays, and/or clock synchronization delays (NTP clock synchronization in LANs is known to have accuracy in the order of 1 ms [22]).

3.5 New metrics

This section describes two new metrics for the characterization of an MPUP system. Section 3.5.1 defines a metric called *file latency* and presents results comparing LDM6 and LDM7 on this metric. Section 3.5.2 defines a metric called *block retransmission ratio* and presents experimental results for this metric.

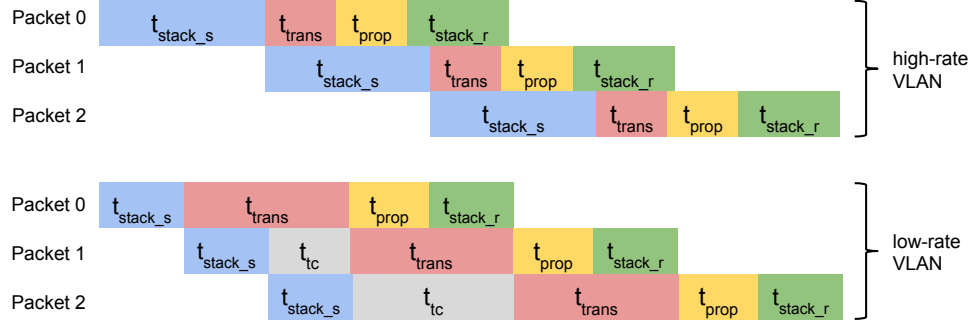


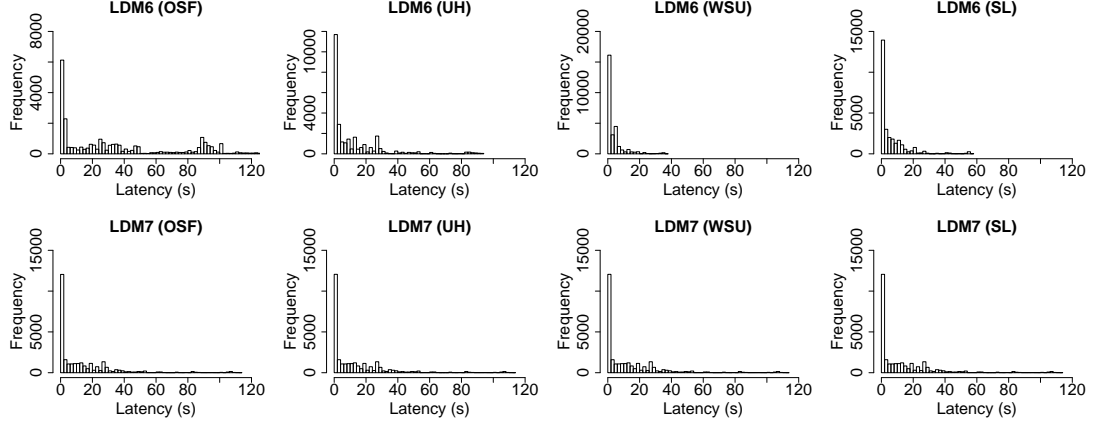
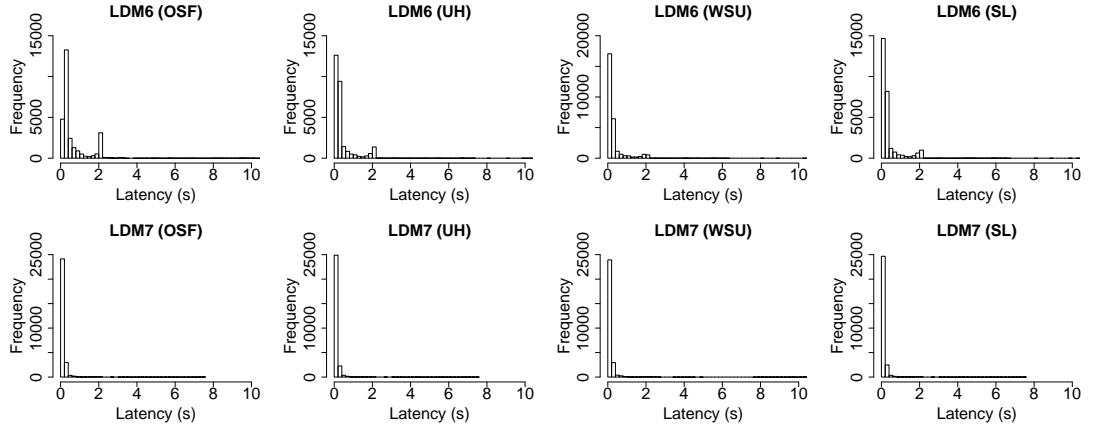
Figure 3.13: The pipelining effect of packetization; small propagation delay assumed (e.g., datacenter networks)

3.5.1 File latency

Most of the IDD feedtypes have a requirement for near-real-time product delivery. Since the IDD feedtypes mostly carry weather data, the value of the data products decreases with time. The throughput metric defined in Chapter 2 was per file-set rather than per file. Our reasons for this using this average file-set throughput metric were provided in Chapter 2. In this section, we compare the latency distribution obtained when using LDM6 vs. LDM7.

The LDM7 receiver log files generated in the experiments described in Section 2.5.5 were reused for this study. For each data product, the receiver log file shows the time at which the product was completely received by FMTP and logged by one of the LDM7 processes at the receiver, as well as the time at which the product was created, which means the time instant when it was inserted into the product queue at the upstream LDM7 server. The difference between these two time values in each log entry is the file latency. The Network Time Protocol (NTP) was used to synchronize clocks of all hosts running LDM. The latency distributions are almost identical for the four nodes in each of the receiver sites of the 5-rack GENI slice used in the experiments. Therefore, we arbitrarily pick one node in each receiver site, and parse the corresponding receiver log to obtain latency.

Fig. 3.14 shows the latency distributions for both LDM6 and LDM7. The upper row shows the latency histograms of four different sites when running LDM6. The lower row shows the latency histograms of the same four nodes when running LDM7. The latency distribution with LDM7 is similar for receivers at different sites.

Figure 3.14: Latency distribution: $m = 16$, $r = 20$ MbpsFigure 3.15: Latency distribution: $m = 16$, $r = 60$ Mbps

Except for the OSF receivers (the OSF rack receivers had the highest RTT of 90ms from the sender), LDM6 outperformed LDM7 for receivers in all other sites when r was set to 20 Mbps. The (average latency, maximum latency) for LDM6 in the WSU, SL, and UH receivers were (3.6 sec, 37.3 sec), (6.1 sec, 56.3 sec), and (11.6 sec, 93.4 sec), respectively. The different latency values can be explained by the sequential serving of TCP segments to each receiver by the sending host. Since the RTT values are 36, 41, and 50 ms, for WSU, SL and UH receivers, respectively, the latency values are correspondingly smallest for the WSU receivers and largest for the UH receivers in a comparison across these three racks. For LDM7, the average latency was approximately the same at 12.7 sec, and the maximum latency was approximately 112.9 sec for receivers at all four sites, since all receivers are served at the same time. The differences in RTT between receivers at the four sites does not

play an important role with LDM7 because the RTT is incurred only once for each product, and RTT is in ms, while the buffering delays at the sending host increase the average and maximum latency values to sec. LDM6 achieved an average latency and maximum latency of 55.2 sec and 315.4 sec in OSF receivers, which are higher than the LDM7 values.

Our reasons for setting the base rate r to 20 Mbps, as stated in Chapter 2, was that at this setting LDM6 and LDM7 achieved the same average throughput, which then allowed for a comparison of resource requirements. But, since LDM7 requires only a single multipoint VLAN for the base multicast, the rate of this VLAN can be higher. Using the 60 Mbps setting for r , we computed the latency distribution, which is shown in Fig. 3.15. With this setting, the average latency in OSF (the worst case) was 0.7 sec for LDM6 and 0.16 sec for LDM7. The maximum latency in OSF was 19.6 sec for LDM6 and 7.6 sec for LDM7. The average latency and maximum latency of LDM7 were both better than for LDM6 for receivers at other sites too.

3.5.2 Block retransmission ratio

The FMTP File Delivery Ratio (FFDR) is a useful metric, and helped us determine the impact of the sender-side maximum retransmission period. Correspondingly, to determine the impact of the receiver timeout factor f_{rcv} , we found FFDR to be insufficient, and found that we needed a metric. If f_{rcv} is too small, the receiver timeout value $\tau_{rcv}(n)$ for file n , as defined in Section 2.2, will be correspondingly small, which could lead to premature retransmission requests. On the other hand, if it is too large, then the FMTP receiver could wait too long and miss the time window during which the sender accepts retransmission requests and serves them. A metric suitable for determining a suitable value for f_{rcv} was found to be block retransmission ratio (BRR) is defined as follows:

$$\Phi_{ij} = \frac{Rx_{ij}}{Tx_{ij}} \quad (3.8)$$

where i is the index for each receiver, and j is the index for each aggregate group of files, Rx_{ij} is the number of retransmitted blocks required by receiver i for the files in group j , and Tx_{ij} is the total number of blocks including the multicast blocks and the blocks

Table 3.3: Experiment A to F for BRR study

Experiment	f_{rcv}	p (%)
A	5	0%
B	10	0%
C	10	1%
D	10	5%
E	20	5%
F	40	5%

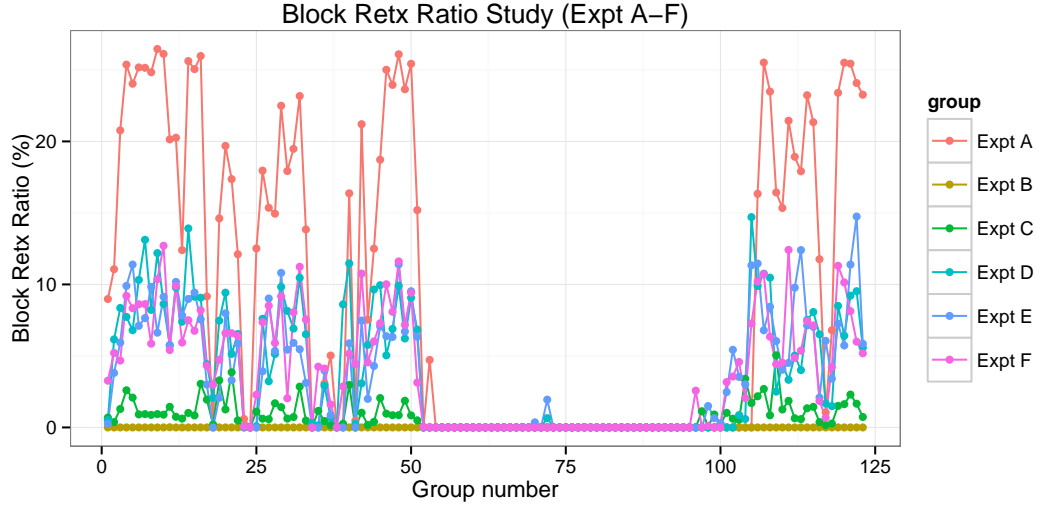


Figure 3.16: Block retransmission ratio per aggregate group; different colors used

retransmitted to receiver i by the sender for the files in group i . Since some files are fairly small, it was deemed better to see aggregate groups of files rather than single files in the definition of this metric.

Six experiments were conducted between two GENI hosts on the same rack (one sender and one receiver) with the base rate r set to 40 Mbps, and sending host buffer size b_{mc} set to 600 MB. The RTT between sender and receiver was 1 ms. Artificial packet losses were injected at random at the receiver with probability p . Table 3.3 shows the f_{rcv} setting and the p values used in the six experiments.

Fig. 3.16 shows six plots corresponding to experiments A through F. The y-axis corresponds to the block retransmission ratio in percentage, and x-axis is the aggregate group index j . There is a significant difference in BRR values between the plots corresponding to experiments A and B. It appears that when $f_{rcv} = 5$ (Experiment A setting), the receiver

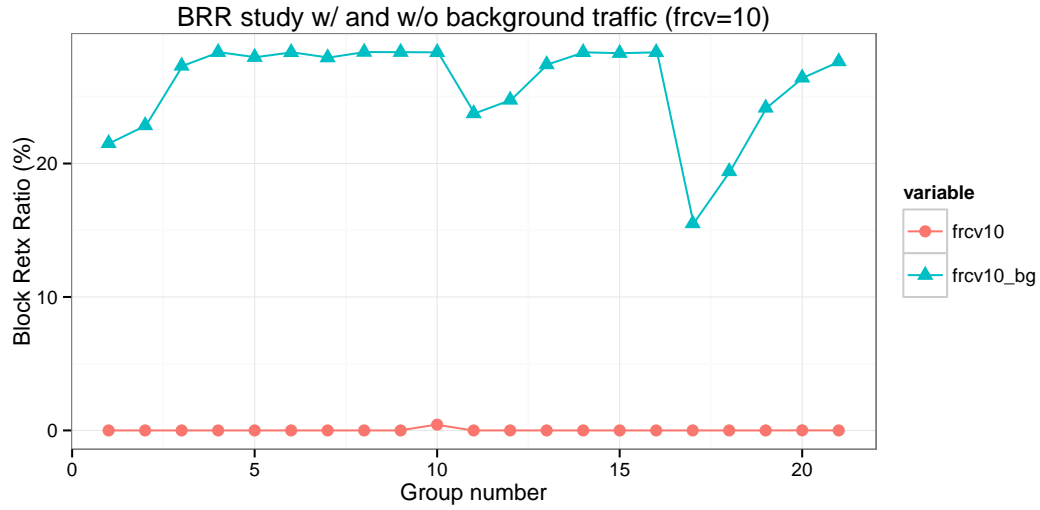


Figure 3.17: A study to determine the impact of background traffic on the choice of f_{rcv}

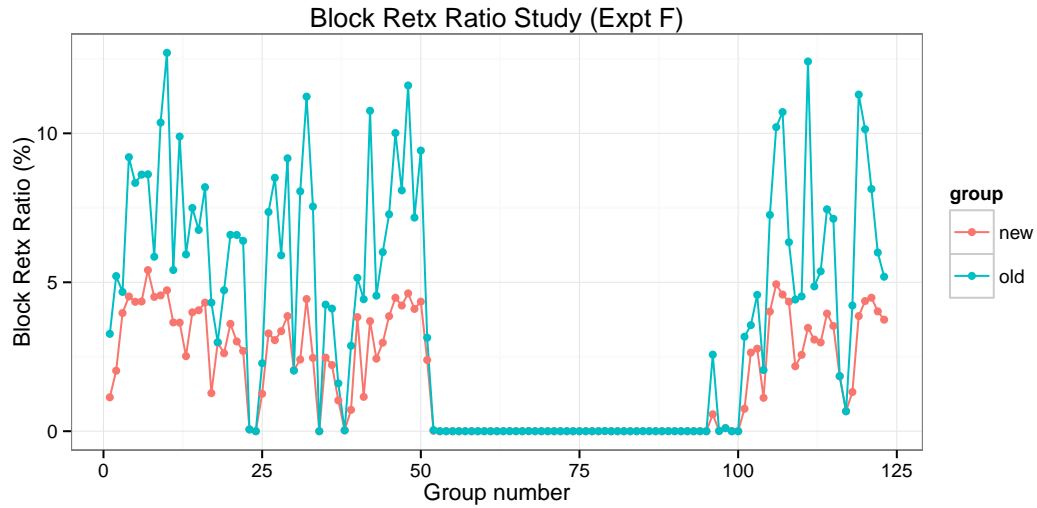


Figure 3.18: Block retransmission ratio with the patched log parser; different colors used

timer times out prematurely and sends requests for block retransmissions when none were needed. When f_{rcv} was increased to 10 (Experiment B setting), the BRR dropped to 0%, which suggests that for this environment, where the path consists of a single switch and there is no interfering traffic, an f_{rcv} value of 10 is sufficient.

To explore this point further, we conducted another experiment, in which a third host was added to the experiment to generate an interfering flow during an ongoing LDM7 file-stream distribution. An `iperf3` server was executed to create the interfering flow by

sending data from the third host to an `iperf3` client on the host running the LDM7 receiver. In other words, the link from the top-of-the-rack switch to the host that was executing both the LDM7 receiver and the `iperf3` client carried both LDM7 and `iperf3` traffic. If `iperf3` packets get interspersed between the LDM7 multicast packets, then an f_{rcv} of 10 could be insufficient to ensure a low BRR. Since the receive-side timer is a product of f_{rcv} and the file transmission delay, if some of the LDM7 packets have to wait in the switch buffer behind `iperf3` packets, then the factor f_{rcv} needs to be larger than if there was no interfering traffic. Fig. 3.17 shows the results. The `frcv10` plot is a flat line at 0%, which corresponds to Experiment B in Fig. 3.16, (without background traffic). Plot `frcv10_bg` shows that BRR increased to over 20% in the presence of the background traffic (`iperf3` flow). This example illustrates the need for online monitoring of the performance of an LDM7 deployment, and dynamic adjustment of the f_{rcv} factor, just as we recommended the use of an EWMA scheme for setting the fixed sender-side maximum retransmission period in Section 3.2. The BRR metric is well suited to determine an appropriate value for the f_{rcv} factor based on the path and traffic conditions of the particular LDM7 deployment. An independent measure of packet loss rate on the path is required because only a BRR value higher than the packet loss rate is indicative of the need to increase f_{rcv} . In the experiment described with background traffic, it is possible that some of the BRR increase to 20% was attributable to actual packet losses, but some portion of it is likely due to dispersion of the LDM7 multicast packets because of intervening `iperf3` packets. When FMTP is run in verbose mode, it records requests for retransmissions of dropped End-of-Products (EOPs), which are indicative of receive timeout. The number of such requests could be saved in an FMTP log file, which would then allow for a separation of the effects of packet loss rate vs. premature receive timeouts on BRR.

Returning back to our analysis of the plots shown in Fig. 3.16, next consider the differences between the BRR values for Experiments B, C, and D. These experiments were run with no background traffic, which means the f_{rcv} setting of 10 was sufficient to ensure zero BRR when there was no artificially injected packet losses (experiment B). To study whether BRR was reflective of packet loss rate, we injected packet losses by setting p to 1% and 5% in experiments C and D, respectively, as shown in Table 3.3. To our surprise, BRR was higher

than p , sometimes reaching over 10% in experiment D, when p was only 5%.

An explanation for this observation is as follows. Analysis of the LDM7 receiver logs showed that the higher BRR values were caused by missing BOPs. If a BOP is dropped, the whole file needs to be retransmitted. Thus, even when the injected loss rate was only 5%, some of the packets dropped were BOPs, which led to retransmissions of all blocks of the files with dropped BOPs. We changed the method used for computing BRR by only considering files for which the BOP was not dropped. The log parser that parsed the receiver log files was thus patched with this fix. Fig. 3.18 shows the BRR for experiment F using the “old” and “new” log parsers. The BRR in the new plot does not exceed the artificial packet loss injection rate p , which was 5%. This fix, was simply to the log parser, and hence does not affect how the MPUP system works. It just improves the definition of BRR to more accurately reflect path loss rates.

Finally, we discuss the plots corresponding to experiments E and F in Fig. 3.16. These experiments were executed to study the relationship between injected packet loss rate p and f_{rcv} , if any. As Table 3.3 shows, f_{rcv} was set to 20 and 40 in experiments E and F, respectively. Thus, results from experiments D, E and F, can be compared as in all three experiments, the injected packet loss rate p was held constant at 5%. The BRR is roughly the same for all three experiments, and therefore a consideration of path packet loss rate is not required while selecting the factor f_{rcv} .

In summary, the BRR proved to be a useful metric. Three observations were made using this metric. First, the receive-side timeout factor f_{rcv} should be dynamically changed based on observed values of the BRR metric because a higher value of f_{rcv} is needed if there is interfering traffic. In other words, if an increase is observed in BRR, f_{rcv} should be increased as it could indicate premature retransmission requests, though it could also indicate packet losses. Second, we observed that the BRR metric definition should not include block retransmissions triggered by a dropped BOP since the inclusion of these transmissions could be misleading as the BRR would be inflated. Third, we found that packet loss rate does not affect the choice of f_{rcv} .

3.6 Conclusions

Three aspects were studied in the work described in this chapter. First, we redefined the FMTP-sender maximum retransmission period, modified the FMTP code, and ran experiments to test the performance of LDM7 with this modified FMTP code. We found that a dynamically updated, but file-independent, timer value was sufficient to achieve 100% FMTP File Delivery Ration (FFDR). This solution is simple, and yet effective. Second, the sender `tc`-layer buffer size value chosen in the experiments described in Chapter 2 was revisited to understand why the LDM7 solution required a much larger buffer than the LDM6 solution. The difference arose because LDM6 uses CTCP, which includes the standard TCP code that limit sthe number of outstanding segments to be the smaller of the congestion window and receive window, and hence restricts the number of segments passed down to the lower layers, while LDM7 uses UDP, which has no such limiting variable. Third, the impact of several factors on LDM7 latency were studied. We found that when propagation delay is negligible and VLAN rate is relatively low, transmission delay and sender `tc` buffering delay are the two main components of latency, while processing delay increases in importance when the VLAN rate is high. Finally, two metrics, file latency and block retransmission ratio, were defined and used to characterize the MPUP system. The block retransmission ratio proved to be a good metric for sizing the receiver FMTP timeout factor.

Chapter 4

File Multicast Transport Protocol (FMTP)

4.1 Introduction

The File Multicast Transport Protocol (FMTP) is a reliable multicast transport protocol designed to support the distribution of file-streams to multiple receivers over rate-guaranteed multipoint virtual networks. FMTP requires the lower layers to support packet multicasting (though this multicast service could be unreliable), and to support a reliable unicast service. FMTP segments files into blocks, and passes the blocks to the lower layers of the network protocol stack for multicasting to all receivers. Since FMTP does not require the multicast service offered by the lower layers to be reliable, FMTP assumes that not all blocks will be multicast successfully to all receivers, and hence accepts requests for block retransmissions from individual receivers, and serves these retransmissions using the reliable unicast service of the lower layers.

FMTP does not have built-in congestion control or flow control. Since FMTP is designed for use over rate-guaranteed virtual networks, FMTP does not adjust its sending rate dynamically, i.e., it does not have congestion-control mechanisms such as those found in TCP. To use FMTP, external controllers are required to set up the rate-guaranteed multipoint virtual network, and to configure the Linux traffic-control `tc` utility at the sender to send

packets at the rate of the virtual network. These setup operations will prevent switch buffer overflows on paths to all receivers and hence FMTP does not require data-plane congestion control.

The rate selected for the multipoint virtual network should be provided to all receivers, which are then expected to schedule their FMTP applications in a manner that avoids receive-buffer overflows. With this assumption, FMTP does not implement any flow-control mechanisms such as TCP's window control. With multiple receivers, feedback on window size is difficult to manage at the sender. Furthermore, slowing down the multicast because of one or two slow receivers is unfair to the multicast group.

Without flow-control, packets can be dropped due to temporary CPU workload increases at receiving hosts. Therefore, FMTP's error control mechanism in which the sender receives NACKs for individual blocks of files from individual receivers, and serves them using the reliable unicast service of the network, is required.

Finally, FMTP allows receivers to dynamically join and leave a multicast group during the transmission of file-streams. The FMTP tracks the number of receivers, and awaits acknowledgment of file reception from all receivers, though the wait time is limited by a timer. Without such a timer, the throughput of file delivery could suffer because of one or more slow receivers.

Section 4.2 introduces the basic aspects of FMTP. Section 4.3 describes the packet structure and message formats in FMTP. Section 4.4 explains the protocol operation from the perspectives of both sender and receiver. Section 4.5 explains the implementation details of FMTP. Section 4.6 reviews related work. Section 4.7 concludes the chapter.

4.2 Protocol overview

When an FMTP application is started at a sending host, the application initiates an FMTP sender. Similarly, when its counterpart FMTP application is started at one or more receiving hosts, FMTP receivers are initiated at these hosts. As part of an initialization phase, each FMTP sender and all FMTP receivers configure the lower layers of their network stacks to send/receive multicast packets. Also, the lower layers of the network stack should be

configured by each receiver to send retransmission requests for lost/dropped blocks of files over a reliable unicast service to the FMTP sender, and to receive the retransmitted blocks. The sender should also configure its lower layers for reliable unicast service to each receiver.

The file-stream distribution procedure starts by the application providing an FMTP sender a pointer to a file (also called product) in the application memory space. Associated with each product is metadata about the product, which is also created by the application. The FMTP sender places the product size (which is part of the metadata), and the rest of the metadata, in a special FMTP packet named Begin-of-Product (BOP) message, and sends the BOP to all receivers via the multicast service. Next, the FMTP sender divides the file into small blocks, each of which fits into a single packet based on the maximum packet-length constraints of the lower layers. The FMTP sender adds a header to each FMTP packet (each carrying a data block), and sends these to the lower layers for multicasting to all receivers. When all the data packets have been passed down to the lower layers, FMTP creates an End-of-Product (EOP) message, and passes it to the lower layers for multicasting.

Upon receiving a BOP, each FMTP receiver passes the product size and metadata carried in the BOP to its application. If the application decides that it wants to receive this product, it responds to the FMTP receiver by sending it a user-space memory location for the product. All the following data packets are directly moved into the user-space memory location to avoid an extra copy. The FMTP receiver checks each FMTP packet header and detects missing blocks. If a missing data block is detected, the FMTP receiver sends a retransmission request back to the FMTP sender via the reliable unicast service. The FMTP sender retransmits the requested data block via the reliable unicast service to just the requesting receiver.

A sender-side maximum retransmission period is set for each file after the BOP is sent. When this period ends (the associated timer expires), the FMTP sender stops serving all pending retransmission requests and sends back rejections. On the FMTP receiver side, a receive timer is set for each file when its BOP is received. If the timer expires before the reception of the corresponding EOP, the FMTP receiver immediately requests retransmissions for all missing blocks and the EOP for that file.

An FMTP receiving application can join a multicast group at any time. As part of its

initialization procedure, the FMTP receiver notifies the FMTP sender while establishing the reliable unicast service. The FMTP sender updates its list of connected receivers, and commits to serve the new receiver if retransmissions are requested. The FMTP receiver starts listening for multicast blocks, and discarding blocks until it receives a BOP. The FMTP receiver sends the product size and metadata in this received BOP to its application. If the application approves reception of the product and provides the FMTP receiver a user-space memory location for the product, the FMTP receiver starts saving file blocks, and continues listening for more files in the file-stream. Similarly, an FMTP receiving application can leave a multicast group any time. The FMTP sender receives notification (through termination of the reliable unicast service), and can then update its list of receivers, and notify the FMTP sending application.

4.3 Packet structure and message formats

4.3.1 FMTP packet structure

There are two types of FMTP packets: data packets and control packets. Data packets are used to carry payload for multicast and retransmission. Control packets are used to carry metadata for a new file, block retransmission requests, or notification of successful reception for a file. Both data packets and control packets have the same packet header structure.

An FMTP packet consists of an FMTP header and payload. For some control packets, the payload part is empty. Fig. 4.1 shows the packet structure.

The **prodindex** field is a 32-bit unsigned value indicating the file index in the file-stream. The first file in a file-stream starts with *prodindex* = 0. For each new file, the **prodindex** field of the FMTP packets associated with the file is incremented by 1. After $2^{32} - 1 = 4294967295$ files, the **prodindex** loops back to 0.

The **seqnum** field is a 32-bit unsigned value indicating the sequence number of a packet within a file. As in TCP, the **seqnum** indicates the total number of bytes sent for that product in previous packets. If an FMTP packet is dropped/errored, the FMTP receiver can detect the loss by inspecting the sequence number in the packet header. The **seqnum** increments by the number of bytes contained in the previous packet. For example, when a new file is

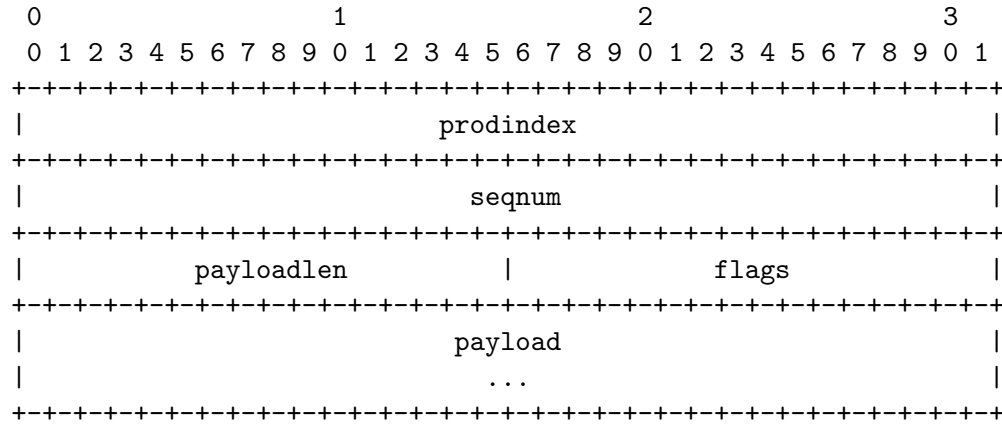


Figure 4.1: FMTP packet structure

sent, the first data packet of the file contains $seqnum = 0$ in its FMTP header. The next packet contains $seqnum = seqnum' + payloadlen'$ in its FMTP header, where $seqnum'$ is the sequence number in the previous packet header, and $payloadlen'$ is the payload length of the previous packet. An assumption made about **seqnum** is that a file seldom exceeds 4 GB. For files larger than 4 GB, the application should divide them into smaller file chunks, and send each chunk as a separate file.

The **payloadlen** field is a 16-bit unsigned value representing the length of the payload (data) section in the FMTP packet. For example, if TCP/IP is used for the reliable unicast service, the maximum **payloadlen** can only be $1500 - 20 - 20 - 12 = 1448$ bytes because Ethernet Maximum Transmission Unit is 1500B, IP header is 20B, TCP header is 20B, and the FMTP header is 12B. If correspondingly a UDP socket is used for the multicast packets, the UDP header is only 8 bytes, which is less than the 12B needed for the TCP header. But the FMTP sender limits the payload size to 1448B to allow for retransmission requests to correspond to single multicast packets. As control packets have no payload, the **payloadlen** field is simply set to 0.

The **flags** field is a 16-bit unsigned value represents the message type of the packet. Fig. 4.2 shows the currently defined values for the flag field.

The **payload** field contains the a block of a file. In some control messages such as EOP, the **payload** field is empty.

4.3.2 FMTP message formats

Each of the messages listed in Fig. 4.2 are described in this section.

Flag	Value
FMTP_BOP	0x0001
FMTP_EOP	0x0002
FMTP_MEM_DATA	0x0004
FMTP_RETX_REQ	0x0008
FMTP_RETX_REJ	0x0010
FMTP_RETX_END	0x0020
FMTP_RETX_DATA	0x0040
FMTP_BOP_REQ	0x0080
FMTP_RETX_BOP	0x0100
FMTP_EOP_REQ	0x0200
FMTP_RETX_EOP	0x0400

Figure 4.2: FMTP message types

FMTP_BOP message is sent by the FMTP sender via the multicast service to notify all receivers that blocks of a new file will soon follow. The payload of the FMTP_BOP message contains the file size, metadata and the size of the metadata, all three of which are provided by the application to the FMTP sender. Fig. 4.3 illustrates the message format of the payload part of FMTP_BOP. The **prodsizes** field contains the file (product) size. It is a 32-bit unsigned value. The **metasize** field contains the size of the following **metadata** field. For example, the LDM7 application includes a 16-byte MD5 signature as the **metadata** for its products, and hence **metasize** is 16 bytes. Currently, FMTP places a limit on the size of metadata since it sends each FMTP_BOP message in a single packet. But this constraint can be relaxed in future releases of FMTP. To construct an FMTP_BOP message, the FMTP sender sets the **prodindex** field in the FMTP header to a corresponding file index (previous **prodindex** + 1). Then it sets the **seqnum** field to 0, the **payloadlen** field to the length of the payload, and the **flags** field to 0x0001.

FMTP_EOP message is sent by the FMTP sender via the multicast service to notify all receivers that all blocks of the file have been sent. If an FMTP receiver does not receive this message before the timer for that file expires (the receiver starts a timer for each file

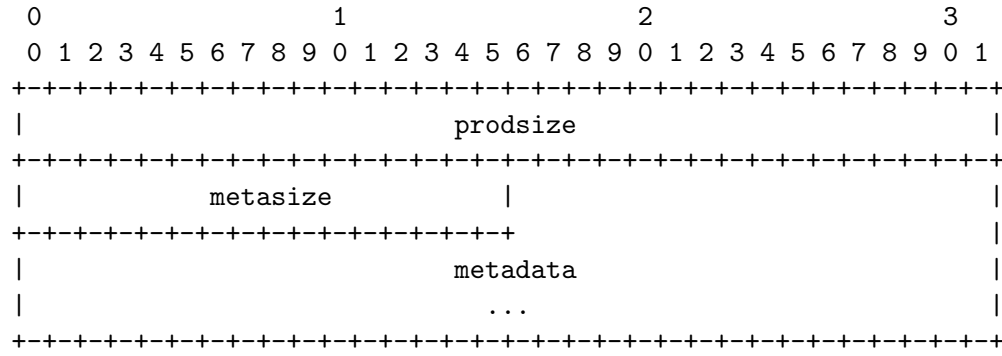


Figure 4.3: FMTP_BOP message format

upon reception of the FMTP_BOP), the FMTP receiver sends an FMTP_EOP_REQ message to the sender requesting retransmission of the EOP, but over the unicast reliable service. An FMTP_EOP message only contains an FMTP header, i.e., the payload is empty. The FMTP sender constructs an FMTP_EOP message by (i) setting the **prodindex** field to a corresponding file index; (ii) setting the **seqnum** field and the **payloadlen** field to 0; and (iii) setting the **flags** field to 0x0002.

FMTP_MEM_DATA message carries a data block of the file. This message is only sent from an FMTP sender to FMTP receivers using the multicast service. The FMTP sender constructs an FMTP_MEM_DATA message by (i) setting the **prodindex** field to a corresponding file index; (ii) setting the **seqnum** field to a corresponding value (number of bytes already sent); (iii) setting the **payloadlen** field to the number of data bytes being carried in the message; and (iv) setting the **flags** field to 0x0004. The FMTP sender fills the payload of all FMTP_MEM_DATA messages to the maximum extent allowed by the lower layers, except for the last FMTP_MEM_DATA message, which carries the remaining few bytes.

One FMTP_RETX_REQ message is sent by an FMTP receiver to the FMTP sender for each block of data for which the receiver requires retransmissions. These messages are sent over the unicast reliable service of the lower layers. The number of FMTP_RETX_REQ messages sent depends upon the size of missing data. The FMTP receiver determines the total number of lost bytes, and divides this number by the maximum-allowed size for the payload of each FMTP FMTP_MEM_DATA message, and then issues one FMTP_RETX_REQ message for each requested FMTP_MEM_DATA message. The size of the last segment could be smaller than the maximum-allowed size for the payload of FMTP FMTP_MEM_DATA messages. The FMTP

receiver uses the `prodindex` field to identify the file, the `seqnum` field to identify the missing block within that file, and the `payloadlen` field according to the size of the `FMTP_MEM_DATA` message being requested. The `flags` field is set to `0x0008` instead. There is no payload in this `FMTP_RETX_REQ` message.

`FMTP_RETX_REJ` message is sent by the FMTP sender to an FMTP receiver upon receipt of an `FMTP_RETX_REQ` message from the receiver that arrived after the maximum retransmission period associated with the file ended at the FMTP sender. In an `FMTP_RETX_REJ` message, the `prodindex` field is set to the same value as in the corresponding `FMTP_RETX_REQ` message. Since this message is designed to reject all the following `FMTP_RETX_REQ` messages for a file from the requesting FMTP receiver, the FMTP sender does not need to precisely indicate which request is being rejected. Hence, both the `seqnum` and the `payloadlen` fields are set to 0. The `flags` field is set to `0x0010`. There is no payload in this `FMTP_RETX_REJ` message. An `FMTP_RETX_REJ` message may not only be triggered by maximum retransmission period timeout, but also by an invalid `FMTP_RETX_REQ` message (e.g., non-existing `prodindex` value or non-existing `seqnum` value).

`FMTP_RETX_END` message is sent by an FMTP receiver to an FMTP sender via the unicast reliable service to indicate that it has received all blocks of the file through multicast or through retransmissions. If the FMTP receiver received all multicast blocks successfully, an `FMTP_RETX_END` message is sent to the FMTP sender right after the `FMTP_EOP` message arrives at the FMTP receiver. Otherwise, the FMTP receiver sends an `FMTP_RETX_END` message to the FMTP sender after all the missing blocks are recovered through retransmissions. If an `FMTP_RETX_REQ` message from a receiver is rejected by the FMTP sender, the FMTP receiver will not send a `FMTP_RETX_END` message to the sender. Upon receiving `FMTP_RETX_END` messages from all receivers, the FMTP sender correspondingly turns off the timer for the maximum retransmission period of the file, and notifies the application that the file is now released back to the control of the application. In an `FMTP_RETX_END` message, `prodindex` field is set to the file index that the FMTP receiver is acknowledging. Both the `seqnum` field and the `payloadlen` field are set to 0. The `flags` field is set to `0x0020`. This message `FMTP_RETX_END` contains no payload.

`FMTP_RETX_DATA` message carries the retransmitted data block that is requested by an

FMTP receiver. This message is only sent from an FMTP sender to an FMTP receiver via the reliable unicast service. The message format is identical to the `FMTP_MEM_DATA` message except that it is sent via the reliable unicast service instead of via the multicast service, and the `flags` field is set to `0x0040`.

`FMTP_BOP_REQ` message requests the retransmission of an `FMTP_BOP` message. This message is only sent from an FMTP receiver to an FMTP sender via the reliable unicast service. An FMTP receiver detects a missing `FMTP_BOP` if it receives an `FMTP_MEM_DATA` message or `FMTP_EOP` message for a particular `prodindex` without having received a preceding `FMTP_BOP` message with the same `prodindex`. Therefore, it issues an `FMTP_BOP_REQ` message to the FMTP sender with the expected `prodindex`. If an `FMTP_BOP` is missing for a file, the FMTP receiver cannot write data blocks received for that file into application memory space because the application would not have had an opportunity to allocate memory space and pass the memory location down to the FMTP receiver. Therefore all data blocks received via multicast while the FMTP receiver awaits the retransmitted `FMTP_BOP` message are simply dropped by the FMTP receiver. The `seqnum` field and the `payloadlen` field are set to 0 and the `flags` field is set to `0x0080`. No payload is contained in this `FMTP_BOP_REQ` message.

`FMTP_RETX_BOP` message carries the retransmitted `FMTP_BOP` message. This message is only sent from an FMTP sender to an FMTP receiver via the reliable unicast service. The FMTP sender retransmits the `FMTP_BOP` message that is requested by an FMTP receiver and replaces the `flags` field in the `FMTP_BOP` message with `0x0100`. Thus, this message is of the same format as the `FMTP_BOP` message.

`FMTP_EOP_REQ` message requests the retransmission of an `FMTP_EOP` message. This message is only sent from an FMTP receiver to an FMTP sender via the reliable unicast service. An FMTP receiver issues this request when it detects loss of an `FMTP_EOP` message when the receiver timer for the file times out, or it receives an FMTP message with a higher `prodindex` for the next product. In the `FMTP_EOP_REQ` message, the `prodindex` field is set to the expected file index. The `seqnum` field and the `payloadlen` field are both set to 0 and the `flags` field is set to `0x0200`. No payload is contained in this `FMTP_EOP_REQ` message.

`FMTP_RETX_EOP` message carries the retransmitted `FMTP_EOP` message. This message is only sent from an FMTP sender to an FMTP receiver via the reliable unicast service. Upon

receiving an `FMTP_RETX_EOP` message, the FMTP sender retransmits the `FMTP_EOP` message with specified `prodindex` after replacing the `flags` field in the `FMTP_EOP` message with `0x0400`. Thus, this message has the same format as the `FMTP_EOP` message.

4.4 Protocol operation

This section describes the detailed interactions of the FMTP sender and FMTP receivers participating in the same multicast group. A synopsis of the protocol operation is as follows:

1. An FMTP sending application instantiates an FMTP sender, which then uses the lower-layer protocols to configure the multicast service, and waits for the FMTP receivers to request connections for the reliable unicast service.
2. An FMTP receiving application instantiates an FMTP receiver, which then configures itself for multicast reception, and issues requests to the FMTP sender to establish a connection for reliable unicast service.
3. The FMTP sending application passes information about a file-stream to the FMTP sender. For each file (product), the FMTP sender sends an `FMTP_BOP` message to the multicast group. Upon receiving the `FMTP_BOP` message, the FMTP receiver extracts information contained in the BOP and notifies its receiving application. If the receiving application wants to receive the product, it provides a memory location in response to the FMTP receiver.
4. The FMTP sender sends one `FMTP_MEM_DATA` message corresponding to each block of the file to the multicast group. Simultaneously, the FMTP sender receives and handles retransmission request messages, (`FMTP_RETX_REQ`, `FMTP_RETX_BOP` and `FMTP_RETX_EOP`). FMTP receivers receive the arriving `FMTP_MEM_DATA` messages and identify missing messages, if any. If a missing message is detected, the FMTP receiver issues an `FMTP_RETX_REQ` message immediately.
5. The FMTP sender sends an `FMTP_EOP` message to the multicast group when all blocks of a file have been multicast. It simultaneously waits for `FMTP_RETX_END` messages

from all FMTP receivers. If the maximum retransmission period times out for the file, or all receivers have sent in `FMTP_RETX_END` messages for the file, the FMTP sender releases the file back to the application.

6. The FMTP sender and receivers repeat the above file multicast and block retransmission steps for the next file in the file-stream.

4.4.1 Underlying protocol layers

As stated earlier, FMTP assumes the availability of a potentially unreliable multicast service and a reliable unicast service from the underlying protocol layers. The FMTP protocol operation in the rest of this section assumes that multicast service is provided by OpenFlow switches at Layer 2 (L2) in which Ethernet frames are replicated and forwarded based on OpenFlow table entries. Providers such as Internet2 now offer dynamic L2 path services. Our research group deployed a multi-domain SDN on which a rate-guaranteed L2 paths were provisioned [23]. Further, our research group provisioned a multipoint VLAN across this multipoint SDN [24].

At the end hosts, a UDP/IP socket is used for sending/receiving the FMTP packets to/from the L2 multipoint VLAN. Sockets offer an easy-to-use programming API, and hence FMTP uses UDP sockets for the multicast service. The UDP header is only examined at the end hosts, as is typically done even with unicast applications. The IP layer is examined only at the end hosts since packet forwarding is based strictly on the VLAN ID carried in the IEEE 802.1q header at all the switches between the sender and the receivers. In other words, there is no Layer-3 (IP-layer) packet forwarding on the paths between the sender and receivers.

IP offers additional value besides the easy-to-use socket API. IP multicast addresses are translated by the kernel into multicast Ethernet MAC addresses through a fixed mapping procedure. The most-significant three bytes of the multicast Ethernet MAC address are fixed (0x01:00:5e), the first bit of the fourth byte is set to 0, and the remaining 23 bits are copied from the least-significant 23 bits of the IP address. The most-significant 4 bits of all multicast IP addresses are 1110. The next five bits of the IP address are ignored, which

leaves the remaining 23 bits that are used in the mapping. This automatic mapping is useful as it allows the Ethernet NICs/drivers of all receivers to be configured to accept Ethernet frames whose destination MAC address matches the multicast MAC address. As soon as an application, or in our case the FMTP receiver, opens a UDP/IP socket with a multicast IP address, one that has 1110 in the most-significant 4 bits, the Ethernet NIC/driver is configured to accept frames with the corresponding multicast MAC address. It is for these two reasons, ease of programming and support for multicast addresses, that the IP layer is used on top of the L2 multipoint VLAN.

A TCP/IP socket is created using the unicast IP addresses of the sender/receiver VLANs used in the end-to-end multipoint VLAN. In other words, the retransmission requests and block retransmissions are sent over the same VLAN as the multicast packets, but because block retransmissions are addressed to the unicast IP address of the retransmission-requesting receiver, no other receiver will accept those frames.

In summary, the lower layers, TCP, UDP, IP and Ethernet/VLAN, jointly offer both the (unreliable) multicast and reliable unicast services required by FMTP. In the description of the FMTP protocol operation in this section, we assume these lower-layer protocols.

4.4.2 FMTP sender initialization

The FMTP upstream application instantiates an FMTP sender using multiple configuration parameters. These parameters include multicast IP address, UDP port number, unicast IP address of the interface to which multicast packets are directed within the sending host, Time-to-Live (TTL) to multicast packets, an unicast IP address for the TCP connection, a TCP port number, and an initial product index.

With the *multicast IP address and UDP port number* specified, the FMTP sender opens a UDP socket. This action causes the kernel to configure the Ethernet NIC/driver to accept frames with the multicast MAC address derived from the multicast IP address. Using Linux `setsockopt`, with the `IP_MULTICAST_IF` option, the unicast IP address of the interface to which multicast packets are sent is added to the UDP socket. The step makes it possible for the IP layer of the sending host to determine the interface to which packets destined to the multicast IP address should be directed. With this action, no special IP routing

table entry needs to be added at the sender for the multicast IP address. Usually, there is a default interface for multicast IP addresses, but if the multipoint VLAN was configured on a different Ethernet interface, then this action of setting an interface for the correct routing of IP packets addressed to the multicast group is required.

The *unicast IP address of the interface to which multicast packets are sent* should be the address of the VLAN configured at the sending host. As part of setting up the end-to-end multipoint VLAN, Linux `vconfig` command is executed with a particular VLAN ID to create this virtual interface at the sending host NIC. A similar action is required at the receivers. In addition, Linux `ifconfig` command is executed at the end hosts to set a private unicast IP address for the newly created VLAN at the sender and receivers. The subnet IDs of the private unicast IP addresses and subnet masks assigned to all hosts on the multipoint VLAN should be the same. Thus, the FMTP applications should learn the private unicast IP addresses of the VLANs at the sender and receivers and pass these values to the FMTP senders/receivers during instantiation.

Linux `setsockopt` has another option called `IP_MULTICAST_TTL`. *Time-To-Live (TTL)* is required in IP multicast networks where packet multicasting is done at the IP layer in IP routers. Since IP routing tables could have loops (because of the use of distributed routing protocols), a TTL field is needed. As multipoint Ethernet VLANs are provisioned by a centralized SDN controller, there should be no packet forwarding loops in the tables, and hence TTL is not needed in the Ethernet or 802.1Q headers. While FMTP was designed from the start to run over rate-guaranteed multipoint VLANs, we allowed for the case that it may be deployed on IP multicast networks. This is not a recommended option because (a) varying levels of congestion on paths to different receivers could adversely impact throughput (because of a lack of an admission control procedure that is necessary for configuring rate-guaranteed multipoint VLANs), and (b) as FMTP does not have congestion control, an aggressive sender could overwhelm other flows.

The next part of initialization is the creation of one TCP connection between the FMTP sender and each FMTP receiver. The FMTP sender opens a TCP socket with the application-provided *IP address for the TCP connection and TCP port number*. The IP address provided for the TCP connection could be the same as the unicast interface IP

address provided for the transmission of multicast packets. But, for the sake of generality, these two unicast IP addresses were allowed to be different. The FMTP sender listens on the TCP socket, and responds to TCP connection requests from FMTP receivers. The FMTP sender maintains and dynamically updates a list of FMTP receivers in the multicast group.

The last parameter set by the application during FMTP sender instantiation is the *initial product index*. The default value of the initial product index is 0. But if this value is non-zero, the FMTP sender simply uses this application-provided value in the FMTP header `prodindex` field for all blocks of the first product. Allowing the application to specify a non-zero value is useful to the application in case the application needs to terminate an FMTP sender instance during a file-stream distribution, and restart a new FMTP sender instance later. When the new FMTP sender instance is started, it can be passed the index of the next product to serve. This initial product index, if specified, is used for the single file-stream served by an FMTP sender. There is no file-stream index because each FMTP sender handles only one file-stream.

4.4.3 FMTP receiver initialization

At each receiving host, an FMTP downstream application, which is counterpart to the FMTP upstream application that instantiates the FMTP sender at the sending host, instantiates an FMTP receiver. The parameters that are specified in this FMTP-receiver instantiation include multicast IP address, UDP port number, unicast IP address of the interface on which multicast packets are received, an IP address for the TCP connection, and a TCP port number. The multicast IP address and UDP port number are the same values used by the FMTP upstream application. The unicast IP address of the interface on which multicast packets are received is that of the VLAN configured in the receiving host, which is part of the end-to-end multipoint VLAN as described in Sec 4.4.2. A reverse path filter may be required on receiving hosts to allow through packets destined to the multicast IP address and the unicast IP address of the VLAN. The IP address for the TCP connection, and TCP port number correspond to those of the FMTP sender. These parameters are used by the FMTP receiver to initiate the setup of a TCP connection to the FMTP sender. The TCP connection is used as a reliable channel for retransmissions.

4.4.4 File multicast and retransmissions

Sender actions The application uses the FMTP sender API call `SendProduct` to send products in a single file-stream. The application can do so at any point after it instantiates an FMTP sender. This event happens asynchronously relative to when FMTP receivers send requests to the FMTP sender for TCP connections (which effectively places receivers into the multicast group). The `SendProduct` function minimally requires a pointer to memory location where the product is stored and the product size. Optional parameters include metadata size and a pointer to the location of the metadata associated with a product. The FMTP sender constructs an `FMTP_BOP` message as described in Section 4.3. Before describing the contents of the `FMTP_BOP` message payload, consider the fields of the FMTP packet header.

The `prodindex` field in the FMTP packet header is set to the default value of 0, or the initial product index, which was provided by the application during instantiation of the FMTP sender. This `prodindex` field in all FMTP packets is what makes FMTP a transport protocol for serving file-streams, not just individual files. The presence of the `prodindex` field in all FMTP packets allows an FMTP sender to simultaneously multicast a product, while serving retransmissions for previously multicast products. The `prodindex` field allows an FMTP receiver to detect a completely missing file, which could happen if the `FMTP_BOP`, all `FMTP_MEM_DATA` messages carrying the blocks of the file, and the `FMTP_EOP` are all dropped/errored. Finally, if the `FMTP_EOP` of one file and the `FMTP_BOP` of the next file are both lost, data blocks of the second file will not be mistakenly aggregated with data blocks of the first file since the `prodindex` will be different. Values of the other fields of the FMTP header of `FMTP_BOP` were described in Section 4.3.

The `FMTP_BOP` payload carries the product size, specified by the application in the `SendProduct` API call, in the `prodsizesize` field. If the application provides the optional parameters, metadata size and a pointer to the location of the metadata associated with a product, in the `SendProduct` API call, these values are carried in the `metasize` and `metadata` fields of the `FMTP_BOP` payload.

We explain why the `FMTP_BOP` is even necessary in FMTP, and the reasons for the fields

carried in its payload using an example application, LDM. The LDM6 version uses TCP, and there is no equivalent of the `FMTP_BOP` message. Instead the TCP connection transports the metadata and the data of each product. The metadata includes an MD5 checksum of the product, and serves both as a signature and as a product name. When the product appears at an LDM6 receiver TCP socket, LDM6 copies the product from kernel space into a temporary location in user space, extracts the MD5 checksum, and uses it as a product name to check whether the product is already in the product queue (PQ). If the product is not present in the PQ, then LDM6 allocates memory for the new product in the PQ, and then LDM6 copies the product data from the temporary location into the newly created memory within the PQ. Therefore, there are two copies of the product: (i) from kernel-space TCP-buffer to the temporary user-space location, and (ii) from the temporary user-space location to the assigned location within the PQ. This solution reduces the number of copies within user-space by 1.

As LDM was a driving application for our design of FMTP, we designed the `FMTP_BOP` to carry metadata on behalf of the application to allow for performance optimizations, such as the reduction in the number of data copies. Exactly how this is done is explained below in the description of the receive-side operation.

After sending the `FMTP_BOP` message to the UDP socket for multicasting, the FMTP sender reads data blocks of the product directly from the location of the product held in the application's user-space memory. The FMTP sender chooses a maximum block size as permitted by the lower-layer protocols. Each block is multicast via the UDP socket in an `FMTP_MEM_DATA` message. An `FMTP_EOP` message is also multicast via the UDP socket. After sending the `FMTP_EOP` message for a file, the FMTP sender sets a timer corresponding to the maximum retransmission period for that file.

While performing the file multicast or soon after, the FMTP sender may receive `FMTP_BOP_REQ` or `FMTP_RETX_REQ` for one or more data blocks from any receiver. These messages will be received over the TCP connections. The FMTP sender responds to an `FMTP_BOP_REQ` with an `FMTP_RETX_BOP`, and responds to each `FMTP_RETX_REQ` with an `FMTP_RETX_DATA` message. An FMTP sender may also receive an `FMTP_EOP_REQ` from any receiver for a file that is currently being multicast (premature receive-side timers can cause

an FMTP receiver to send this message even before the FMTP sender has sent the `FMTP_EOP` message for a file), or for a file whose multicast phase is complete. In either case, the FMTP sender will send an `FMTP_RETX_BOP` message to the requesting receiver via its TCP connection.

After the multicast phase for a file, the FMTP sender awaits an `FMTP_RETX_END` message from each receiver, which is an acknowledgment of complete reception of a file. The FMTP sender maintains a table of such acknowledgments for each product. When the FMTP sender receives this acknowledgment message from all receivers for a file (recall that the FMTP sender maintains a list of receivers based on requests for TCP connections), or the retransmission period for the file ends, the FMTP sender notifies the application that it no longer needs access to the file. At this point, the application is free to release the memory in which the file was held for use by another product.

The FMTP sender then waits for the next `SendProduct` call from the application. The `SendProduct` call for another file could arrive while the FMTP sender is still serving retransmissions or waiting for the `FMTP_RETX_END` messages from all receivers. Thus, file multicasting through the UDP socket co-exists with retransmissions on the TCP connections.

If while waiting for the next `SendProduct`, the FMTP sender receives an `FMTP_RETX_REQ` for a block of a file that has already been released to the application, the FMTP sender sends an `FMTP_RETX_REJ` message to the receiver.

Receiver actions The FMTP receiver actions begin upon receiving an `FMTP_BOP` message for a file on its multicast (UDP) socket. The FMTP receiver extracts the `prodindex` of the file, and checks it against a variable that contains the most-recent `prodindex` received through the multicast socket. If the `prodindex` of the new file does not exceed the most-recent `prodindex` by 1, the FMTP receiver construes that one or more files were fully lost, and hence sends `FMTP_BOP_REQ` messages for the missing files. Upon receiving `FMTP_RETX_BOP` (or `FMTP_BOP`) messages, the FMTP receiver will start reception of data blocks for the new files. A newly starting FMTP receiver will not have a valid entry in the most-recent `prodindex` variable, and hence will accept an `FMTP_BOP` with any `prodindex`.

Next, the FMTP receiver starts a timer for the product, which was described in Section 2.2 of Chapter 2, to enable detection of lost `FMTP_EOP` messages, and then the FMTP receiver sends the `prodsizes`, `metasizes`, and `metadata` extracted from the `FMTP_RETX_BOP` (or `FMTP_BOP`) payload up to its FMTP downstream application. The application in turn processes these values internally, and returns a response back to the FMTP receiver providing it a memory location into which the product data should be stored, or instructing it to drop the product.

If the FMTP receiver is instructed to drop the product, it simply drops all received `FMTP_MEM_DATA` messages carrying blocks of the product and the `FMTP_EOP` message. If, on the other hand, the FMTP receiver is provided a memory location, as a confirmation that the FMTP downstream application wants to receive the product, the FMTP receiver saves the `prodsizes` from the `FMTP_BOP` message, processes every received `FMTP_MEM_DATA` message for the product. Two variables associated with the file are created/updated to store the `seqnum` and `payloadlen` from the FMTP header of each `FMTP_MEM_DATA` message. Using these numbers, the FMTP receiver can detect missing blocks and send `FMTP_RETX_REQ` messages to the FMTP sender.

If an `FMTP_MEM_DATA` is received for a file with a missing `FMTP_BOP`, the message is dropped because the FMTP receiver would not have had an opportunity to notify the FMTP downstream application to allocate a memory location into which the data blocks can be written.

Comparing the total number of received bytes with the saved `prodsizes`, the FMTP receiver can detect successful reception of the whole file. Moreover, the FMTP receiver will receive an `FMTP_EOP` message for the file. If one or more blocks at the end of the file were dropped/errored, the arrival of the `FMTP_EOP` message is used to trigger `FMTP_RETX_REQ` messages for these missing blocks. Upon receiving the corresponding `FMTP_RETX_DATA` messages, the FMTP receiver writes the payload into the appropriate application memory space.

If the receive-side timer for the file expires before the arrival of the `FMTP_EOP` message, the FMTP receiver sends an `FMTP_EOP_REQ` message along with `FMTP_RETX_REQ` messages for missing data blocks, if any. Upon the reception of `FMTP_RETX_DATA` messages, if any,

the FMTP receiver writes the payload data into the appropriate application memory space. Upon the reception of `FMTP_RET_X_EOP`, the FMTP receiver changes the status of the product, and awaits retransmitted blocks if any are still pending.

If all blocks were successfully received, either through the original multicast, or through retransmissions, the FMTP receiver issues the `FMTP_RET_X_END` message to the FMTP sender. The FMTP receiver notifies the application of successful reception of the file using `prodindex` to identify the file.

The FMTP receiver may receive an `FMTP_RET_X_REJ` message, in which case the file was not completely received. In this case, the FMTP receiver notifies the application of failed reception for the file using `prodindex` to identify the file.

The FMTP receiver is designed to support concurrent file reception. In other words, an FMTP receiver could receive `FMTP_BOP` messages for other products while still serving a previous product. The FMTP receiver maintains per-file variables and hence can handle multiple file receptions simultaneously.

If an FMTP receiver receives an `FMTP_MEM_DATA` message whose `prodindex` is not in the set of products under current reception (which would happen if `FMTP_BOP` message for that product was missing), the FMTP receiver sends an `FMTP_BOP_REQ` for that product to the sender.

4.4.5 Transmission and retransmission timeout mechanisms

The FMTP sender maintains a per-file timer to limit the maximum retransmission period for the file. The timer prevents an FMTP sender from spending too much of its compute/network capacity serving a few receivers with high packet loss rates as this could adversely affect the multicast delivery of new files. This maximum retransmission period could be file-dependent or file-independent based upon the sizes of files and the rate of the underlying multipoint VLAN. If transmission delays dominate, then a file-dependent solution is required, but if the propagation delay is the dominant factor, then a file-independent timer is sufficient.

In experiments conducted with LDM7 for an IDD feedtype, we found the file-independent solution to be sufficient, as described in Chapter 3. We recommended the use of a dynamic system in which a controller carries out a post-facto analysis to determine the ideal timeout

value that should have been used based on product creation times and product release times logged at the LDM7 sender for already transmitted files in a specified time interval. This computed ideal value is then used in an EWMA scheme to update a running average value in order to compute a new maximum retransmission period for the files that are to be transmitted in the next interval. This new value is then used for all files transmitted in the next interval. This method is represented as follows

$$c = \frac{7}{8} \times c' + \frac{1}{8} \times c_{ideal}, \quad (4.1)$$

where c' is the c value used in the past interval, c_{ideal} is the ideal value that should have been used in the past interval, and c is the new value computed for use in the next interval. This process is repeated at the end of each time interval. We used $\frac{7}{8}$ and $\frac{1}{8}$ as is currently used in the TCP retransmission timeout algorithm.

On the FMTP receiver side, there is a timer to monitor per-file receiving period ($\tau_{rcv}(n)$) for each file n . This timer is an estimate of the amount of time required for an FMTP receiver to receive the whole file after it has received the first message. Upon receiving an FMTP_BOP message for a file n , the FMTP receiver computes $\tau_{rcv}(n)$ using a factor f_{rcv} as follows:

$$\tau_{rcv}(n) = f_{rcv} * S_n / r_{mc} \quad (4.2)$$

This timer is required to generate a retransmission request if the FMTP_EOP message is missing for a file. The file receive-timeout factor f_{rcv} should be an integer value greater than 1 so that if individual packets are caught in switch buffers behind other packets, the FMTP receiver avoids sending premature retransmission requests. Chapter 3 described our experiments with this factor for LDM7/FMTP.

4.5 Implementation

We have implemented FMTP with C/C++. It was developed for Linux/Unix systems. The FMTP code has been tested on the following mainstream OS distributions: Ubuntu LTS

12.04+, CentOS 5+, Mac OSX 10.9+, and FreeBSD 9+. The latest source code can be found in UCAR's `github` repository [25]. This code contains the following: (i) sending-side program, (ii) receiving-side program, (iii) a test application with sending- and receiving-sides, (iv) wireshark packet dissector, and (v) doxygen-style documentation.

Thread Model After the sending-side multicast thread completes its initialization phase, it spawns two additional threads: a timer thread, and a coordinator thread. The multicast thread is responsible for multicasting products, and the timer thread is responsible for maintaining a timer for each product. The coordinator thread is a supervisor that accepts incoming TCP connections and forks a new retransmission thread for each receiver. The new retransmission thread is responsible for handling retransmission requests from the associated receiver. Therefore, on the sending side, there will be $3 + N$ threads eventually, where N is the number of receivers. The receiving-side main thread will fork into four threads: a multicast handler thread, a retransmission handler thread, a retransmission requester thread and a timer thread. The multicast thread is responsible for FMTP messages received via multicast. The retransmission thread is responsible for handling retransmitted blocks received on the unicast (TCP) connection. The retransmission requester thread is responsible for parsing internal messages, and constructing and sending corresponding retransmission request messages. The timer thread maintains the receive-side timer for each product.

Timed Queue and Message Queue Queues are key data structures in the implementation. Two types of queues are used: timed queues and message queues. Timed queue is an unordered queue that does not guarantee first-in first-out (FIFO) services. The sequence with which queued entries depart this queue depends upon the timeout period associated with each entry. One timer is entered in this queue for each product. A timed queue is needed at the sender and at each receiver. When the shortest timer expires, actions are executed for the corresponding product. For example, if a receive-side timer expires for a product, and the product EOP has not been received, the FMTP receiver will send an `FMTP_EOP_REQ` message to the FMTP sender.

Message queue is a common FIFO queue, which is used for inter-thread communications. The receiving-side multicast thread and retransmission requester thread are two separate threads, which implies the execution of these two threads is asynchronous. The multicast

thread needs to send information identifying missing blocks so that the requester thread can send retransmission requests for these blocks. A message queue is used for this purpose. The requester thread keeps querying the message queue for new requests. It either retrieves the oldest request and continues to process, or blocks on an empty queue.

Retransmission Metadata Retransmission Metadata (**RetxMeta**) is an auxiliary data structure maintained per product on the sending side. It helps the retransmission thread locate particular blocks for which retransmissions are requested. When the application passes a pointer to the memory location of a product to the FMTP sender, the FMTP sender formats a new **RetxMeta**, and inserts this structure into a **RetxMeta-List**. Before the timer expires, all retransmission threads have access to this list. If a request for a block of the product is received, the retransmission thread searches the **RetxMeta-List**, and extracts the corresponding **RetxMeta**. This data structure contains a pointer to the memory location of the product. The retransmission thread accesses the block by offsetting the product pointer with the sequence number parsed from the FMTP header of the **FMTP_RETX_REQ** message. Also, the **RetxMeta** for a product holds a list of outstanding receivers that have not yet sent in their **FMTP_RETX_END** messages for that product. When an **FMTP_RETX_END** message is received by a retransmission thread from its corresponding receiver, the retransmission thread removes the receiver from the list of receivers maintained for that product in the **RetxMeta**. When a retransmission thread finds that its receiver is the last one to send a **FMTP_RETX_END** message for a product, that retransmission thread erases the **RetxMeta** for that product from the **RetxMeta-List**, and clears the corresponding timer.

Per-product Bitmap We have implemented a per-product bitmap to track data blocks of a product at the FMTP receiver. A per-product bitmap is a data structure that contains as many bits as number of blocks in a product. When a data block arrives, the FMTP receiver calculates the block index based on sequence number and payload length, and then uses the index to locate the bit and sets it to **true**. The arrival of the EOP for the product invokes an integrity check on the bitmap. A bit remaining **false** denotes a lost data block. The bitmap thus provides the receiver a second method to identify losses, and issue retransmission requests. The first method is the use of **seqnum** to identify lost blocks since in-sequence arrival is guaranteed on a provisioned multipoint VLAN.

Integration with LDM7 An important goal of this FMTP development effort was to integrate it with LDM6 to create LDM7 for improved distribution of meteorological data in UCAR’s IDD project. We integrated FMTP into LDM7 as a pluggable module in addition to the original network library so that LDM7 can decide whether to use our (multicast) FMTP solution, or unicast TCP connections.

Function overloading was used in our C++ implementation for the `SendProduct` call to allow applications to specify a required minimal set of parameters (pointer to memory location where the product is stored and the product size), as well as optional parameters (metadata size and a pointer to the location of the metadata associated with the product), as was required by LDM7. In Section 4.4, we described how the upstream LDM7 server uses the optional fields and sends information to its downstream LDM7 servers in the `FMTP_BOP` message.

4.6 Related work

The Scalable Reliable Multicast (SRM) [26] protocol is based on a principle of receiver-based reliability, which means each receiver is individually responsible for detecting losses and recovering lost data. In FMTP also, each receiver is responsible for detecting losses and sending retransmission requests, but the FMTP sender does maintain a list of receivers and expects a final acknowledgment from all receivers for each file. While this feature makes the FMTP solution less scalable than the SRM approach in which sender does not perform such receiver tracking, FMTP requires this tracking in order to support its API that was designed to avoid one memory copy (for high-performance implementations). The FMTP sender needs to know when all receivers have received the file so that it can release its hold on that file and allow the application to reclaim that memory location. Second, in SRM, each receiver periodically multicasts its highest received sequence number of the data block that it has received. All listening receivers use this information to identify their own lost blocks, e.g., if a receiver’s highest received sequence number is smaller than the value multicast by another receiver, the first receiver deduces that it needs to request retransmissions. FMTP uses a different method for loss detection, which is through the transmission of EOPs and receiver

timeouts. Third, SRM receivers multicast repair requests and retransmissions to the whole group in order to suppress duplicate requests from other receivers. If a receiver missed a packet, it is able to retrieve the retransmission from another close-by receiver without having to request it from the original sender. A receiver waits for a random period before sending out a repair request to leverage retransmissions from other repair requests for the same missing data. A disadvantage is that this random backoff time interval adds to file-delivery latency. For a stream of small files, it could cause a throughput drop. FMTP concurrently recovers missing blocks even during the multicast, which somewhat hides propagation delay to the sender. The FMTP solution has only one sender, which does have the drawback of propagation delay in wide-area deployments. On the other hand, only FMTP receivers need to be executed at the receiving hosts.

As discussed in Section 2.6, coding-based reliable multicast schemes, such as ALC and FLUTE, can provide reliable data delivery without positive or negative acknowledgments from receivers to the sender. Another coding-based reliable multicast scheme [27] proposed a layered design in which each layer provides a different degree of redundancy. Each receiver can choose different encoding levels based on its QoS requirements, and the quality of the path from the sender to the receiver. Receivers enduring poor network quality can use a layer with higher redundancy to achieve the required reliability. This idea of differentiating good service quality from poor service quality is useful to increase group performance. Such coding based methods waste bandwidth and compute cycles in return for high scalability, and are hence typically used in applications with millions of users. Since FMTP is designed to serve multicast groups with just hundreds of receivers, we chose to use retransmission based schemes rather than coding.

A multicast-based replication for Hadoop HDFS [28] [29] is proposed to reduce the network traffic caused by HDFS write operations. A congestion-controlled reliable multicast socket (CCRMSocket) is developed to replace the native pipelined replication method. CCRMSocket uses IP multicast with the TCP acknowledgment mechanism to provide reliable multicast service. This work demonstrates the application of reliable multicast in data center-networks. FMTP could also be used in data-center applications.

4.7 Conclusions

A new reliable multicast transport protocol, named FMTP, was designed and implemented to serve real-time scientific file-stream distribution. This protocol relies on the network offering a packet multicast service. Our FMTP implementation uses a UDP socket to send message blocks to the network multicast service, and unicast per-receiver TCP connections to retransmit missing message blocks from the sender. Multicast and retransmissions are handled by FMTP simultaneously. The FMTP receiver uses a file-associated bitmap to track message blocks. The FMTP receiver also has a per-file receive timeout ($\tau_{rcv}(n)$) to detect losses of one-or-more message blocks at the end of a file, and missing End-of-Product (EOP) messages. The FMTP sender has a maximum retransmission period for each file to limit the resources consumed for retransmissions. FMTP is designed to be used over rate-guaranteed multipoint virtual networks and hence it does not have built-in congestion control or flow control. FMTP can operate in lossy networks and yet save compute and network resources when compared to the application-layer multicast solution.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we presented our work towards solving a wide-area real-time scientific data-distribution problem. We developed a cross-layer Multicast-Push Unicast-Pull (MPUP) architecture that defines functions at three layers: (i) link layer, (ii) transport layer, and (iii) application layer. It combines (i) a multicast-push function at the transport layer, (ii) a unicast-pull at the transport and application layers, and (iii) rate-guaranteed link-layer network multicast. We also developed a new reliable multicast transport protocol called File Multicast Transport Protocol (FMTP). Next, we integrated FMTP into an application, Local Data Manager (LDM), to create LDM version 7, as an example realization of the MPUP architecture. Finally, we executed LDM7 on the NSF-sponsored GENI testbed and compared in performance on throughput, latency, and resource requirements (bandwidth and CPU usage) with LDM version 6, which is an application-layer multicast solution. LDM6 runs on the IP-routed network, while LDM7 requires an underlying Software Defined Network across which rate-guaranteed multipoint virtual networks can be provisioned.

LDM7 can be gradually introduced to users who are already using LDM6 to various file-stream subscriptions. For receivers who do not have access to the new OpenFlow/SDN network services, the sender can continue using the LDM6 ALM solution, even as LDM7 servers concurrently offer the same file-streams via multipoint VLANs to those subscribers who have connectivity to an SDN. LDM7 usage can be slowly expanded to cover an increasing

number of receivers as the OpenFlow/SDN network services spread in availability.

5.2 Future work

Future work items include the following: (i) security-enhanced FMTP, and (ii) integration of LDM7 control module with a client to send dynamic requests to an SDN controller for the setup, modification, and release of underlying multipoint VLANs.

A weakness in multicast is security. With IP multicast, any host can join a multicast group as there is no explicit setup phase. If a malicious host joins the multicast group, it can start spoofing data, and perform other types of attacks. While multipoint VLANs require a setup phase, during which users can be authenticated, if one of these hosts are compromised after authentication, the compromised host can be used to inject spoofed data onto the multipoint VLAN. Even if firewall filters are set to prevent a receiver from receiving packets from any host other than the legitimate multicast application sender, source address spoofing is possible. To mitigate this threat, security enhancements are needed in both the control-plane and data-plane. One potential solution is to use IPsec for authentication and encryption, if confidentiality is required, to each data-plane packet. To reduce one memory copy on the receive side, in LDM7, received message blocks are written directly into the product queue. This features exposes a potential vulnerability for stack smashing and code injection attacks. To defend against such attacks, all FMTP receivers should verify the signature of a received message block as well as the identity of the sender. This requires a modification to FMTP.

The MPUP architecture allows receivers to dynamically join and leave a multicast group. The SDN controller should therefore take action to change the topology of the multipoint VLAN. For a new receiver to join the VLAN, the SDN controller needs to update flow tables in the switches to provision a new segment (potentially with policing and scheduling) in the multipoint VLAN. Similarly, when a receiver drops subscription to a feedtype, the SDN controller needs to release the portion of the multipoint VLAN to this receiver. All these actions are only performed when the SDN controller is notified by the sending LDM7.

The sending LDM7 translates incoming subscriptions from receivers and notifies the SDN controller via a message interface.

Appendices

Appendix A

Linux kernel implementation of the network stack

In the Linux network stack, to transmit a packet, the IP layer calls the function `dev_queue_xmit()`, whose source code is part of the file `net/core/dev.c`. This function queues the `sk_buff` data structure, which holds the packet, in the ring buffer of the hardware device. If a `tc` queueing discipline is configured, the `dev_queue_xmit()` function invokes the virtual method `qdisc->enqueue`. The function `dev_queue_xmit()` returns a status indicating whether the `enqueue` function was successful or not. If the `enqueue` function was not successful, e.g., due to a lack of buffer space, the packet will be dropped. In other words, the function in the `tc` layer neither blocks the calling function (in other words, by waiting for buffer space to free up), nor does it just notify the calling function that there is insufficient space. Instead, it simply drops the packet if there is no buffer space.

Tracing the code to understand how the return status from the `enqueue` function is handled by the transport layer, we found that the responsible function to be `tcp_transmit_skb()`. The return status from the `tc` layer is propagated up to this TCP function `tcp_transmit_skb()` as `err`. Only when `err` is zero, indicating `NET_XMIT_SUCCESS`, will `tcp_transmit_skb()` return control to its calling function directly. For all other values of `err`, the TCP sender enters a Congestion Window Reduced (CWR) state by executing the `tcp_enter_cwr()` function. In this function, the TCP sender halves the congestion window

to reduce the sending rate.

In contrast, in CTCP, since the congestion-control code has been removed, the CTCP sender will not enter the CWR state or take any action even when the `err` value is non-zero. The function `ctcp_cong_avoid()` in the file `tcp-ctcp.c` is called when the sender receives an ACK. It is the core function that controls the size of the congestion window, `cwnd`, and correspondingly the maximum number of outstanding (unacknowledged) segments. For every ACK received, if an RTT measurement was ongoing for the corresponding segment, the `ctcp_cong_avoid()` function checks if the newly measured RTT value is lower than the current RTT. If so, a new `cwnd` is computed based on the new BDP using the `scale` factor (which is another configurable parameter in CTCP). If the newly measured RTT value is higher than the current RTT, `cwnd` remains unchanged. The TCP sending function `tcp_transmit_skb()`, which is common to Standard TCP and CTCP, computes the minimum of two parameters: `cwnd` and the receiver's flow window, to determine the number of packets to send to the `tc` buffer.

In UDP, function `udp_send_skb()`, whose source code is part of the file `net/ipv4/udp.c`, invokes the IP-layer function `ip_send_skb()`, which in turn invokes `dev_queue_xmit()`. Unlike TCP, the UDP-function function `udp_send_skb()` does not , in any way, react to the returning status `err` from the `dev_queue_xmit()` function.

In *summary*, both CTCP and UDP are similar in their lack of reaction to a dropped packet by the `tc` layer, when its buffer is full. Standard TCP is the only transport layer protocol that reduces its congestion window if a packet is dropped by the `tc` layer.

Bibliography

- [1] Local Data Manager. <http://www.unidata.ucar.edu/software/ldm/>.
- [2] Internet Data Distribution. <http://www.unidata.ucar.edu/software/idd/>.
- [3] B. Fenner and D. Meyer. Multicast Source Discovery Protocol (MSDP). RFC 3618 (Experimental), October 2003.
- [4] Sylvia Ratnasamy, Andrey Ermolinskiy, and Scott Shenker. Revisiting IP multicast. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 15–26, New York, NY, USA, 2006. ACM.
- [5] Open Networking Foundation. <https://www.opennetworking.org/>.
- [6] On-Demand Secure Circuits and Advance Reservation System (OSCARS). <http://www.es.net/OSCARS/docs/index.html>.
- [7] T. Akiyama, Y. Kawai, Y. Teranishi, R. Banno, and K. Iida. SAPS: Software defined network aware pub/sub – a design of the hybrid architecture utilizing distributed and centralized multicast. In *Computer Software and Applications Conference (COMP-SAC), 2015 IEEE 39th Annual*, volume 2, pages 361–366, July 2015.
- [8] S. Chen, X. Ji, M. Veeraraghavan, S. Emmerson, J. Slezak, and S. Decker. A cross-layer multicast-push unicast-pull (MPUP) architecture for reliable file-stream distribution. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*. IEEE, 2016.
- [9] Jie Li, M. Veeraraghavan, S. Emmerson, and R.D. Russell. File multicast transport protocol (FMTP). In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 1037–1046, May 2015.
- [10] A.P. Mudambi, X. Zheng, and M. Veeraraghavan. A transport protocol for dedicated end-to-end circuits. In *Communications, 2006. ICC '06. IEEE International Conference on*, volume 1, pages 18–23, June 2006.
- [11] LuisE. Nieto-Barajas and Tapen Sinha. Bayesian interpolation of unequally spaced time series. *Stochastic Environmental Research and Risk Assessment*, 29(2):577–587, 2015.
- [12] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61:5 – 23, 2014. Special issue on Future Internet Testbeds Part I.

- [13] Xiang Ji, Yicheng Liang, Malathi Veeraraghavan, and Steve Emmerson. File-stream distribution application on software-defined networks (SDN). In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 2, pages 377–386, July 2015.
- [14] S. Tepsuporn, F. Al-Ali, M. Veeraraghavan, X. Ji, B. Cashman, A. J. Ragusa, L. Fowler, C. Guok, T. Lehman, and X. Yang. A multi-domain SDN for dynamic layer-2 path service. In *Proceedings of the Fifth International Workshop on Network-Aware Data Management*, NDM '15, pages 2:1–2:8, New York, NY, USA, 2015. ACM.
- [15] Kyu-haeng Lee, Chong-kwon Kim, Soo-Hyung Lee, and Won-Tae Kim. Rateless code based reliable multicast for data distribution service. In *Big Data and Smart Computing (BigComp), 2015 International Conference on*, pages 150–156. IEEE, 2015.
- [16] B. Adamson, C. Bormann, M. Handley, and J. Macker. NACK-Oriented Reliable Multicast (NORM) Transport Protocol. RFC 5740 (Proposed Standard), November 2009.
- [17] M. Luby, M. Watson, and L. Vicisano. Asynchronous Layered Coding (ALC) Protocol Instantiation. RFC 5775 (Proposed Standard), April 2010.
- [18] T. Paila, M. Luby, R. Lehtonen, V. Roca, and R. Walsh. FLUTE - File Delivery over Unidirectional Transport, RFC 6726, November 2012.
- [19] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.
- [20] James F Kurose and Keith W Ross. *Computer networking: a top-down approach*. Addison-Wesley, 2007.
- [21] LDM Data Product. <https://www.unidata.ucar.edu/software/ldm/ldm-current/basics/data-product.html>.
- [22] NTP Clock Quality. <http://www.ntp.org/ntpfaq/NTP-s-sw-clocks-quality.htm>.
- [23] S. Tepsuporn, F. Al-Ali, M. Veeraraghavan, X. Ji, B. Cashman, A. J. Ragusa, L. Fowler, C. Guok, T. Lehman, and X. Yang. A multi-domain SDN for dynamic layer-2 path service. In *Proceedings of the Fifth International Workshop on Network-Aware Data Management*, NDM '15, pages 2:1–2:8, New York, NY, USA, 2015. ACM.
- [24] Xiang Ji. On Using Software Defined Networks for File-stream Distribution. diploma thesis, University of Virginia, May 2016.
- [25] UCAR FMTP Project. <https://github.com/Unidata/vcmtp>.
- [26] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *ACM SIGCOMM*, page 342, August 1995.
- [27] Christian Esposito, Aniello Castiglione, and Francesco Palmieri. Dealing with Reliable Event-Based Communications by Means of Layered Multicast. In *Algorithms and Architectures for Parallel Processing*, pages 572–581. Springer, 2015.

- [28] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.
- [29] Jiadong Wu and Bo Hong. Multicast-based replication for Hadoop HDFS. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*, pages 1–6. IEEE, 2015.