Research for Congestion Control in a Reliable Scalable Message-oriented Middleware

Tang Tao
Computational-engineering,
Dresden University of technology,
Dresden,
Germany,
`tonycbr2005@hotmail.com`

February 26, 2007

Abstract

This paper presents congestion control and some related problems in message-oriented middleware. Here i describe two congestion control mechanisms, (1)driven by a publisher hosting broker(PDCC), (2)driven by a subscriber hosting broker(SDCC). PDCC use a Feedback loop between endpoints and downstream SHBs to monitor congestion, and limit publication rate of new messages to prevent it.SDCC monitor rate of progress at a recovering SHB and limit rate of NACKs during recovery. Also i've described the different outcome of using and not using the congestion control mechanism described above to evaluate them.

Contents

1 Introduction

Message-oriented middleware is basically an application level routing structure, mainly dealing with information dissemination, filtering and transfering optimization. This mechanism is developed in the context of publish/subscribe messaging model. There can be many nodes that play a role of publisher or subscriber in the network. Publishers send messages downstream, subscribers express interest in messages by providing a predicate/filter, that can be executed on each message, and only those messages that match any filters are delivered to the subscriber. During that phase, different nodes may have different rate of sending or receiving messages. For instance, One intuitive problem can be that the message waiting queue in a node might be larger and larger, some congestion might happen, that might bring down the whole system in extreme case. This paper focuses on the cause of congestion in more details and introduce protocols to alleviate congestion problem for reliable and scalable messaging middleware [1].

The structure of this is organized as follows: section 2 gives an basic knowledge and some terminology we will use in later chapter. section3.1 introduce what the problem is and how do they happen, section 3.2 describe how does congestion control protocol works. section 4 evaluate the effect after using CCP. section 5 gives an overview of related work. section 6 gather the questions discussed in lecture.

2 Basics

2.1 Why scalable and reliable

The author of [1]explained the reason why Message-Oriented Middleware(MOM) is scalable is mainly due to 1)Asynchronous communication and loose synchronization. Asynchronous model is usually preferable than tight synchronization for many applications and provides more convenience to scale up and down. 2)its Publish/Subscribe communication with filtering that can dynamically change the subset of messages that are interested in. 3)its Overlay network of message brokers. They can be added or reduced with little coupling relation.

And the reason why Message-Oriented Middleware(MOM) is reliable is mainly due to its 1)Guaranteed delivery sematics for messages. exactly-once delivery of messages are provided to subscribers to satisfy service agreements and message interdependencies.[1, 3, 4] 2)Resending messages lost due to failure.

2.2 System model

Broker is one of the central concept used all around the paper. Authors in [1] describe it like this "Scalable messaging middleware is deployed as an overlay network of application level routers, which we refer to as brokers." It's basically an application router, usually, multiple routing paths are between many brokers, and the routing protocol used in overlay network both (1)balances the load amongst the available paths, and (2)routes around failed paths for high availability. While still that's no guarantee that system can sustain itself after some congestion or failures because of the ignorance of message rate. Since incompatible message rate might cause message queue overflow and in addition, resending messages might bring even heavier load to system. These all draw much attention in researching congestion control.

Author in[1]assume a model where brokers can perform 3 roles (1)publish hosting broker(PHB)is an edge of the network broker which publishing clients connect to. likewise,(2) subscriber hosting broker(SHB) is an edge broker which subscribing clients connect to. (3) intermedia broker(IB) is a broker which is inside the network and host no clients. Brokers actually play multiple roles , but for simplicity, here assume that each broker has only one role.

Like it's illustrated in the Figure 1: several publishers are aggregated to publishing endpoints, which is also called pubends. Each pubend represents an ordered stream of messages, and maintain the stream in a persistent storage. Pubends then send messages downstream to SHBs via IBs. IBs can play roles more than forwarding the messages, but also perform filtering on data messages so that IBs don't need to forward messages that don't match any of target subscribers' predicate/filters. Besides that, each IB can cache stream data and respond to NACKs. NACKs are forwarded to PHB only if they can't be satisfied by an IB. Obviously, caching will ease the workload to a great extent.

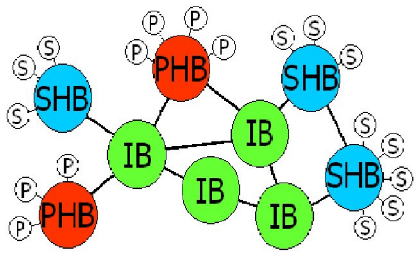


Figure 1: : What are PHBs, SHBs, IBs like.[4]

The congestion control protocols presented here deal with congestion starting from the pubend up to 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 pubends stream. Hence, the protocols adjust the message rate to the slowest link or broker inside the system, but not to the slowest subscribing application[1]. Here the brokers in the system are assumed trusted, thus the system can protect itself against malicious slow subscribers by disconnecting them.

Adjusting the message rate are performed both at PHB and SHB so that the waiting queue in all brokers will not overflow. Adjusting in SHB is described by Author in [1]:

- 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 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."

And later we will see in more details how NACK window works.

Adjusting in PHBs are done by using Feedback loop between pubends and downstream SHBs to monitor congestion and Limiting publication rate of new messages to prevent congestion. Just keep in mind first that lack of feedback is not used as a trigger for CC(congestion control), because it can be false positive feedback if some SHBs are down or partitioned from the rest of the network[1, 4, 4].

These protocols have been implemented in the context of the Gryphon system[1, 2], which supports highly-scalable content-based publish/subscribe. Authors in[2]describe Gryphon as "a distributed computing paradigm for message brokering, which is the transferring of information in the form of streams of events from information providers to information consumers ". In Gryphon, the flow of streams of events is described via an information flow graph. The information flow graph specifies the selective delivery of events, the transformation of events, and the generation of derived events as a function of states computed from event histories. For this, Gryphon derives from and integrates the best features of distributed communications technology and database technology. The Gryphon approach augments the publish-subscribe paradigm with the following features:

1. Content-based subscription, in which events are selected by predicates on their content rather than by pre-assigned subject categories;
2. Event transformations, which convert events by projecting and applying functions to data in events;
3. Event stream interpretation, which allows sequences of events to be collapsed to a state and/or expanded back to a new sequence of events; and

4. Reflection, which allows system management through meta-events.

The Gryphon can be used in many scenarios, such as stock exchange or a weather forecasting, where there is a constantly varying number of sub-applications supplying events, and a varying number consuming events. The suppliers and consumers may not necessarily know each other; instead the suppliers just supply certain type of information to any one may be interested in the subset of these information. for instance which get from[2]:in a stock exchange, one consumer may be interested in all stock trades greater than 1000 shares, and another in specific market trends, such as all stock trades representing a drop of more than 10 points from the previous day's high. It is an extension of publish-subscribe technology.

3 Congestion Control

In this section, i first illustrate what the congestion control problem is, and then introduce the congestion control protocol that is implemented in the system.

3.1 Congestion Control Problem

Let's see two examples first.

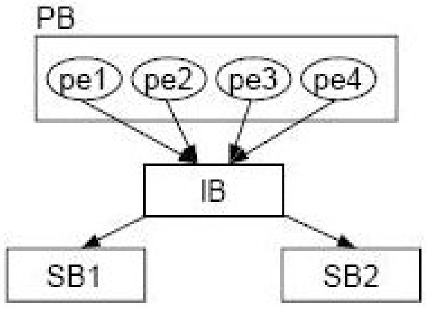


Figure 2: : A simple broker topology [1].

Like it is illustrated in Figure 2: 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 , 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. It's allowed for the path from a pubend to a particular SHB to change, but assume that paths change infrequently.[1]

As mentioned before, CCP for each link is not sufficient to prevent congestion collapse in the system as a whole. Just like two examples got from [1]. 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 percent idle in the steady-state.

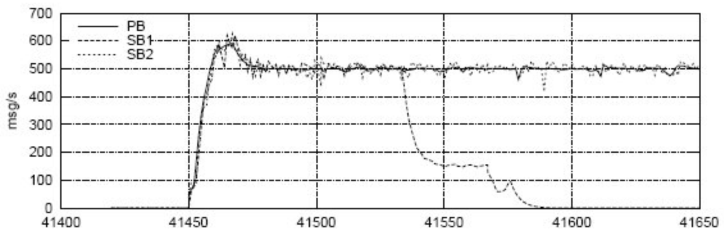


Figure 3: : Collapse with IB-SB1 link restricted to 60KB/s[1].

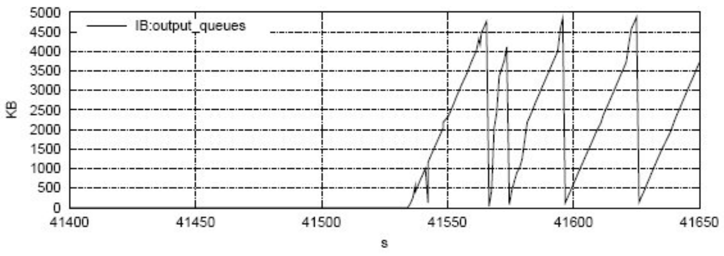


Figure 4: : Queue utilization at broker IB during collapse[1].

In Fig. 3, after running the system without any congestion, the bandwidth on the ib-sb1 link is throttled to 60 KB/s 2. 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 (see Fig. 4) from the time of collapse and eventually overflow will causes message loss. Lost messages need to be recovered (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.[1]

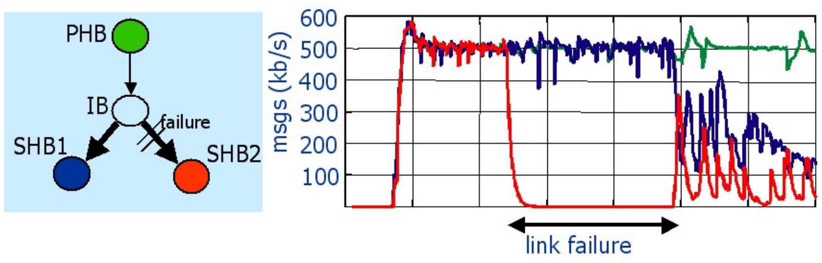


Figure 5: : Collapse with PB-IB link restricted to 250 KB/s[1].

In Fig. 5, the bandwidth on the pb-ib link is restricted to 250 KB/s. In the absence of failure, this does not cause congestion. Then the author 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.[1]

3.2 Congestion Control Protocol

Here presents the congestion control mechanism in detail. The solution consists of two parts[1]:

1. The PHB-driven cc protocol (PDCC) ensures that pubends do not cause congestion due to a too high publication rate. Feedback loop is used between pubends and downstream SHBs to monitor congestion and Limit publication rate of new messages to prevent congestion

2. The SHB-driven cc protocol (SDCC) handles the rate at which SHBs try to recover after a failure. This protocol Monitor rate of progress at a recovering SHB and Limit rate of NACKs during recovery .

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.

3.2.1 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, they are described by author of [1]as followed:

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 pubends message stream (m pubend), for example, the latest assigned message time-stamp. This message is periodically sent down the dissemination tree by pubend to query SHBs' status.

Upstream Congestion Alert Messages (UCA). UCA messages tell the pubend about congestion, and that also indicate UCA won't be sent if there is no congestion according to spectific SHBs' metric. SHBs observe their own throughput and decide whether they should respond the DCQ with a UCA message. If yes, 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. And that may cause pubend to reduce their message publication rate. 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, 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.

The behavior of the three types of brokers (PHB, IB, and SHB) are describe by Author in [1] like this: when processing these messages in turn.

"*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 (tnouca), 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)."

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 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. For this algorithm, your can check Figure5 in paper[1] for more details.

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

$$t = \frac{r_{pubend}}{r_{SHB}} \tag{1}$$

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 $=$ 0.1 and obtain $\bar{t}$.

$$\bar{t} = (1-\alpha) * \bar{t} + \alpha * t \tag{2}$$

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

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 t drops below unity by a threshold, the SHB assumes that it has started falling behind because of congestion.

$$\bar{t} < 1 - \Delta t_{nonrec} \tag{3}$$

In rare cases, an SHB could be slowly falling behind because t stays below 1 (but above $1-\Delta t_{nonrec}$) 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 buildup. Therefore, an SHB periodically compares its current position in its message stream m_{SHB} to the pubends 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 *overlinet* is above the threshold:

$$m_{SHB} < m_{pubend} + \Delta t_s \tag{4}$$

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

$$\bar{t} < 1 + \Delta t_{rec} \tag{5}$$

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

And if you are interested in this part, there are formal formula inference and description in more details about how to use DCQ and UCA to adjust congestion problems in the paper[1].

3.2.2 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 nwnd0. During recovery, the NACK window is adjusted depending on the change in recovery rate r_{SHB},

$$r_{SHB} = \frac{nwnd}{RTT} \tag{6}$$

where nwnd is the NACK window size and RTT is an estimate of the round trip time needed to satisfy a NACK. The NACK window is managed as followed: When r_{SHB} increases by at least a factor αNACK, the NACK window is opened by one additional NACK per RTT. When rSHB decreases by at least a factor β NACK, the NACK window is reduced by one NACK:

$$nwnd_{new} = nwnd_{old} \pm size_{NACK} \tag{7}$$

4 Evaluation

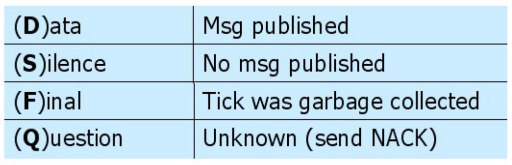
The paper[1] have implemented the PDCC and SDCC mechanisms as an extension on top of the guaranteed delivery service provided by the Gryphon Broker. The implementation comes with a number of configuration parameters [1] that influence the congestion control protocols. Here let's begin by giving an introduction to Gryphons guaranteed delivery service and then discuss the SHB and PHB implementation.

Guaranteed Delivery Service.

Motivations for guaranteed delivery include (1) service agreements (e.g., it is unacceptable for some stock traders not to see a trade event that others see), and (2) message interdependencies. Because the messages may be used by the subscribing application to accumulate a view where missing or reordered messages could cause an incorrect state to be displayed.where missing or reordered messages could cause an incorrect state to be displayed.

Author in [3] describe it like this:

"A guaranteed delivery service provides exactly-once delivery of messages to subscribers. Each publisher is the source of an ordered event stream. A subscriber who remains connected to the system is guaranteed a gapless ordered filtered subsequence of this stream. A filtered subsequence is gapless if, for any two adjacent events in this subsequence, no event occurring between these events in the original stream matches the subscribers filter. The guarantee is honored as long as the subscriber remains connected, even in the presence of intermediate broker and link failures."



(**D**)ata	Msg published
(**S**)ilence	No msg published
(**F**)inal	Tick was garbage collected
(**Q**)uestion	Unknown (send NACK)

Figure 6: : four states for tick.[4]

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

[1]check Table 1 in [1] for more detail

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.

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 certain time[2]. 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.

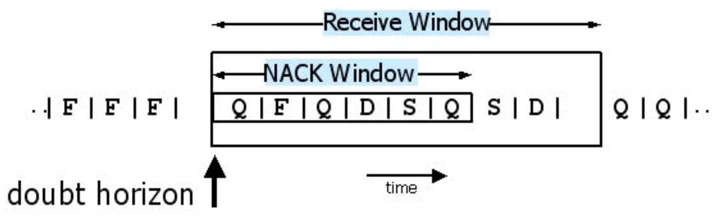


Figure 7: : double horizon [4].

SHB Implementation. In 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:

$$t = dh_{rate} = \frac{ticks}{time} \tag{8}$$

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. Here the rate of process of double horizon can be used as an congestion metric. I've mentioned the metric before, here's just an remind in more detail.[1]

Then i can present another two example which are done under similar condition as previous one, while adding congestion control protocol to see their performance.

[2]$t_{silence}$ ms reference Table 1 in [1])

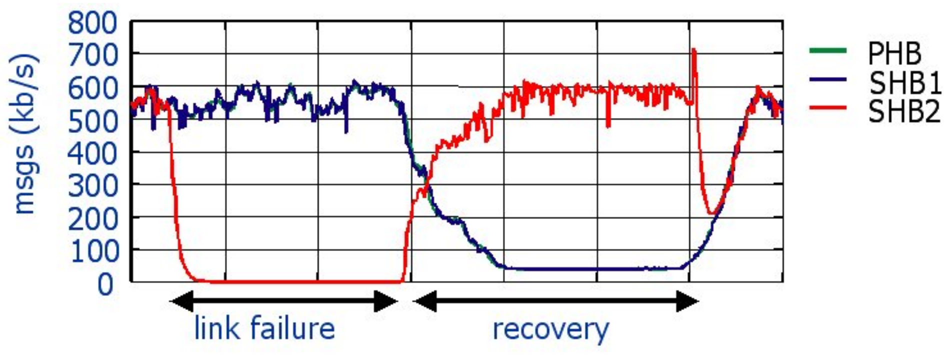


Figure 8: : E1: Congestion control after a IB-SB1 link failure[4].

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. 8 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 sb2s 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.[1]

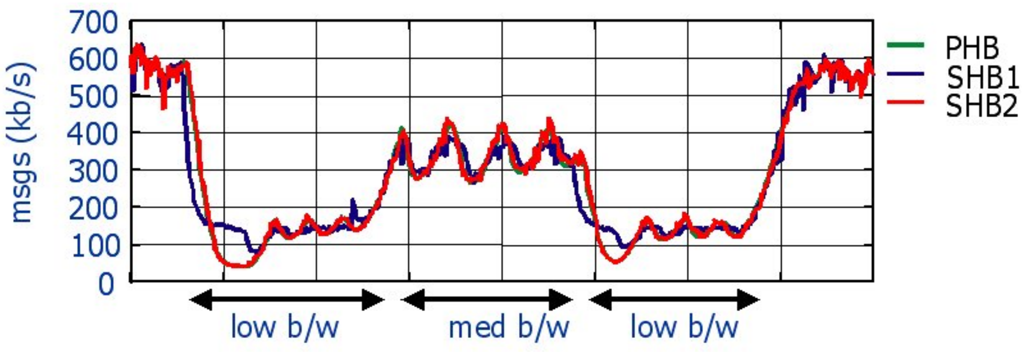


Figure 9: : E2: Congestion control with dynamic bandwidth restrictions [1].

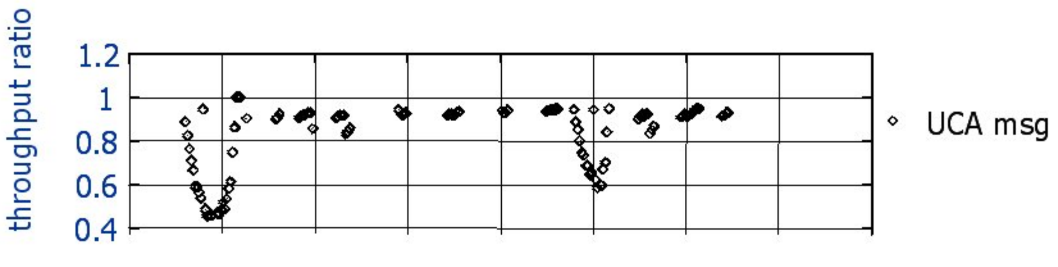


Figure 10: : E2:UCA messages received at pubend [4].

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 . After that, the limit is increased to 150 KB/s for 120 s , 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 . Even when the available bandwidth is severely restricted, 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.[1]

Figure 10 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. later, the doubt horizon rate decreases to half of 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.

5 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[6]. 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.
- 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.[7] 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 where a cost-benefit approach helps to find an optimal solution.

For more related work like Reliable Multicast and Multicast ABR ATM, check the paper[1].

6 Discussion

There are several question that are discussed during my presentation.

Question 1: Does the real case contain many more brokers in the network?

- Surely, There can be many brokers, take PHB as an example: Publishers are aggregated to publishing endpoints (pubends), multiple of PHBs, SHBs and IBs are interconnected. That is illustrated in Figure 1 which has given an simple impression what it is like in the real case. Furthermore, even a single broker can play several roles, while Author in [1] assume each broker plays only one role for simplicity of exposition.

Question 2:How to detect congestion? Who decide congestion has happened?

- SHB can observe the message throughput, and are responsible to decide whether to send an UCA message as an indication of congestion. Sudden reduction of throughput can be inferred that messages queue has built up. And also, we can use the rate of progress of double horizon as metric which is better in my mind, because it's independent from filtering and actual publication rate.

Question 3: If an SHB was down how does it respond to congestion happened? How does it send UCA upstream to IB or PHB to indicate the situation?

- First, all experiment carried on are based on the assumption of ensuring that all SHBs that are trying to recover messages they missed due to failure are eventually successful. And of course, when a SHB was down, it can do nothing. So the problem here is not how SHB respond, because during the period of down, congestion doesn't happen. Congestion happens during the recovering process, at that period, SHB send NACK ticks which is initialized as (Q)uestion to request the messages it missed. Then it can use CC protocol, sends UCA to eventually adjust the publishing rate according to throughput.

Question 4: How to make sure congestion control messages get through the congested places?

- 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. During implementation, DCQ and UCA messages are treated as high-priority messages in the event broker.

Question 5: How is it possible to cut off slow subscribers?

- The congestion control protocols presented in the paper 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 pubends 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* (if cut off means disconnect here)them. While how to disconnect them is not explained in paper, my intuitive thinking is that we can add a throughput monitor, checking every node throughput, then we can use a predicate to justify them. If throughput has reached an unacceptable low level, the node is cut by monitor or by system administrator manually.

7 Conclusion

In this paper, I 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 SHBdriven 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 applicationlevel 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] Peter R. Pietzuch1, Sumeer Bhola, *: Congestion Control in a Reliable Scalable Message-Oriented Middleware.* In Proceedings of ACM/IFIP/USENIX International Middleware Conference, 202-221, 2003. 1, 2.1, 2.2, 2.2, 2, 3.1, 3, 4, 3.1, 5, 3.1, 3.2, 3.2.1, 3.2.1, 4, 1, 4, 2, 4, 9, 4, 5, 6

[2] R. Strom and G. Banavar and T. Chandra and M. Kaplan and K. Miller and B. Mukherjee and D. Sturman and M. Ward, *Gryphon: An information flow based approach to message brokering.* International Symposium on Software Reliability Engineering (ISSRE '98), 1998 2.2, 2.2

[3] S. Bhola and R. Strom and S. Bagchi and Y. Zhao and J. Auerbach, *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, 2002. . 2.1, 4

[4] Perter R. Pietzuch, Sumeer Bhola, *presentation slides of Congestion control in a Reliable Scalable Message-Oriented Middleware.* University of Cambridge, Computer Laboratory. IBM T J Watson Reseach Center 2003 2.1, 1, 2.2, 6, 7, 8, 10

[5] Brakmo, L.S., OMalley, S.W., Peterson, L.L. *: TCP Vegas: New Techniques for Congestion Detection and Avoidance.* In: Proc. of ACM SIGCOMM. 1994 5

[6] Jacobson, V., Karels, M.J. *: Congestion Avoidance and Control.* In: Proc. of ACMSIGCOMM. (1988) 314C332 5

[7] Chawathe, Y., McCanne, S., Brewer, E.A. *: RMX: Reliable Multicast for Heterogeneous Networks.* In: INFOCOM, Tel Aviv, Israel, IEEE (2000) 795C804 5