

# Scalably Supporting Durable Subscriptions in a Publish/Subscribe System

Sumeer Bhola      Yuanyuan Zhao      Joshua Auerbach  
{sbhola, yuanyuan, josh}@us.ibm.com  
IBM T.J. Watson Research Center

## Abstract

*We describe algorithms to scalably support durable subscriptions in a publish-subscribe system. Durable subscriptions are guaranteed exactly-once message delivery, despite periods of disconnection from the system. Our algorithms persistently log each message only once in the system, and can support administratively specified 'early-release' policies that reclaim persistent storage in the presence of misbehaving durable subscribers. To efficiently recover messages missed by a disconnected durable subscriber, without refiltering messages published while the subscriber was disconnected, we persistently log filtering information in a manner optimized for the read/write pattern of durable subscriptions. Consolidation of data-structures across all subscribers that are done with catching up (after a disconnection), helps the system support a larger number of subscribers.*

*We experimentally demonstrate the low-latency and scalability properties of our implementation, both in the presence and absence of failures.*

## 1 Introduction

This paper describes novel algorithms to scalably support durable subscriptions in a publish-subscribe system. These algorithms have been implemented in Gryphon [7, 9, 10, 14], a highly-scalable content-based publish-subscribe system, that employs a redundant overlay network of brokers.

Durable subscribers can disconnect and reconnect from the system, and are guaranteed exactly-once message delivery, i.e., messages published while they were disconnected will be delivered on reconnection, and no messages will be lost due to failures inside the system. In message queuing systems, such as IBM's MQseries, this is also known as persistent messaging. An example of usage of durable subscriptions is stock trading applications, where all orders to trade must arrive reliably at the application processes that will execute the trades, and also be recorded reliably by data

backup applications, at multiple locations, for disaster recovery. Since disconnection of a durable subscriber may be involuntary, for instance, due to failure of the subscriber itself, such subscribers typically consume messages within the context of a transaction. In the rest of this paper we will refer to messages published by applications as events, to distinguish them from control messages sent inside the overlay network.

Previously [10] we have described a scalable algorithm for exactly-once event delivery to non-durable subscribers, i.e., a subscriber will miss events for the period of disconnection. This is the only known algorithm that can perform filtering of events at intermediate nodes (brokers) in the overlay network, which improves network utilization, but logs each event only once and does not perform store-and-forward routing of events.

An obvious, but undesirable, way to extend this solution for durable subscriptions is the following:

Every edge-broker to which durable subscribers connect, which we call a subscriber hosting broker (SHB), maintains a persistent event log for each durable subscriber in which each event that matches the subscriber is placed. While the subscriber is connected, events are delivered in-order from this log to the subscriber, and are removed from the log when the subscriber acknowledges consuming them. At all times, whether the subscriber is connected or not, new events are added to the end of the log.

This is the typical solution adopted at SHBs by current Message Queuing products <sup>1</sup> However, this solution has significant disadvantages:

1. An event is logged at all SHBs, and multiple times at an SHB, depending on the number of subscribers that match the event.
2. A durable subscriber has to always reconnect to the SHB that is maintaining its persistent event log. This decreases system availability if an SHB crashes and does not recover immediately. Due to the fanout properties of publish-subscribe, a majority of brokers in

---

<sup>1</sup>This is based mainly on anecdotal evidence, since messaging product vendors do not publicly reveal such technical details.

typical deployments are SHBs. Hence, increasing system availability, such as by hosting SHBs on highly fault-tolerant hardware, increases costs significantly.

In contrast, the novel features of the solution we present in this paper are:

1. *Only once event logging*: Each event needs to be logged only once, at the broker hosting the publisher of the event (the publisher hosting broker or PHB). Scalability of event recovery is achieved by caching events at intermediate brokers and SHBs, and nack consolidation. Some of these caches may use persistent storage to support large cache sizes, but the absence of an event from a cache does not impact correctness. Availability can be increased by hosting the PHBs, which are fewer in number, on fault-tolerant hardware.
2. *Persistent Logging of filtered information*: An SHB persistently logs event timestamps and the list of matching subscribers, in an optimized manner<sup>2</sup>. This aids in fast recovery of messages missed by a reconnecting subscriber, by avoiding refiltering of messages.
3. *Consolidated data-structures for 'non-catchup' subscribers*: An SHB consolidates most of its data-structures and message processing for all subscribers that are not in *catchup mode*, i.e., they are not recovering events in the past. We expect that, under normal operation, a high-percentage of durable subscribers will be connected and not in catchup. Hence, consolidation allows each SHB to support a large number of subscribers.
4. *Releasing persistent storage despite long-duration disconnected subscribers*: Typical event expiration models, such as in the Java Message Service (JMS) API [3], only support an optional publisher specified event expiration time interval. If the event has not expired (or has infinite expiration), persistent storage used by it cannot be *released* until the event has been acknowledged by all subscribers. Our model also supports administratively specified *early-release* policies. Such policies guard against storing an event indefinitely in the presence of incorrect or malicious durable subscribers that are disconnected for a long duration. The algorithm ensures that reconnecting durable subscribers that may have missed some event due to early-release get an explicit gap notification.
5. *Extensible for 'reconnect-anywhere' subscribers*: Since the persistent filtered log is only a performance optimization, and events are retained at the PHB, a

durable subscriber reconnecting to a different SHB can be accommodated by retrieving the events it may have missed (from the PHB or intermediate caches) and refiltering the events. We do not elaborate on this aspect of our design and implementation in this paper.

We have implemented our subscription model and algorithms in the context of the Gryphon system, and present results that show high performance and linear scalability in the presence of subscriber disconnections/reconnections. We show system behavior with SHB failure. Results with intermediate broker and PHB failure are similar to those in [10] and not presented here. For programmers writing to the Java Message Service (JMS) API, we have also implemented JMS durable subscriptions on top of our model.

The rest of the paper is organized as follows: Section 2 describes the system model seen by durable subscribers. Section 3 recaps our stream routing protocol and describes event retention and release. Section 4 describes the important parts of the subscriber hosting broker: (1) the persistent filtering subsystem, (2) the consolidated data-structures for non-catchup subscribers. Section 5 discusses experimental results. Section 6 discusses related work and we conclude in section 7.

## 2 System Model for Subscribers

Each publisher hosting broker (PHB) maintains one or more publishing endpoints (*pubends*). Each persistent event published to this broker is assigned to a pubend based on some criteria such as the identity of the publisher, the importance of the event etc. Each pubend maintains a persistent and ordered event stream, that is indexed by the timestamp assigned to the event when it was added to this stream. Conceptually, a stream represents information for every time tick<sup>3</sup>, whether there is an event at that time tick or not. This aspect of the model is the same as [10].

When a durable subscriber, say  $s$ , first connects to the system, it is provided a starting point (a timestamp) for each pubend in the system. This set of (pubend, timestamp) pairs is essentially a Vector Clock [15], and we refer to it as the *Checkpoint Token* (CT) of subscriber  $s$ . We use the notation  $CT(s, p)$  to refer to the current timestamp value for pubend  $p$ .

Subscriber  $s$  is delivered monotonically increasing timestamp information for each pubend  $p$ , using messages,  $m$ , containing a timestamp,  $m.t$ . Let  $t_0$  be the timestamp of the message preceding  $m$  from pubend  $p$ , delivered to  $s$ , or in case  $m$  is the first ever message from  $p$  delivered to  $s$ , let  $t_0$  be equal to  $CT(s, p)$ . There are three kinds of messages:

<sup>2</sup>Each event causes a log record write of length  $8 + 16 * n$  bytes, where  $n$  represents the the number of matching subscribers ( $n > 0$ ).

<sup>3</sup>time ticks are fine-grained enough to ensure no 2 events occur at the same time.

1. Event message: This contains an event with timestamp  $m.t$  that matches its subscription. It also guarantees that there were no events that matched its subscription in the interval  $(t0, m.t)$ .
2. Silence message: This message does not contain an event, but guarantees that there is no event that matched its subscription in the interval  $(t0, m.t]$ .
3. Gap message: This message indicates that there may have been some events in the interval  $(t0, m.t]$  that matched its subscription, but information about what is in the interval was discarded due to early-release.

Once the subscriber has consumed message  $m$ , and all preceding messages from pubend  $p$ , it sets  $CT(s, p) = m.t$ . Periodically, a subscriber should send its current CT to the SHB as an acknowledgment of all messages with timestamps less than or equal to the CT. When the subscriber reconnects after a disconnection it presents its current CT to the SHB as the point from which it wants to resume message delivery.

If the subscriber loses its current CT, it is possible for it to reconnect using an older CT. However, if it has acknowledged events later than this CT, the subscriber may get gap messages in lieu of events it has already acknowledged. Most durable subscribers want to be able to recover from their own failure without receiving such gaps. Hence they store their CT in a persistent store, and update it in the context of the transaction that consumes one or more messages.

This model is more flexible than JMS since it supports explicit gap messages, and does not require the CT to be stored by the messaging system. Storing the CT inside the messaging system results in distributed transactions to update the CT consistently with the application's state, which decreases performance and availability.

### 3 Event Retention and Release Protocol

Message routing and recovery is accomplished using a tree of knowledge and curiosity streams, as described in [10]. The root of this tree is a pubend. Intermediate knowledge streams serve as caches of data that increase scalability of recovery, by responding to nacks, and curiosity streams consolidate nacks from multiple SHBs. The knowledge stream is an extension of that described in our earlier work, and contains four kinds of ticks: Q (unknown), S (silence), D (data), and L (lost). The D tick represents an event published by an application. An S tick means that either there was no event at this timestamp or it was filtered upstream and is therefore not relevant. An L represents that the pubend has discarded information about whether this tick was S or D.

The release protocol described next is responsible for converting an increasing prefix of ticks at the pubend into L.

We define some terms used in this section and the rest of the paper.

**Definitions**  $T(p)$  represents the current time at pubend  $p$ . We loosely define a subscriber to be in *catchup mode* if it is still consuming events that it missed because it was disconnected. Subscribers in catchup mode can be consuming events at different points in a knowledge stream, while non-catchup mode subscribers are all at the same point in the stream.

An SHB maintains a single consolidated knowledge stream that delivers events in-order to all connected subscribers that are not in catchup mode. The timestamp  $latestDelivered(p)$ , at an SHB, represents the latest event that it has delivered to all non-catchup subscribers<sup>4</sup>. Since there is non-zero latency in the system  $latestDelivered(p) < T(p)$ .

The predicate  $catchup(s, p)$  is true for a subscriber in catchup mode and false otherwise. It is false when subscription  $s$  is first created, and becomes true the instant the subscriber disconnects. On reconnection, it remains true, until the subscriber has recovered all messages till  $latestDelivered(p)$ , at which point it transitions to false, and remains false till the next disconnection.

**Release Protocol** An SHB *releases* a timestamp once all durable subscribers hosted by it, whether they are connected or not, have acknowledged that timestamp. At each node in the knowledge graph, the *release protocol* maintains (1) the minimum timestamp that all downstream SHBs have released, and (2) the minimum timestamp of all  $latestDelivered(p)$  at downstream SHBs.

The pubend is the root node, hence the minimum values at the pubend are the minimum across all SHBs. Let the current latestDelivered and release timestamps at pubend  $p$  be  $T_d(p), T_r(p)$ . These are used to decide when to change a tick to L. By definition, the invariant  $T_r(p) \leq T_d(p)$  holds.

Even in the absence of any early-release policy, a tick at time  $t$  can be changed to L by the pubend when  $t \leq T_r(p)$ . Since the system uses less resources to support connected subscribers that are not in catchup mode, it considers such subscribers well-behaved. The pubend ensures that such subscribers do not receive any gap messages (due to early-release), by not changing any tick  $t$  to L if  $t > T_d(p)$ . Therefore, the early-release policies control what happens to ticks in the range  $(T_r(p), T_d(p)]$ . We consider one example of such a policy.

**Example: PHB Controlled Policy** Each pubend  $p$  has a maximum retention time,  $maxRetain(p)$ , after which it discards an event even if some disconnected durable subscribers have not received it. More precisely, the pubend

<sup>4</sup>The SHB is described in more detail in the next section.

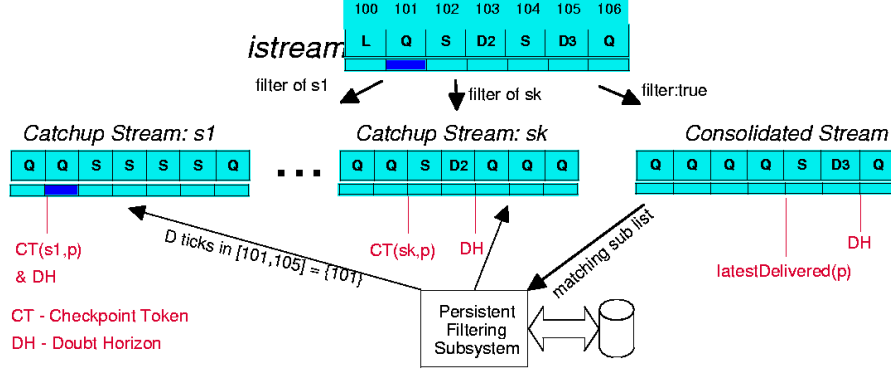


Figure 1. The Knowledge Graph inside an SHB

changes a tick  $t$  to L when the following condition becomes true:

$$t \leq T_r(p) \vee [(t \leq T_d(p)) \wedge (T(p) - t > \text{maxRetain}(p))]$$

A subscriber in catchup mode is at risk of receiving a gap message if its  $CT(s, p)$  falls behind  $T(p)$  by more than  $\text{maxRetain}(p)$ .

## 4 Subscriber Hosting Broker

In this section, we describe the functions performed by the subscriber hosting broker (SHB) that are critical to scalably support durable subscribers.

A connected durable subscriber can be either in catchup or non-catchup mode for a pubend  $p$ . The SHB maintains a separate (knowledge and curiosity) stream for each connected subscriber that is in catchup mode. For non-catchup subscribers the SHB maintains one consolidated stream (constream). When the subscriber switches from catchup to non-catchup, the separate catchup stream is discarded. Under normal conditions, an SHB operates with a high percentage of non-catchup subscribers, hence consolidating the streams saves a significant amount of system resources.

This is depicted in figure 1, where subscribers  $s_1$  through  $s_k$  are in catchup mode for pubend  $p$ , and subscribers  $s_{k+1}$  through  $s_n$  are non-catchup. The figure shows knowledge stream values in square boxes. The corresponding curiosity streams are in thin boxes, with shaded boxes representing nacks.

The istream contains knowledge received from upstream, and consolidated nacks from downstream. A message containing knowledge ticks enters the SHB at the istream, where it is accumulated. The changes are then filtered through the filters for  $s_1$  through  $s_k$  and accumulated

in their knowledge streams. Finally, the changes are accumulated into the constream. The constream delivers messages in order to the Persistent Filtering Subsystem (PFS) and the non-catchup subscribers.

### 4.1 Consolidated Stream

The consolidated stream maintains the following timestamp values:

- $\text{latestDelivered}(p)$  - The timestamp of the latest event that has been delivered to all non-catchup subscribers.
- Doubt Horizon ( $DH$ ) - The highest timestamp such that all ticks between  $\text{latestDelivered}(p)$  and the timestamp are not Q. All events between  $\text{latestDelivered}(p)$  and  $DH$  are delivered to subscribers in sequence.
- $\text{released}(s, p)$  - the timestamp of the latest event which durable subscriber  $s$  has acknowledged.
- $\text{released}(p)$  - the highest timestamp that can be released. It equals the following:

$$\min[\text{latestDelivered}(p), \min_{\forall s}(\text{released}(s, p))]$$

In the figure,  $\text{latestDelivered}(p)$  is equal to 104. Since the doubt horizon is now 106, the constream will deliver D3 (as an event message) to non-catchup subscribers that match D3, and to the PFS and then advance  $\text{latestDelivered}(p)$  to 106. The last hop from the SHB to a subscriber is a FIFO link, and delivery of a message to the subscriber is complete, as soon as it is enqueued onto this FIFO link, i.e., it does not involve waiting for an acknowledgment from the subscribing application. In contrast, delivery of a message to the PFS is complete only after it has successfully

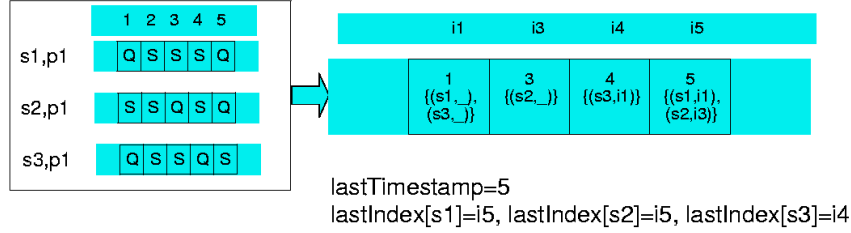


Figure 2. Persistent Filtering Subsystem

logged the timestamp and matching subscriber ids to persistent storage.

A subscriber that has not matched any event for a period of time is delivered a silence message to prevent its CT from lagging far behind. Note that the constream never delivers a gap message to a subscriber, since a tick value at time  $t$  cannot be changed to L until  $t \leq latestDelivered(p)$ .

The  $latestDelivered(p)$  and  $released(s, p)$  timestamps are maintained in persistent storage since they need to survive SHB crashes. Our implementation maintains these in database tables. When the constream recovers after a crash, it resumes delivery to non-catchup subscribers and the PFS from  $latestDelivered(p)$  onwards.

A new subscriber, i.e., one that is connecting for the first time, is given a starting point (CT) such that  $CT(s, p) = latestDelivered(p)$ . Due to this, a new subscriber is in non-catchup mode.

**Catchup Stream and Switchover to non-catchup** On reconnection, subscriber  $s$  provides the SHB its checkpoint token ( $CT(s)$ ) as the desired starting point for resumption of delivery.

Typically,  $CT(s, p)$  at reconnect time is less than  $latestDelivered(p)$ . Since the consolidated stream is only capable of delivering events  $> latestDelivered(p)$ , the SHB creates a catchup stream with the doubt horizon set to  $CT(s, p)$ . The catchup stream uses the PFS to retrieve information about which timestamps greater than  $CT(s, p)$  matched subscriber  $s$ . For instance, in figure 1,  $CT(s1, p) = 100$ , and the catchup stream has been told by the PFS that the only  $D$  tick for  $s1$  in the timestamp range  $[101, 105]$  is at timestamp 101. Consequently, the stream has set the tick value for 102-105 to S, and the one at 101 to Q, and initiated a nack for 101. Our implementation includes a flow control scheme, between the SHB and the subscribing client, to control the rate of nacks initiated, so as not to overwhelm the client with catchup event messages. Our congestion control scheme, described in [14], ensures that brokers allocate enough resources to catchup.

When the doubt horizon of the catchup stream becomes  $\geq latestDelivered(p)$ , the catchup stream is removed and

the subscriber starts getting messages delivered from the constream.

## 4.2 Persistent Filtering Subsystem

The persistent filtering subsystem (PFS) stores information about which events match a durable subscriber. This is an important optimization since it avoids retrieving and refiltering events that did not match the subscriber.

At a concrete level, the PFS stores Q, S and L knowledge ticks for each subscriber. A precise PFS implementation stores a Q tick for subscriber  $s$  only if there is an event at that timestamp which matches the subscriber. An imprecise implementation may represent some S ticks as Q, which does not affect correctness of the delivery protocols. It can be used to trade off PFS write performance with respect to the cost of retrieving and refiltering unnecessary events. Our current implementation is precise.

The L ticks represent a prefix of time where everything is L, and are the same across all subscribers, while Q and S ticks will typically be different across subscribers. Our description focuses on how the PFS stores and retrieves Q and S ticks.

**API** The write API of the PFS is used by the constream, while the read API is used by the catchup streams. Each write call contains a monotonic timestamp representing the time of a Q tick, and a list of subscribers for which it is Q. For subscribers not in this list, the tick value is S.

A catchup stream for subscriber  $s$  reads every Q and S tick for this subscriber, starting from timestamp  $CT(s, p)$ , up to the value of the timestamp  $latestDelivered(p)$  when the subscriber catches up. Instead of reading one Q tick at a time, the PFS supports a batch read API to read a large number of Qs. The S ticks between adjacent Q's are implicitly represented in the read buffer. Typically, the read buffer used by a catchup stream is large enough to retrieve all the Q and S ticks for the time period the subscriber was disconnected using one read operation. While the catchup stream is retrieving and delivering the events corresponding to this batch of Qs, the value of  $latestDelivered(p)$  may advance further. This triggers another read operation by the

catchup stream, after it has successfully delivered all events corresponding to the previous read.

**Implementation** The PFS uses the Log Volume described in [8], which is similar in some ways to Log-structured File Systems. A Log Volume can contain multiple Log Streams, in our case one for each pubend known to the SHB. Each Log Stream implements a write API that supports (1) appending a record to the stream, where each such appended record is assigned a unique monotonic index number, and (2) chopping (discarding) all the records up to some index number. The Log Volume multiplexes multiple log streams onto a single file, and supports efficient retrieval of records by index number.

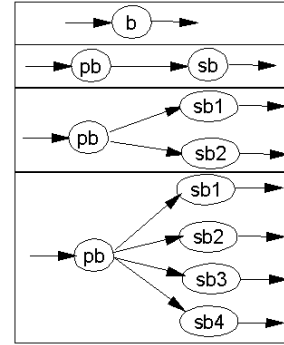
Say there are  $n$  subscribers and  $m$  pubends. The PFS needs to maintain Q and S ticks for each subscriber, pubend pair. Typically  $n \gg m$ , therefore it is important to compact the knowledge streams across all subscribers. This compaction is done, for a given pubend, by mapping Q and S ticks for all subscribers onto a single Log Stream such that one record is written for each timestamp that has a Q tick for at least one subscription. This optimizes write throughput. No record is written for a timestamp for which all subscribers had an S tick, i.e., no subscription matched.

Each record written contains the timestamp of the Q tick, and the list of subscribers for which the tick value is Q. For each subscriber in this list of subscribers, the previous index of a Q tick record is also written as part of the record. Figure 2 shows an example with three subscribers, s1, s2 and s3 and their knowledge streams for time 1-5. The record written for the Q tick at time 1, has an index  $i_1$  and similarly for time 3, 4, 5. The special  $\_$  index represents the first record in the stream for a subscriber. The PFS maintains *lastIndex* and *lastTimestamp* metadata in a database table. The *lastTimestamp* represents the latest Q tick written by the PFS, and *lastIndex(s)* represents the index of the latest Q tick record that contains this subscriber. This structuring of the data is used to perform batch reads. For instance, say s3 wants to know its knowledge stream for time interval [1, 10]. Knowing that the last timestamp it has seen is 5 the PFS sets all ticks from [6, 10] in the read buffer to Q. Next, all ticks with timestamp greater than the record at *lastIndex(s3)* and  $\leq$  *lastTimestamp* are S. Then following the backpointer from  $i_4$ , the PFS retrieves  $i_1$ . This means that ticks 1 and 4 are Q and 2 and 3 are S.

By doing a large batch read that retrieves all the Q ticks up to *lastTimestamp* into a read buffer, the PFS does not need to repeatedly traverse the same Q tick for the subscriber.

## 5 Experimental Results

We have implemented these algorithms in the Gryphon system. The Gryphon system is mostly Java code, for portability,



**Figure 3. Topology for scalability experiments**

with native libraries for network I/O. Connections between brokers in the overlay network are implemented using TCP. The persistent data-structures in the SHB are stored in DB2 version 7 [2], and accessed using a JDBC API [4] driver that uses shared memory to communicate with the DB2 server located on the same machine.

The experiments in this section run brokers on IBM RS/6000 F80 servers (with 6 processors), running AIX, with SSA160 disk drives. All published events contained a 250 byte application payload. Including headers, the size of the event was 418 bytes. We disabled early-release in these experiments, since we wanted to observe system behavior when no gap messages are delivered to subscribers. Here is a summary of our results.

1. The end-to-end event latency for a 5 hop broker network is 50ms, of which 44ms is due to event logging at the PHB. Since our system logs an event only once, the end-to-end latency is low.
2. The system scales linearly from 20K events/s with 1 SHB to 79.2K events/s with 4 SHBs in the absence of any subscriber disconnection. Under moderate subscriber disconnection/reconnection, it scales from 17.6K events/s to 69.6K events/s. A microbenchmark with the same workload demonstrates that the PFS is over 5x faster than logging events at the SHB.
3. Under SHB failure that forces all subscribers to remain disconnected simultaneously for about 40s, there is little overhead on the PHB during catchup due to nack consolidation. However, the SHB rate reduces to about 10K events/s when all subscribers have a separate catchup stream (compared to 20K events/s with only the constream). This demonstrates the importance of stream consolidation.

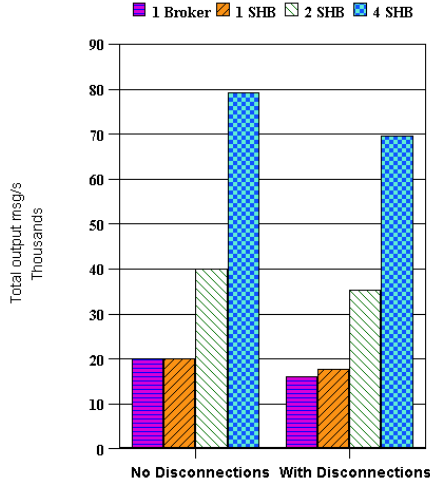


Figure 4. Peak Event Rate

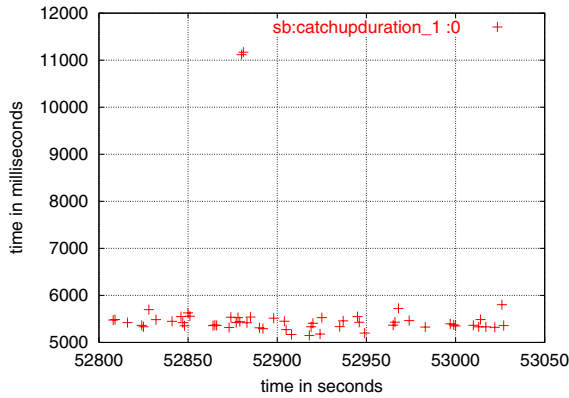


Figure 5. Catchup Duration

## 5.1 Scalability Results

Figure 3 shows the topologies used in the scalability experiments. The goal is to add subscriber hosting brokers and measure the ability of the system to support a larger number of subscribers, while keeping the per subscriber event rate constant. The input event rate in all topologies is 800 events/s, distributed equally over 4 pubends, and subscriptions are such that each subscriber receives 200 events/s.

We show a 1 broker, a 2 broker network with 1 subscriber hosting broker (SHB), and 2 SHB and 4 SHB networks. The 1 broker and 1 SHB networks are compared to demonstrate that the capacity of the 1 SHB network is similar to the 1 broker network (as the CPU overhead of logging to disk is negligible).

The first set of bar charts in figure 4 show the aggregate rate to subscribers with no subscriber disconnection.

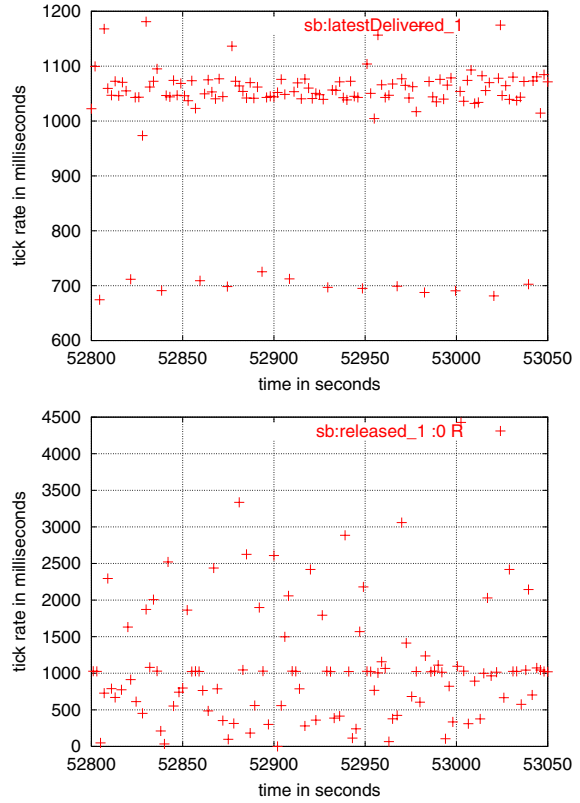


Figure 6. latestDelivered(p) and released(p) rate with disconnections

tion/reconnection. It scales almost linearly from 20K events/s for 1 SHB to 79.2K events/s for 4 SHBs. The CPU idle time at the PHB decreases slightly from 69% to 59% when going from 1 SHB to 4 SHBs. The second set of bar charts in figure 4 show the aggregate rate to subscribers when each subscriber independently disconnects every 300s, remains disconnected for 5s (so it misses 1000 events), and then reconnects. The aggregate rate increases almost linearly from 17.6K events/s to 69.6K events/s (an increase from 88 subscribers to 348 subscribers) when going from 1 SHB to 4 SHBs. With 348 subscribers using 4 SHBs, at least 1 subscriber is reconnecting and catching up in the system at any instant of time. The peak performance in the presence of subscriber disconnections, 69.6 events/s, is about 88% of the peak in the absence of disconnections. In these experiments, updates to *released(s)* (maintained in DB2) are performed periodically every 250ms.

### 5.1.1 latestDelivered(p), released(p) and catchup durations

We examine detailed behavior of the SHB for the 2 broker network with 1 PHB and 1 SHB and subscriber dis-

connection/reconnection. Figure 5 shows the catchup duration of the subscribers, on the Y axis, for a time interval of about 250 seconds (from 52800-53050). Catchup durations are usually between 5 and 6 seconds. Figure 6 shows the rate of advance (on the Y axis), of  $latestDelivered(p)$  and  $released(p)$  for 1 of the 4 pubends. Since  $latestDelivered(p)$  is not affected by disconnected subscribers it steadily advances at a rate close to 1000 tick milliseconds every second of real time. The periodic drop in rate to about 700 tick ms every second, is due to periodic garbage collection in the Java VM running the SHB. In comparison,  $released(p)$  shows much larger variation since subscriber disconnection causes it to stop advancing.

### 5.1.2 PFS Microbenchmark

To compare PFS performance versus logging an event for each subscriber, at the SHB, we ran a microbenchmark which represented the preceding no disconnection 1 SHB experiment: 800 events/s input rate, 100 subscribers, 200 events/s per subscriber, 418 byte messages (250 byte payload). For each subscriber both the PFS and the event log is synced every 200 events, i.e., every second of the workload, and maintains information for the last 1000 events, i.e., the last 5 seconds. The benchmark represents 100s of real time, i.e., a total of  $800 \times 100$  events are received at the SHB. The PFS ran the benchmark in 11088ms. Compared to event logging for each subscriber, PFS logged 25x less data, and was over 5x times faster in completing the benchmark.

## 5.2 JMS auto-acknowledge Results

The above results are with each subscriber maintaining its own CT value. In contrast, for a subscriber  $s$  using the JMS API, the SHB needs to maintain  $CT(s)$  in persistent storage (DB2). Whenever the JMS durable subscriber commits after consuming some events, the corresponding changes to the  $CT(s)$  vector at the SHB are committed to the database. The JMS auto-acknowledge mode is the most severe, since the subscriber commits after consuming each event, so  $CT(s)$  is updated and committed for each event.

In our experiments with a single SHB, we measured the peak aggregate rate for 25 subscribers and 200 subscribers, which was 4K events/s and 7.6K events/s respectively. The bottleneck at the SHB for JMS auto-acknowledge is the update and commit throughput of the database, and independent of our SHB implementation. To get the rate of 7.6K events/s to 200 subscribers, the SHB used 4 JDBC connections each associated with a thread. Requests to update  $CT(s)$  were assigned to one of the threads based on the subscriber id. Each thread explicitly batched all the waiting requests into one database transaction. To improve performance, the hardware write-cache in the SSA disk controller

was utilized. The cache is battery-backed and hence writes in the cache are safe despite crash failures.

## 5.3 SHB failure and recovery

SHB failure and recovery forces all subscribers into catchup mode at the same time.

**Test Setup:** Our test setting is the 2 broker (1 PHB and 1 SHB) network used in the scalability experiments, with the same input rate, number of pubends, and per subscriber rate (200 events/s). We used 40 subscribers, each with its own TCP connection to the SHB, spread over 5 client machines. Since each client machine runs 8 subscribers, the aggregate rate received at each machine under normal operation is 1600 events/s.

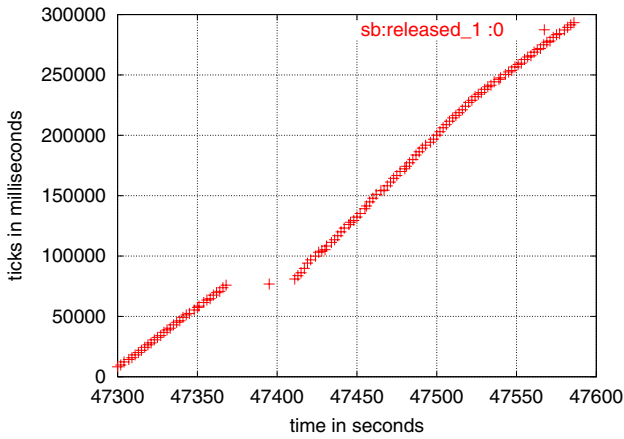
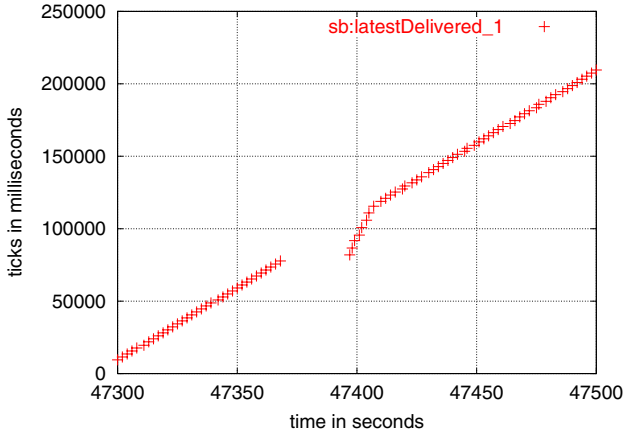
After running normally for a period of time, the SHB is failed for 25 seconds. After the SHB recovers, we delay reconnection of the 40 durable subscribers until the constream has nacked and received all the events it missed. We then reconnect all the durable subscribers. This delay separates the effect of constream nacking from the catchup streams nacking, and enables us to observe the differences in behavior. Due to this delay, the durable subscribers are disconnected for about 36-40s (mean=37.55s, standard deviation=1.16s).

**Results:** The first plot in Figure 7 shows the value of  $latestDelivered(p)$  for one of the pubends. There is no change for about 25 seconds while the SHB is down. When the SHB recovers, the slope is much higher, about 5 times the normal rate, as the constream nacks the events it missed. When the constream has retrieved all the missed events, the slope returns to normal.

Immediately after the  $latestDelivered(p)$  slope returns to normal, the test setup reconnects all the durable subscribers. Till this point in time the value of  $released(p)$  does not change, as can be seen in the second plot of Figure 7. During catchup, the value of  $released(p)$  advances at a rate slightly higher than the normal slope, and returns to normal at the end of catchup. Notice the slight change in slope between 47500 and 47550 seconds.

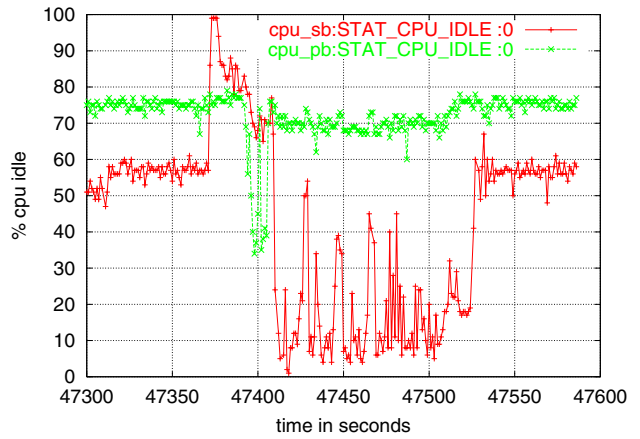
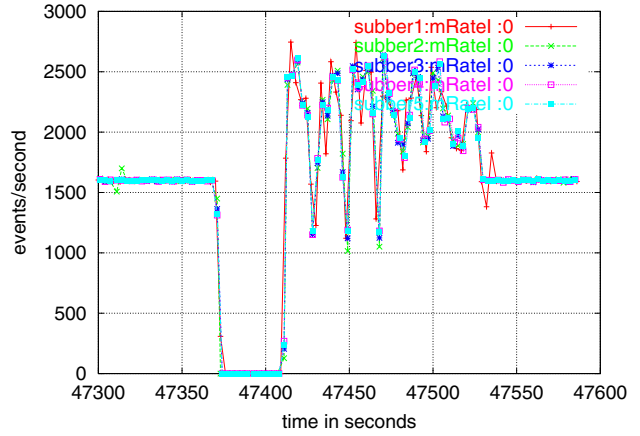
The average catchup duration across all subscribers is 116 seconds. This value is high since all subscribers are in catchup mode at the same time, and maintain separate catchup streams. During this catchup duration, a catchup stream nacks and retrieves the 37 seconds of events the subscriber missed because it was disconnected, and the 116 seconds of events sent by a pubend while the subscriber was catching up.

The first plot in Figure 8 shows the aggregate event rate at each of the 5 machines running the subscribers. The normal rate is 1600 events/s, before the SHB crashes. After the subscribers reconnect the rate varies but is on average higher than 1600 events/s. This variation in rate is primarily due to PFS reads that are in time synchrony. These



**Figure 7. latestDelivered(p) and released(p) with SHB crash and recovery**

experiments use a read buffer that can hold 5000 Q ticks, which represents 25 seconds of events missed by a subscriber (since the subscriber matches 200 events/s). Each read fills up the buffer to its maximum size or with the maximum Q ticks available, whichever is smaller. A new read does not occur until the catchup stream has nacked all the Q ticks in the current buffer. Because of this large read buffer, 87% of the read operations on the PFS read all the Q ticks up to the *lastTimestamp(p)* value maintained by the PFS. As discussed in section 4.2, this means that most reads are very efficient. The reads performed in the early part of catchup read a full buffer of 5000 Q ticks, but as catchup progresses, the reads return fewer Q ticks and are of shorter duration. Since all subscribers have the same event rate and have missed the same time period of messages, they all perform PFS reads in approximate time synchrony. While doing a read, the event rate for that subscriber drops since new nacks cannot be initiated, and because CPU resources are used for the PFS read. Since reads are in time synchrony, the event rate for all subscribers drops at the same time. As



**Figure 8. message rate & cpu utilization with SHB crash and recovery**

catchup progresses, the reads become shorter, and the variation in event rate becomes lower.

The second plot in Figure 8 shows the CPU idle time during SHB crash/recovery. During the time the SHB is performing catchup for durable clients, the CPU utilization at the PHB is only slightly higher. This increase is due to servicing nacks for events needed by durable subscribers. However, the increase is small due to nack consolidation for events needed by multiple durable subscribers. In comparison, the CPU idle time for the SHB shows a significant drop while the subscribers are in catchup, and then returns to normal. This shows that the effect of subscriber catchup is largely localized to the SHB.

## 6 Related Work

Most Internet scale pub/sub systems, such as SIENA [11], only offer best-effort delivery and no subscriber durability. The Elvin system [16] supports disconnected subscribers by using a proxy client that

stores events on behalf of disconnected subscribers. The subscriber has to reconnect to the same proxy, and the same event may be logged multiple times in the system, proportional to the number of proxy clients. They do not discuss how they handle the failure case of the proxy disconnecting from the system, and do not present any experimental results.

Exactly-once delivery to durable subscribers is offered by most commercial messaging systems such as IBM's MQseries, SonicMQ, Tibco Rendezvous TX, FioranoMQ, OpenQueue. Though most technical details of these systems are not available, such systems are typically message queuing systems that use a store and forward approach which involves logging an event at every hop in a multi-broker network. Some performance data is available in the public domain at vendor web sites [1, 6, 5], however, these studies only measure single-broker peak performance in the absence of subscriber disconnection/reconnection and broker failures. Due to this reason, as well as the fact that the hardware used in each of these studies is different, we do not compare them with our results.

There is a large body of work on reliable multicast in the networking literature. Most of it, such as SRM [12], LBRM [13], deals with building an end-to-end reliable protocol in which all messages are of interest to all members of the group. Each application that is receiving from the group is responsible for its own reliability, by detecting missing sequence numbers, and sending nacks. In contrast, pub/sub messaging systems allow the subscribers to receive a subset of published messages, which makes the problem and consequently the solution very different from reliable multicast.

## 7 Conclusion

We have developed algorithms for scalably supporting durable subscriptions in a pub/sub system that is organized as an overlay network. Each event only needs to be logged once in the whole system, regardless of the number of subscribers that are interested in it and where they are located. Caching events at intermediate nodes, and the persistent filtering subsystem, ensure scalability of handling reconnecting durable subscribers. An administrative *early-release* model allows events to be discarded from persistent storage to guard the system against incorrect or misbehaving subscribers. We have presented experimental results that show high performance and scalability.

Future work includes experimentally examining the effect of different event cache sizes and management policies, on the catchup rate of reconnecting subscriptions.

**Acknowledgements** We thank the anonymous reviewers, and our shepherd, JoAnne Holliday, for their comments.

## References

- [1] A benchmark comparison between fioranomq and smq. In [http://www.fiorano.com/products/fmq/performance\\_comparison.htm](http://www.fiorano.com/products/fmq/performance_comparison.htm).
- [2] Db2 product family. In [www.software.ibm.com/data/db2](http://www.software.ibm.com/data/db2).
- [3] Java (tm) message service. In <http://java.sun.com/products/jms/>.
- [4] Jdbc api. In <http://java.sun.com/products/jdbc>.
- [5] Sonicmq: High performance messaging with jms. In [http://www.sonicsoftware.com/white\\_papers/sonic40\\_vs\\_mqseries52.pdf](http://www.sonicsoftware.com/white_papers/sonic40_vs_mqseries52.pdf).
- [6] Sun one: High performance jms messaging. In [http://www.sun.com/software/products/message\\_queue/wp\\_JMSperformance.pdf](http://www.sun.com/software/products/message_queue/wp_JMSperformance.pdf).
- [7] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the Principles of Distributed Computing, 1999*, pages 53–61, May 1999.
- [8] S. Bagchi, R. Das, and M. Kaplan. Design and evaluation of a logger-based recovery subsystem for publish-subscribe middleware. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2002)*, 2002.
- [9] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, 1999*, pages 262–272, 1999.
- [10] S. Bholra, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2002)*, pages 7–16, 2002.
- [11] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, December 1998.
- [12] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, November 1996.
- [13] H. W. Holbrook, S. K. Singhal, and D. R. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. In *Proceedings of ACM SIGCOMM*, pages 328–341, August 1995.
- [14] P. Pietzuch and S. Bholra. Congestion control in a reliable scalable message-oriented middleware. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*.
- [15] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7:149–174, 1994.
- [16] P. Sutton, R. Arkins, and B. Segall. Supporting disconnectedness - transparent information delivery for mobile and invisible computing. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2001.