TCP-Friendly Many-to-Many End-to-End Congestion Control

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science

by

Innocenty Sukhov

supervised by

Prof. Danny Dolev

Institute of Computer Science

The Hebrew University of Jerusalem

Jerusalem, Israel

September 19, 2003

Acknowledgments

I would like to tender thanks to all my colleagues who contributed to this project by advice and support, most of all, Ilya Shnaiderman, for his determination in promoting the project and bringing it to the full completion.

I am extremely grateful to my academic supervisor Prof. Danny Dolev for his guidance and valuable assistance, original ideas and troubleshooting, personal concern and encouragement.

Special thanks to Dr. Tal Anker, who defined and promoted the project, for his helpful comments and insights.

Last, but not least, a great thank to my wife Inna who was with me through all my ups and downs, for her infinite love, patience, and understanding.

Abstract

The work addresses the issue of a TCP-friendly congestion control mechanism for many-to-many communication environment. Lack of congestion control is an inhibitor to the deployment of WAN applications that involve a collaboration of groups of processes in the Internet environment.

Recent efforts targeted unicast WAN congestion control (TFRC). We extend that approach to a single rate many-to-many multicast applications that operate using a middleware framework. The mechanism was implemented within a group communication middleware and tested in a multi-continent environment. The measurements prove that the proposed approach is robust, efficient and TCP-friendly, and provides fairness among processes that compete for shared resources.

Contents

Chapter 1

Introduction

1.1 IP Multicast and Congestion Control

As the Internet becomes more diversified in its capabilities, it becomes feasible to develop and offer services and applications that were not possible under earlier generations of Internet technologies. The Multicast Backbone (MBONE) and IP-multicast are two Internet technologies that have enabled a wide range of new applications. Using multicast, large-scale conferencing involving hundreds to thousands of participants is possible over the Internet. As multicast technologies become more widely deployed, we expect to see new multicast-based applications that demand more bandwidth and higher speed. Many of these applications will require reliable data transfer.

Congestion control is an integral part of any best-effort Internet data transport protocol. It is widely accepted that the end-to-end congestion control mechanisms employed in TCP [1] have been one of the key contributors to the success of the Internet. A conforming TCP flow is expected to respond to congestion indication (e.g., packet loss) by drastically reducing its transmission rate and by slowly increasing its rate during steady state. This congestion control mechanism encourages the fair sharing of a congested link among multiple competing TCP flows. A flow is said to be TCP-compatible or TCP-like if it behaves similar to a flow produced by TCP under congestion [2]. At steady state, a TCP-compatible flow uses no more bandwidth than a conforming TCP connection running under comparable conditions.

Unfortunately, most of the multicast schemes proposed so far do not employ end-to-end congestion control. Since TCP strongly relies on other network flows to use congestion control schemes similar to its own, TCP-incompatible multicast traffic can completely lock out competing TCP flows and monopolize the available bandwidth. Furthermore, multicast flows insensitive to existing congestion (especially congestion caused by their own traffic) are likely to cause simultaneous congestion collapses in many parts of the Internet [3]. Because of the potential far-reaching damage of TCP-incompatible multicast traffic, it is highly unlikely that transport protocols for large-scale reliable multicast will become widely accepted without TCP-like congestion control mechanisms.

The main challenge of congestion control for reliable multicast is scalability. To respond to congestion occurring at many parts of a multicast tree within a TCP time-scale, the sender needs to receive immediate feedback regarding the receiving status of all receivers. However, because of the potentially large number of receivers involved, the transmission of frequent updates from the receivers directly to the sender becomes prohibitively expensive and non-scalable.

This work is concerned with the flow and congestion control of best-effort multicast traffic.

1.2 Congestion Control in Group Communication Systems

Congestion control is a key element that enables WAN deployable applications to function in the Internet environment. Many applications of this kind involve a group of processes that are to collaborate by communicating via multicast.

Group communication systems (GCSs) are now widely recognized as powerful building blocks for supporting consistency and fault-tolerance in distributed applications.

There are several middleware implementations that offer group communication services with a variety of different semantics ([4, 5, 6, 7, 8, 9]).

In this work we address the issue of TCP-friendly congestion control mechanism for group communication middleware.

The specific group communication middleware that we focus on is the *Xpand* [10] WAN group communication system (GCS). However, our results are applicable, with some implementation adaptation, to other middleware systems as well. The services we exploited in order to add congestion control to Xpand were the group membership and the ability to detect message losses, which are inherent to any reliable multicast scheme.

Xpand and similar group communication systems typically use UDP for all their communication needs. This is due to the usage of native IP multicast infrastructure for message delivery. As a consequence, Xpand lacks the TCP built-in congestion and flow control mechanisms, which may lead to unfair bandwidth share, unstable transmission rate or even severe network congestion in a WAN environment. Moreover, usage of TCP congestion control mechanism over UDP would neither scale and nor provide the optimal results in a GCS.

In order to overcome these drawbacks, Xpand was supplied with congestion control mechanism based on the "TCP-Friendly Rate Control Protocol (TFRC)" [11], with modifications that extend it to a many-to-many framework and to general message traffic, without focusing on multi-media applications only. TFRC is an equation-based unicast congestion control mechanism, in which the equation derived from a model of TCP long term throughput ([12]) is used to limit the sender's transmission rate. We have extended the TFRC equation-based approach from unicast to many-to-many message exchange, where every member can be either a sender, or a receiver, or both.

A recent paper (*TFMCC* [13]) that addresses a similar problem focuses on directly expanding TFRC to deal with multicast. We take a different approach using the advantage of the built-in hierarchy in Xpand to address scalability issues, thus using a more natural approach to achieve scalability in multi-cluster based system. This approach enables building a simpler feedback mechanism with no need for feedback separation, as well as a simpler round-trip time estimation. Chapter 2 provides for more detailed comparison between Xpand and TFMCC.

Xpand was implemented and tested over the WAN environment described in [14]. To the best of our knowledge, no group communication system[1] has implemented an end-to-end congestion

[1] In the current context, a group communication system refers to middleware that supports many-to-many communication with reliable transmission services, membership services, ordering guarantees on message delivery, etc.

control mechanism.

Note that flow control mechanisms for group communication have already been thoroughly studied (See [16]). Xpand implements not only the congestion control mechanism, but also contains a built-in flow control mechanism that follows the standard techniques known in the literature ([17, 18, 19]).

Performed measurements validated the implemented congestion control mechanism. The experiments results showed that our extended TFRC model is applicable to WAN and to many-to-many communication patterns. The resulting Xpand behavior is TCP-friendly, and fairness is achieved among multiple Xpand senders sharing a bandwidth.

1.3 Overview of the Thesis

The rest of the Thesis is organized as follows:

- Chapter 2 is a survey of related work
- Chapter 3 briefly describes the Xpand's architecture and real wide area network environment we used for testing and experiments
- Chapters 4 and 5 give general design of congestion control mechanism in Xpandand detailed algorithm description, respectively
- in Chapter 6 we present intra-LAN flow control mechanism for a special case when all group members reside on the same LAN
- Chapters 7 and 8 describe performance evaluation results for congestion control in wide area environment, and flow control in intra-LAN environment, respectively
- Chapter 9 concludes the thesis
- Chapter 10 provides implementation related details which can be useful for those who will work on Xpand in the future

(See [15])

Chapter 2

Related Work

Congestion and flow control have attracted a lot of research in computer networking. There are numerous approaches to introducing congestion control mechanisms for multicast applications.

2.1 LAN vs WAN Environment

Typically, in a WAN it is a network that limits a transmission rate: link bandwidth is scarce, and routers may drop packets. On the contrary, in a LAN bandwidth is usually plentiful, and it is an ability of a receiving host to handle incoming traffic that becomes a performance bottleneck.

Most of the existing GCSs [20] are limited to LAN environment. Numerous instances of GCSs for LAN exist, a non-exhaustive list including Transis [21], Horus [6], Ensemble [7], Amoeba [22], Totem [5], etc.

Those systems are stable and show good performance in a local network environment, but were not designed to be deployed in a WAN. There are a few obstacles that make these systems impractical for wide area networks. Some of the obstacles are listed below.

- *Scalability.* Some of the data structures and protocols of those systems do not scale with the number of participants. For example: in Transis, as in many other GCSs, each participant collects acknowledgments from *all* other participants. This procedure will cause low performance when the number of participants reaches a few hundreds.

  Furthermore, ample amount of uniform bandwidth is implicitly assumed. This is why all LAN oriented GCSs lack congestion control mechanism, which makes them impractical in WAN.

- *Round Trip Time (RTT).* The message acknowledgment mechanism and the message completion mechanism in these systems were built on the assumption that RTT is short and varies insignificantly.

- *Low Loss Rate and Symmetric Bandwidth.* The flow control mechanism in a GCS for LAN assumes there are similar network resources for all participants and a low percentage of message losses.

To be applicable in a WAN environment, underlying protocols must be explicitly designed to overcome the emerging difficulties.

2.2 Single Rate vs Layered Multicast

In this work we focus on a single rate multicast congestion control in which the slowest receiver limits the transmission rate. The major alternative approach uses layered multicast scheme ([23, 24, 25]). With layered multicast, data can be delivered to receivers with different rates. A receiver adapts to available bandwidth by joining one or more multicast groups.

Our choice of a single rate is accounted for by the fact that group communication framework requires synchronization between all receivers. Nevertheless, the proposed approach could be extended to multi-layer multicast by employing the described technique for rate control on each layer.

2.3 Congestion Control Mechanisms For Single Rate Multicast

One possible classification criterion for TCP-friendly schemes is whether they adapt their offered network load based on a congestion window or on their transmission rate.

Algorithms that belong to the *window-based* category use a congestion window at the sender or at the receiver(s) to ensure TCP-friendliness. Similar to TCP, each packet transmitted consumes one slot in the congestion window, while each packet received or the acknowledgment of a packet received frees one slot. The sender is allowed to transmit packets only when a free slot is available. The size of the congestion window is increased in the absence of congestion indications and decreased when congestion occurs.

Rate-based congestion control achieves TCP-friendliness by dynamically adapting the transmission rate according to some network feedback mechanism that indicates congestion. It can be subdivided into simple *AIMD* schemes and *model-based* congestion control.

Simple AIMD schemes mimic the behavior of TCP congestion control. This results in a rate that displays the typical short term sawtooth like behavior of TCP. This makes simple AIMD schemes unsuitable for continuous media streams.

Model-based congestion control uses a TCP model such as the one presented in [26] instead of a TCP-like AIMD mechanism. By adapting the sending rate to the average long-term throughput of TCP, model-based congestion control can produce much smoother rate changes that are better suited for the aforementioned type of traffic. Such schemes do not mimic TCP's short-term sending rate but are still TCP-friendly over longer time scales.

2.3.1 Rate-Based Approaches

Many rate-based congestion control protocols mimic TCP's additive increase - multiplicative decrease behavior to achieve TCP-fairness, while others (e.g. Xpand) implicitly or explicitly adjust their rate according to a model of TCP traffic. A very obvious approach to TCP-friendly congestion control is to directly apply TCP's congestion control mechanism, but without the associated reliability mechanism.

LTRC

The Loss Tolerant Rate Controller for Reliable Multicast [27] uses an AIMD scheme, but does not directly copy TCP congestion control. Loss threshold values are used to determine whether the sender is allowed to increase its rate, or should maintain or decrease the rate. Rate modifications are performed periodically based on a timer value that corresponds to the time it takes to detect a loss.

To avoid the difficult task of determining the round-trip time to the receivers, the authors propose to use a pre-configured value. In addition, the maximum sending rate is limited to the rate given by the simple TCP model ([26]) for the worst receiver. Since a large number of parameters have to be preset and are not automatically adjusted during the multicast session, this protocol is not well suited for dynamic environments with changing network conditions. Receivers report the current loss rate along with NACKs without a suppression mechanism which can result in a NACK implosion at the sender. Furthermore, the protocol suffers from *the loss path multiplicity problem* [28].

TRAM

The Tree-based Reliable Multicast protocol [29] forms a dynamic tree structure as a basis for reliable one-to-many multicast transmissions. TRAM performs rate-based traffic shaping to allow for smooth changes of the sending rate between a pre-configured minimum and maximum rate.

Congestion reports to the sender are generated either by so-called repair heads, the inner nodes of the tree, when their data cache occupancy exceeds a certain limit or by receivers when they experience packet loss while receiving a certain number of packets (a so-called ACK window). Congestion reports for the same ACK window are aggregated on the way to the sender at the repair heads. Upon receiving a congestion report, the sender reduces the current rate by half. In the absence of congestion reports, the sending rate is increased by a fraction of the difference of the current rate and the highest rate previously reached.

The fairness of the congestion control mechanism clearly depends on parameters such as minimum and maximum rate and the size of the ACK window. Using preset values may be inappropriate for some network conditions. The authors give no clear evidence as to why their increase mechanism based an a previously achieved rate is safe to deploy. Like the previous congestion control scheme TRAM suffers from the loss path multiplicity problem.

TEAR

TCP Emulation At Receivers [30] is a hybrid protocol that combines aspects of window-based and rate based congestion control. TEAR receivers calculate a fair receive rate which is sent back to the sender, who then adjusts the sending rate. To this end, the receivers maintain a congestion window that is modified similarly to TCP's congestion window. Since TCP's congestion window is located at the sender, a TEAR receiver has to try to determine from the arriving packets when TCP would increase or decrease the congestion window size. Additive increase and window reductions caused by triple duplicate ACKs are easy to emulate. However, due to the lack of acknowledgments, timeout events cannot be estimated accurately.

In contrast to TCP, the TEAR protocol does not directly use the congestion window to determine the amount of data to send but calculates the corresponding TCP sending rate. This rate is roughly a congestion window worth of data per round trip time. To avoid TCP's sawtooth-like rate shape, TEAR averages the rate over an epoch, which is defined as the time between consecutive rate reduction events.

To prevent further unnecessary rate changes caused by noise in the loss patterns, a smooth rate is determined by using a weighted average over a certain number of epochs for the final rate. This value is then reported to the sender, which adjusts the sending rate accordingly. Since the rate is determined at the receivers and TEAR refrains from acknowledging packets, it can be used for multicast as well as for unicast communication, provided a scalable scheme to determine the round-trip time and to report the rates used in the multicast case. For multicast congestion control, the TEAR sender has to adapt the rate to the minimum of the rates reported by the receivers.

Due to the close modeling of TCP's short-term behavior, TEAR shows TCP-friendly behavior while avoiding TCP's frequent rate changes.

TFMCC

TFMCC [13] is an equation-based multicast congestion control mechanism that extends the TCP-friendly TFRC [11] protocol from the unicast to the multicast domain. TFMCC uses the same control equation [26] as Xpand does. Each receiver measures the loss event rate, estimates the RTT to the sender, and uses the control equation to calculate an acceptable sending rate from the sender to itself.

In order to eliminate a large number of unnecessary messages, *current limiting receiver (CLR)* is selected per group. CLR is the receiver that the sender believes currently has the lowest expected throughput of the group. The CLR is permitted to send immediate feedback without any form of suppression, so the sender can use the CLR's feedback to increase the transmission rate. The CLR will change if another receiver sends feedback indicating that a lower transmission rate is required. It will also change if the CLR leaves the group.

To solve the feedback implosion problem, TFMCC uses mechanisms based on randomized timers. Time is divided into feedback rounds. At the start of each feedback round, each receiver sets a randomized timer. If the receiver hears feedback from another receiver that makes it unnecessary for it to send its own feedback, it cancels its timer. Otherwise when the timer expires, a feedback message is sent. TFMCC uses such a mechanism based on exponentially weighted random timers. Timers are biased in favor of low-rate receivers. When the feedback timer expires, the receiver unicasts its current calculated sending rate to the sender. If this rate is lower than previous feedback received, the sender echoes the feedback to all receivers.

When a receiver sees echoed feedback from another receiver, it decides whether or not to cancel its feedback timer. The decision may rely on the feedback timer bias (timer is canceled on receipt of first feedback for this round). Another possibility is to cancel the timer only if the echoed feedback indicates a rate lower than the receiver wanted to report. TFMCC uses a slows tart mechanism to more quickly approach its fair bandwidth share at the start of a session.

During slows tart, the sending rate increases exponentially, whereas normal congestion control allows only a linear increase. An exponential increase can easily lead to heavy congestion. A simple measure is to limit the increase to a certain multiple d of the minimum rate received by any of the receivers. Since a receiver can never receive at a rate higher than its link bandwidth, this effectively limits the overshoot to d times that bandwidth. Slow start is terminated as soon as any one of the receivers experiences its first packet loss.

2.3.2 Window-Based Approaches

The domain of window-based unicast congestion control is well covered by TCP. There are two main problems that have to be solved in order to use window-based congestion control for multicast. First, protocols should prevent drop-to-zero of the rate due to the aforementioned loss path multiplicity problem. The second problem is how to free slots in the congestion window. Clearly it is not possible for the sender to receive acknowledgments for each packet from each receiver, as this would cause an acknowledgment implosion.

In the following we will present several window-based congestion control approaches for multicast transmission. In particular we will focus on how the two main problems are solved for each.

RLA and LPR

The Random Listening Algorithm [31] extends TCP SACK by introducing some enhancements for multicast. For each receiver, the multicast sender stores the smoothed round-trip time and the measured congestion probability. A loss is detected by the sender via identification of discontinuous acknowledgments or via timeout. Based on these loss indications, the number of receivers n with a high congestion probability is tracked.

If congestion is detected, the window is halved in the following two cases: (1) if the previous window cut was made too long ago (the authors propose an interval of twice the moving average of the window size times the smoothed round-trip time of the corresponding receiver) or (2) if a generated uniform random number r is less than or equal to $1/n$.

When a packet has been acknowledged by all receivers, the congestion window *cwnd* is incremented by $1/cwnd$, identical to TCP. A TCP-like retransmission scheme with fast-recovery is also included in RLA.

With the above mechanisms, RLA avoids the loss path multiplicity problem, while achieving statistical long-term TCP fairness.

Linear Proportional Response [32] is a probabilistic loss indication filtering scheme that is an improvement over the corresponding RLA mechanism. The probability with which a multicast source reduces its congestion window size is proportional to the loss probability at the receiver; i.e., the window size is halved when r is greater than $X_i / \sum_{i=1}^{n} X_i$ where X_i is the number of losses at receiver i.

The LPR scheme achieves better fairness of multicast sessions towards competing unicast sessions than does the window adjustment indication scheme of RLA. Even though we are not aware of any extensive measurements in real scenarios, the mathematical proofs and the simulations in [32] give high evidence that when combined with the window adjustment mechanism of RLA, LPR achieves good TCP-friendliness.

MTCP

MTCP [33] is a reliable one-to-many multicast protocol that uses window-based end-to-end congestion control to achieve TCP-friendliness. The protocol is based on multi-level logical tree, where the root is the sender, and the other nodes in the tree are receivers. The sender multicasts data to all receivers, and the latter send acknowledgments to their parents in the tree. A parent in the logical tree structure stores a received packet until receipt is acknowledged by all of its children. Upon receiving a packet, a child (which may be a parent for other participants) transmits an acknowledgment to its parent using unicast.

MTCP employs TCP-like congestion window. Each tree node estimates the minimum bandwidth available in the multicast routes from the sender to its children by maintaining a TCP-like *cwnd*. A node maintains its *cwnd* using TCP-Vegas [34] congestion control mechanisms such as *slow start* and *congestion avoidance*, based on original RTT estimation scheme. In order to estimate RTT, parent node periodically polls its children by sending a probe packet.

The only differences with TCP-Vegas are (1) the *cwnd* is incremented when a parent node receives ACKs for a packet from all of its children, and (2) receiver send NACKs for missing packets and parent node immediately retransmits the packets reported missing.

The sender regulates its rate based on the congestion status of all receivers on the tree. Information about congestion status of each receiver (i.e size of its *cwnd*) propagates along the paths of the logical tree from the leaves to the root (the sender).

In MTCP, the loss path multiplicity problem [28] is avoided by means of the aggregation at the intermediate nodes. Each node forwards the information about the bottleneck link of its children to its parent. Therefore the sender will receive information about the overall bottleneck link rather than about uncorrelated packet loss.

The main drawback to MTCP is its complexity and required setup of a tree structure where each node has to perform package storage, repair, and congestion monitoring functionality.

NCA and PGMCC

Nominee-based Congestion Avoidance (NCA) presented in [35] and Pragmatic General Multicast Congestion Control (PGMCC) [36] are two approaches to congestion control that share the same fundamental idea: they select as a group representative the bottleneck receiver with the worst network connection. This receiver acknowledges every packet received and thereby allows the sender to use a TCP-style congestion control algorithm. It is important to note that in this approach congestion control and packet repair are treated independently of each other. Thus, the approach can be used in combination with a large number of mechanisms that establish reliability, as well as for unreliable data transmission.

The most challenging aspect of NCA and PGMCC is how to select the group representative. Similarly to Xpand, in both approaches, each receiver calculates the data rate at which it is able to receive by using the TCP rate formula [26]. The information about the acceptable rate is conveyed back to the sender either piggy-backed on NACKs (PGMCC) or accumulated in a tree structure of routers that always forward the report of the participant with the lowest acceptable data rate (NCA). From those reports the sender selects as the representative the participant with the lowest acceptable rate and uses a TCP-like congestion control mechanism to this participant.

This approach seems very promising, since it closely mimics the behavior of unicast TCP and therefore should lead to fairness with regard to TCP flows if the proper representative is chosen.

The main problem is that the selection process is based on a rough estimate of the acceptable data rate. Further insight is needed whether network conditions exist where the wrong representative is selected. This could lead to unfair behavior against other flows. The author of PGMCC indicates that this may occur when a set of receivers has lossy links with a low round-trip time and congested links with a high round-trip time.

RMTP

RMTP [37] is a reliable one-to-many multicast protocol with end-to-end congestion control. A multicast tree, rooted at the sender and spanning all the receivers, is set up at the network layer. The key idea in RMTP is to group receivers into local regions and to use *Designated Receiver* as a representative of the local region.

Designated Receiver is responsible for sending ACKs and NACKs periodically to the sender, for processing ACKs from receivers in its domain, and for retransmitting packets within its domain.

RMTP has been designed basically for two-tier system. However, to enhance scalability, the protocol supports further subdivision of local regions, which creates a multi-level hierarchy.

The protocol employs window-based scheme for congestion avoidance. RMTP uses retransmission requests (NACKs) from receivers as an indication of possible network congestion. NACKs propagate along the tree of representatives from leaves to the root (the sender). If during fixed-length time period number of NACKs exceeds a certain threshold, congestion window is set to 1. Otherwise, the sender increases congestion window by 1.

RMTP was not designed to be TCP-friendly and no TCP-friendliness tests have been conducted.

TMTP

TMTP [38] is a reliable one-to-many multicast protocol. TMTP organizes a group participants into hierarchy of subnets or domains. All the group members in the same subnet belong to a domain and a single domain manager acts as a representative on behalf of the domain for that particular group. The domain manager is responsible for recovering from errors and handling local retransmissions.

The domain managers are organized into a control tree rooted at the sender. Each node has at most K other domain managers as children, resulting in a tree with maximum degree K. The value of K is chosen at the time of group creation and limits the processing load on the nodes of the control tree.

The tree is used for sending ACKs/NACKs from a child to its parent. The transmission rate is controlled using a sliding window based protocol. The fixed size window consists of transmitted by not yet acknowledged messages. A domain manager holds packets in its buffer until all of its children have acknowledged them. If the children fail to acknowledge packets, the domain manager's window will not advance and its buffers will eventually fill up. As a result, the domain manager will drop and not acknowledge any new data from the sender, thereby causing back pressure to propagate up the tree which ultimately slows the flow of data.

TMTP was not designed to be TCP-friendly and no TCP-friendliness tests have been conducted.

Spread

Spread [8] and Xpand [10] are the only two operational WAN oriented GCSs today, which makes it especially interesting to compare with each other.

Spread builds a global overlay network. An edge in the overlay graph is considered as two links, one in each direction of the edge. Standard AIMD ([39]) congestion control is deployed on each virtual link, separately for each direction. Xpand, on the contrary, relies on existent subnet structure, and employs end-to-end congestion control mechanism described in Chapter 5.

As regards to flow control, Spread exploits the global cost-benefit scheme ([40]). Every virtual router (node in an overlay network) advertises periodically the costs of its outgoing links, more often when the links are congested, less often when they are not. The cost of a link reflects the congestion level of that link, computed through some exponential formula based on the link's buffer utilization. Cost updates are forwarded through a link-state mechanism, reliably on each link, until they reach all the other nodes in the graph.

Each node regulates the sending rate of its own clients (applications) based on the current information about the costs of the links that the client uses in its multicast. It is not the congested router that slows down the throughput, but rather the initial node, responsible for the sending client.

Xpand uses window-based flow control in general, and AIMD scheme for a special case when all group members reside on the same LAN.

The window-based scheme makes use of fixed length buffer (the window). A message being sent is stored in the buffer until all receivers acknowledge its receipt. If a slow receiver defers acknowledgments, sender's window fills up with unacknowledged messages. When the window becomes full, the sender stops transmission altogether until the 'oldest' message is acknowledged by all receivers.

Intra-LAN flow control in Xpand is described in details in Chapter 6.

Chapter 3

Environment

The Xpand group communication system is a middleware for distributed many-to-many applications and is fully described in [10, 41]. This work presents only the essential features of the system and the relevant assumptions.

The processes participating in Xpand are grouped into clusters so that all members of each cluster belong to the same LAN. The clusters are spread over a WAN.

Xpand distinguishes between two types of processes: *regular* and *delegate*. An application instance linked to an Xpand library is a regular process. In each LAN, a delegate is a designated daemon that serves as a representative of its LAN to all the other Xpand clusters and may be spawned by the first application instance in the cluster.

Among other benefits, this hierarchical design allows for better scalability of the congestion control mechanism. The relationships between Xpand's components and its environment are shown in fig. 3.1. The congestion control mechanism is integrated within Xpand's core module.

3.1 Application Layer Multicast

As it was noted before, Xpand reliable multicast service relies on the availability of a many-to-many communication substrate. The network service that supports many-to-many communication in IP networks is IP multicast. There is a limitation on using IP multicast since it is only partially deployed in various networks, which creates isolated regions supporting IP multicast.

In order to overcome this difficulty and to "bridge" regions supporting IP multicast, Xpand introduces a layer called *Application Layer Multicast* (ALM), which is used only as an interim solution. This layer constructs a message distribution tree incorporating delegate nodes, each node being chosen from a region that supports a native multicast. The tree construction is performed automatically and is beyond the scope of this work (See [41, 10]).

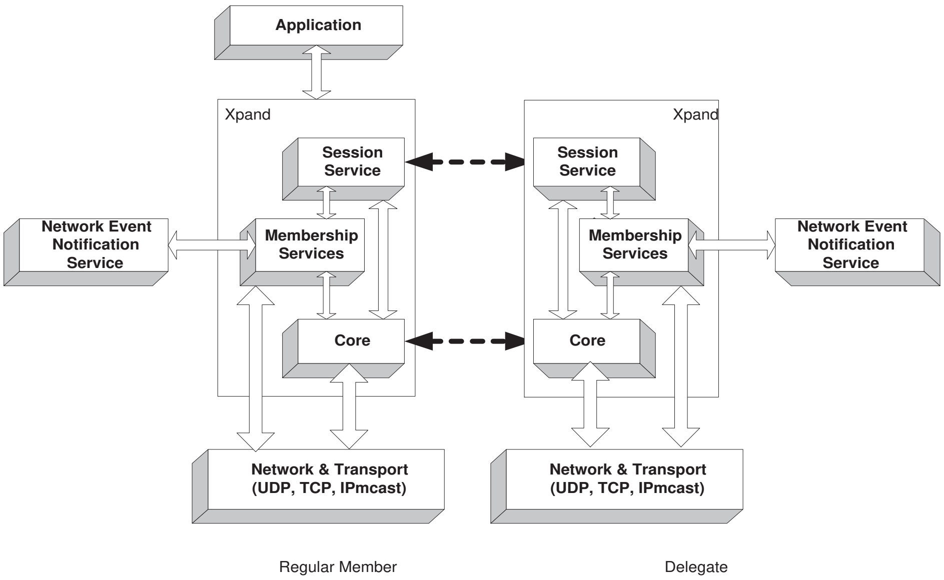


Figure 3.1: Xpand's layer structure

3.2 RON Testbed Environment

In our wide area network experiments we used the Netbed's RON ([14]) wide-area nodes. The tests involved 6 sites, 5 located in North America and one in Jerusalem, Israel. The nodes themselves are Celeron/733 or Pentium III/1.1G machines running FreeBSD 4.7 / Linux 2.4 operating systems, connected via commodity Internet and Internet2. The links have diverse bandwidth, delays, and packet loss rates (see Section 10). Since there is no IP multicast among RON machines, Xpand builds an overlay network (Application Layer Multicast (ALM)) over the nodes. With ALM, messages are delivered by unicast among the nodes. The ALM tree created for our specific set of RON nodes is shown in fig. 3.2.

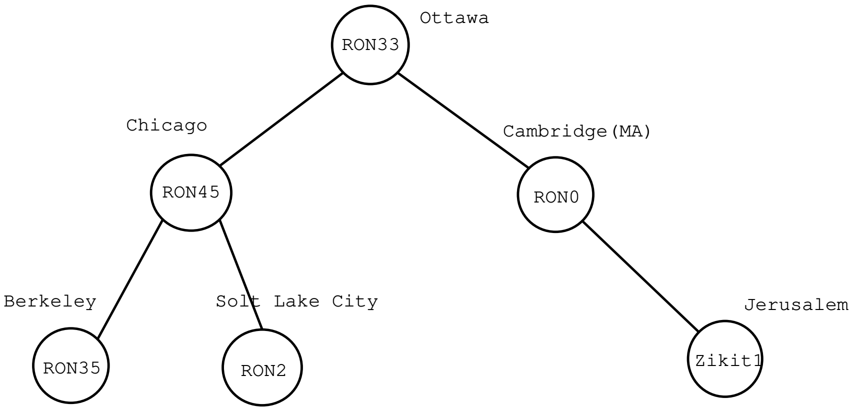


Figure 3.2: ALM layout

Chapter 4

Design of Congestion Control Mechanism in Xpand

The design of the congestion and flow control mechanisms takes advantage of Xpand's hierarchical structure. In Xpand, both senders and receivers from each LAN are represented by the LAN's delegate. We designed many-to-many single rate mechanism in which senders' delegates cooperate with receivers' delegates in order to make the message flow to be TCP-friendly.

To introduce the congestion control mechanism into Xpand, some parameters characterizing the network are measured by the delegates and used to adjust the senders' transmission rate in accordance with the network conditions. In [11], a message flow is defined to be TCP-friendly if under the same conditions its sending rate is within a factor of two of the sending rate of TCP flow. To achieve that, Xpand must be able to estimate the appropriate TCP rate. To reach that goal, we use Eq. 4.1 from [12] characterizing TCP steady state throughput as a function of packet loss rate, estimated round trip time and TCP retransmission timeout:

$$B(p, RTT, RTO) \approx \frac{s}{RTT\sqrt{\frac{2bp}{3}} + RTO \cdot \min\left(1, 3\sqrt{\frac{3bp}{8}}\right) \cdot p\left(1 + 32p^2\right)} \,, \tag{4.1}$$

where

- B is the desired transmission rate in bits/sec
- s is the average packet size in bits
- RTT is the estimated round trip time in seconds
- $p \in [0, 1]$ is the loss event rate
- $RTO = 4 * RTT$ is the TCP retransmission timeout value in seconds recommended in [11] and $b = 2$ is the number of packets acknowledged by a single TCP acknowledgement

In order to use this equation, we designed a RTT estimation scheme and a mechanism for loss rate measurement. These parameters are to be continuously evaluated and translated into an appropriate transmission rate by each receiving delegate. *Appropriate* means here a rate that TCP would achieve according to the equation, given the values of RTT and loss rate.

However, our model calls for many sending processes and several sending delegates. For a given multicast group G, a delegate generally represents a set S_G of all local G senders. The congestion control mechanism provides the delegate with aggregate transmission rate for S_G. Further, the sender delegate has to fairly distribute the aggregate rate among particular senders. To do this, the delegate should know transmission needs of each local sender. Thus, each local sender must continuously estimate its application transmission rate and deliver the estimation to the delegate.

4.1 Design Decisions

Xpand design distributes the load of calculating the congestion parameters between senders' delegate and receivers' delegates. Each calculation is performed at the optimum location, as detailed below.

- **RTT Estimation** is performed by the sender's delegate for each (*remote delegate, group*) pair. The RTT estimation does not require synchronized clocks.
- **Loss Rate Estimation**. The best place to measure the loss rate is at the receivers' delegates, since the latter have accurate information on message losses. The loss rate is estimated for a (*receiver delegate, sender delegate, group*) tuple.
  Furthermore, since in most cases there are many more receivers than senders, it is reasonable to distribute the loss rate calculations among the receivers. The hierarchical structure requires the calculation to be done at the receivers' delegates.

Loss rate calculations require RTT estimation, which is done by senders' delegate and is passed to receivers' delegates. As a result, a receiver's delegate gets RTT calculated half RTT ago, which may slow down the responsiveness of the scheme. This inaccuracy is smoothed by using weighted RTT average.

We assume that the RTT and the calculated loss rate are uniform for all receivers belonging to the same group within the same LAN. This is a safe assumption since message loss and RTT in LAN are negligible compared to those in WAN.

- **Local senders detection**. The senders' delegate has to identify the potential senders in its LAN and to obtain estimation of their sending rate demands.
- **Local sender's rate allocation**. The senders' delegate distributes the aggregate transmission rate among set of all its local senders. We use the "Max-Min fairness" ([42]) criteria to allocate the rates.
- **Local sender's traffic shaping**. We use a token-bucket-like mechanism in order to enable the sender to conform to the assigned rate.
- **"Slow start" mechanism**. We incorporate a slow start mechanism into the local sender's rate limit mechanism so that it would be TCP-friendly even prior to a loss event.

- **Single LAN group**. The congestion control mechanism should cover a special case when all group members reside on the same LAN.
- **Issues with Multicast Delivery Tree**. Different multicast routing protocols use different ways to build a multicast distribution tree in the Internet environment. We assume that different multicast trees are built for different groups, thus RTTs and loss rates should be measured separately for each group.

Chapter 5

The Congestion Control Mechanism Description

We start by giving the general algorithm flow and then describe each step in details. Our mechanism is designed for many-to-many communications, while the algorithm is presented from a single source LAN perspective and for a group[1] (G). The source LAN delegate (SD) communicates with the delegates (RD) of LANs that have receivers for the given group. Every delegate implements the same mechanism and thus imposes congestion control over all senders and receivers. At the same time, Xpand message flows are friendly to TCP flows that use the same links.

The basic steps of the algorithm follow:

1. RTTs are computed at the senders' delegate per receivers' delegate and per multicast group and afterward are passed to the receivers' delegates;
2. Loss rate is measured at every receivers' delegate per sender delegate and group: $\{SD,\ G\}$. In these measurements, the receivers' delegate uses the RTT estimation obtained from SD;
3. The maximum acceptable transmission rate is calculated by RDs from the equation using the measured loss rate and delivered RTT. RD sends the rate value to the corresponding SD;
4. SD obtains the acceptable (restricting) rates reported by all remote delegates and chooses the minimal value to be the aggregate transmission rate for the senders belonging to the group within its LAN;
5. SD adjusts the local senders' transmission rate according to the aggregate transmission rate.

[1] congestion control among multiple groups is achieved implicitly.

The RTT Calculation:

A sender multicasts packet P to the entire group G;

Upon receiving P, SD records P's arrival time T_{out};
/* T_{out} approximates the sending time */

Upon receiving P, RD records P's arrival time T_{arr};

RD sends ACK for P to SD at time T_{ack} containing $\Delta = T_{ack} - T_{arr}$;

Upon receiving the ACK for P at time T_{now}, SD calculates $RTT = (T_{now} - T_{out}) - \Delta$;

Figure 5.1: RTT estimation at the sender's delegate

5.1 RTT Estimation by Senders' Delegate

For a given group G, RTT is measured by SD for every LAN that contains members of G. Intra-LAN delays are negligible in a WAN environment. We assume that all the messages flowing from the sending LAN to specific receiving LAN (targeted for the same group) traverse the same route. We also assume that the control traffic from RD to SD takes the same route, which may be different from SD to RD route. As a consequence, all G senders within SD LAN have essentially the same RTT to RD. Thus, for each group we actually measure RTTs from SD to RD. The algorithm is straightforward, as can be seen in Fig. 5.1.

Note: RTT is *not* updated on *retransmitted* messages, since the original and the retransmitted messages have the same sequence number.

This scheme allows RTT variations to change the transmission rate within approximately one round trip time, which is important for the scheme responsiveness.

5.2 Receiver Side Loss Rate Estimation

Loss rate is measured by a receivers' delegate (RD). Regular receivers do not participate in loss measurements. RD may receive multiple flows from the same sending LAN for a specific group. All such flows are aggregated and considered by RD as a single meta-flow to improve the statistics of message loss rate and to address scalability issues.

A packet loss in our implementation is detected when the sequence number of a received message within a specific flow is out of the order. Since we assume packets (sent from particular LAN, for specific group) to traverse the same multicast (unicast) delivery path, packet reordering is infrequent and has no significant influence on the overall picture.

The model in [12] requires that two loss events be separated by at least RTT seconds to be statistically independent. A loss event starts on a packet loss and continues RTT seconds after it. To determine whether a lost packet should start a new loss event or be considered as a part of a current loss event, we compare the estimated arrival time of a lost packet with the time associated with the beginning of the last lost event[2]. To approximate the arrival time of a lost packet, we

[2]In TFRC ([43, 11]) the term "inter-loss interval" is used, defined as the interval from the beginning of one loss event until the beginning of the next loss event.

use the suggested interpolation to infer its nominal "arrival time" (see [43, 11] for details). In the calculation that follows we ignore packets that arrive or are lost within RTT seconds of the time associated to the lost packet that started the loss event.

As opposed to the TFRC requirement, in Xpand retransmitted and original packets have the same sequence number. Such an approach does not take into account loss of retransmitted packets, thus underestimating the actual loss.
However, if loss rate without retransmissions is, say, at most 10%, than it would be reasonable to assume that among retransmitted packets loss rate is about the same. Thus, loss rate including loss of retransmitted packets should be 10% + 1%, which is well within a statistical error.

In order to compute the loss rate, we need to count the number of packets contiguously received between two consecutive loss events. If a loss event is determined to have started at time T_1 and the next loss event started at time T_2, then the number of packets between the loss events is the total number of packets in all n flows that have arrived within period $[T_1 + RTT,\ T_2)$.

The Xpand approach matches the model presented in [12], whereas the original specification ([43, 11]) also counts packets that arrive within the first RTT seconds of the loss event, thus underestimating the actual loss rate.

To calculate the loss rate p, we first calculate I_{mean}, i.e. the average number of packets received between two consecutive loss events. This is done by using moving average of the past average I_{past} and the most recent estimation (with a quotient 0.5).

$$I_{\text{mean}} = \frac{1}{2} I_{\text{past}} + \frac{1}{2} I_{\text{sample}} \ , \tag{5.1}$$

where I_{sample} is the number of packets that have arrived since the end of the last loss event. Immediately after a loss event, the value of I_{sample} is smaller than I_{past} which results in an inaccurate value of I_{mean}. To deal with this undesirable effect we do not update I_{mean} until either a new loss event occurs, or $I_{\text{sample}} \geq I_{\text{past}}$.

At the beginning of a new loss event I_{mean} becomes I_{past}. Equation 5.1 reflects the average once a new loss event takes place.

The quotient is chosen to provide responsiveness, on the one hand, and to filter out samples that are too far from average value due to inaccuracy of measurements, on the other. One of the original goals of [43, 11] was to achieve congestion control that is TCP-friendly, while allowing a more smooth traffic rate (than TCP's). This is an important property for various applications, such as multimedia, etc. Since for group communications this smoothness is not an important property, we used a simpler equation (Eq. 5.1) that produces a TCP-friendly rate change response, though not that smooth.

Finally, RD computes the loss rate as $p = 1/I_{\text{mean}}$. Given values of p and RTT, RD calculates corresponding transmission rate (Eq. 4.1) and sends it to SD.

SD computes updated aggregate rate by taking the minimum over all acceptable transmission rates from relevant RDs. It then distributes this rate among its local senders according to their needs and notifies them about their new sending rate[3] (See Section 5.3). Each local sender uses a token-bucket like mechanism in order to shape its traffic to comply with an obtained sending rate.

[3]The two-way communication between SD and its local sender regarding sending rate allocation is done via a reliable channel and is beyond the scope of this work.

5.3 Multiple Senders per LAN

As it was pointed out above, given a source LAN, we determine transmission rate separately for each group, since we assume the possibility of different multicast distribution trees for different groups.

Given an aggregate group transmission rate for a particular group, there is a problem of partitioning the rate among multiple local senders. Due to its key role in a LAN, the only reasonable candidate to distribute transmission rate among local senders is senders' delegate. The delegate will use a "Max-Min fairness" ([42]) criteria to ensure fair rate allocation.

5.3.1 Determining local senders' rate demands

We describe here a scheme that enables SD to learn its local senders' demands for transmission rate. Vector of senders' demands is used as an input to the fair rate distribution algorithm.

The demand evaluation scheme must be responsive enough to avoid under-utilization of the resource (transmission rate), e.g. a case when one sender receives a rate higher than it actually needs, whereas the other one receives a rate lower than it needs.

Application transmission rate may change more frequently than the aggregate rate does, since the latter is updated relatively infrequently and is smoothed, as opposed to common bursty traffic pattern of an individual sending application.

Thus, SD must sample[4] its local senders' demands more frequently than it receives aggregate rate updates. Each time a new vector of senders' demands is collected, the delegate recomputes the sending rate partition.

Each local sender determines its application demand by measuring the rate at which Xpand sending window fills up and gets drained. To avoid unnecessary fluctuations when evaluating a transmission demand, the sender uses weighted "history" of its window's occupancy. To provide fast responsiveness, however, the weight of the "history" should be lower than weight of recent values.

When the window is full, the local sender cannot measure its application's transmission demand, since the application is blocked. In this case, a special approach is taken to estimate transmission rate demand. There is no point requesting a larger rate from SD if the last rate request has not been satisfied yet. Since SD uses the Max-Min fairness criteria, increasing the requested rate will not increase the allocated rate.
If the delegate has granted to the sender requested rate amount and the window is still full, the sender doubles its requested rate demand. This process continues until the window opens up.

When a new sender starts transmitting, it is always permitted to start sending at some predefined initial rate (32 kbit/sec). The delegate will recognize the presence of the new sender and will take it into account by recalculating the sending rate partition.

5.3.2 Taking retransmissions into account

SD is in charge for retransmission of lost packets to remote delegates. In a WAN environment, it is possible for retransmissions to consume considerable percentage of a bandwidth. If SD retransmits

[4] The sender can explicitly ask for transmission demand update from its delegate if its local application demand changes by at least some δ amount.

without taking into account maximal acceptable rate for the LAN, the aggregate outgoing rate will be higher than it should to stay TCP-friendly.
Thus, retransmissions should be taken into account when partitioning LAN's aggregate rate between local senders. For example, when partitioning the aggregate rate among the senders, SD could consider itself as an additional sender, and provide for proper rate share for itself. This rate can be initialized to some default value (5% of aggregate outgoing traffic), and further evaluated and updated dynamically.

5.4 Providing Reliability for Rate Update Messages

Reliable delivery of rate request and update messages between sender and its delegate is important. Suppose aggregate transmission rate for a LAN has dropped significantly. The delegate recalculates new rate distribution and sends rate updates to the senders. Naturally, every sender's new rate is much slower than it was before. If some of the rate update messages get lost, some of the senders will keep transmitting faster than they actually should. Therefore, the LAN aggregate transmission rate is probably much higher than it should be in order to provide TCP-friendliness.

If the aggregate rate increases, and some of the update messages from the delegate get lost, some of the senders will continue to transmit with slower rate, and the available transmission rate is under-utilized. In the opposite direction, if rate request message from a sender to delegate gets lost, either the rate is under-utilized (if the sender's demand decreases), or rate distribution fairness is broken (if the sender's demand increases).

We rely upon Xpand to provide reliability of rate request/update messages as follows:

- **Sender to Delegate Rate Request Messages**. Special multicast group is created for each pair *(local group sender, local delegate)*. Then, a sender uses Xpand's reliable multicast mechanism to safely deliver its rate demand to the delegate.

- **Delegate to Sender Rate Update Messages**. Special multicast group is created for each multicast group which has senders in the local LAN. The members are all local senders to the group and the delegate. Then, delegate uses Xpand to reliably deliver (by multicast) rate distribution to all local senders.

Chapter 6

Intra LAN Flow Control

So far we dealt with congestion control in a WAN environment. Typically, in a WAN it is a network that limits a transmission rate: link bandwidth is scarce, and routers may drop packets. In a LAN, bandwidth is usually plentiful, and it is an ability of a receiving host to handle incoming traffic that becomes a performance bottleneck.

Thus, we are to introduce a flow control scheme which limits fast sender from swamping slow receivers with data when both the sender and the receivers reside on the same LAN.

Originally, Xpand used window-based flow control. The window-based scheme makes use of fixed length send buffer (the window). A message being sent is stored in the buffer until all receivers acknowledge its receipt. If a slow receiver defers acknowledgments, sender's window fills up with unacknowledged messages. When the window becomes full, the sender stops transmission altogether until the 'oldest' message is acknowledged by all receivers.

Intra-LAN performance evaluation tests showed satisfactory, but not very good results for this scheme. When approaching the physical limits of a network, transmission rate becomes unstable.

Our equation-based scheme for regulation of transmission rate cannot be used in a LAN environment: RTT is small, loss rate is far too low, whereas the measurement mechanism is coarse, thus introducing large relative error.

A flow control scheme is independent of a congestion control scheme. Flow control gives rate limit L_F based on capabilities of receivers, whereas congestion control rate limit L_C is based on current network state. Actual transmission rate for a sender should be as shown below:

$$\min\{L_F, L_C\}.$$

6.1 NACK Driven TCP-like AIMD Scheme

To control sending rate in a local network environment, we use the AIMD [39] approach. It is a generalization of the increase-by-one decrease-to-half strategy used in TCP for congestion window adjustment. With AIMD, increase value and decrease ratio are parameters. That is, in the congestion avoidance state, the window size is increased by α per window of packets acknowledged and it is decreased to β of the current value when there is congestion indication.

In Xpand, however, we increase and decrease transmission rate (L_F) itself rather than transmission window, as it is done in the original AIMD scheme.

A delegate treats $NACK$s from **local receivers** for messages sent by **local senders** (*intra-LAN NACKs*) in a special way. When sender and receiver reside on different LANs, $NACK$ from the receiver (*inter-LAN NACK*) usually indicates a congestion in a WAN. In contrast, *intra-LAN NACK* hints at the receiver's shortage of buffer space rather than congestion in the local network.

As in the TCP congestion control algorithm [19], a sender has the *slow start* and the *congestion avoidance* phases.

In the slow start phase, transmission rate is doubled every time period Δ if no packet loss occurred in the LAN (loss is indicated by $NACK$ arrival).

When in the congestion avoidance phase, the sender increases its transmission rate linearly every Δ.

Every time a sender's delegate receives *intra-LAN NACK*, transmission rate is halved. *Inter-LAN NACKs* do not trigger this mechanism. Remote $NACK$s are due to an internetwork congestion and are dealt with by the congestion control scheme (loss rate measurements and acceptable transmission rate calculation).

6.2 The Algorithm Details

Let's consider delegate D which has local senders for group G.

- On the one hand, remote delegates determine for D transmission rate $R_G(C)$, due to congestion control in WAN.

- On the other hand, D's local senders to G regulate transmission rate $R_G(F)$, due to flow control in the LAN.

Resulting transmission rate for the group G is $R_G = \min\{R_G(C), R_G(F)\}$. D partitions R_G among all its local senders according to their needs. We described in the previous sections how $R_G(C)$ is obtained. Now, we shall consider estimation of $R_G(F)$.

- Flow control mechanism works in cycles of constant duration Δ, which are scheduled by D. Δ should be close to RTT in a LAN ($0.2...0.5ms$) for the best responsiveness. However, we must avoid multiple rate decreases in response to arrival of several back-to-back intra-LAN NACKs referring to the same loss event. Thus, Δ should be long enough to cover most of 'related' NACKs. Taking into account these considerations and due to coarse granularity of computer clock we use $\Delta = 10ms$.

- At the beginning of a cycle D has aggregate rate $R_G = \min\{R_G(C), R_G(F)\}$, and this rate has been partitioned between all local senders to G.

- Flow control can be in one of two phases: *slow start* or **c***ongestion avoidance*. When in the *slow start*, a rate is doubled every Δ if no loss was observed during this period. In the *congestion avoidance* rate is increased linearly every Δ if no loss occurred during this period. In both phases, if loss is detected (arrival of *intra-LAN NACK*), the rate is halved, and phase is set to *congestion avoidance*.

- The *slow start* is executed only if no remote members exist. With the *slow start* the rate can increase significantly before we obtain first feedback from remote receivers, causing congestion or making it more severe.
- The *slow start* phase is preempted by the *congestion avoidance* when either
  - *intra-LAN NACK* arrives. Then, $R_G(F)$ is halved
  - $R_G(C)$ becomes less than $R_G(F)$, making it pointless to further increase $R_G(F)$.
  - Remote member joins the group
- The *congestion avoidance* is preempted by the *slow start* phase when either
  - last remote member leaves the group
  - there are no remote members, and all local senders are silent during long enough time. More precisely, when *all* local G senders are silent for at least *Max Inactivity Period,* $R_G(F)$ is de-initialized to $minR_F$. The motivation behind this decision is that during a long period of inactivity, network conditions can change considerably, and thus fast rate increase is unsafe.
- Let T be sum of all local G senders rate demands. If during the cycle D does not receive *intra-LAN NACKs* for G,
  - $if\,((R_G(F) \leq R_G(C))\,AND\,(R_G(F) < T))$
    - $if\,(in\,slow\,start)$
      - · $then\ \ R_G(F) = 2 * R_G$
      - · $else\ \ R_G(F) = R_G + minR_F$, where $minR_F$ is a minimal aggregate rate for a group within a LAN
  - $else\,if\,(R_G(F) > R_G(C))$
    - do *congestion avoidance* algorithm

We don't want to cancel the slow start due to $R_G(F) > T$ only. The motivation here is to provide high responsiveness in a case when the value of T increases suddenly and becomes larger than $R_G(F)$.

- If during the cycle D receives an *intra-LAN NACK* to G,
  - $R_G(F) = \frac{R_G}{2}$
  - do *congestion avoidance* algorithm
- Every time remote delegate X leaves G, D is notified by *CONGRESS* membership service. D then removes X's rate advertisement $R_G(C, X)$, since there is no more G members in X's local LAN.
- Rate advertisement $R_G(C, X)$ from remote delegate X can get lost due to congestion in a WAN. To recognize this situation, $R_G(C, X)$ has an expiration time. If no updates arrive from X during this period, we assume that severe congestion has been created in a network, and set $R_G(C)$ to $minR_F$.

- If there is **no more** remote G members, $R_G(C)$ is set to infinity and D enters the *slow start* phase for G. This enables local senders to promptly achieve maximal transmission rate (which can be high in a LAN):
  - $R_G = R_G(F)$
  - $R_G(C) = \infty$
  - do *slow start* algorithm
- Every time **first** remote member joins the group
  - $R_G(C) = minR_F$
  - do *congestion avoidance*
- Every time value of $R_G = \min\{R_G(C), R_G(F)\}$ changes, rate distribution protocol should be run.
- **Initialization**. Let's consider the standard situation when a regular member S joins G (becoming the first G member in the LAN) and starts sending. Since S is the first G member in the LAN, D has neither $R_G(C)$ nor $R_G(F)$ estimations yet. Until $R_G(C)$ is obtained from one of the remote delegates, D assumes
  - if (there exist remote G members)
    - $R_G(C) = minR_F$
    - do *congestion avoidance*
  - else
    - $R_G(C) = \infty$
    - do *slow start*
  - $R_G(F)$ is initialized to $minR_F$, a minimal aggregate rate for a group within a LAN, which is also acceptable for a WAN (say, $32kbit/sec$).

Chapter 7

Wide Area Performance Results

The congestion control was tested in multi-continent (WAN) setting of RON testbed Network. The tests included six sites: Zikit1 (Jerusalem), RON0 (Boston), RON2 (Salt Lake City), RON33 (Ottawa), RON35 (Berkeley), RON45 (Chicago). All measurements were carried out within the same time-window every day. The time-window was chosen to be very early morning time in the North America, which is midday in Israel.

Since there is no IP multicast connectivity among these sites, a proprietary application layer multicast (ALM) mechanism was used ([10]). The obtained message distribution tree is presented in Fig. 3.2. RON33 was selected to be the tree root by the algorithm used in the dynamic ALM tree construction.

In Tables 7.1, 7.2, 7.3 columns represent senders and rows represent receivers. These numbers were collected during the first test (see below).

Table 7.1 presents the median round trip time (RTT) among the sites as was measured by the senders. The messages from the sender to the receiver were sent along the ALM tree, and the ACKs were sent via unicast back to the sender.

Table 7.2 presents the median loss ratio as calculated by the receivers. One can notice that loss ratio increases along the ALM paths as the number of ALM overlay links increases.

Table 7.1: Round Trip Time, ms

Site	Zikit1	RON0	RON2	RON33	RON35	RON45
Zikit1		170.92	282.15	205.25	277.99	213.99
RON0	170.62		120.18	51.67	122.20	62.82
RON2	260.60	91.95		57.65	77.15	57.87
RON33	236.30	50.00	85.70		85.59	26.68
RON35	299.13	121.32	96.58	76.23		59.80
RON45	242.47	62.29	58.75	25.27	59.64	

Table 7.2: Loss rate (%)

Site	Zikit1	RON0	RON2	RON33	RON35	RON45
Zikit1	0.00	0.00	2.90	0.00	7.63	2.33
RON0	0.33	0.00	3.42	0.00	7.40	2.47
RON2	4.20	2.87	0.00	3.39	6.35	1.96
RON33	0.33	0.00	3.31	0.00	7.84	2.67
RON35	4.80	3.26	3.34	4.36	0.00	1.50
RON45	1.15	0.82	0.71	0.63	4.85	0.00

Table 7.3: TCP-friendly rate (Mb/s)

Site	Zikit1	RON0	RON2	RON33	RON35	RON45
Zikit1			0.16		0.07	0.24
RON0	0.96		0.33		0.16	0.80
RON2	0.13	0.49		0.69	0.30	1.02
RON33	0.69		0.48		0.22	1.79
RON35	0.10	0.34	0.42	0.43		1.17
RON45	0.34	1.60	1.84	4.58	0.50	

Table 7.3 presents the calculated (theoretical) rate using the Eq. 4.1. There are some blank cells in the table, since no message was lost on the corresponding links, and the reason is that the rate was limited by another receiver.

In the following sections we present performance measurements of the congestion control mechanism within Xpand system. The measurements validate the congestion control mechanism, its TCP-friendliness and its applicability to multicast in WAN.

7.1 Rate Restriction

In the first test, we let each sender in turn send messages for 60 seconds with an interval of five seconds between senders. Figure 7.1 presents the received rate at Zikit1 for each sender. In order to evaluate the congestion control mechanism we examined the performance of Zikit1. The variation in receiving rate is caused by the changes in rate forced by the slowest receiver.

Figure 7.2(a) shows the slowest receiver as perceived by Zikit1. It is clear that the sending rate is limited by the rate reported by the slowest receiver. We present only the two slowest receivers that affect the sending rate, since other receivers have reported a much higher receiving rate.

Figure 7.2(b) presents the loss rate and the reported rate calculated by RON35 regarding Zikit1 messages over the same time period. One can see that the dominating factor in the rate reported

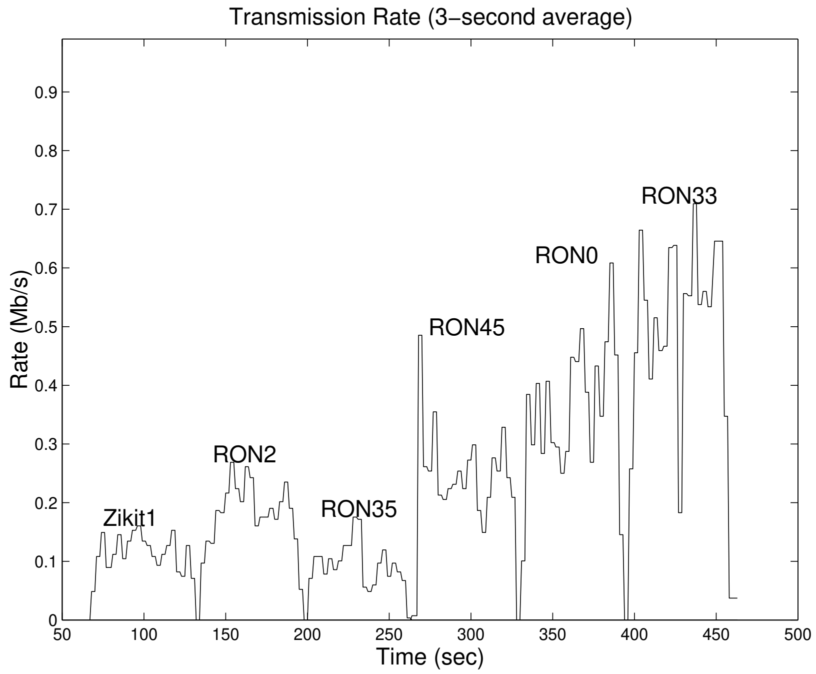


Figure 7.1: Sequential sending

by the receiver is the loss rate, since the RTT remains relatively stable during that period.

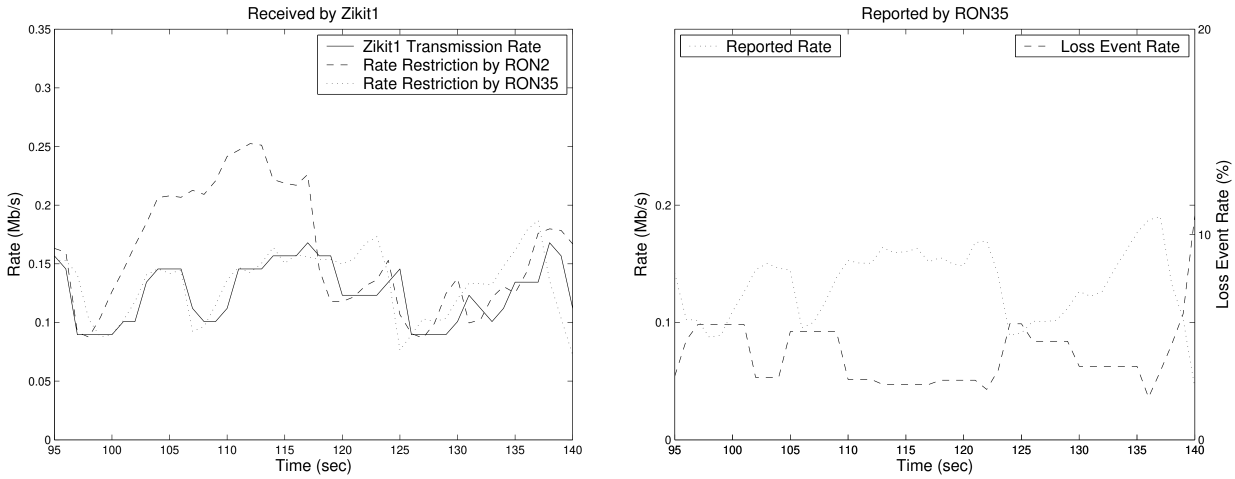
7.2 Rate Adaptation to Membership Changes

In this test all RON computers except RON35 and Zikit1 were members of group G. The application running at RON45 sent as much traffic as was the restrictions reported by other delegates. In Fig. 7.3 we see that when RON35 joins G the sending rate of RON45 drops. Afterwards, when Zikit1 joins G, the rate drops even more significantly. 60 seconds later, when Zikit1 leaves G, the rate increases and then increases again when RON35 leaves G.

7.3 Fairness Among Senders in a LAN

In Fig. 7.4(a) we see aggregate rates generated by Zikit1, Zikit2, Zikit3 while all RON computers receive the messages. Zikit1-3 all residing in the same LAN at the Hebrew University of Jerusalem. Zikit1 sends messages through the entire duration of the test, while Zikit2 (for 180 seconds) and Zikit3 (for 60 seconds) send messages for a shorter period. The graph also presents the rate by which Zikit's LAN was limited by the other delegates. One can see that the influence of the reported rate, which in turn depends on loss rate, is much higher than the impact of the number of senders.

In Fig. 7.4(b) we present the achieved sending rate of three senders sharing the same aggregate transmission rate during this test. The results clearly show that Max-Min fairness is achieved. The measured results were averaged over three seconds intervals. The results were the same on smaller



(a) Zikit1 is limited by RON2 and RON35 (b) Reported rate vs. loss rate

Figure 7.2: Rate limitation factors

scale interval, but for presentability we chose a relative large one.

7.4 TCP-friendliness

Figure 7.5 presents the TCP-friendliness test results. In this test we ran six cycles of the following scenario: every cycle started with 60 seconds transmission of Xpand, following which a TCP session was run for an additional 60 seconds.

In this test, only Zikit1 and RON35 were the group members. Both Xpand and TCP messages were sent by Zikit1. The achieved rate was averaged over 3 seconds interval. The average rate over the periods when both flows shared the network resources are also presented in the figure.

In all the cycles the performance of Zikit1 showed that the bandwidth limited by our congestion control mechanism is comparable to that of TCP and therefore the message flow is TCP-friendly.

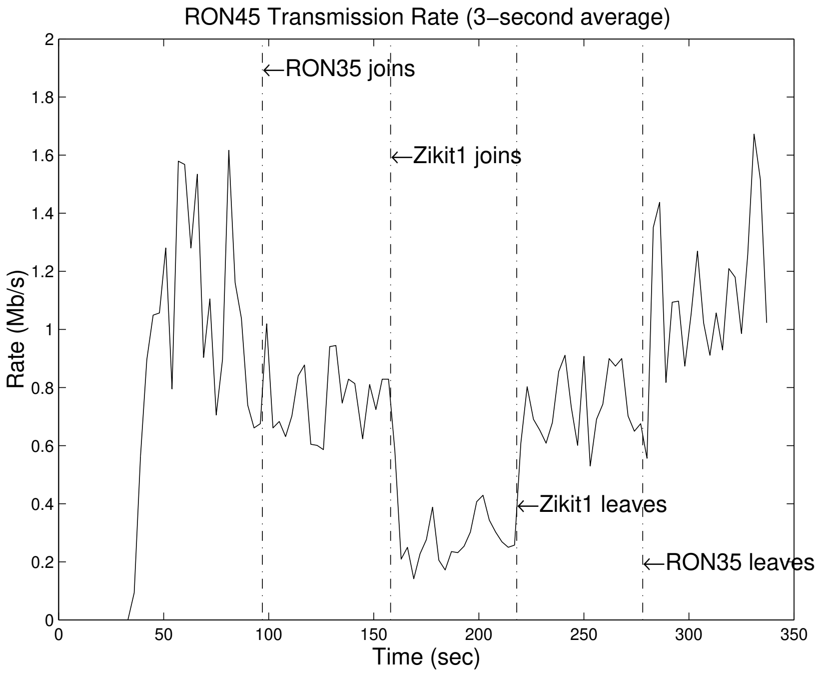
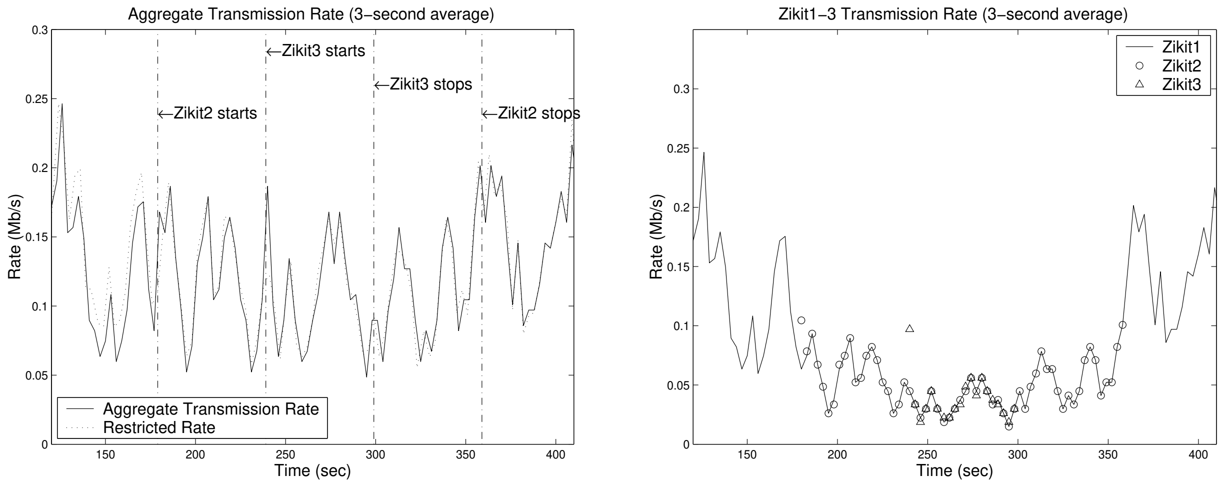


Figure 7.3: Responsiveness to membership changes



(a) Aggregate Rate (b) Particular Rate

Figure 7.4: Multiple senders aggregation and fairness

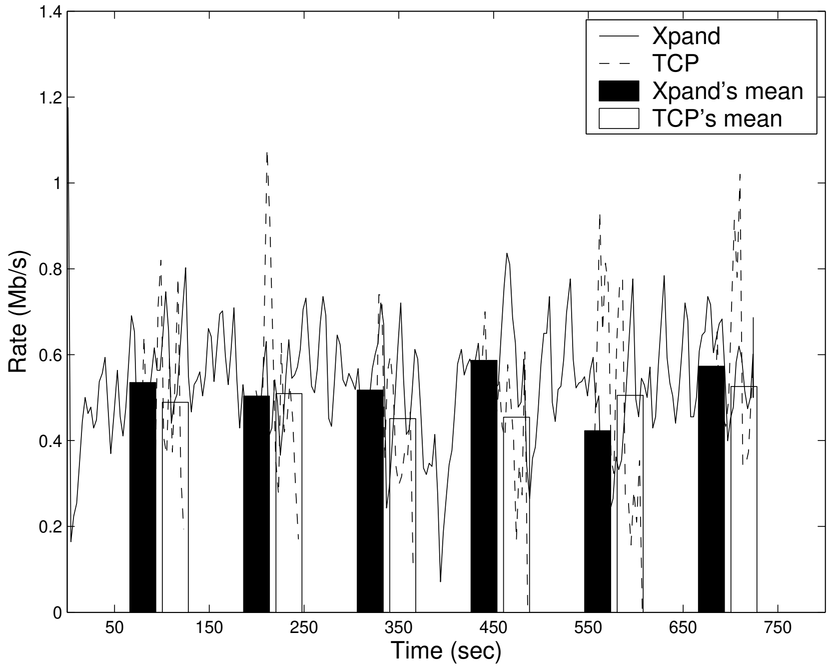


Figure 7.5: TCP-friendliness

Chapter 8

Local Area Performance Results

The main goal of this work is design, implementation, and performance evaluation of end-to-end many-to-many congestion control mechanism in wide area environment. However, the situation when all members of a multicast group reside on the same LAN is very typical and thus presents an important special case for any flow control mechanism.
When all the members reside on the same LAN, it is usually ability of receiving host to handle incoming data what limits the overall performance (transmission rate) since the bandwidth is plentiful. The TCP throughput equation cannot be used in this case because round trip time and loss rate are far too small in local area networks, whereas the RTT and loss rate event measurements mechanisms in Xpand are relatively coarse. The primary goals of flow control in LAN are:

- to achieve performance comparable with that of TCP
- to demonstrate friendliness in presence of TCP flows

As a secondary goal, we wish to evaluate the rate distribution mechanism (described in 5.3) in a LAN environment, when the rate distribution algorithm is run due to changes in senders' rate demands vector rather than changes in the aggregate available rate.

This section is dedicated to Xpand performance analysis within a LAN. The experiments were conducted over an idle 100 Mbit/sec switched local area network. We used machines with Pentium III 500MHz processor, 256 MBytes memory, running Linux 2.4.9-13 operating system. The local area testing topology is shown on figure 8.1.

8.1 Test Scenarios

In all the tests participants join a single group (thus becoming members of the group) prior to the measurements. During the performance evaluation period no membership changes occur. Two types of scenarios were tested:

- Xpand only scenarios

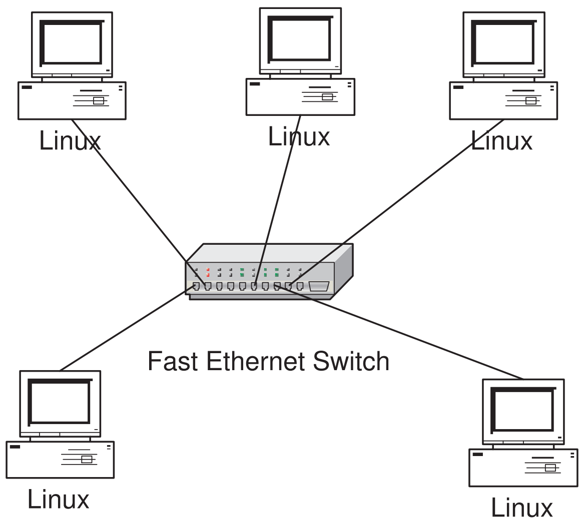


Figure 8.1: LAN Tests Topology

  - several members, single sender
  - several members, each member is a sender

- Xpand vs TCP scenarios
  - Xpand competes for the bandwidth with TCP flows

A sending member is pre-configured to emit a burst of messages every 10 milliseconds (the maximal clock resolution on machines being used for the testing). Number of messages per a burst and size of a message was chosen to obtain transmission rate of over 100 Mbit/sec (the network capacity). In other words, each sender is trying to push on the wire as much data as possible.
Every message that is sent to a specific group is received by all the members of that group. Note that all the tests were performed multiple times, during different hours and different days, on an idle network.
To evaluate 'friendliness' of Xpand in presence of TCP flows, we conducted point-to-point performance tests between pair of participating machines (Xpand sender and Xpand receiver). We used the Netperf toolkit [44] to test the performance of TCP.

8.2 Single Sender Test

In the single sender test there are 5 group members, one of them is sending (as fast as possible) during 120 seconds, the others only receive. The main objective of the test is to examine the maximal Xpand performance in a LAN environment. The results are shown on figure 8.2. As we can see, Xpand achieves transmission rate of 20..60 Mbit/sec with average value of 40 Mbit/sec. In comparison with TCP, which performs steadily 93..94 Mbit/sec in the same conditions, Xpand demonstrates more than twice lower transmission rate as well as significant rate instability.

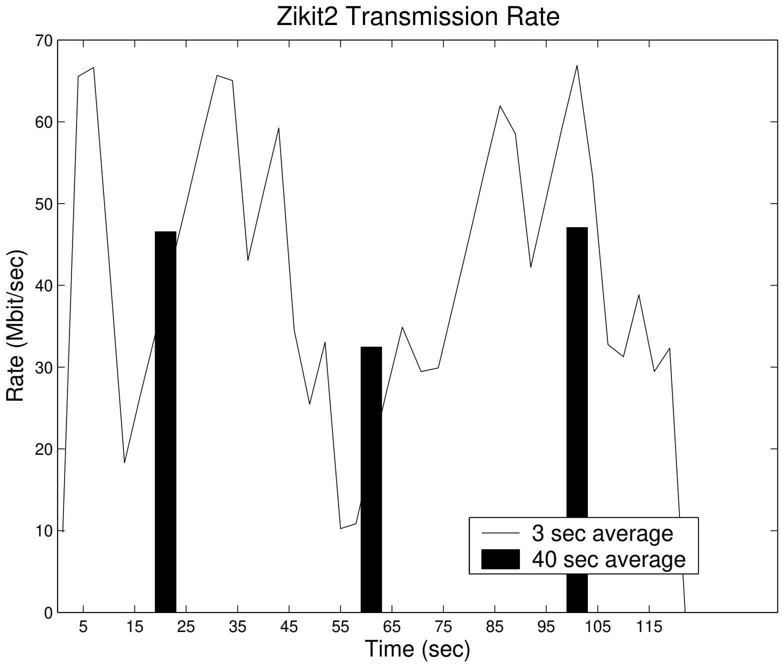


Figure 8.2: 1 Sender, 5 Receivers

Lower transmission rate could be caused by lack of fine-tuning in the system. Proper profiling will alleviate the problem. Note that periodical sharp rate drops decrease overall performance as well.

One possible reason explaining the rate fluctuations is that local delegate (who is responsible for completion of missed messages within its LAN) accepts multiple feedback (*NACK*s) from receivers, as opposed to TCP. If one message gets lost, the delegate will get corresponding *NACK* from each receiver. Furthermore, a NACK is retransmitted periodically by each receiver until missing message eventually arrives.

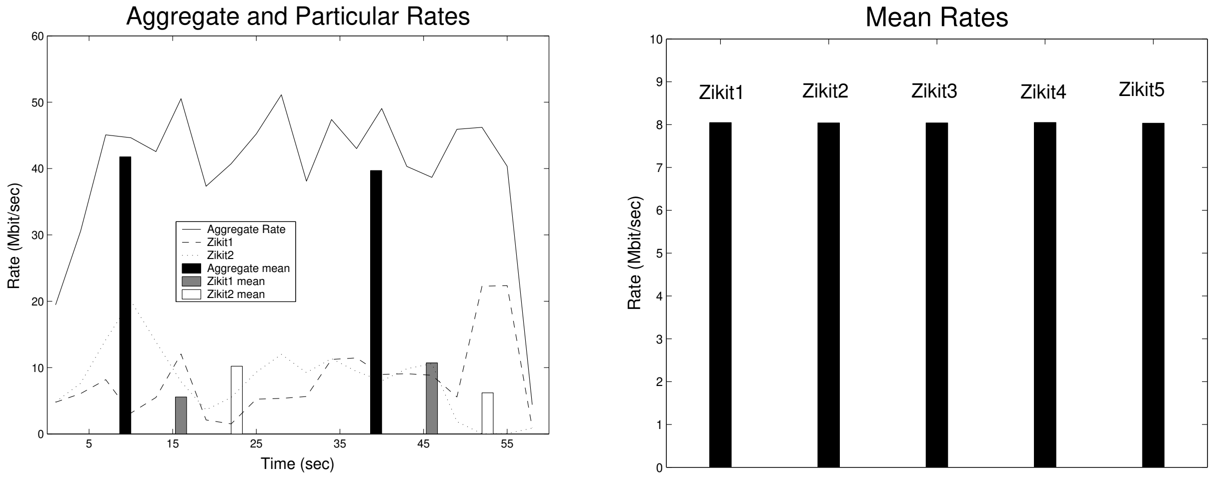
Intra-LAN flow control mechanism, described in Chapter 6, reacts to intra-LAN *NACK* occurrence by halving the aggregate transmission rate. If two *NACK*s that refer to the same loss event trigger rate drop twice, unjustified performance decrease will happen.

Although a delegate does not decrease rate upon intra-LAN *NACK* arrival if not enough time (less than 100 ms in current version) elapsed since previous rate decrease, still there exists a possibility a rate is decreased multiple times in response to the same loss event. This point should be investigated in a future.

8.3 Multiple Senders Test

In this scenario there are 5 group members, and all 5 members are sending data as fast as possible. The goal of the test is to investigate maximal performance of Xpand when several senders compete for bandwidth, and fairness of the rate distribution mechanism described in Chapter 6.

As we can see on the figure 8.3(a), aggregate transmission rate achieved by the 5 senders is very close to that of the single sender test. Furthermore, figure 8.3(b) shows that average rates of the



(a) Aggregate and Particular Transmission Rates (b) Averaged Particular Transmission Rates

Figure 8.3: 5 Senders

senders are in fact the same, whereas at any given moment instant rates can differ (compare Zikit1 and Zikit2 performances on figure 8.3(a)).

We can thus conclude that presence of multiple senders does not hurt Xpand performance, and the rate distribution algorithm is fair.

It should be noticed that with multiple senders aggregate rate fluctuations are less prominent than in the single sender test. One of the reasons is more even distribution of loss events in a time due to lack of synchronization between the senders. This randomization causes less false rate decreases.

8.4 TCP Friendliness Tests

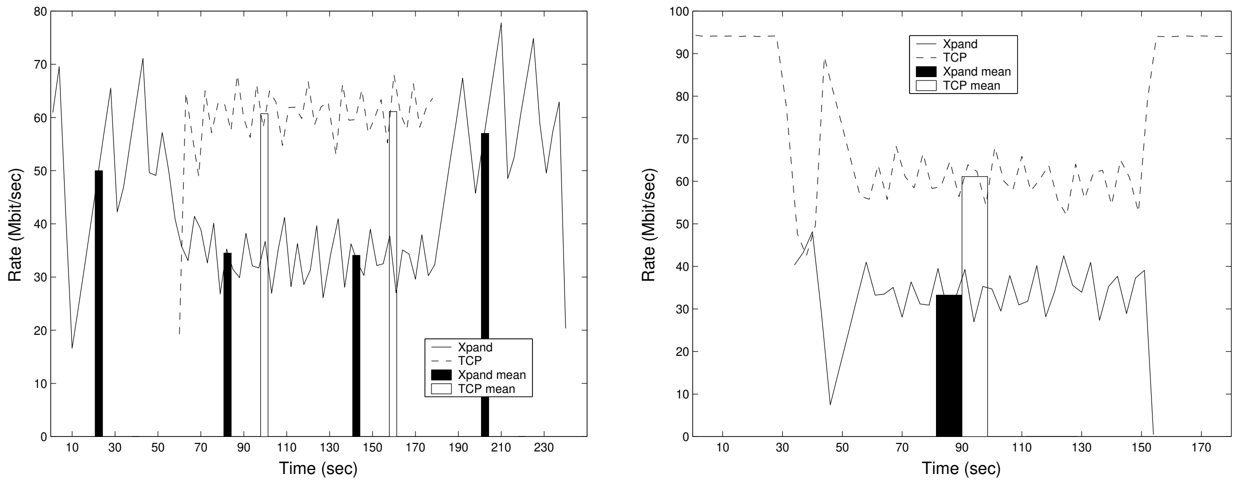
The following series of tests demonstrate fairness between Xpand and TCP when the two are competing for bandwidth. We conducted 4 types of tests: single Xpand flow competes with 1,2,4, and 8 TCP flows respectively. Every test has been run in two scenarios:

- a Xpand flow starts and runs 60 sec, then TCP flow(s) join it for 120 sec, and finally the TCP flows stop, leaving the Xpand flow running alone for last 60 sec
- TCP flow(s) start and run 30 sec, then a Xpand flow joins them for 120 sec, and finally the Xpand flow stops, leaving the TCP flows running for last 30 sec

8.4.1 Xpand vs Single TCP Flow

On the figure 8.4(a) we can see that when TCP flow joins, Xpand's aggregate rate falls 30%. Competing for the bandwidth, TCP uses twice as much of the resource than Xpand does. When TCP stops, Xpand's performance raises back to the level before TCP emerged.

Figure 8.4(b) shows that when Xpandflow joins, TCP's aggregate rate falls about 30%. Competing for the bandwidth, TCP again uses twice as much of the resource than Xpand does. When



(a) TCP joins Xpand

(b) Xpand joins TCP

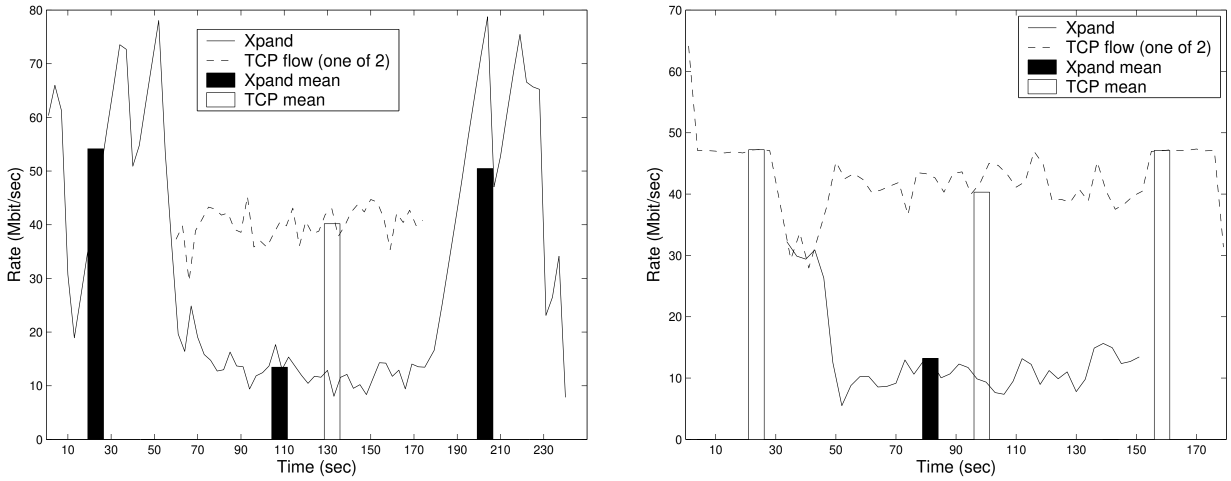
Figure 8.4: Xpand Competes for the Bandwidth with a single TCP Flow

Xpand stops, TCP's performance raises back to the level before Xpand emerged. Note the clear correlation between rate increases for one flow and rate decreases for the other, which reflects struggle for the limited resource (the bandwidth) between the two flows.

As expected, average transmission rates for Xpand and TCP in the two scenarios are the same.

8.4.2 Xpand vs Several TCP Flows

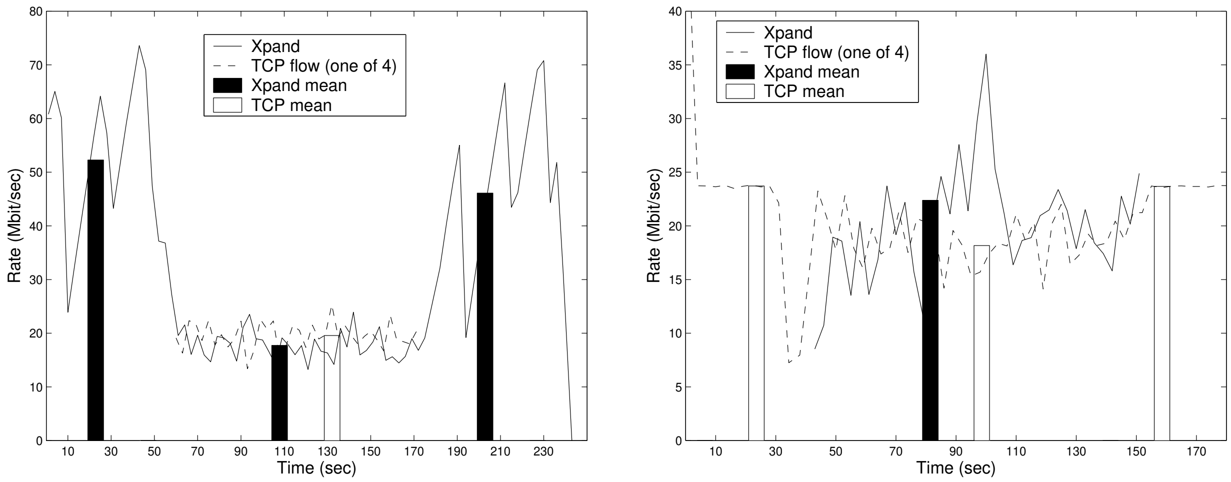
In the following three experiments we examine Xpand's fairness when competing for bandwidth with 2,4, and 8 TCP flows (figures 8.5, 8.6, and 8.7 respectively). Each graph contains a curve representing Xpand transmission rate and a curve representing rate of one of the TCP flows. We don't show all TCP flows for the sake of graph readability. Results for any two TCP flows are very similar, in terms of average performance as well as in instant behavior.



(a) TCP flows join Xpand

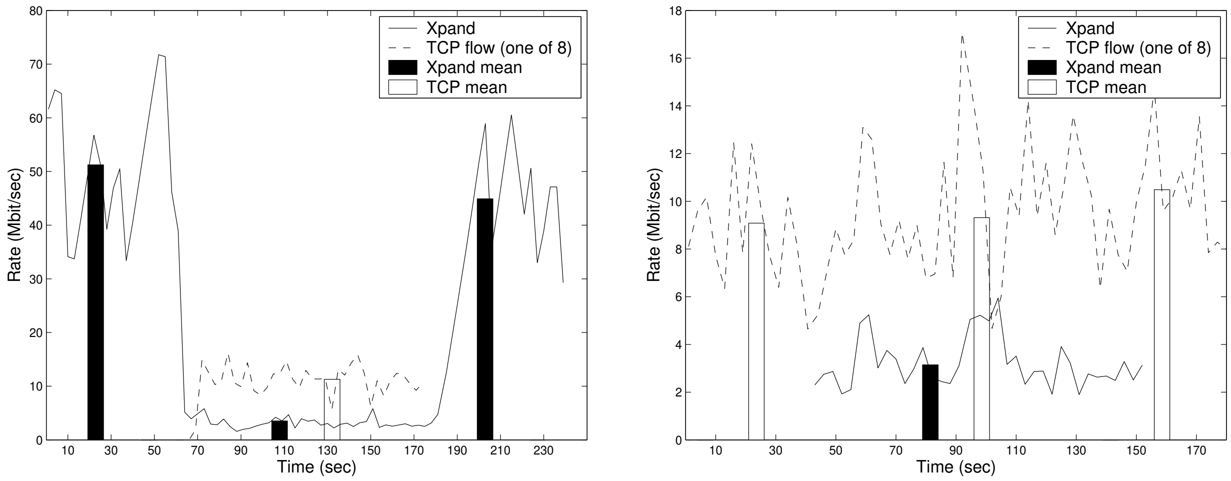
(b) Xpand joins TCP flows

Figure 8.5: Xpand Competes for the Bandwidth with 2 TCP Flows



(a) 4 TCP flows join Xpand (b) Xpand joins 4 TCP Flows

Figure 8.6: Xpand Competes for the Bandwidth with 4 TCP Flows



(a) 8 TCP flows join Xpand (b) Xpand joins 8 TCP Flows

Figure 8.7: Xpand Competes for the Bandwidth with 8 TCP Flows

The following observations:

- in most of the tests Xpand's transmission rate is about 2-3 times lower than that of any of the TCP flows (except the test with 4 TCP flows, where Xpand's performance is roughly the same as that of TCP);
- when TCP flows join, Xpand drops its transmission rate more readily than TCP does when Xpand joins;

- allow us to conclude that Xpand demonstrates reasonable fairness (*TCP-friendliness*) when competing for bandwidth with several TCP flows within a LAN.

Chapter 9

Conclusions and Future Work

The original Xpand implementation deployed simple window-based congestion/flow control scheme. With this scheme, a sender holds a fix-sized window of sent but not yet acknowledged messages. A sender is allowed to transmit as long as the window is not full. Thus, transmission rate is limited only by ACKs arrival rate. When an acknowledgment for some message is deferred, the window fills up and eventually becomes full, causing the sender to stop transmission. Testing showed that this scheme is neither stable in LAN nor TCP-friendly in WAN. In fact it is as bandwidth greedy as UDP is, making its employment in a wide area environment infeasible.

To overcome this problem, new explicit congestion control mechanism was designed and implemented within Xpand. The new scheme is based on the equation from [12] which characterizes TCP steady state throughput as a function of packet loss rate, estimated round trip time and TCP retransmission timeout. In order to use this equation, we designed a RTT estimation scheme and a mechanism for loss rate measurement. These are continuously evaluated and translated into an appropriate transmission rate. *Appropriate* rate is a rate that TCP would achieve according to the equation, given instant values of RTT and loss rate.

The thesis presented the congestion control mechanism: general design, underlying algorithms, relevant protocols, and implementation details as well as results of performance evaluation testing. One of the main difficulties was to embed the new mechanisms into a large, complex operational system. Whenever possible, we took an advantage of existing data structures, message formats, and protocols. In cases when existing implementation did not provide enough functionality, we introduced new modules, message types and extended the available protocols to meet new requirements.

Our implementation proves that it is possible to create a many-to-many congestion control mechanism that is TCP-friendly and provides a built-in fairness among senders that share resources. TCP-friendliness is achieved both in a LAN and in a WAN environments.

Another important result is that the overhead imposed by the scheme implementation is minor in terms of both bandwidth and CPU consumption:

- Inter-LAN ACKs are sent by unicast and not by multicast as it is done in the original version. With unicast, info attached to an ACK relates only to the ACK receiver.

- Some additional fields were introduced into inter-LAN ACK message to allow exchange of congestion control related info between a sender and the receivers (e.g. RTT, acceptable transmission rate, etc).

- No new inter-LAN control message types were introduced. Moreover, amount of control messages sent per unit of time stays the same as in the original version.
- CPU consumption by protocol members is reasonable. Even on humble Pentium II 500MHz Linux machines we observe at most 10% of CPU usage on peak loads.
- Finally, no performance degradation (in terms of throughput) is observed after introduction of the new congestion control mechanism.

Future research will focus on improving the results, possible directions being

- to improve responsiveness of the mechanism to message losses and group membership changes.
- in our current implementation, retransmissions are not considered to be a part of the aggregate sending rate, though a conservative approach was taken. Future work should find an appropriate way to incorporate retransmissions in the congestion control mechanism while allowing for heterogeneous (unicast and multicast) retransmission schemes.
- AIMD intra-LAN flow control mechanism should be fine-tuned. As a result, Xpand should achieve TCP performance in a LAN on the one hand, and take more fair bandwidth share on the other (currently, Xpand is too conservative, achieving twice as less transmission rate than any of the competing TCP flows).
- As the tests show, Xpand demonstrates conservative TCP-friendliness in WAN as well, which is partly due to conservative model underlying the throughput equation. Future work could conduct closer investigation of factors that define the extent of TCP-friendliness in Xpand, and use these parameters for dynamic self-tuning.

Chapter 10

Appendix

The congestion and flow control mechanisms were embedded into the large operational system. Whenever possible, we took advantage of existing data structures, functions, and protocols. In other cases we adapted current modules or created new ones. To make life of future generations of Xpand developers easier, we provide here implementation-related details.

10.1 A Regular Member Complies with Specified Transmission Rate

We changed transmission service module (files **src/regular/core/transmission_service.c**, **src/regular/core/transmission_service.h**), mainly function *ts_get_next_message()*:

- explicit $SENT$ and $APPLICATION\_SENT$ pointers
  (fields *Sequence_Number sent* and *Sequence_Number application_sent* in *struct GroupInfo*, file **src/regular/data_struct/group_info.h**) to the group data structure. See fig. 10.1 for Xpand sender window structure.

  – *Sequence_Number sent* points to the last message put on a wire.
    All the messages to the left of *sent* (with sequence numbers less than *sent*) have already been sent. Messages between *ack* and *sent* have been sent but not yet acknowledged.

    ∗ initialized in file **src/regular/data_struct/groupds.c**, function *gr_join()*, to *SN_PREDECESSOR(0)*.

    ∗ updated in file **src/regular/core/transmission_service.c**, function *ts_get_next_message()* to sequence number of the most recently transmitted message.

  – *Sequence_Number application_sent* points to the last message which has been accepted from the application for sending. Messages with sequence numbers between *sent* and *application_sent* have not been transmitted yet.

    ∗ initialized in file **src/regular/data_struct/groupds.c**, function *gr_join()*, to *SN_PREDECESSOR(0)*.

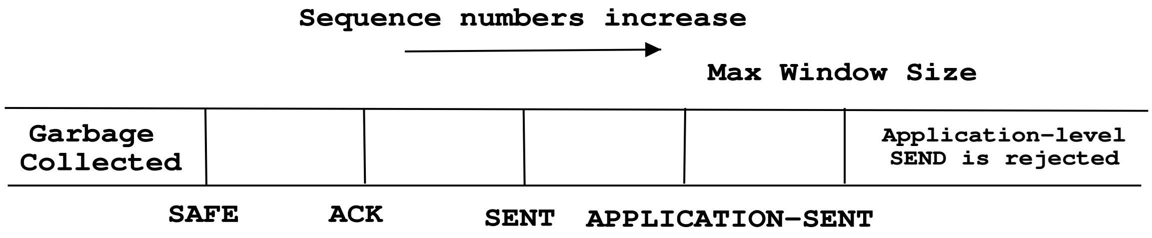


Figure 10.1: Xpand Sender Window Management

    * incremented in file **src/regular/core/core.c**, function *C_send()* with every message accepted from an application.

- new fields in the *struct GroupInfo* (file **src/regular/data_struct/group_info.h**):
  - *unsigned long allowed_rate*: maximal allowed transmission rate (bits/sec) for a group sender, obtained from the local delegate.
    * initialized in file **src/regular/data_struct/groupds.c**, function *gr_reset_values()* to *ULONG_MAX*, which is a temporary hack and should be changed to *CC_MIN_REGULAR_TRX_RATE*, defined in **src/regular/core/congestion_control.h**.
    * updated in file **src/regular/core/congestion_control.c**, function *cc_process_rate_reply_message()* every time a local delegate sends rate share update for specific group.
    * updated in file **src/regular/core/core.c**, function *C_set_max_rate()*. For debug purposes only (used in textbfexample/user.c).
  - *double last_timestamp*: last time a message was sent for specific group by this regular member. Used for complying with specified transmission rate limit. More specifically, given amount of time T elapsed since last message transmission and rate limit R, the regular can send message of size $R * T$ without rate limit violation.
    * initialized to *now time* in file **src/regular/data_struct/groupds.c**, function *gr_reset_values()*.
    * updated in file **src/regular/core/transmission_service.c**, function *ts_get_next_message()* every time a data message is chosen for transmission.
  - *long bytes_in_bucket*: rate remainder Rem since the previous message transmission. Message of size $R * T + Rem$ can be sent without rate limit violation.
    * the value of *long bytes_in_bucket* is limited by system-wide parameter *long bucket_size* (file **src/regular/data_struct/parameterds.h**).
    * initialized to zero in file **src/regular/data_struct/groupds.c**, function *gr_reset_values()*.
    * updated in file **src/regular/core/transmission_service.c**, function *ts_get_next_message()* every time a message is chosen for transmission and available rate is underutilized.

- file **src/regular/core/transmission_service.c**, *struct Transmission_Service_Info*. Instead of a single list of all existing groups we manage two linked lists of *active* groups, i.e. those that have at least one message to send (fields *struct s_dbl_ll_handle *ready* and *struct s_dbl_ll_handle *suspended* of *struct Transmission_Service_Info*, file **src/regular/core/transmission_service.h**):

- *ready* list: list of all *Internal_Group_Ids* that have messages to send and sending will not violate rate limitation.
  - initialized in file **src/regular/core/transmission_service.c**, function *ts_init()*
  - a group is inserted in *ts_get_next_message()* when *suspended* group becomes *ready*, and in *ts_add_to_ready_list()* when *idle* group (neither *ready* nor *suspended*) becomes *ready*.
  - a group is deleted in *ts_get_next_message()* when *ready* group becomes *suspended* due to rate limitation, or when a group has no messages to send; in **src/regular/core/group_state_machine.c**, function *pre_leave_state()* (via *ts_drop_group()*) when a regular member leaves specified group.
- *suspended* list: list of all *Internal_Group_Ids* that have messages to send, but cannot send right now due to rate limitation.
  - initialized in file file **src/regular/core/transmission_service.c**, function *ts_init()*
  - a group is inserted in *ts_get_next_message()* when *ready* group becomes *suspended*
  - a group is deleted in *ts_get_next_message()* when a group has no messages to send; in **src/regular/core/group_state_machine.c**, function *pre_leave_state()* (via *ts_drop_group()*) when a regular member leaves specified group.

Groups that have nothing to send appear neither in the ready list nor in the suspended list, thus shortening groups list search time.

The both lists hold only *Internal_Group_Ids*. Access to corresponding message queues is accomplished through *GetGroupInfo(reg, gid)* function.

The *ready* list has a pointer (*last_peek*) that traverses the list in the round robin manner (for fairness), while looking for a message to send. It always points to the successor of the most recently inspected group. When a group enters the *ready* state, corresponding entry in the *ready* list should be created, and the *ready* flag in *GroupInfo struct* (file **src/regular/data_struct/group_info.h**) should be turned on.

The two group lists are scanned periodically by the *ts_get_next_message()* function which is called by the scheduler every Δ ms.

- actually, every Δ ms total of M messages can be sent. M is a system parameter, as well. First, protocol control messages are sent (e.g. ACKs,$NACK$s, etc), and eventually N regular data messages.

These N back-to-back calls to *ts_get_next_message()* we define as a *round*. At the beginning of every round we pass over the *suspended* groups list to find groups which have become ready for sending, and move them to the *ready* list. Every time *ts_get_next_message()* is called, we pass through the *ready* groups list, starting where we finished last time, until we either found a message eligible for sending, or all the ready groups list is scanned, and no ready message was found. When examining a particular group,

- we first check whether it has at least one message to send (its $APPLICATIION\_SENT$ pointer must be greater than $SENT$ pointer). If it does not, we remove the group from the *ready* list.

- we then check whether sending the first message in the group's queue complies with rate limitation. If it does not, we move the group to the *suspended* list. If it does, the message is removed from the group's queue and is returned by *ts_get_next_message()* for transmission.

How we decide whether a message can be sent

For each group, we keep a time when the last message was sent (field *long bytes_in_bucket* in *struct GroupInfo*, file **src/regular/data_struct/group_info.h**, see above). When considering a group, we know how much time elapsed from the last send (say, Δ) and maximal allowed transmission rate(R). Then, if size of the first message is at most $\Delta * R$, it can be sent without rate limit violation. Otherwise, it is not eligible for sending.
For more flexibility, in addition to Δ and R we use a *token bucket* for rate shaping, which saves rate remainder since the previous message transmission: field *long bytes_in_bucket* of *struct GroupInfo*, file **src/regular/data_struct/group_info.h** (see above).

Eligible for sending message of size S may be smaller than $\Delta * R$. We define $bytes\_in\_bucket = bytes\_in\_bucket + \Delta * R - S$. This variable saves amount of rate underutilization till next inspection of the group. Then, on the next attempt to send a message (of size S), the message will be eligible if $S \leq \Delta * R + bytes\_in\_bucket$.

10.2 Regular Member Estimates Application Sending Rate

A regular member must have an estimation of a rate R_a with which an application above passes data for transmission. The regular then requests from local delegate to allocate it R_a out of total aggregate rate for the local *LAN*, to allow the application to send as fast as needed.
Estimation of R_a should be obtained frequently enough to be up-to-date with the application needs, and to give back excess rate as fast as possible, to make it available for other senders.

The congestion control module (file **src/regular/core/congestion_control.h**, **src/regular/core/congestion_control.c**) is in charge for measuring application sending rate in a regular member.

- system parameter *long appl_rate_measurement_period*, file **src/regular/data_struct/parameterds.h**. Defines duration of a *congestion control round.* Congestion control round is a time period between two consecutive samples of application sending rate (i.e. invocations of function *cc_measure_appl_rate()*).
- function *cc_measure_appl_rate()*, file **src/regular/core/congestion_control.c**, is called periodically (every *appl_rate_measurement_period* ms) by the scheduler. Performs measurements of application sending rate.
- file **src/regular/data_struct/group_info.h**, *struct GroupInfo*:
  - *unsigned long written_bytes_counter*: number of bytes passed from an application to the core since the beginning of the last congestion control round. Used to compute application sending rate.
  - *double smoothed_app_rate*: weighted history of measured instant application sending rate (bit/sec). This rate is requested by a regular from local delegate. $smoothed\_app\_rate = \alpha * smoothed\_app\_rate + (1 - \alpha) * last\_measured\_appl\_rate$.

    Smoothing quotient α is a system parameter defined in **src/regular/data_struct/parameterds.h**. Currently $a = 1/8$ to give most of the weight to the recent instant rate measurement.

– *enum* { *CC_INACTIVE, CC_BLOCKED, CC_JUST_UNBLOCKED, CC_NORMAL, CC_SILENT*} *state*: states of a group sender.

  A regular is sending actively for a group if it has sent at least one message during last two congestion control rounds.

  * *CC_INACTIVE*: regular is not sending actively to the group. The initial state (initialized in file **src/regular/data_struct/groupds.c**, function *gr_reset_values()*). Entered from *CC_SILENT* after a full congestion control round of inactivity.
    Quitted to *CC_JUST_UNBLOCKED* when *gc_update()*
    (file **src/regular/core/garbage_collection.c**) unblocks the group.
  * *CC_BLOCKED*: entered when sender window is full and call to *W_Send()* blocks the application. The state is set to *CC_BLOCKED* in function *cc_block_group()*, file **src/regular/core/congestion_control.c**. *cc_block_group()* is called in *C_send()*, file **src/regular/core/core.c**, when the core recognizes that sending window is full (using *cc_check_window()*).
    Quitted to *CC_JUST_UNBLOCKED* when *gc_update()*
    (file **src/regular/core/garbage_collection.c**) unblocks the group.
  * *CC_JUST_UNBLOCKED*: entered from *CC_BLOCKED* or *CC_INACTIVE* when garbage collector (file **src/regular/core/garbage_collection.c**) recognizes that sending window is not full (using *cc_check_window()*) after a full round of activity.
    Quitted to *CC_NORMAL* after an additional round of activity
    (file **src/regular/core/congestion_control.c**, function *cc_measure_appl_rate()*).
  * *CC_NORMAL*: sender window is not full. Entered from *CC_SILENT* or *CC_JUST_UNBLOCKED* in function *cc_measure_appl_rate()*, after a full round of activity.
    Quitted (in function *cc_measure_appl_rate()*) to *CC_BLOCKED* using *cc_block_group()* in *C_send()* when the core recognizes that a sender window is full (via *cc_check_window()*).
    Quitted (in function *cc_measure_appl_rate()*) to *CC_SILENT* after a full round of inactivity (when *written_bytes_counter* is zero).
  * *CC_SILENT*: entered (in function *cc_measure_appl_rate()*) from *CC_NORMAL* after a full round of inactivity (when *written_bytes_counter* is zero).
    Quitted (in function *cc_measure_appl_rate()*) to *CC_INACTIVE* after an additional round of inactivity.
    Quitted (in function *cc_measure_appl_rate()*) *CC_NORMAL* after an active round (when *written_bytes_counter* is not zero).

  We stop measuring application rate as soon as the application becomes blocked (message buffer is full). In the *CC_BLOCKED* state, there is no application rate updates. We can resume rate evaluation only if the application stays unblocked during two last consecutive rounds.

  When the garbage collector unblocks a previously blocked group (somewhere between two consecutive rounds), the group enters the *CC_JUST_UNBLOCKED* state. If till the beginning of the next round the application does not block, it enters the *CC_NORMAL* state, but the measurement is not made yet. If till the beginning of another round the application does not block, the state stays *CC_NORMAL*, and rate estimations are resumed. If the application gets blocked, sender returns to the *CC_BLOCKED* state.

  If a non blocked group sends nothing during a round, it becomes *CC_SILENT*. If a *CC_SILENT* group sends nothing during additional round, it stops to be active sender and is removed from active senders list.

These states are relevant for active senders only. A group that is not active (does not appear on the active groups list) is in the *CC_INACTIVE* state. When inactive group becomes active, its initial state is *CC_JUST_UNBLOCKED*, since we need at least two rounds to start measurements.

See fig. 10.2 for the full state transition diagram.

- file **src/regular/core/congestion_control.h**, *struct Congestion_Control_Info*
  - *struct s_dbl_ll_handle *active_groups*: list of groups to which this regular member has been sending actively. A regular is sending actively to a group if it has sent at least one message to the group during last two congestion control rounds.
    - initialized in *cc_init()*, file **src/regular/core/congestion_control.c**
    - a group is inserted in *cc_add_to_active_groups()*, file **src/regular/core/congestion_control.c**. Called from *C_send()* (via *cc_activate_group()*) when sending window is not full.
    - a group is deleted in *cc_measure_appl_rate()*, file **src/regular/core/congestion_control.c**, when the group passes from *CC_SILENT* to *CC_INACTIVE* state.
    - a group is deleted in *cc_drop_group()* (called from *pre_leave_state()*, file **src/regular/core/group_state_machine.c**) when a regular member leaves specific group.
  - *double round_start*: start time of the current congestion control round. Used to determine exact duration D of last congestion control round ($D = now\ time - round\_start$). Then, given D and amount of data accepted from an application (*unsigned long written_bytes_counter*, see above), we can compute instant application sending rate estimation as $written\_bytes\_counter/D$.
    - initialized in *cc_init()*, file **src/regular/core/congestion_control.c** to -1.0 (to let know the value is not initialized yet).)
    - updated in *cc_measure_appl_rate()*, file **src/regular/core/congestion_control.c** to *now time*.

Every Δ ms (system parameter *long appl_rate_measurement_period*, defined in file **src/regular/data_struct/parameterds.h**) the scheduler invokes the function *cc_measure_appl_rate()* to measure R_a.
During a congestion control round (period between two consecutive invocations of *cc_measure_appl_rate()*) we count a number of bytes accepted from an application. Then, at the end of each round, we know how much data (say, S bytes) was accepted from the application since the previous call to *cc_measure_appl_rate()*. Then, we calculate $R_a = \frac{S}{\Delta}$.
This approach works fine for unlimited message buffer. In a real system, message buffer size is limited. The problem is that when an application generates data faster than Xpand can send it, the buffer fills up, the application eventually gets blocked and stops sending. Thus, new messages aren't put into the buffer, and measured value of R_a will be less than actual application need. All in all, if message buffer is full, it have no space to grow, and thus there is no way to know that application has more data to send.
To overcome the problem, the following technique is used (see fig. 10.2 for full state transmission diagram):

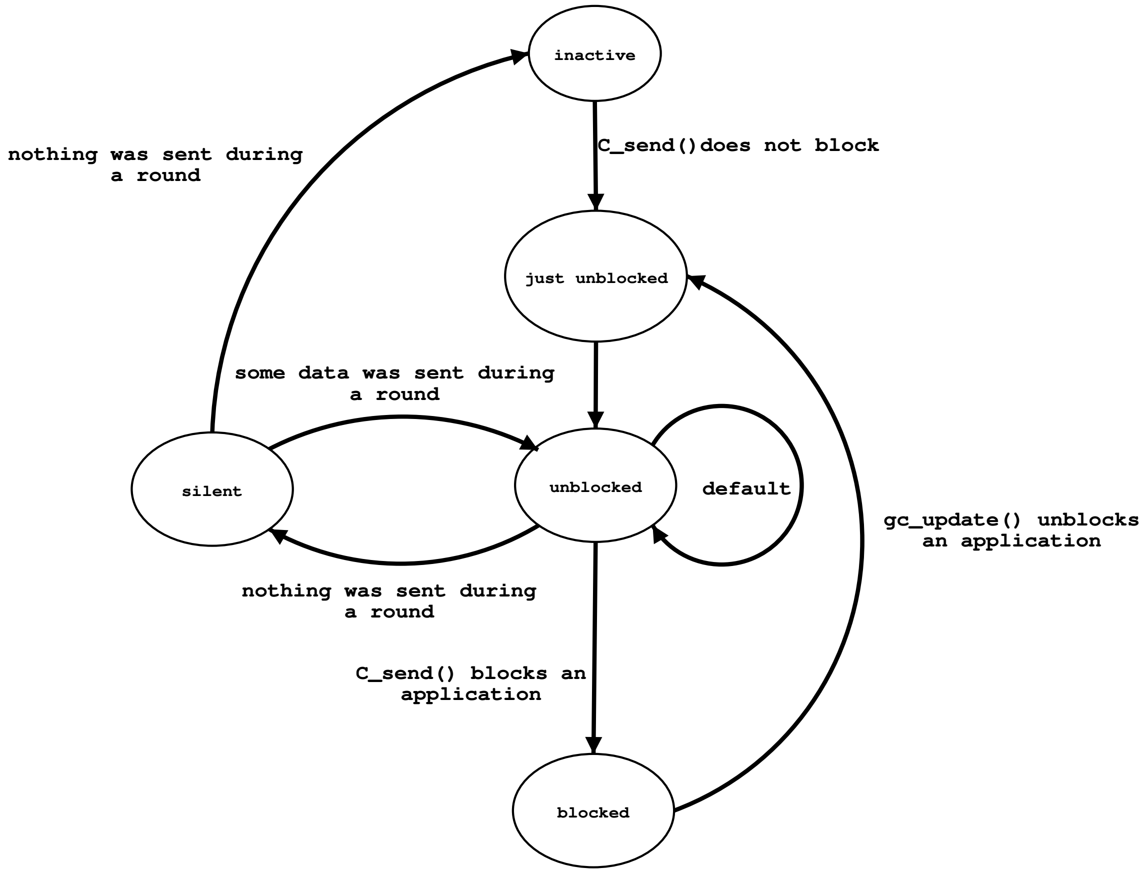


Figure 10.2: Active Sender State Machine

- (The *unblocked* state) as long as the message buffer is not full, we measure R_a as described above.
- (The *blocked* state) if at the beginning of a round we note that the buffer has became full
  - we set $R_a = 2 * R_a$
  - we don't update R_a until one of the following happens:
    - delegate provides us with the required rate R_a and buffer still remains full. Then, we double rate demand again and return to the previous bullet
    - message buffer is not full during two adjacent rounds. Then, R_a can be measured as in the regular case.

To smooth unavoidable fluctuations in R_a estimation from round to round, we don't use value R_a itself, but rather its weighted average:$R = \alpha R + (1 - \alpha) * R_a$, where parameter $\alpha = \frac{1}{8}$ (α is a system parameter and defined in src/regular/data_struct/parameterds.h as *smoothing_quotient*) to give most of the weight to recently measured application rate R_a.

According to the above algorithm, when a group changes state from *unblocked* to *blocked*, it doubles rate request R and waits for either unblock event, or receipt of the required rate. If upon receipt of the required rate the group is still blocked, it doubles R again. The fine point here is not to double request R immediately, but wait for a while to let the application time to send with increased rate. System parameter *min_rate_request_increase_interval* (file src/regular/data_struct/parameterds.h) defines minimal time interval between to consecutive rate increase requests.

10.3 Delegate Distributes Aggregate Transmission Rate Among Multiple Senders

This is a stand-alone module: **fair_rate_partition.h, fair_rate_partition.c**. Function *long frp_distribute_rate()* is used by a delegate to partition aggregate transmission rate for a group between all local senders to this group, in accordance with their requests.

frp_distribute_rate() is called from

- *check_rate_changed()*, called from *fc_cycle_handler()*, file **src/delegate/core/flow_control.c**, when group aggregate rate changes
- *fc_process_rate_request_message()*, file **src/delegate/core/flow_control.c**, when vector of senders' requests changes (upon arrival of *MS_RATE_REQUEST* message from one of local senders).

Available aggregate rate is shared using the *Min-Max Fairness Criteria*: small users get all they want, large users evenly split the rest. More formally, given n local senders, a vector $D = D[1], ..., D[n]$ of senders' rate demands, and total amount R of available rate, the module produces vector $S = S[1], ..., S[n]$ of rate distribution for each one of n senders, such that:

- $\sum_{i=1}^{n} S[i] = \min(R, \sum_{j=1}^{n} D[j])$
- if $D[j] \geq D[i]$ then $S[j] \geq S[i]$

In addition to the general rules of fairness, *Min-Max Fairness* algorithm provides additional features:

- Every requester i is guaranteed to get at least $\min(R/n, D[i])$ units of rate
- For each two requesters i, j such that $D[i] > D[j]$, $S[i] > S[j] \Rightarrow S[j] = D[j]$

10.4 Changes in the Bound Protocol

The *bound protocol* in Xpand provides for cumulative positive acknowledgements between delegates. Every Δ ms a receiving delegate sends *bound messages* to all remote sending delegates.

In the previous implementation *bound messages* were sent by multicast to all remote delegates that have local senders to specific multicast group. In that way, a *bound message* for some group contained *ACK* info for *all* senders to the group. Upon arrival of the *bound message*, a sending delegate had to extract relevant info (related to its local senders).

We changed this approach since we want to use *bound messages* for passing congestion control related info between sender and receiver delegates. *Bound messages* are sent by unicast, separately to each remote sender delegate. A *bound message* contains only relevant info for the destination *LAN*. An outgoing *bound message* contains:

- for each group G, for each sender S_G to G, maximal sequence number of a message accepted from S_G. If the sending delegate realizes that some receiver missed part of the messages, the *retransmission protocol* is activated to recover the loss.
- maximal sequence numbers of messages sent by local senders. Thus, remote receivers can know whether they missed some messages. If they did, remote delegates initiate the *nack protocol* to complete the lost messages.
- *RTT* measurement related data
  - time elapsed from arrival of the message being *ACK*ed until issuing this bound message
  - value of *RTT* measured by sender delegate (this value is needed by receiver delegate for loss rate estimation)
- transmission rate limit for senders

Following changes were introduced:

- file **src/common/message.h**, definition of bound message
  - *struct MS_BOUND_MESSAGE*. Bound message header, followed by *groups amount* per group headers.
    - bound message id
    - groups amount
  - *struct MS_BOUND_PER_GROUP_HEADER*. Per group header, followed by *number of senders* per sender entries.
    - group id (IP address)
    - number of senders
    - *RTT* (from the *ACK* sender to the *ACK* receiver), measured by the *ACK* sender. Used by the *ACK* receiver for loss rate estimation
    - rate limit(bit/sec) for the group, calculated by receiver
  - *struct MS_BOUND_ENTRY*. *ACK* for specific sender.
    - sender IP and port
    - max known sequence number from this sender
    - amount of time elapsed from *ACK*ed message arrival to receiver and until putting this *ACK* on the net
- file **src/delegate/core/bound_protocol.c**, *struct BP_INFO*. We define fields as follows

- *Active_List *subnets_waiting_for_bound_message*, which is a list of all subnets that should be sent a bound message.
  * initialized in file **src/delegate/core/bound_protocol.c**, function *bp_init()*.
  * subnet is inserted in file **src/delegate/core/bound_protocol.c**, function *AddToSubnetsWaitingForBoundMessagesList()*. The function is called periodically by the scheduler. The function passes over all known subnets (array of subnets $dlg->SubNet\_handle->SubNets$), for each subnet passes over all active groups. If a delegate has local members for group G which is active in subnet SN, SN is appended to *Active_List *subnets_waiting_for_bound_message*.
  * subnet is deleted in file **src/delegate/core/bound_protocol.c**, function *bp_get_next_bound_message()*, which is called periodically by the scheduler. When a subnet is chosen for sending *bound message*, the subnet is deleted from *Active_List *subnets_waiting_for_bound_message*.
    After a while function *AddToSubnetsWaitingForBoundMessagesList()* will find the subnet and will put it into the end of the subnets list.
  * subnet is deleted in file **src/delegate/core/bound_protocol.c**, function *bp_stop()* when last local group member leaves the group.
- *Sequence_Number bound_id*: an increasing counter put in every outgoing bound message (instead of per group counter in the previous implementation). A receiver of bound message checks its *bound_id*. If *bound_id* of the message is not greater than maximal *bound id* seen so far, it is a duplicate/obsolete message and should be dropped.
  * initialized to zero in file **src/delegate/core/bound_protocol.c**, function *bp_init()*.
  * incremented in file **src/delegate/core/bound_protocol.c**, function *bp_get_next_bound_message()* upon creating another bound message.
  * inspected file **src/delegate/core/message_completion_service.c**, function *mcs_process_bound_message()* upon bound message arrival.

• file **src/delegate/core/bound_protocol.c**

- function *static void AddToSubnetsWaitingForBoundMessagesList(void *par)* is called periodically by the scheduler. The function adds to the *subnets_waiting_for_bound_message* list all those subnets that have local members for the same groups that local subnet does.
- function *bp_get_next_bound_message()* is called periodically by message scheduler. Picks a remote subnet from the *subnets_waiting_for_bound_message* list and builds a bound message for group members of the subnet that have receivers in the local *LAN*.
- function *build_bound_entries_for_remote_subnet()* builds bound entries for specific remote subnet. These entries include
  * all *(remote sender, group)* pairs, where *remote sender* belongs to the remote subnet and is a member of multicast *group*, which has receivers in the local *LAN*.
  * all *(local sender, group)* pairs, where *local sender* belongs to the local subnet and is a member of multicast *group*, which has receivers in the remote *LAN*.

• file **src/delegate/core/message_completion_service.c**

- function *mcs_process_bound_message()* processes bound message arrived from a receiver and initiates recovery of messages the receiver missed.

10.5 RTT Measurements

To calculate TCP-friendly transmission rate using equation 4.1, round trip time from sender to receiver should be known. Moreover, RTT is used in loss rate estimation. In Xpand, sender delegate measures RTT to every remote receiver delegate (i.e. per {*remote delegate, group*} tuple) as follows:

1. Regular member multicasts a data message M to group G.
2. Local sender delegate receives M and notes its arrival time T_d, which approximates the time when the message has been transmitted by the regular (since inra-LAN delay is negligible in a WAN environment).
3. Remote receiver delegate accepts M and notes its arrival time T_a
   - file **src/delegate/data_struct/dlg_msginfo.h**: added a new field to *struct Dlg_MsgInfo*: *double arrival_time*
     - When M is accepted by a local sender delegate, this is the message departure time approximation (T_d).
       * set in file **src/delegate/core/message_store_service.c**,
         function *mss_insert_message()*: whenever a new data message is accepted by a local delegate, message arrival time (T_d) is stored in *double arrival_time* field of corresponding *struct Dlg_MsgInfo*.
     - When M is accepted by remote receiver delegate, this is the message arrival time (T_a).
       * set in file **src/delegate/core/message_store_service.c**,
         function *mss_insert_message()*: whenever a new data message is accepted by a remote receiver delegate, message arrival time (T_a) is stored in *double arrival_time* field of corresponding *struct Dlg_MsgInfo*.
4. Remote receiver delegate emits $BOUND$ message (i.e. ACK) to the sender delegate, acknowledging arrival of M. The $BOUND$ message contains in one of its fields (see file **src/common/message.h** or section 10.4 for the details) amount of time elapsed from M arrival to the receiver delegate until issuing the $BOUND$ message related to M (i.e. *current time*$-T_a$).
   - file **src/common/message.h**: new field added to *struct MS_BOUND_ENTRY*: *unsigned long delta*.
     - in $BOUND$ messages from receiver delegate to sender delegate this is amount of time (in milliseconds) the message was stuck at the receiver delegate.
     - set in file **src/delegate/core/bound_protocol.c**, function
       build_bound_entries_for_remote_subnet() when building an ACK message for specific sequence number.
     - inspected and used for RTT calculation in file
       src/delegate/core/message_completion_service.c,
       function *mcs_process_bound_message()* upon arrival of a $BOUND$ message from remote receiver delegate.

5. Sender delegate receives *BOUND* message and notes its arrival time (T_{now}).
6. Sender delegate computes *RTT* estimation for the {*receiver delegate, group*} tuple ($T_{now} - T_d - delta$).
7. Sender delegate updates weighted average for the *RTT*.

- the three above actions are performed in file **src/delegate/core/message_completion_service.c**, function *mcs_process_bound_message()*
- Sender delegate keeps *RTT* estimations for each {*remote LAN, group*} pair, where there exists at least one local sender for the *group*, and *remote LAN* has at least one *group* member (i.e. receiver).
  - in file **src/delegate/data_struct/group_info.h**, *struct GroupInfo* there is a list of all *LAN*s which have members of specific group (*LanInfo **LANs*).
  - in file **src/delegate/data_struct/group_info.h**, we add new fields to the *struct LanInfo*:
    - *unsigned long rtt_recvd.* This is a smoothed *RTT* estimation (in microseconds) from specific remote *LAN* to the local *LAN*. Measured by remote sender delegate, delivered to the local delegate within a *BOUND* message. The local delegate uses this value for rate limit calculations and loss measurements.
    - *unsigned long rtt_my.* Smoothed *RTT* estimation (in microseconds) from local *LAN* to specific remote *LAN*. Measured by local sender delegate, delivered to remote receiver delegate within a *BOUND* message. Receiver delegate uses this value for acceptable rate calculations and loss measurements.

10.6 Loss Rate and Transmission Rate Limit

In Xpand, a *LAN* means a {*subnet, group G*} tuple. That is, a *LAN* is viewed as a delegate and set of those local regular members that are *G* members. Thus, when we refer to a *LAN*, specific multicast group is implied.

Receiver delegate measures loss rate and calculates transmission rate limit for every sending *LAN*.

- new fields added to *struct LanInfo* (file **src/delegate/data_struct/group_info.h**) to support loss rate estimation:
  - *unsigned long rtt_recvd*: smoothed estimation of *RTT* from specific remote *LAN* to the local *LAN*. Measured by remote sender delegate, delivered to the local receiver delegate within a *BOUND* message. Local sender delegate uses *RTT* for transmission rate limit calculations and loss measurements.
    - initialized to zero in file **src/delegate/data_struct/groupds.c**, function*gr_add_lan()* when local delegate joins the group.

    * set in file **src/delegate/core/message_completion_service.c**, function *mcs_process_bound_message()* upon *BOUND* message arrival from remote delegate (using *RTT* value in the *BOUND* message).
    * used in file **src/delegate/core/message_store_service.c**, functions *mss_put()* (for counting amount of data and messages since *double last_loss_event_time + RTT*) and *update_loss_rate()* (for recalculation of loss rate).
- *unsigned long rtt_my*: smoothed estimation of *RTT* from the local *LAN* to remote *LAN*. Measured by local sender delegate, delivered to remote receiver delegate within a *BOUND* message. Remote receiver delegate uses *RTT* for transmission rate limit calculations and loss measurements.
    * initialized to zero in file **src/delegate/data_struct/groupds.c**, function *gr_add_lan()* when local delegate joins the group.
    * set in file **src/delegate/core/message_completion_service.c**, function *mcs_process_bound_message()* upon *BOUND* message arrival from remote delegate (using *delta* value in the *BOUND* message).
    * used in file **src/delegate/core/bound_protocol.c**, function *build_bound_entries_for_remote_subnet()* when building a *BOUND* message to pass *RTT* value to remote receiver delegate.
- *double loss_rate*: loss rate estimation for a message flow from remote *LAN* to the local *LAN*. Used in calculation of transmission rate limit.
    * initialized to zero in file **src/delegate/data_struct/groupds.c**, function *gr_add_lan()* when local delegate joins the group.
    * updated in file **src/delegate/core/message_store_service.c**, function *update_loss_rate()*, upon arrival of out-of-order data message from remote sender.
    * used in file **src/delegate/core/bound_protocol.c**, function *build_bound_entries_for_remote_subnet()* when calculating TCP-friendly rate limit (function *tcp_rate_formula()*, defined in file **src/delegate/core/message_store_service.c**) for remote sender and putting it into a *BOUND* message.
- *double avrg_loss_interval*: smoothed *loss interval length* (number of messages accepted in a row since *last_loss_event_time + RTT*) for message flow from remote *LAN* to the local *LAN*.
    * initialized to zero in file **src/delegate/data_struct/groupds.c**, function *gr_add_lan()* when local delegate joins the group.
    * updated in file **src/delegate/core/message_store_service.c**, function *update_loss_rate()*, upon arrival of out-of-order data message from remote sender.
    * used in calculation of loss rate estimation (*double loss_rate*): $loss\_rate = \frac{1}{avrg\_loss\_interval}$.
- *double last_loss_event_time*: time of the most recent loss event. Used to decide whether a lost message starts a new loss event or it is a part of the current loss event. Accordingly to the model [12], lost message starts new loss event if its nominal arrival time is at least *RTT* ms later than beginning of a previous loss event.
    * initialized to zero in file **src/delegate/data_struct/groupds.c**, function *gr_add_lan()* when local delegate joins the group.

  * updated in file **src/delegate/core/message_store_service.c**, function *update_loss_rate()*, when new loss event occurs.
  * used in file **src/delegate/core/message_store_service.c**, function *update_loss_rate()* to decide whether a lost message starts a new loss event.
  * used in file **src/delegate/core/bound_protocol.c**, function *build_bound_entries_for_remote_subnet()* to improve *double loss_rate* in case when long run of in-order messages makes it possible.

- *unsigned long packets_num*: number of messages that arrived from remote LAN in a row (without a loss) since *last_loss_event_time* + RTT. Used for calculation of average message size (average message size is one of the parameters required in equation 4.1).
  * initialized to zero in file **src/delegate/data_struct/groupds.c**, function *gr_add_lan()* when local delegate joins the group.
  * incremented in file **src/delegate/core/message_store_service.c**, function *mss_put()* each time new in-order data message arrives.
  * set to zero in file **src/delegate/core/message_store_service.c**, function *update_loss_rate()* at the beginning of a new loss event.
  * used in file **src/delegate/core/message_store_service.c**, function *update_loss_rate()* to calculate *double avrg_loss_interval*.
- *unsigned long trx_rate_to_send*: transmission rate limit (bits/sec) for remote sender, calculated locally based on corresponding loss rate and RTT values. Delivered to remote sender delegate within a $BOUND$ message.
  * initialized to zero in file **src/delegate/data_struct/groupds.c**, function *gr_add_lan()* when local delegate joins the group.
  * updated in file **src/delegate/core/bound_protocol.c**, function *build_bound_entries_for_remote_subnet()* every time TCP-friendly rate limit is recalculated for remote sender LAN.
  * used in file **src/delegate/core/bound_protocol.c**, function *build_bound_entries_for_remote_subnet()* when building a $BOUND$ message for remote sender delegate, to deliver the rate limit.
- *unsigned long trx_rate_recvd*: transmission rate limit (bits/sec) for local senders to specific remote LAN. Calculated by remote receiver delegate based on corresponding loss rate and RTT values. Delivered to local sender delegate within a $BOUND$ message.
  * initialized to zero in file **src/delegate/data_struct/groupds.c**, function *gr_add_lan()* when local delegate joins the group.
  * updated in file **src/delegate/core/message_completion_service.c**, function *mcs_process_bound_message()* every time TCP-friendly rate limit is accepted with incoming $BOUND$ message from remote receiver delegate.
  * set to zero in file **src/delegate/core/message_store_service.c**, function *update_loss_rate()* at the beginning of a new loss event.
  * used in file **src/delegate/core/flow_control.c**. function *get_min_WAN_rate()* when determining a globally acceptable transmission rate for specific group (field *unsigned long cc_trx_rate* of *struct FlowControlInfo*), which is a minimum over all such rate limit values advertised to local sender delegate by remote receiver delegates.

- *total_data_rcvd*: cumulative total data size (in bytes) received from specific remote LAN since $last\_loss\_event\_time + RTT$. Used for average message size calculation (average message size is one of the parameters required in equation 4.1).
  - initialized to zero in file **src/delegate/data_struct/groupds.c**, function *gr_add_lan()* when local delegate joins the group.
  - updated in file **src/delegate/core/message_store_service.c**, function *mss_put()* every time in-order data message arrives from specific remote LAN. Increased by the message size.
  - used in file **src/delegate/core/bound_protocol.c**, function *build_bound_entries_for_remote_subnet()* to calculate an average message size (needed to calculate a TCP-friendly rate).
- *double rate_adv_expiration*: time left until specific rate limit advertisement expires.
  - set in file **src/delegate/core/message_completion_service.c**, function *mcs_process_bound_message()* to the maximal value *now time + FC_RATE_ADVERTISEMENT_EXPIRATION_PERIOD* (defined in **src/delegate/core/flow_control.h**), every time new rate advertisement arrives from a remote receiver.
  - inspected in file **src/delegate/core/flow_control.c**, function *check_expired_advertisements()* every *flow control cycle (FC_CYCLE_DURATION ms)*. When *current time* reaches *double rate_adv_expiration* value, corresponding rate limit advertisement expires. Expiration implies WAN congestion, therefore *unsigned long cc_trx_rate* (rate limit imposed by WAN conditions) is set to the minimum value *FC_MINIMAL_RATE* (defined in **src/delegate/core/flow_control.h**).
  - *unsigned long saved_effective_trx_rate*: effective transmission rate, minimum of the WAN imposed limit (due to *congestion control*) and LAN imposed limit (due to *flow control*): $\min(fc\_trx\_rate, cc\_trx\_rate)$.
    - initialized in **src/delegate/core/flow_control.c**, function *fc_init()* to the minimum value *FC_MINIMAL_RATE* (defined in **src/delegate/core/flow_control.h**).
    - updated in **src/delegate/core/flow_control.c**, function *check_rate_changed()* at the end of every flow control round (*FC_CYCLE_DURATION* ms).
    - used in **src/delegate/core/flow_control.c**, function *check_rate_changed()* to recognize a change in transmission rate limit for a group. Every time rate limit changes, the rate distribution algorithm is run to partition the new rate (function *frp_distribute_rate()* is called from *check_rate_changed()*, file **src/delegate/core/flow_control.c**).
  - *double rate_demand[MAX_HOST_NUM]*: vector of rate demands from local regular members (local group senders). A local group member's demand is indexed by its *Internal_EndPoint_Id*. An entry is zero unless corresponding endpoint is a local group member *AND* has reported its rate demand at least once.
    - initialized (zeroed) in **src/delegate/core/flow_control.c**, function *fc_init()*.
    - updated in **src/delegate/core/flow_control.c**, function *fc_process_rate_request_message()* upon arrival of rate demand update from one of local senders.

  - · corresponding entry is unset (zeroed) in file **src/delegate/core/member_state_machine.c**, function *member_leaves_group()* every time specific local regular member leaves the group.
    - · used in **src/delegate/core/flow_control.c**, function *check_rate_changed()* to recognize a change in transmission rate limit for a group. Every time rate limit changes, the rate distribution algorithm is run to partition the new rate (function *frp_distribute_rate()* is called from *check_rate_changed()*, file **src/delegate/core/flow_control.c**).
    - · used in **src/delegate/core/flow_control.c**, function *fc_process_rate_request_message()* to recompute rate distribution (function *frp_distribute_rate()* is called from *fc_process_rate_request_message()*, file **src/delegate/core/flow_control.c**) every time one of the local senders updates its rate demand (i.e. *double rate_demand[MAX_HOST_NUM] changes*).
  - ∗ *double rate_partition[MAX_HOST_NUM]*: vector of aggregate rate partition between all local senders, calculated by *frp_distribute_rate()*. A local group member's rate share is indexed by its *Internal_EndPoint_Id*.
    - · initialized (zeroed) in **src/delegate/core/flow_control.c**, function *fc_init()*.
    - · updated in **src/delegate/core/fair_rate_partition.c**, function *frp_distribute_rate()* when either rate limit for the group changes (after a *BOUND* message arrival with advertised rate limit less than current global minimum) or vector of rate demands (*double rate_demand[MAX_HOST_NUM]*) changes (after a *RATE REQUEST* message arrival from local sender with update of rate demand).
    - · used in **src/delegate/core/flow_control.c**, function *fc_send_rate_partition()* to deliver (with *RATE REPLY* messages) corresponding rate share to each local sender.

- File **src/delegate/data_struct/group_info.h**, *struct MemberInfo*: new fields were added to support loss rate estimation:
  - – *Sequence_Number latest_received_packet*: maximal sequence number seen so far from specific sender for specific group.
    - ∗ initialized in file **src/delegate/data_struct/groupds.c**, function *gr_add_member()* to *SN_PREDECESSOR(0)* whenever a new member joins the group.
    - ∗ updated in file **src/delegate/core/message_store_service.c**, function *mss_insert_message()* when a message arrives with sequence number greater than seen so far.
    - ∗ used in file **src/delegate/core/message_store_service.c**, function *update_loss_rate()* for interpolation of nominal arrival time of a first lost message.
  - – *double latest_received_packet_timestamp*: time when last message has been received from specific sender for specific group. Used for interpolation of nominal arrival time of a first lost message.
    - ∗ initialized to zero in *gr_add_member()*, file **src/delegate/data_struct/groupds.c**.
    - ∗ updated in *mss_put()*, file **message_store_service.c** every time a data message from local sender arrives.

    - used in file **src/delegate/core/message_store_service.c**, function *update_loss_rate()* for interpolation of nominal arrival time of a first lost message.
- *double last_message_sent_timestamp*: timestamp (in milliseconds) of last message sent by this regular member. Used to recognize long inactivity periods.
    - initialized to *now time* in *gr_add_member()*, file **src/delegate/data_struct/groupds.c**.
    - updated in *mss_put()*, file **message_store_service.c** every time a data message from local sender arrives.
    - inspected in file **src/delegate/core/flow_control.c**, function *check_local_senders_inactivity()*. If *all* members are inactive during *FC_MAX_INACTIVITY_PERIOD* (defined in file **src/delegate/core/flow_control.h**) and there is no remote members, than flow control phase switches to *FC_SLOW_START* for specific group.
- function *update_loss_rate()* (file **message_store_service.c**) implements loss detection and loss rate measurements as well as transmission rate calculations. Every time a data message arrives, it is passed to the *mss_put()* function, which calls *update_loss_rate()*. According to the message sequence number we know whether it has arrived out of order. Given *RTT* provided by sender delegate, receiver delegate can determine whether the lost message starts new loss event. If it does, loss event rate is recalculated.

10.6.1 Handling Retransmitted Messages

According to the model,

- retransmitted message has its own sequence number and is considered as independent message in the message flow
- out-of-order messages should not start loss event (they should be reordered).

In current implementation of Xpand

- retransmitted message has the same sequence number as the original message.
- no effort is made to reorder out-of-order messages (any out-of-order message triggers message loss processing)

If retransmitted message arrives *after* the original message has arrived, loss event should not be triggered (as if the 'hole' has been eliminated).

If retransmitted message arrives *before* the original message, it means that either message reordering or message loss occurred. Since Xpand does not support message reordering, arrival of retransmitted message before its original is treated as a message loss.

If retransmitted message has a sequence number **within** a 'hole' this loss is considered to be a part of the recent loss event.

If retransmitted message arrives **out of order**, we handle it the same way as non retransmitted message. In this case, we cannot give good estimation of nominal lost message arrival time. For nothing better, we use the arrival time of the retransmission message as the estimation.

If retransmitted message arrives **in order**, we still handle this event as possible loss, since arrival of retransmitted message implies loss of original message.

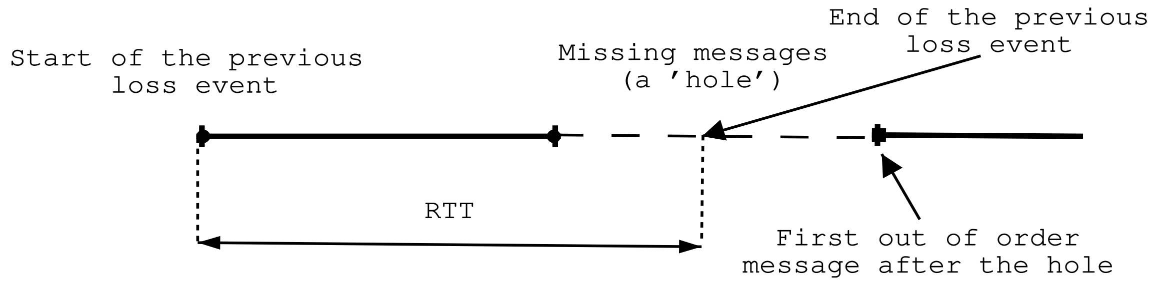


Figure 10.3: Sequence of lost messages ('a hole') starting within the previous loss event

Consider a situation when a message that should start new loss event is in the middle of a 'hole' (i.e. *previous loss event time + rtt* falls within a run of lost messages, see fig. 10.3). Although we can interpolate to obtain sequence number of the message that starts the loss event, there is no point in starting new loss event since probably no messages arrived within [*previous loss event + rtt, new loss event*]. We don't start new loss event unless the first message in a hole is at least RTT sec apart from the previous loss event.

10.6.2 Future Work

Packet reordering should be properly handled. Arrival of out-of-order message should not trigger message loss processing immediately: missing messages might just be reordered and will arrive in a while.

Similar improvement should be done for the *NACK protocol*: we should not generate a *NACK* immediately, but wait for a while in a hope the missing packet has been reordered and will arrive intact soon.

10.7 AIMD Flow Control in a LAN

In addition to the equation-based congestion control in a WAN environment, Xpand deploys a TCP-like AIMD flow control within a *LAN*. These two mechanisms impose two independent limits on sender transmission rate:

- *unsigned long cc_trx_rate*, defined in **src/delegate/core/flow_control.h**, is a rate limit imposed by WAN conditions (message loss). This is a minimum over all rate limits advertised by remote receivers.
  - initialized to *FC_UNLIMITED_RATE* in file **src/delegate/core/flow_control.c**, function *fc_init()*. *FC_UNLIMITED_RATE* is defined in **src/delegate/core/flow_control.h** (4.3 Gbit/sec) and the intention is to allow any transmission rate, as high as needed.
  - updated in file **src/delegate/core/flow_control.c**, function *check_expired_advertisements()* when rate limit advertisement from remote *LAN* expires. In this case we assume network congestion and set *unsigned long cc_trx_rate* to be min($FC\_MINIMAL\_RATE$, cc_trx_rate).
  - updated in file **src/delegate/core/flow_control.c**, function *check_rate_changed()*, where a minimum over all rate limits from remote receivers is recomputed periodically.

  - updated in file **src/delegate/core/lan_state_machine.c**, function *enable_delegate()*, where it is set to *FC_MINIMAL_RATE* when first remote member joins the group.
  - updated in file **src/delegate/core/member_state_machine.c**, function *member_leaves_group()*, where *unsigned long cc_trx_rate* is set to *FC_UNLIMITED_RATE* when last remote member leaves the group.
  - used in file **src/delegate/core/flow_control.c**, function *check_rate_changed()* to compute effective WAN-wide transmission rate limit for specific group (as $\min(fc\_trx\_rate,\ cc\_trx\_rate)$).
    * this value is saved in *unsigned long saved_effective_trx_rate* (defined in **src/delegate/core/flow_control.h**) each time function *check_rate_changed()* is called.
    * *unsigned long saved_effective_trx_rate* is used in functions *check_rate_changed()* and *fc_process_rate_request_message()* when running rate partitioning algorithm via call to function *frp_distribute_rate()*.

- *unsigned long fc_trx_rate*, defined in **src/delegate/core/flow_control.h**, is a rate limit imposed by *LAN* conditions (*intra-LAN NACK*s).
  - initialized to *FC_MINIMAL_RATE* in file **src/delegate/core/flow_control.c**, function *fc_init()*. *FC_MINIMAL_RATE* is defined in **src/delegate/core/flow_control.h** (32 Kbit/sec).
  - updated in file **src/delegate/core/flow_control.c**, function *check_intra_lan_nack()*, where *unsigned long fc_trx_rate* is halved upon *intra-LAN NACK* occurrence.
  - updated in file **src/delegate/core/flow_control.c**, function *check_intra_lan_nack()*, where *unsigned long fc_trx_rate* is doubled if no *intra-LAN NACK* has occurred during a *flow control cycle* (*FC_CYCLE_DURATION* ms, time period between two consecutive invocations of *fc_init()*) and if the flow control phase is *FC_SLOW_START*.
  - updated in file **src/delegate/core/flow_control.c**, function *check_intra_lan_nack()*, where *unsigned long fc_trx_rate* is increased by *FC_MINIMAL_RATE* if no *intra-LAN NACK* has occurred during a *flow control cycle* and if the flow control phase is *FC_CONGESTION_AVOIDANCE*.
  - used in file **src/delegate/core/flow_control.c**, function *check_rate_changed()* to compute effective WAN-wide transmission rate limit (*unsigned long saved_effective_trx_rate*) for specific group (as $\min(fc\_trx\_rate,\ cc\_trx\_rate)$).

Intra-LAN flow control mechanism works in cycles (*FC_CYCLE_DURATION* is defined in src/delegate/core/flow_control.h). At the end of each cycle we check whether an *intra-LAN NACK* has occurred. A *NACK* is *intra-LAN* if both the *NACK* originator and the *NACK* addressee reside on the same subnet. Occurrence of an *intra-LAN NACK* causes reduction of fc_trx_rate to the half of its previous value. Lack of *intra-LAN NACK*s allows either linear increase (in the *congestion_avoidance phase*), or doubling of fc_trx_rate (in the *slow_start phase*).

- function *fc_cycle_handler()*, file **src/delegate/core/flow_control.c**, handles flow control cycle bookkeeping. It is registered at the event scheduler in function *datastruct_init()*, file **src/delegate/core/core_delegate.c** to be called every *FC_CYCLE_DURATION* ms.

- function *mcs_process_nack()*, file **message_completion_service.c**, recognizes *intra-LAN NACK* occurrence.
- macro *ST_GET_RECEIVERS_AMOUNT(st)*, file **src/delegate/data_struct/summary_table.h**, says how many remote delegates are members of the group (to recognize the situation when there are only local group members).
- macro *NO_REMOTE_MEMBERS(gr)*, file **src/delegate/core/flow_control.h**, says whether a group has only local members (actually uses *ST_GET_RECEIVERS_AMOUNT(st)* macro)
- file **src/delegate/core/flow_control.h**, *struct FlowControlInfo*. Per group data structure for flow control support:
  - *unsigned long intra_LAN_NACK:1*: *intra-LAN NACK* occurrence flag.
    - initialized (cleared) in *fc_init()*, file **src/delegate/core/flow_control.c**
    - cleared at the beginning of each cycle in *check_intra_lan_nack()*, file flow_control.c
    - set in *mcs_process_nack()*, file src/delegate/core/message_completion_service.c every time *intra LAN NACK* is detected
    - examined in *check_intra_lan_nack()*, file **src/delegate/core/flow_control.c** at the end of every *flow control cycle* to decide whether to increase or to decrease *unsigned long fc_trx_rate*.
  - *enum { FC_SLOW_START, FC_CONGESTION_AVOIDANCE } phase*: flow control phase. Either *slow start* or *congestion avoidance*.
    - initialized in *fc_init()* to *FC_SLOW_START*.
    - set to *FC_SLOW_START* in *check_local_senders_inactivity()* when all local members have been silent for a while, and there is no remote members.
    - set to *FC_CONGESTION_AVOIDANCE* in *check_intra_lan_nack()* when *intra-LAN NACK* occurs.
    - set to *FC_SLOW_START* in *check_intra_lan_nack()* when no *intra-LAN NACK* occurs during a *flow control cycle AND* $fc\_trx\_rate \leq cc\_trx\_rate$) *AND* ($fc\_trx\_rate < aggregate\_rate\_demand$).
    - set to *FC_CONGESTION_AVOIDANCE* in *check_intra_lan_nack()* when no *intra-LAN NACK* occurs during a *flow control cycle AND* $fc\_trx\_rate > cc\_trx\_rate$).
    - set to *FC_CONGESTION_AVOIDANCE* in **src/delegate/core/lan_state_machine.c**, function *enable_delegate()*, when first remote member joins the group.
    - set to *FC_SLOW_START* in **src/delegate/core/member_state_machine.c**, function *member_leaves_group()*, when last remote member leaves the group.
- file **src/delegate/data_struct/group_info.h**, *struct GroupInfo*. New field is introduced: *Active_List *local_regular_members*, list of all local regular group members. Used in the flow control module to check local senders inactivity (by inspecting the *last_message_sent_timestamp* field of each member's *struct MemberInfo*).
  - initialized in file **src/delegate/data_struct/groupds.c**, function *gr_join()*.
  - a member is inserted in **src/delegate/core/member_state_machine.c**, function *enable_local_regular()*
  - deletion: file **src/delegate/core/member_state_machine.c**, function *member_leaves_group()*.

10.8 Intra-LAN Reliable Channel

Within a *LAN* regular senders and a local delegate exchange messages related to the congestion control algorithm:

- a regular member sends periodically its rate request to a local delegate.
- local delegate provides local senders with rate share accordingly to their demands.

We use Xpand infrastructure to deliver these messages reliably.

10.8.1 General

- in file **src/common/message.h**, new message types are defined:
  - *MS_RATE_REQUEST*: is sent by a regular sender to local delegate to update rate demand.
  - *MS_RATE_REPLY*: is multicast by a delegate to all local group members to deliver rate share of each sender. Every message refers to one specific group.
- in file **src/common/message.h** data structures for the new message types are defined:
  - *struct MS_RATE_REQUEST_MESSAGE*. Contains fields for:
    - group id (multicast IP address)
    - sender id (IP address and port)
    - sender's rate demand
  - *struct MS_RATE_REPLY_MESSAGE*. Contains fields for:
    - group id
    - number of trailing *MS_RATE_REPLY_ENTRY*'s
  - *struct MS_RATE_REPLY_ENTRY*. Refers to specific group sender. Contains fields:
    - sender id (IP address and port)
    - sender's rate share

10.8.2 Delegate Side

- every time application rate changes enough, a regular sends to local delegate rate request message
- delegate accepts a rate demand update, recalculates rate distribution and multicasts updated rate shares to all local senders. Function calls chain:
  - *pds_read()* reads a rate request message from a socket
  - *pds_dispatch()* primary processing of an arrived message
  - *ms_ntoh_header()* converts a message common header to host byte order
  - *ms_ntoh_message()* converts a message to host byte order

- new function *ms_ntoh_rate_request_message()* (file **src/common/message.c**) converts a *MS_RATE_REQUEST* message to host byte order
- *pds_dispatch_control_message()* primary processing of a control message
- new function *fc_process_rate_request_message()* (file **src/delegate/core/flow_control.c**) stores rate demand update from particular group sender.
- new function *frp_distribute_rate()* (file **src/delegate/core/fair_rate_partition.c**) runs rate distribution algorithm to recalculate aggregate rate share for all active local senders of the group
- new function *fc_send_rate_partition()* (file **src/delegate/core/flow_control.c**) builds and sends *MS_RATE_REPLY* message to all local group senders.
  - uses reliable delegate-to-LAN channel. Actual sending is done with *d2l_send()* function, which multicasts a message to all local members of specified group.

10.8.3 Regular Side

- every time application rate changes enough for some group, a regular sender delivers to the delegate rate request message for the group
  - we say that a rate changed *enough* if $\frac{|old\_rate - new\_rate|}{old\_rate} \geq min\_rate\_change, min\_rate\_change \in (0.0...1.0)$, where *double min_rate_change* is defined in file **regular/data_struct/parameterds.h**.
    - Minimal relative change in smoothed application transmission rate which allows a regular to send a rate request update message to local delegate.
  - in function *appl_rate_changed_enough()* (file **src/regular/core/congestion_control.c**) if smoothed application rate change amount is large enough, rate request message delivery is initiated:
    - function *cc_report_new_rate_demand* (file **src/regular/core/congestion_control.c**) builds *MS_RATE_REQUEST_MESSAGE* and calls *cc_reliable_channel_send()* to actually send the message to the delegate using the intra-LAN reliable channel.
- every time *MS_RATE_REPLY_MESSAGE* arrives from local delegate for some group, a regular updates its transmission rate limit for the group accordingly
  - *cc_process_rate_reply_message()* (called from *dispatch_control_message()*, called from *pds_dispatch()*, called from *pds_read()*).

Bibliography

[1] V. Jacobson, "Congestion avoidance and control," *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, vol. 18, 4, pp. 314–329, 1988.

[2] B. Braden et al, "Recommendations on queue management and congestion avoidance in the internet," *RFC 2309*, p. 16, 1998.

[3] S. Floyd and K. Fall, "Router mechanisms to support end-to-end congestion control," February 1997.

[4] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: A Communication Sub-System for High Availability," in *22nd Annual International Symposium on Fault-Tolerant Computing*, July 1992, pp. 76–84.

[5] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos, "Totem: A Fault-Tolerant Multicast Group Communication System," *Communications of the ACM*, vol. 39, no. 4, April 1996.

[6] R. van Renesse, K. P. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System," *Communications of the ACM*, vol. 39, no. 4, April 1996.

[7] M. Hayden, *The Ensemble System*, Phd thesis, Cornell University, Computer Science, 1998.

[8] Y. Amir, C. Danilov, and J. Stanton, "A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication," in *Proceedings of ICDSN'2000*, 2000.

[9] Gregory Chockler, Idit Keidar, and Roman Vitenberg, "Group communication specifications: a comprehensive study," *ACM Computing Surveys*, vol. 33, no. 4, pp. 427–469, 2001.

[10] T. Anker, G. Chockler, I. Shnaiderman, and D. Dolev, "The Design and Performance of Xpand: A Group Communication System for Wide Area Networks," Tech. Rep. 2001-56, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, August 2001, URL: http://leibniz.cs.huji.ac.il/research/, See also the previous version TR2000-31.

[11] M. Handley, "Tcp friendly rate control (tfrc): Protocol specification," RFC 3448, Internet Engineering Task Force, Network Working Group, January 2003.

[12] Jitendra Padhye, Victor Firoiu, Donald F. Towsley, and James F. Kurose, "Modeling TCP Reno performance: a simple model and its empirical validation," *IEEE/ACM Transactions on Networking*, vol. 8, no. 2, pp. 133–145, April 2000.

[13] Jorg Widmer and Mark Handley, "Extending equation-based congestion control to multicast applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. 2001, pp. 275–286, ACM Press.

[14] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002, USENIX Association, pp. 255–270, url: www.emulab.net.

[15] ACM, *Communications of the ACM 39(4), special issue on Group Communications Systems*, April 1996.

[16] Shivakant Mishra and Lei Wu, "An evaluation of flow control in group communication," *IEEE/ACM Transactions on Networking (TON)*, vol. 6, no. 5, pp. 571–587, 1998.

[17] Sally Floyd and Kevin Fall, "Promoting the use of end-to-end congestion control in the Internet," *IEEE/ACM Transactions on Networking*, vol. 7, no. 4, pp. 458–472, 1999.

[18] S. Floyd, "Congestion Control Principles," RFC 2914, Internet Engineering Task Force, Network Working Group, September 2000.

[19] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control," RFC 2581, Internet Engineering Task Force, Network Working Group, April 1999.

[20] ACM, *Communications of the ACM 39(4), special issue on Group Communications Systems*, April 1996.

[21] D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communication," *Communications of the ACM*, vol. 39, no. 4, April 1996.

[22] M. F. Kaashoek and A. S. Tanenbaum, "An evaluation of the Amoeba group communication system," in *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996, pp. 436–447.

[23] John Byers, Michael Frumin, Gavin Horn, Michael Luby, Michael Mitzenmacher, Alex Roetter, and William Shaver, "FLID-DL: Congestion Control for Layered Multicast," in *Second Int'l Workshop on Networked Group Communication (NGC 2000)*, November 2000.

[24] Lorenzo Vicisano, Luigi Rizzo, and Jon Crowcroft, "TCP-like congestion control for layered multicast data transfer," in *INFOCOM (3)*, 1998, pp. 996–1003.

[25] Steven McCanne, Van Jacobson, and Martin Vetterli, "Receiver-driven layered multicast," in *ACM SIGCOMM*, New York, Aug. 1996, vol. 26,4, pp. 117–130, ACM Press.

[26] Jitedra Padhye, Victor Firoiu, Don Towsley, and Jim Krusoe, "Modeling TCP throughput: A simple model and its empirical validation," in *ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, Vancouver, CA, 1998, pp. 303–314.

[27] Todd Montgomery, "A loss tolerant rate controller for reliable multicast," August 1997.

[28] Supratik Bhattacharyya, Donald F. Towsley, and James F. Kurose, "The loss path multiplicity problem in multicast congestion control," in *INFOCOM (2)*, 1999, pp. 856–863.

[29] M. Kadansky, D. Chiu, J. Wesley, and J. Provino, "Tree-based reliable multicast (tram)," January 2000.

[30] I. Rhee, V. Ozdemir, and Y. Yi, "Tear: Tcp emulation at receivers – flow control for multimedia streaming," 2000.

[31] Huayan Amy Wang and Mischa Schwartz, "Achieving bounded fairness for multicast and TCP traffic in the internet," in *SIGCOMM*, 1998, pp. 81–92.

[32] James F. Kurose S. Bhattacharyya, Donald F. Towsley, "A novel loss indication filtering approach for multicast congestion control," 2000.

[33] Injong Rhee, Nallathambi Ballaguru, and George N. Rouskas, "MTCP: Scalable TCP-like congestion control for reliable multicast," in *INFOCOM*. IEEE, Mar. 1999.

[34] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson, "TCP vegas: New techniques for congestion detection and avoidance," in *SIGCOMM*, 1994, pp. 24–35.

[35] Sneha Kumar Kasera, Gísli Hjálmtýsson, Donald F. Towsley, and James F. Kurose, "Scalable reliable multicast using multiple multicast channels," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 294–310, 2000.

[36] Luigi Rizzo, "pgmcc: a TCP-friendly single-rate multicast," in *SIGCOMM*, 2000, pp. 17–28.

[37] Sanjoy Paul, K. Sabnani, J.C. Lin, and S. Bhattacharyya, "Reliable multicast transport protocol (RMTP)," *IEEE Journal on Selected Areas in Communications*, April 1997.

[38] Rajendra Yavatkar, James Griffoen, and Madhu Sudan, "A reliable dissemination protocol for interactive collaborative applications," in *ACM Multimedia*, 1995, pp. 333–344.

[39] Y. Yang and S. Lam, "General aimd congestion control," 2000.

[40] Yair Amir, Baruch Awerbuch, Claudiu Danilov, and Jonathan Stanton, "Global flow control for wide area overlay networks: A cost-benefit approach," in *OPENARCH 2002*, June 2002, pp. 155–166.

[41] T. Anker, D. Dolev, and I. Shnaiderman, "Ad Hoc Membership for Scalable Applications," in *16th Intl. Conference on Distributed Computing Systems*, October 2002.

[42] D. P. Bertsekas and R. G. Gallager, *Data Networks*, Prentice-Hall, 1992.

[43] Sally Floyd, Mark Handley, Jitendra Padhye, and Jorg Widmer, "Equation-based congestion control for unicast applications," in *SIGCOMM 2000*, Stockholm, Sweden, August 2000, pp. 43–56.

[44] Hewlett-Packard Company., *Netperf: A Benchmark for Measuring Network Performance*, May 1993-1996, URL: http://www.netperf.org/.