Congestion Control in a Reliable Scalable Message-Oriented Middleware

Peter R. Pietzuch[1]* and Sumeer Bhola[2]

[1] University of Cambridge Computer Laboratory,
Cambridge, United Kingdom
Peter.Pietzuch@cl.cam.ac.uk
[2] IBM T.J. Watson Research Center,
Hawthorne, NY, USA
sbhola@us.ibm.com

Abstract. This paper presents congestion control mechanisms for reliable and scalable message-oriented middleware following the publish/subscribe communication model. We identify the key requirements of congestion control in this environment, how it differs from congestion control for the Internet, and propose a combination of two congestion control mechanisms, (1) driven by a publisher hosting broker (PDCC), (2) driven by a subscriber hosting broker (SDCC). SDCC decouples the notion of a receive window and a NACK window, and is used by subscriber hosting brokers in recovery mode. PDCC implements a scalable and low latency feedback loop between a publisher hosting broker and all subscriber hosting brokers, which is used to adjust the rate of publishing new messages, to allow brokers in recovery to eventually catch up, and other brokers to keep up. We present a detailed experimental evaluation of our implementation of these mechanisms in the Gryphon system by injecting network failures and link congestion.

1 Introduction

Message-oriented middleware is an important building block for enterprise applications. Its asynchronous model is preferable to tight synchronization, for application integration and information dissemination to many users. Reliability of message delivery, despite failures in the messaging middleware, and scalability of the middleware, are important requirements for many such applications.

This paper focuses on congestion control mechanisms for reliable and scalable messaging middleware. Our mechanisms are developed in the context of a publish/subscribe messaging model, since it is a challenging model due to its one-to-many nature and message filtering semantics. Subscribers express interest in messages by providing a predicate/filter, that can be executed on each message, and only messages that match the filter are delivered to the subscriber. Scalable messaging middleware is deployed as an overlay network of application-level routers, which we refer to as *brokers*. Typically, multiple routing paths are

* Work done while at IBM T.J. Watson Research Center.

ACM/IFIP/USENIX Int. Middleware Conference 2003.

possible between a pair of brokers, and the routing protocol used in the overlay both (1) balances the load amongst the available paths, and (2) routes around failed paths, for high availability. However, the overlay routing protocol cannot ensure that the system has enough capacity, say after some failure, to continue processing messages at the rate at which they are being published. In addition, messages lost due to failures need to be resent, which requires more capacity for retrieving them (potentially from stable storage), processing them at intermediate brokers, and network bandwidth for sending them on each network link.

Congestion control has been an active research area for Internet protocols, especially for point-to-point communication using TCP. There is also work that addresses end-to-end congestion control for IP multicast, such as using layered multicast [1], or single-rate schemes that adjust the sender's rate to the slowest receiver [2, 3]. However, these schemes are for best-effort delivery, and assume that all receivers get the same set of messages. In contrast, for reliable publish/-subscribe, (1) the congestion control mechanism needs to ensure that the overlay network allocates enough capacity for message retransmission to satisfy receivers who are lagging behind, (2) message filtering may occur at intermediate brokers, due to which each receiver may get a different subset of messages from the sender, and (3) message retransmissions, to overcome losses, can originate not only at the sender, but from caches located on the path from the sender to the receiver.

There are also important differences in the system environment compared with Internet protocols running on IP routers, such as:

1. Message processing is bursty due to (1) application-level scheduling, since brokers run on commercial off-the-shelf (COTS) hardware and software, and (2) variable processing cost of performing content filtering on a message.
2. The ratio of maximum queue size at a node, to the message processing throughput, is typically much higher for a broker compared to an Internet router. This is due to (1) lower routing throughput because of content filtering, and (2) larger queues to handle burstiness. Hence queue overflows occur only when significant congestion already exists. Since the congestion control mechanism should not allow significant congestion to build up, it cannot use message loss caused by queue overflows as a trigger for congestion control.
3. Broker software and inter-broker routing protocols are completely under the control of the messaging vendor. Hence, we are not constrained to a congestion control scheme which treats the entire overlay network as a black-box. At the same time, the congestion control scheme should not depend intimately on a particular broker architecture, since that would hinder congestion control being part of a future protocol standard for publish/subscribe routing.

1.1 System Model

We assume a model where brokers can perform 3 roles (1) *publisher hosting broker* (PHB) is an edge of the network broker to which publishing clients connect, (2) *subscriber hosting broker* (SHB) is an edge broker to which subscribing

clients connect, and (3) *intermediate broker* (IB) is a broker which is inside the network and does not host clients. Brokers can perform multiple roles, but for simplicity of exposition we will assume that each broker has only one.

Each PHB hosts one or more publishing endpoints, referred to as *pubends*. Each pubend represents an ordered stream of messages, and maintains this stream in persistent storage. Messages published from different publishing clients may be assigned to the same pubend. This pubend decides on a position for the message in the persistent stream and logs the message to persistent store. After that, the pubend sends the message downstream towards SHBs. The IBs forward data and control messages to the SHBs, and may also perform filtering on data messages such that an SHB does not receive messages that do not match any of its subscribers' filters. One approach to performing such filtering, while preserving the guarantee of in-order exactly-once delivery to subscribers, is described in our previous work [4]. IBs also cache stream data and can respond to NACKs. NACKs not satisfied by an IB are forwarded towards the pubend.

The congestion control protocols presented here deal with congestion starting from the pubend upto and including the SHBs that receive messages from the pubend. The goal is to ensure that eventually all SHBs are being able to receive and deliver in-order *recent* messages from a pubend's stream. Hence, the protocols adjust the message rate to the slowest link or broker inside the system, but not to the slowest subscribing application. This design choice allows the system to protect itself against very slow or malicious subscribers by disconnecting them. The number of brokers is typically 3–4 orders of magnitude lower than the number of clients. We assume that the brokers are trusted.

We perform adaptation at both PHBs and SHBs to control congestion, such that broker queues do not overflow. Adaptation at an SHB is done by

- decoupling the notion of a NACK window and a receive window, where the receive window is used to bound the memory consumption at an SHB and the NACK window is used for congestion control.
- using a NACK throughput metric (similar to the send throughput metric in TCP Vegas [5]) to adjust the NACK window. Adjustment is additive increase and additive decrease. The NACK throughput adjusts to NACK responses received from the pubend or intermediate caches at IBs.

Adaptation at a PHBs is done using explicit feedback from the SHBs. Lack of feedback is not used as a trigger for congestion control since it can give false positives if some SHBs are down or partitioned from the rest of the network.

- The messages in a pubend stream are assigned timestamps based on real-time, and the message rate at SHBs is computed in terms of these timestamps. This normalizes the rate at different SHBs regardless of which subset of messages they receive, and the current pubend rate.[3]

[3] The brokers do not need to have synchronized clocks, but the difference in clock rates should be small.

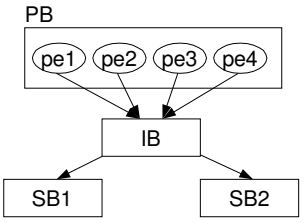

Fig. 1. A simple broker topology

- The protocol distinguishes between SHBs that are in recovery and those not in recovery, and scalably monitors the slowest SHB rate for both recovery and non-recovery using high-priority congestion-alert messages that get consolidated at intermediate brokers. The target rate for SHBs in recovery mode is such that they will eventually catch up to the current time.
- The PHB shapes the rate of new messages sent by a pubend when congestion occurs, using an additive increase and hybrid (multiplicative and additive) decrease that accounts for burstiness in the overlay network.

These protocols have been implemented in the context of the Gryphon system [6, 7, 4], which supports highly-scalable content-based publish/subscribe. The Gryphon system has been deployed for information dissemination in a wide-area environment, such as score updates for tennis Grand Slam events, and the Sydney Olympics, for tens of thousands of concurrently connected subscribers.

The paper is organized as follows: Section 2 illustrates the congestion problem using some examples of congestion collapse. Section 3 describes our congestion control protocols, and Sect. 4 discusses their implementation in the Gryphon broker. Section 5 describes experimental results that show the effectiveness of our protocols. Section 6 describes related work and we conclude in Sect. 7.

2 The Congestion Control Problem

In this section, we illustrate the congestion control (cc) problem in publish/-subscribe overlay networks by showing two instances of congestion collapse in Gryphon without any congestion control mechanism. This helps to characterize the points of congestion, and motivates the design presented in the next section.

Figure 1 shows a simple overlay network with 1 publisher hosting broker, PB, that hosts 4 pubends, connected to 2 SHBs, SB1 and SB2, through an intermediate broker IB. In this example there is only 1 path from PB to SB1, but in general there could be multiple paths. For instance, the Gryphon system organizes the overlay network into trees of cells [4], where each cell can have multiple equivalent brokers which balance the load and provide lightweight failover. The load balancing is accomplished by routing messages from different pubends on different paths through the same cell. We allow for the path from a pubend to a particular SHB to change, but assume that paths change infrequently.

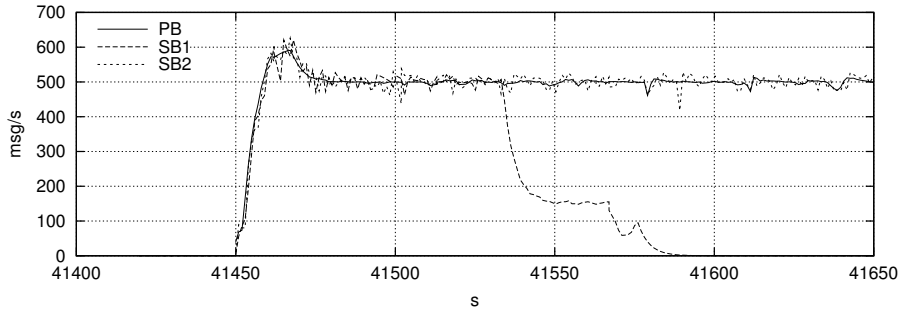

Fig. 2. Collapse with IB-SB1 link restricted to 60 KB/s

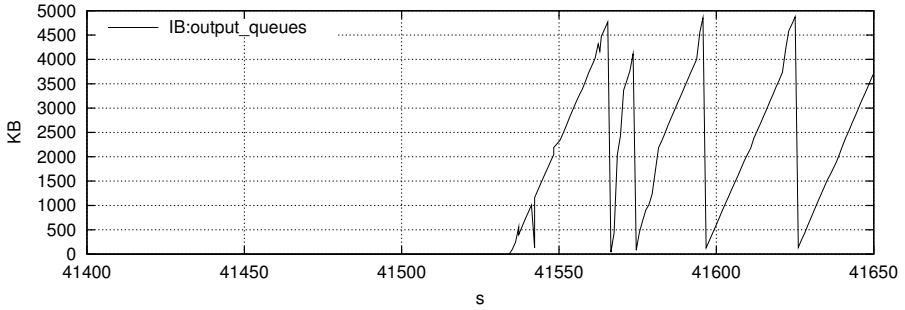

Fig. 3. Queue utilization at broker IB during collapse

In Gryphon, links between brokers are implemented using TCP connections, which means that there is a congestion control protocol running on each link. This is not sufficient to prevent congestion collapse in the system as a whole. We illustrate this for a simple overlay using two examples. In both examples, the PB is handling an aggregate publish rate of 500 msg/s, split over the 4 pubends, where each message has a message header and carries a 100 byte application payload. The trivial filter of 'true' is used on all inter-broker links, i.e., all data messages need to be eventually received at both SB1 and SB2. Each subscriber receives 2 msg/s and there are 250 subscribers each at SB1, SB2, which is an aggregate subscriber rate of 500 msg/s at each SHB. This is a modest rate, and the CPU at the SHBs is 95% idle in the steady-state.

In Fig. 2, after running the system without any congestion, we throttle the bandwidth on the IB-SB1 link to 60 KB/s[4]. The X-axis is elapsed time in seconds, and the Y-axis shows the aggregate smoothed rate for the pubends (PB) and the subscribing applications at SB1 and SB2 respectively. With no congestion control the pubends continue accepting published messages, and sending them, at the same rate. This causes queues to build up at IB (cf. Fig. 3) and eventually overflow which causes message loss. Lost messages need to be recov-

[4] KB/s stands for (binary) Kilo Bytes per second.

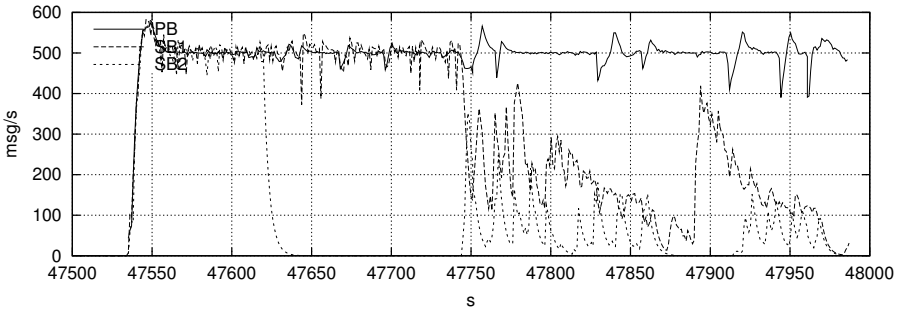

Fig. 4. Collapse with PB-IB link restricted to 250 KB/s

ered (using NACKs) before delivering later messages, but many of the messages retransmitted due to NACKs also get lost. The result is that the aggregate rate for subscribers connected to SB1 drops to zero.

In Fig. 4, the bandwidth on the PB-IB link is restricted to 250 KB/s. In the absence of failure, this does not cause congestion. We then fail the link IB-SB2 for 120 s. When the link comes back, the NACKs initiated by SB2 cause congestion on the PB-IB link and the rate for subscribers at both SB1 and SB2 drops much below the rate at which pubends are sending new messages, and never recovers.

These examples demonstrate how congestion on a network link can cause collapse. Congestion at an IB or SHB, due to not having enough CPU capacity, can also appear to be network congestion to an upstream broker since its outgoing queues will build up. However, CPU congestion at a PHB due to the overhead of processing NACKs may not readily appear as queue buildup, since NACKs are small and each NACK message can request retransmission of a large number of data messages. We want to control all three kinds of congestion (1) on a network link, (2) at an IB or SHB, (3) at the PHB, while ensuring that all SHBs that are trying to recover messages they missed due to failure are eventually successful.

3 Congestion Control Protocols

In this section, we present our congestion control mechanism in detail. The solution consists of two parts: (1) The *PHB-driven cc protocol (PDCC)* ensures that pubends do not cause congestion due to a too high publication rate. A feedback loop between pubends and SHBs with downstream and upstream messages is created to monitor congestion in the broker network. (2) The *SHB-driven cc protocol (SDCC)* handles the rate at which SHBs try to recover after a failure. This protocol only involves the SHBs by monitoring their recovery rate. These two congestion control mechanisms can be used independently from each other. Both protocols need to distinguish between recovering and non-recovering SHBs in order to ensure that SHBs will eventually manage to recover successfully.[5]

[5] Informally, an SHB is recovering when it is re-requesting previous messages sent by the pubend. A non-recovering SHB is only receiving new messages.

3.1 PHB-Driven Congestion Control

The PDCC mechanism regulates the rate at which new messages are published by a pubend. The publication rate is adjusted depending on the observed throughput at the SHBs. It is the responsibility of the SHBs to calculate their own congestion metric based on throughput and notify the pubend whenever they think that they are suffering from congestion. The PDCC protocol uses two kinds of control messages between brokers to exchange congestion information:

Downstream Congestion Query Messages (DCQ). DCQ messages trigger the congestion control mechanism. They are generated by a pubend and sent down the message dissemination tree to all SHBs. DCQ messages carry a (1) pubend identifier (`pubendID`), (2) a monotonically increasing sequence number (`sequenceNo`), and (3) the current position in the pubend's message stream (`m_pubend`), e.g. the latest assigned message timestamp.

Upstream Congestion Alert Messages (UCA). UCA messages tell the pubend about congestion. They flow upwards the message dissemination tree from SHBs to the pubend and are generated by SHBs in response to DCQ messages. They are aggregated at IBs so that the pubend eventually receives a single UCA message. Apart from a pubend identifier and a sequence number, they contain the (3) minimum throughput rates observed at recovering (`minRecSHBRate`) and (4) non-recovering (`minNonRecSHBRate`) SHBs. The sequence number associates a UCA message with the DCQ message that triggered it.

For the PDCC scheme to be efficient, it is important that (1) DCQ and UCA messages have very low loss rates and (2) their queuing delays are much lower than the maximum delays that can occur in the system, even when the system in congested. Since DCQ and UCA messages are small in size and are sent at a larger time-scale compared to data messages, they consume little resources in the system. In our implementation, described in Sect. 4, DCQ and UCA messages are treated as high-priority messages in the event broker. Note that for fairness with other applications sharing the network, we rely on the fairness properties of TCP for inter-broker connections. We will describe the behavior of the three types of brokers (PHB, IB, and SHB) when processing these messages in turn.[6]

Publisher Hosting Brokers (PHB). The PHB triggers the PDCC mechanism by periodically sending out DCQ messages. The sequence number in the DCQ message is used to match it to the corresponding response coming from the SHBs in form of a UCA message. The interval (e.g. 1 s) at which DCQ messages are dispatched determines the interval at which the pubend will receive UCA responses when there is congestion. The higher the rate of responses, the quicker the protocol will adapt to congestion.

When the PHB has not received any UCA messages for a certain period of time (t_{nouca}), it assumes that the system is currently not congested. It then increases the publication rate when the rate is throttled (i.e. the publishers could publish at a higher rate). For the increase of the publication rate, we use a hybrid

[6] To simplify the discussion, we will assume that each PHB only hosts a single pubend.

scheme with additive and multiplicative increase. The new rate r_{new} is calculated from the old rate r_{old} according to

$$r_{\text{new}} = \max \left[r_{\text{old}} + r_{\text{min}},\ r_{\text{old}} + f_{\text{incr}} * (r_{\text{old}} - r_{\text{decr}}) \right]. \quad (1)$$

In (1), r_{decr} is the publication rate at the time of the last decrease in rate, f_{incr} is a multiplicative increment, and r_{min} is the minimum increase in rate. Initially, we used a purely additive scheme that resulted in a very slow increment, but experiments showed that a more optimistic approach gave a higher message throughput. The multiplicative use of f_{incr} allows the increase to be faster than a fixed additive increase. However, when the publication rate is already close to the optimal value, it is necessary to limit the increase. This done by keeping track of the publication rate at which the increase started (r_{decr}) and using it to restrict the multiplicative increase. As shown in the experiments in Sect. 5, this scheme leads to the publication rate probing whether the congestion condition has disappeared and, if not, oscillating around the optimal operating point.

When the PHB receives a UCA message, a decision is made whether to decrease the current publication rate. The rate is kept constant if the sequence number of the received UCA message is smaller than the sequence number of the DCQ message that was sent after the last decrease. This means that the system did not have enough time to adapt to the last decrease in rate and more time should pass before another congestion control decision can be made. Moreover, the rate is not reduced if the throughput value in the UCA message is larger than the value in the previous message. In this case, the congestion situation is improving, and further reduction in rate is deemed to be unnecessary. Otherwise, the publisher rate is decreased according to

$$r_{\text{new}} = \max \left[f_{\text{decr1}} * r_{\text{old}},\ (r_{\text{decr}} + f_{\text{decr2}} * (r_{\text{old}} - r_{\text{decr}}) \right] \text{ iff } r_{\text{decr}} \neq r_{\text{old}} \quad (2)$$

$$r_{\text{new}} = f_{\text{decr1}} * r_{\text{old}} \text{ otherwise} \quad (3)$$

where f_{decr1} and f_{decr2} are two multiplicative decrement factors. The first term in (2) multiplicatively reduces the rate by a factor f_{decr1}, whereas the second term reduces the rate relative to the previous decrement r_{decr}. As in (1), the second term prevents an aggressive reduction in rate when congestion is encountered for the first time after an increase. Since the PDCC mechanism constantly tries to increase the publication rate in order to achieve a higher rate, it will eventually cause SHBs to send UCA messages. This should not result in a strong reduction of the rate. Taking the maximum of the two decrement values tries to keep the publication rate close to the optimal operating point that is supported by the system. However, if the congestion level does not improve after a reduction, the publication rate is reduced again. This time a multiplicative decrease is performed (3) since the condition $r_{\text{decr}} = r_{\text{old}}$ now holds.

Intermediate Brokers (IB). The aggregation logic of UCA messages at IBs must ensure that (1) multiple UCA messages from different SHBs are consolidated such that the minimum rate at any SHB is passed upstream in a UCA

```
1  processDCQ(dcqMsg):
2    sendDownstream(dcqMsg)
3
4  initialization:
5    minNonRecSHBRate ← ∞, minRecSHBRate ← ∞, seqNo ← 0
6
7  processUCA (ucaMsg):
8    minNonRecSHBRate ← MIN(minNonRecSHBRate, ucaMsg.minNonRecSHBRate)
9    minRecSHBRate ← MIN(minRecSHBRate, ucaMsg.minRecSHBRate)
10   IF ucaMsg.seqNo > seqNo THEN
11     sendUpstream(ucaMsg.seqNo, minNonRecSHBRate, minRecSHBRate)
12     minNonRecSHBRate ← ∞, minRecSHBRate ← ∞, seqNo ← ucaMsg.seqNo
```

Fig. 5. Processing logic for DCQ and UCA messages at IBs

message. This enables the pubend to adjust its publication rate to provide for the worst congested SHB in the system. Moreover, (2) when UCA messages occur for the first time, they should be immediately sent upstream so that the pubend responds to new congestion as quickly as possible.

The algorithm for processing DCQ/UCA messages is shown in Fig. 5: IBs relay DCQ messages down to their children (line 2) and aggregate UCA responses. An IB keeps track of the minimum observed throughput values for non-recovering and recovering SHBs, and the maximum sequence number of the UCA messages that it has received (line 5). When a new UCA message arrives, the throughput minima are potentially updated (lines 8–9). A new UCA message is only sent upstream if the sequence number of the received message is larger than the maximum sequence number stored at the IB (line 10). This ensures that UCA messages with the same sequence number coming from different SHBs are aggregated before being sent upstream. However, the first UCA message with a new sequence number immediately triggers a new UCA message so that the pubend is quickly informed about newly detected congestion. Future UCA messages from other SHBs having the same sequence number will be aggregated, and will contribute towards the throughput minima in the next UCA message. After a UCA message is sent, both minimum throughput values are reset (line 12).

Subscriber Hosting Brokers (SHB). A SHB uses the ratio of pubend and SHB message rate as a metric for detecting congestion.

$$t = \frac{r_{\text{pubend}}}{r_{\text{SHB}}} \tag{4}$$

To allow for burstiness in the throughput due to application-level scheduling and network anomalies, we smooth t using a standard first-order low pass filter with an (empirical) value of $\alpha = 0.1$ and obtain $\bar{t}$.

$$\bar{t} = (1 - \alpha)\,\bar{t} + \alpha\, t \tag{5}$$

In addition, we need to distinguish between recovering and non-recovering SHBs. We describe how a SHB detects that it is recovering in Sect. 4.

Non-Recovering Brokers. A non-recovering SHB should receive messages at the *same rate* at which they are sent by the pubends. If the smoothed throughput ratio $\bar{t}$ drops below unity by a threshold, the SHB assumes that it has started falling behind because of congestion.

$$\bar{t} < 1 - \Delta t_{\text{nonrec}} \tag{6}$$

In rare cases, an SHB could be slowly falling behind because $\bar{t}$ stays below 1 (but above $1 - \Delta t_{\text{nonrec}}$) for a long time. Unless there is already significant congestion in the system, this will not cause overflow if queue sizes are large. Nevertheless, an SHB needs a mechanism to detect even very slow queue build-up. Therefore, an SHB periodically compares its current position in its message stream m_{SHB} to the pubend's message stream position (m_{pubend}), as given in the last DCQ message. If the difference is larger than Δt_s, the SHB will send a UCA message, even though its throughput ratio $\bar{t}$ is above the threshold:

$$m_{\text{SHB}} < m_{\text{pubend}} + \Delta t_s \tag{7}$$

Recovering Brokers. A recovering SHB must receive messages at a *higher rate* than the publication rate, otherwise it will never manage to successfully catch-up and recover all previous messages. Often, there is an additional requirement to maintain a minimum recovery rate $1 + \Delta t_{\text{rec}}$ that ensures a timely recovery. Thus, a recovering SHB will send a UCA message if

$$\bar{t} < 1 + \Delta t_{\text{rec}} \tag{8}$$

holds. The value of Δt_{rec} influences how much of the congested resource will be used for recovery messages instead of new data messages.

3.2 SHB-Driven Congestion Control

The SDCC mechanism manages the rate at which an SHB requests missed data by sending NACKs upstream. An SHB maintains a NACK window to decide which parts of the message stream should be requested. Then, the NACK window is opened and closed additively depending on the level of congestion in the broker network. The change in recovery rate throughput is used for detecting congestion.

An SHB starts with a small NACK window size $nwnd_0$. During recovery, the NACK window is adjusted depending on the change in recovery rate r_{SHB},

$$r_{\text{SHB}} = \frac{nwnd}{RTT}, \tag{9}$$

where $nwnd$ is the NACK window size and RTT is an estimate of the round trip time needed to satisfy a NACK.

Table 1. Configuration parameters for the PDCC and SDCC protocols

Param.	Description	Value
$size_{\text{NACK}}$	minimum size of NACK	100 tickms
t_{silence}	interval for sending explicit silence messages	1000 ms
t_{nouca}	interval without UCA messages before rate increase for PDCC	2000 ms
$r_{\min}$	minimum rate increase for PDCC scheme	2 msgs
f_{incr}	multiplicative increment for PDCC scheme	0.05
f_{decr1}	multiplicative decrement for PDCC scheme	0.5
f_{decr2}	multiplicative decrement w.r.t. previous increment for PDCC	0.25
α	smoothing factor for low pass filter	0.1
Δt_{nonrec}	threshold value for UCA messages for non-recovering SHBs	50 tickms
Δt_{rec}	threshold value for UCA messages for recovering SHBs	1000 tickms
Δt_s	threshold value for lag in message stream for SHBs	4000 tickms
$nwnd_0$	initial size of NACK window	100 tickms
α_{NACK}	recovery rate increase before NACK window is increased	0.1
β_{NACK}	recovery rate decrease before NACK window is decreased	0.3

The NACK window is managed in a similar fashion to TCP Vegas [5]: When r_{SHB} increases by at least a factor α_{NACK}, the NACK window is opened by one additional NACK per RTT. When r_{SHB} decreases by at least a factor β_{NACK}, the NACK window is reduced by one NACK:

$$nwnd_{\text{new}} = nwnd_{\text{old}} \pm size_{\text{NACK}} \tag{10}$$

4 Implementation in the Gryphon Broker

We have implemented the PDCC and SDCC mechanisms as an extension on top of the guaranteed delivery service provided by the Gryphon Broker [4]. The implementation comes with a number of configuration parameters (cf. Table 1) that influence the congestion control protocols. They were either explained in Sect. 3 or are referred to below. We begin by giving an introduction to Gryphon's guaranteed delivery service and then discuss the SHB and PHB implementation.

Guaranteed Delivery Service. Under exactly-once delivery semantics, the message stream is subdivided into discrete intervals called *ticks*. Each tick potentially holds a data message and is in one of four states: (1) (d)ata, when it contains a published message, (2) (s)ilence, when no message was published or was filtered upstream, (3) (f)inal, when it is no longer needed, and (4) (q) unknown, when its state is unknown. Ticks are fine-grained such that no two data messages can be assigned to the same tick. This is achieved by using a millisecond granularity clock that is enhanced with a counter to assign unique timestamps to messages. Therefore, a tick can be converted into a real-time timestamp assigned by the pubend.

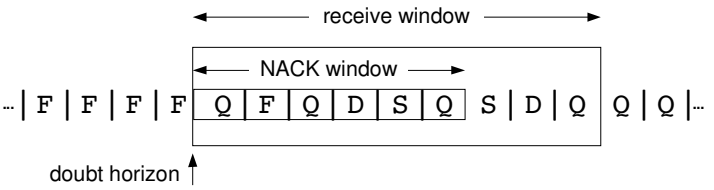

Fig. 6. Example of an `EdgeOutputStream` at an SHB

When no messages are published, ticks in the stream are assigned the silence state. A data message is prefixed with all silence ticks since the last message so that brokers can update their message streams. A pubend will send an explicit silence message containing silence ticks when no data messages were published for a certain interval. This is done every t_{silence} ms (cf. Table 1). Explicit silence messages ensure that SHBs know that no messages were published.

The message stream at an SHB is initialized with all ticks in the unknown state. The SHB then attempts to resolve all unknown ticks to either data or silence states by sending NACK messages upstream. Once a tick has been successfully processed by the SHB, the receipt is acknowledged and its state changes to final. Each SHB maintains a *doubt horizon*, which is the position in the stream until which there are no unknown ticks. All ticks before the doubt horizon either already were or can be delivered to the client applications.

SHB Implementation. In our PDCC implementation, SHBs use the rate of progress of the doubt horizon in the message stream (dh_{rate}) to detect congestion. Since the message stream contains seconds worth of ticks, the rate is measured as "tick seconds" per second:

$$t = dh_{\text{rate}} = \frac{\text{ticks}}{\text{time}} \tag{11}$$

An SHB maintains a consolidated message stream that is used to service all subscribers, and is represented using an `EdgeOutputStream` object. This stream maintains two windows, a *receive window* and a *NACK window*. The lower bound of both windows is the doubt horizon, and so they advance together with the doubt horizon. The receive window is a range of ticks such that only ticks that fall within this window are processed, and messages containing information about ticks outside the window are ignored. Thus, the receive window bounds the memory usage of the `EdgeOutputStream`. In the SDCC mechanism, the NACK window is a subset of the receive window that determines which unknown ticks in the `EdgeOutputStream` can be NACKed. The NACK window size ($nwnd$) is altered depending on the rate of progress of the window through the stream (cf. Sect. 3.2). Figure 6 shows an example of an `EdgeOutputStream` with a receive window and a NACK window. The doubt horizon points to the first unknown tick. The three unknown ticks that are within the NACK window will trigger NACK messages. Any messages referring to ticks outside the receive window will be ignored and, consequently, NACKed once the receive window has advanced.

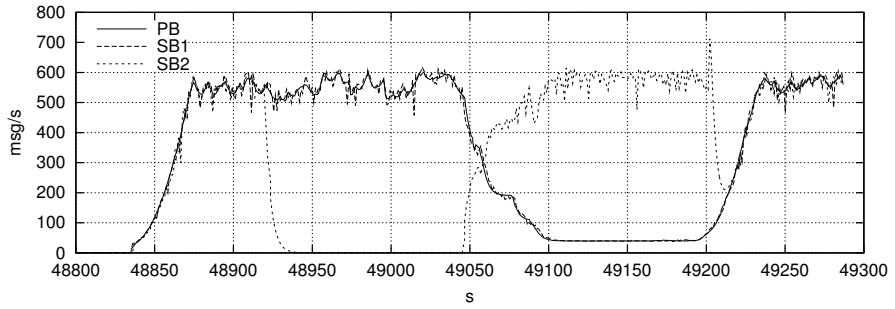

Fig. 7. E1: Congestion control after a IB-SB1 link failure

As mentioned previously, an SHB must be able to determine whether it is currently recovering or not. A solution where the SHB considers whether it is sending NACK messages would be too sensitive because a single lost message would force the PDCC mechanism to go into recovery mode. Instead, we implemented a scheme in which an SHB claims to be recovering only if it has been ignoring messages with ticks outside its receive window. This means that its doubt horizon has been lagging behind by a significant amount and the SHB is recovering all previous ticks in its receive window.

PHB Implementation The PDCC protocol requires pubends to be able to shape the rate of messages coming from publishers. We implemented a simple scheme that is compatible with the Java Message Service (JMS) API [8] used between clients and a Gryphon broker. The pubend keeps track of the number of messages published by a publisher in a time interval and stops sending ACK messages once the target rate has been reached. Holding back acknowledgments to publishers prevents them from publishing new messages. In the future, we plan to implement window flow control between clients and brokers.

5 Experimental Results

In this section, we discuss our experiments that evaluate the PDCC and SDCC mechanisms under congestion in two different topologies. The experimental setup was a network of dedicated broker machines running AIX connected via Ethernet links. Various broker metrics were used to create the plots shown here. Physical link failure was simulated by flushing a broker's output queues and closing its TCP connection. Network congestion was created as bandwidth limits on links. In all experiments, the PHB had 4 pubends and the broker queue sizes were large to maximize throughput (input queues: ~ 24 MB; output queues: 5 MB).

E1: CC after Link Failure (Simple Topology). The first experiment is a re-run of the failure experiment from Sect. 2. However, now the PDCC and SDCC schemes ensure that the system recovers successfully. Figure. 7 shows

that the publication rate of PB is reduced by the PDCC mechanism after the IB-SB2 link comes back up ($t = 49045$) because most of its bandwidth is used by the broker SB2 for recovery. After SB2 has finished recovering ($t = 49205$), PB can increase its publication rate. The spike in SB2's rate close to the end of the recovery phase occurs because the IB caches recent ticks in its message stream and is therefore able to satisfy some of the final NACKs more quickly.

The plot in Fig. 8[7] shows the behavior of the NACK window during recovery, as controlled by the SDCC mechanism. Initially, the NACK window has a small value (200 tickms) and is progressively increased until an optimal operation point ($\sim$ 900 tickms) is found. The NACK window further increases towards the end of the recovery process because of cached ticks at the IB.

E2: CC with B/W Limits (Simple Topology). We investigated how well the PDCC mechanism can adapt to an alternating bandwidth limit. At first, the IB-SB1 link is restricted to 60 KB/s for 120 s ($t = 63500..63620$). After that, the limit is increased to 150 KB/s for 120 s ($t = 63620..63740$), and then reduced to 60 KB/s again. As can be seen from the publication rate in Fig. 9, the PDCC scheme attempts to determine the optimal rate that can be supported by the link bottlenecks. It quickly adapts to new bottlenecks and keeps the queue utilization low (Fig. 10). Even when the available bandwidth is severely restricted ($t = 63500, 63740$), the output queues at the IB do not increase above 1 MB. The publication rate oscillates around the optimal point since the PHB is constantly probing the system to see whether the congestion situation has improved. The r_{decr} mechanism ensures that it stays close to the optimal value.

Figure 11 shows the doubt horizon rate from UCA messages received at the pubend. When there is no congestion during the first 120 s, no UCA messages are received except for transient messages at start-up. These messages occur when the doubt horizons at the SHBs start advancing causing the doubt horizon rate to stay below the threshold for a short time. At $t = 63540$, the doubt horizon rate decreases to half its previous value because of the link bottleneck. Once the publication rate at the pubend has been reduced sufficiently, the doubt horizon rate starts increasing again. When the link bottleneck is constant, UCA messages with doubt horizon rates slightly below the "real-time" rate of 1 ticksec/s are received periodically and prevent the publication rate from increasing further.

E3: CC with B/W Limits and Link Failures (Complex Topology). To evaluate how multiple sources of congestion in different parts of the network are handled, we set up a complex broker topology. This topology consists of 1 PHB, 3 IBs, and 5 SHBs (Fig. 12). It is asymmetric with different paths lengths from the SHBs to PHB. IBs have to perform a non-trivial amount of aggregation of UCA messages that are sent upstream by SHBs in different parts of the network. The following experiments had 100 subscribers per SHB so that under normal conditions the message rate at an SHB was 2/5 of the PHB's rate.

The first experiment consists of link bottlenecks and link failures (leading to subsequent recovery phases). Throughout the entire run, the IB1-SB1 link

[7] This plot is a re-run of E1 with a b/w limit on the PB-IB link of 500 KB/s.

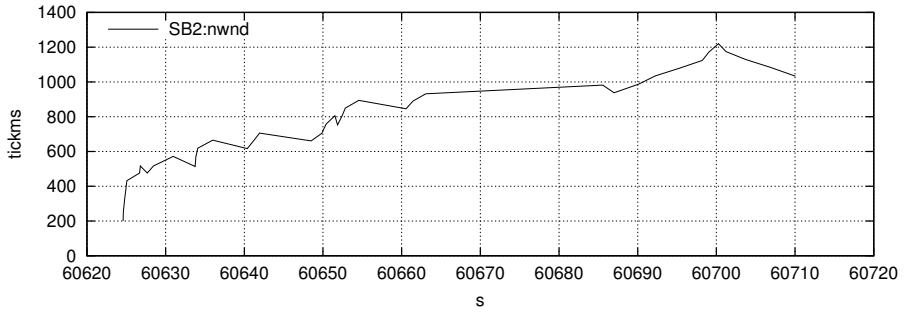

Fig. 8. E1: NACK window behavior after the IB-SB1 link failure.

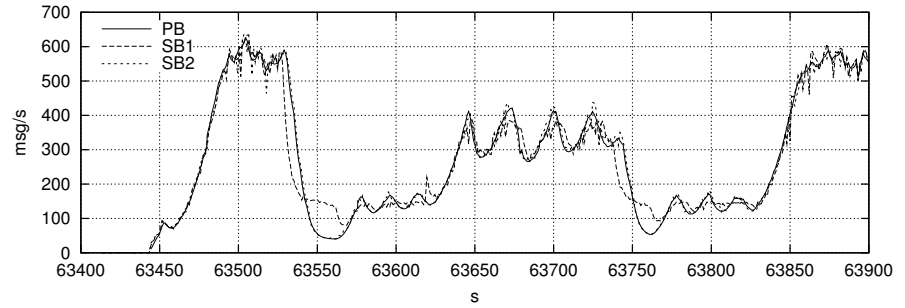

Fig. 9. E2: Congestion control with dynamic bandwidth restrictions

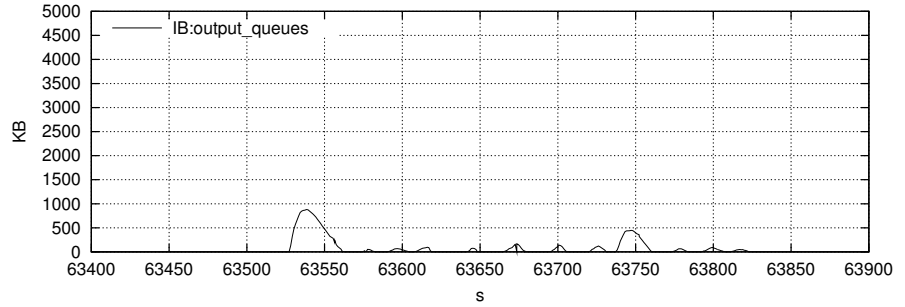

Fig. 10. E2: Output queue utilization at broker IB

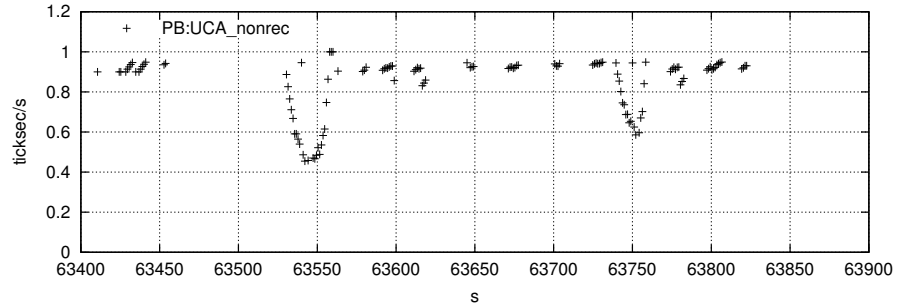

Fig. 11. E2: UCA messages received at pubend

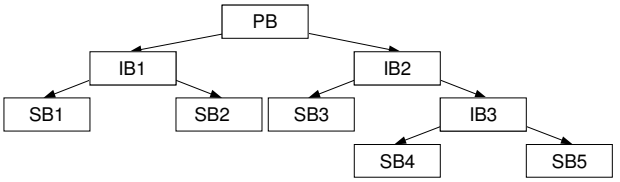

Fig. 12. A complex broker topology

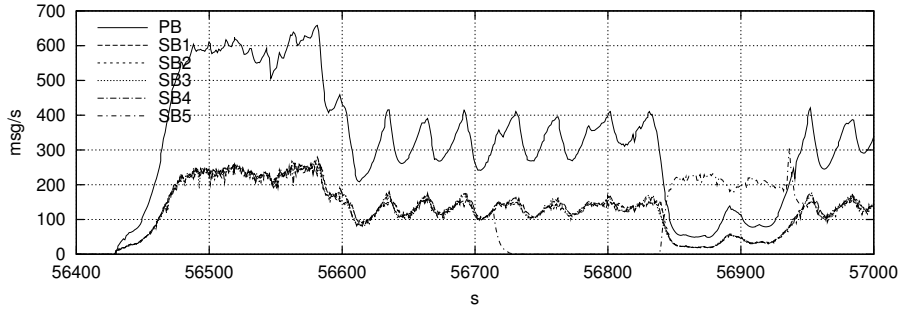

Fig. 13. E3: Congestion control with bandwidth restrictions and link failures

is restricted to 250 KB/s, which does not cause congestion in the absence of failures. The PB-IB1 link is limited to 150 KB/s at $t = 56590$, and, after 120 s, the IB3-SB5 link is failed ($t = 56710..56830$). The message rates observed at the SHBs and PHB are shown in Fig 13. At the beginning, the PB broker publishes messages at a rate of around 500 msgs/s. The subscribers connected to each SHB observe an aggregate message rate of 200 msg/s. When the first bandwidth limit comes into effect, all SHBs receive messages at a reduced rate because PB adjusts its publication rate. After the failure of the IB3-SB5 link, PB drops its rate even further to enable SB5 to recover all lost messages. Even though the link bottleneck and failure occur in different parts of the network at the same time,

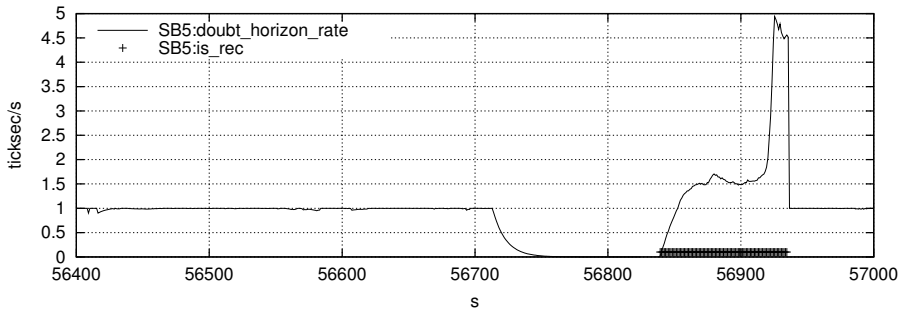

Fig. 14. E3: Doubt horizon rate with bandwidth restrictions and link failures

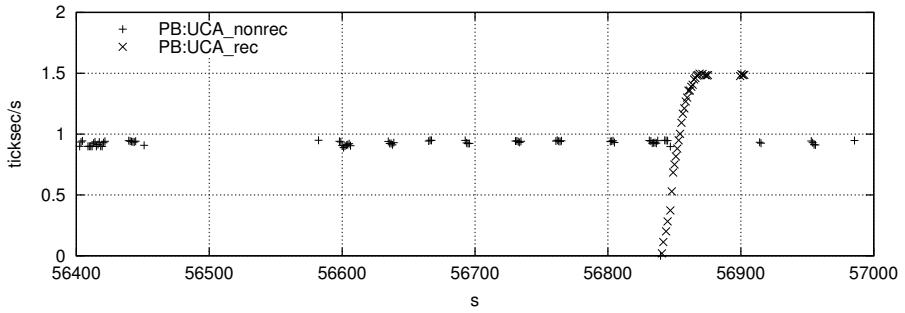

Fig. 15. E3: UCA messages received at pubend

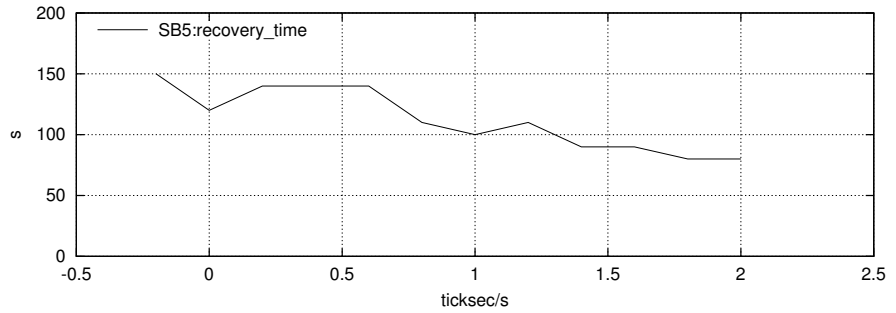

Fig. 16. E4: Variation of recovery time with Δt_{rec} threshold

the PDCC scheme drives the publication rate by the worst congestion point in the system and successfully prevents queues from building up.

The doubt horizon rate, as observed at SB5, is shown in Fig. 14. Since the doubt horizon rate is independent from the publication rate, it stays close to 1 ticksec/s until the IB3-SB5 link is failed at $t = 56710$. The value of Δt_{rec} in this experiment is 0.5 ticksec/s. After the link failure, the SHB switches to recovery mode (is_rec in Fig. 14) and the doubt horizon rate is kept above Δt_{rec}. Close to the end, the rate peaks to about 5 ticksec/s when SB5 reaches the point in the message stream at which the pubend reduced its rate. Now, more ticks in the stream are silence ticks without data, which enables the SHB to recover faster.

Figure 15 shows the consolidated UCA messages received at PB from recovering and non-recovering SHBs. After a startup effect, messages from the non-recovering SHBs (UCA_nonrec) arrive at regular intervals due to the link bottleneck PB-IB1. When SB5 starts recovering ($t = 56830$), it sends UCA messages (UCA_rec) in recovery mode whenever its rate drops below Δt_{rec}.

E4: CC with Different Recovery Times (Simple Topology). The next experiments show how the duration of recovery is influenced by the threshold value Δt_{rec}. The higher this value, the earlier the SHB will send UCA messages so that more of the congested resource is used for recovery messages. Figure 16 plots how recovery time varies with the Δt_{rec} values ranging from $-0.2 \,..\, 2.0$ ticksec/s

at 0.2 ticksec/s steps. A value $\Delta t_{\text{rec}} \leq 0$ is not used in practice, as it can result in an infinite recovery time. The PB-IB1 link was throttled to 500 KB/s.

The plot shows a clear correlation between Δt_{rec} and the recovery duration. However, it is not linear for two reasons: (1) With Δt_{rec} above 1.8 ticksec, the bandwidth-restricted link PB-IB is saturated with resent data messages. (2) When Δt_{rec} is less than or equal to 1 ticksec/s, there is an interaction between the 4 pubends: Even though the low threshold value does not require the pubends to reduce their publication rate by much, some pubends tend to consume a larger fraction of the bottleneck bandwidth, reducing the available bandwidth for the remaining pubends. These pubends observe a very low doubt horizon rate and thus reduce their publication rate more than necessary. The reason for this unfairness between pubends is that we employ first-come first-serve scheduling of messages in brokers and the PDCC protocol cannot synchronize rate reduction at different pubends inside a distributed system. We intend to investigate if the unfairness properties of the TCP Vegas equation are causing this [9].

6 Related Work

TCP. TCP comes with a point-to-point, end-to-end cc algorithm with a congestion window that uses additive increase, multiplicative decrease (AIMD) [10]. *Slow start* helps to open the congestion window more quickly. Packet loss is the only indicator for congestion in the system and *fast retransmit* enables the receiver to signal packet loss by ACK repetition. Modern TCP implementations such as TCP Vegas [5] attempt to detect congestion before packet loss occurs by using a throughput-based congestion metric. Since TCP Vegas is widely used, we decided to base our NACK throughput metric on this.

Reliable Multicast. Multicast protocols are comparable to pub/sub due to their one-to-many semantics, but typically have no filtering at intermediate nodes, and do not ensure that all leaves in the tree will eventually catch up to the sender. Congestion control is usually implemented at the transport level relying on router support. It must often adhere to existing standards to ensure fairness and compatibility with TCP [11, 12]. Since there are many receivers, scalable feedback processing is important, e.g. by feedback suppression [13]. Our approach does not discard information by consolidating feedback in a scalable way. Multicast cc schemes can be divided into (1) *sender-based*, in which all receivers support the same rate, and (2) *receiver-based* schemes, in which receivers have different rates by requesting transcoded versions of the data [14]. Since we can make few assumptions about our data, a receiver-based approach is hard.

The pgmcc [2] protocol forms a feedback loop between the sender and the "worst" congested receiver. The sender chooses this receiver depending on receiver reports in NACKs. The cc protocol for SRM [15] is similar except that here the feedback agent can give positive and negative feedback, and a receiver locally decides whether to send a congestion notification upstream to compete for becoming the new agent. An approach that does not rely on network support except minimal congestion feedback in NACKs is LE-SBCC [3]. A cascaded fil-

ter model transforms the NACKs from the multicast tree to appear like unicast NACKs before feeding them into an AIMD module. However, no consolidation of NACKs can be performed. All these schemes use a loss-based congestion metric that is not a good indicator for congestion in an application-level network.

Multicast ABR ATM. The ATM Forum Traffic Management Spec. [16] includes an available bit rate (ABR) category for traffic through an ATM network. At connection set-up, Forward and Backward Resource Management (FRM/-BRM) cells are exchanged between the sender and the receiver and modified at intermediate ATM switches depending on their resource availability. All involved parties agree on an acceptable cell rate depending on congestion in the system.

Multicast ABR requires flow control for one-to-many communication: A FRM cell is sent by the source and all receivers in the multicast tree respond with BRM cells that are consolidated at ATM switches [17]. Different ways of consolidating feedback cells have been proposed [18]. These algorithms have a trade-off between timely responsive and the creation of "consolidation noise" when new BRM cells do not include feedback from all downstream branches. Our consolidation logic at the IBs tries to balance this trade-off by aggregating UCA messages with the same sequence number and short-cutting new UCA messages. The scalable flow control protocol in [19] follows a "soft" synchronization approach where BRM cells triggered by different FRM cells can be consolidated at a branch point.

Overlay Networks. Congestion control for application-level overlay networks is sparse, mainly because application-level routing is a new research focus. A hybrid system for application-level reliable multicast in heterogeneous networks that addresses congestion control is RMX [20]. Here, a receiver-based scheme with the transcoding of application data is suggested. In general, global flow control in an overlay network can be viewed as a dynamic optimization problem [21] where a cost-benefit approach helps to find an optimal solution.

7 Conclusion

The problem of congestion control in messaging systems has received little attention so far. In this paper, we have presented a scalable congestion control scheme for a reliable message-oriented middleware. We have separated our scheme into a PHB-driven protocol that restricts the rate of new data messages, and a SHB-driven protocol that limits the rate of NACKs. Both protocols were implemented as part of the Gryphon Broker, an industrial-strength message-oriented middleware. The proposed solution addresses the special requirements of application-level overlay routing of messages, and filtering of messages at intermediate brokers in the network, and introduces little overhead into the system. A number of experiments with simple and complex topologies were used to show that the system quickly adapts to congestion and ensures that queue utilization is low.

Future work will investigate how to dynamically adapt the interval between DCQ messages and how to take advantage of the doubt horizon rate in UCA messages. Using the doubt horizon rate will help the system realize the severity of the congestion and allow it to adjust its rate faster to adapt to it.

References

1. McCanne, S., Jacobson, V., Vetterli, M.: Receiver-driven Layered Multicast. In: Proc. of ACM SIGCOMM. Volume 26,4. (1996) 117–130
2. Rizzo, L.: pgmcc: A TCP-Friendly Single-Rate Multicast Congestion Control Scheme. In: Proc. of ACM SIGCOMM, Stockholm, Sweden (2000)
3. Thapliyal, P., Li, S., Kalyanaraman, S.: LE-SBCC: Loss-Event Oriented Source-based Multicast Congestion Control. Technical report, RPI-ECSE (2001)
4. Bhola, S., Strom, R., Bagchi, S., Zhao, Y., Auerbach, J.: Exactly-once Delivery in a Content-based Publish-Subscribe System. In: Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'2002). (2002) 7–16
5. Brakmo, L.S., O'Malley, S.W., Peterson, L.L.: TCP Vegas: New Techniques for Congestion Detection and Avoidance. In: Proc. of ACM SIGCOMM. (1994)
6. Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R.E., Sturman, D.C.: An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In: Proc. of the 19th IEEE Int. Conf. on Distributed Computing Systems, 1999. (1999) 262–272
7. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching Events in a Content-based Subscription System. In: Proc. of the Principles of Distributed Computing, 1999. (1999) 53–61
8. Sun: Java^{TM} Message Service. In: http://java.sun.com/products/jms/. (2001)
9. Hasegawa, G., Murata, M., Miyahara, H.: Fairness and Stability of Congestion Control Mechanisms of TCP. In: Proc. of INFOCOM'99. (1999)
10. Jacobson, V., Karels, M.J.: Congestion Avoidance and Control. In: Proc. of ACM SIGCOMM. (1988) 314–332
11. Floyd, S., Fall, K.: Promoting the Use of End-to-end Congestion Control in the Internet. IEEE/ACM Trans. on Networking **7** (1999) 458–472
12. Golestani, S.J., Sabnani, K.K.: Fundamental Observations on Multicast Congestion Control in the Internet. In: INFOCOM (2). (1999) 990–1000
13. DeLucia, D., Obraczka, K.: Multicast Feedback Suppression Using Representatives. In: INFOCOM (2). (1997) 463–470
14. Yang, Y.R., Lam, S.S.: Internet Multicast Congestion Control: A Survey. In: Proc. of ICT, Acapulco, Mexico (2000)
15. Shi, S., Waldvogel, M.: A Rate-based End-to-end Multicast Congestion Control Protocol. In: Proc. of 5th IEEE Symposium on Comp. and Comm. (ISCC). (2000)
16. Sathaye, S.: ATM Forum Traffic Management Specification 4.0. ATM Forum af-tm-0056.000 (1996)
17. Roberts, L.: Rate-based Algorithm for Point to Multipoint ABR Service. ATM Forum Contribution 94-0772R1 (1994)
18. Fahmy, S., Jain, R., Goyal, R., et al.: Feedback Consolidation Algorithms for ABR Point-to-Multipoint Connections in ATM Networks. In: Proc. of IEEE INFOCOM. Volume 3. (1998) 1004–1013
19. Zhang, X., Shin, K.G., Saha, D., Kandlur, D.D.: Scalable Flow Control for Multicast ABR Services in ATM Networks. IEEE/ACM Trans. on Netw. **10** (2002)
20. Chawathe, Y., McCanne, S., Brewer, E.A.: RMX: Reliable Multicast for Heterogeneous Networks. In: INFOCOM, Tel Aviv, Israel, IEEE (2000) 795–804
21. Amir, Y., Awerbuch, B., Danilov, C., et al.: Global Flow Control for Wide Area Overlay Networks: A Cost-Benefit Approach. In: OpenArch'02. (2002) 155–166