

# Exactly-once Delivery in a Content-based Publish-Subscribe System

Sumeer Bhola

Robert Strom

Saurabh Bagchi

Yuanyuan Zhao

Joshua Auerbach

{sbhola, robstrom, sbagchi, yuanyuan, josh}@us.ibm.com

IBM T.J. Watson Research Center

## Abstract

*This paper presents a general knowledge model for propagating information in a content-based publish-subscribe system. The model is used to derive an efficient and scalable protocol for exactly-once delivery to large numbers (tens of thousands per broker) of content-based subscribers in either publisher order or uniform total order. Our protocol allows intermediate content filtering at each hop, but requires persistent storage only at the publishing site. It is tolerant of message drops, message reorderings, node failures, and link failures, and maintains only “soft” state at intermediate nodes. We evaluate the performance of our implementation both under failure-free conditions and with fault injection.*

## 1 Introduction

This paper discusses the *guaranteed delivery* service of the Gryphon system. Gryphon is a scalable, wide-area content-based publish-subscribe system, employing a redundant overlay network of *brokers* [2, 3].

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 subscriber’s filter. The guarantee is honored as long as the subscriber remains connected, even in the presence of intermediate broker and link failures.

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. That is, the messages may be used by the subscribing application to accumulate a view (e.g., a

snapshot of a sporting event), where missing or reordered messages could cause an incorrect state to be displayed.

Existing solutions to this problem, such as store-and-forward routing, DCP [11], or group multicast are either less efficient, or work with a different set of assumptions. These differences are discussed in more detail in Section 5.

The main contributions of this paper are:

1. A *knowledge graph* abstraction that models propagation of knowledge from publishers to subscribers through filter and merge operations, and propagation of demands for knowledge in the reverse direction.
2. A protocol based on this knowledge model that tolerates broker crashes and dropped and re-ordered messages, that does not require hop-by-hop reliability of messages, and that requires stable storage only at the publishing broker and only *soft-state* [10] everywhere else.
3. Experimental results demonstrating: a low, constant overhead of our protocol compared to best-effort delivery, localized effects of failures without “nack explosions,” and rapid recovery using alternate paths, when available, or after repair of the failures.

The rest of the paper is organized as follows: Section 2 describes the abstract knowledge model, and section 3 describes the algorithm and its implementation in the Gryphon system. Section 4 presents the experimental results from this implementation. Section 5 presents related work in the area of reliable message delivery. Section 6 presents our conclusions.

## 2 Abstract Model

We represent the Guaranteed Delivery service as an abstract *knowledge graph* such as the one in Figure 1. The knowledge graph is a directed acyclic hypergraph whose nodes contain state called *streams* and whose hyperedges

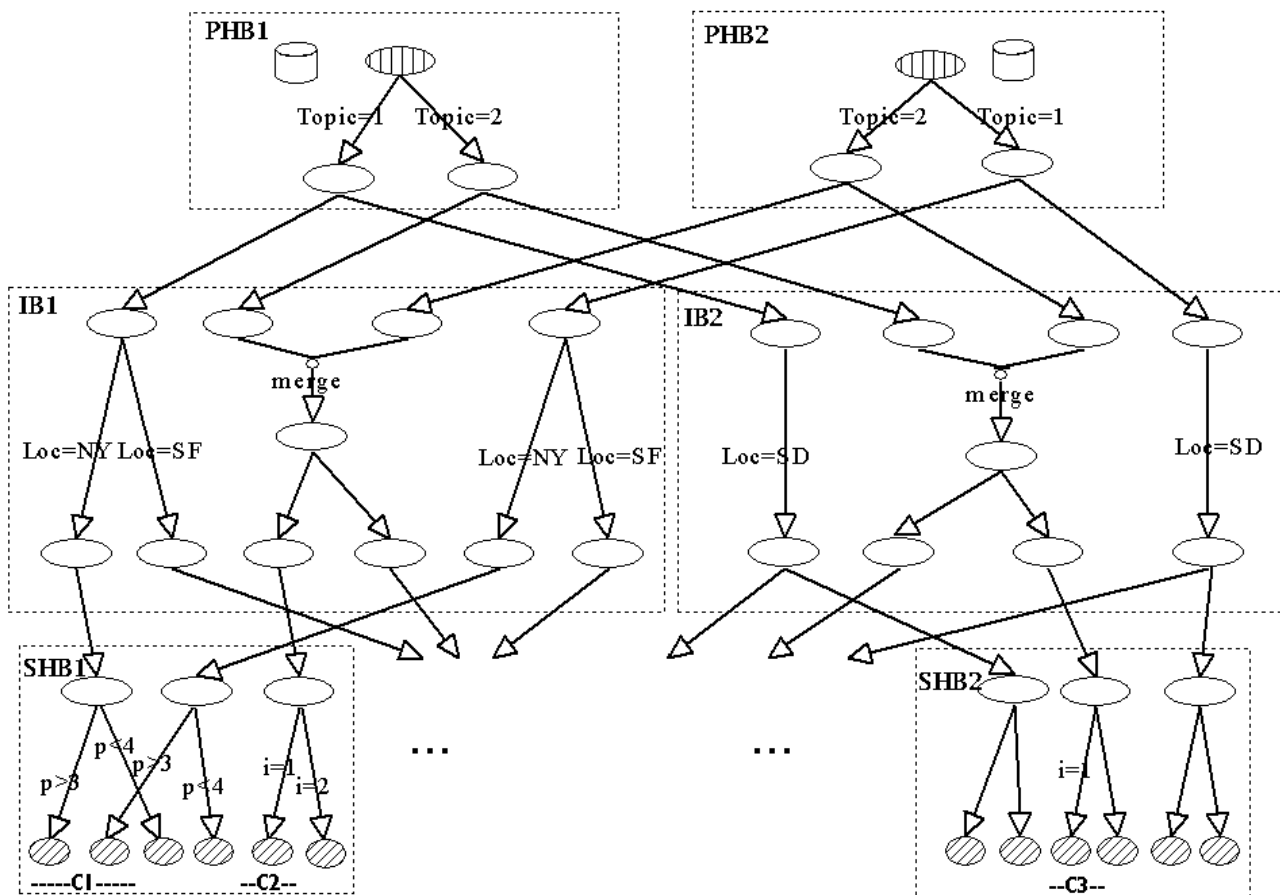


Figure 1. Example: Knowledge Graph

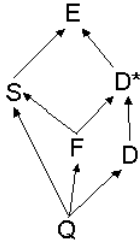
are labelled with *operations*. Each operation is one of *filter* (shown as an optional filter predicate) or *merge*. The streams contain *knowledge* about which ticks of time contain data messages and which do not, and *curiosity* about how urgently the stream requires each tick's knowledge.

The nodes of the hypergraph are partitioned into a collection of *virtual brokers*. In the figure, these are shown by the dotted rectangles labelled PHB1, PHB2, IB1, etc. (the significance of these acronyms is discussed in Section 3). The edges connecting nodes in different virtual brokers can be viewed as *virtual links*. The mapping of virtual brokers and links to physical broker machines and physical network connections is described in Section 3.

Knowledge is created at source nodes, called publisher endpoints or *pubends* (shown with vertical hash-marks in the figure), as a consequence of a client's publishing mes-

sages to a broker hosting that pubend. Knowledge is consumed at sink nodes, called subscriber endpoints or *subends* (shown with diagonal hash-marks in the figure), and then delivered to subscribing clients. Knowledge propagates from pubends to subends (*downstream*), under the control of each edge's operation. Curiosity propagates along the edges in the reverse direction (*upstream*), also under the control of the edge's operation.

The knowledge graph models: (1) the propagation of knowledge and curiosity, (2) the sending of messages over links, and (3) the loss of knowledge and curiosity due to failures. Failures include broker crashes, link outages, dropped or reordered messages on links, but not Byzantine failures. In this section, we assume that the abstract knowledge graph does not change dynamically, although the mapping of abstract nodes in the graph to physical brokers may change.



**Figure 2. Knowledge State Lattice**

This assumption has been weakened in our actual implementation in order to support dynamic subscription. These modifications are beyond the scope of this paper.

## 2.1 Streams, Knowledge, and Curiosity

Each node corresponds to a stream consisting of a *knowledge stream* and a *curiosity stream*. Both knowledge and curiosity streams are keyed by time, which is discretized into *ticks*. Each tick  $t$  is associated with a *knowledge value*,  $K_t$ , and a *curiosity value*,  $C_t$ .

Each  $K_t$  value can be one of the following, and the values/states are organized into a lattice shown in figure 2.

- Q: meaning lack of knowledge.
- D: meaning that a data message was published during tick  $t$ . The data in the message is part of the value.
- D\*: meaning that a data message was published during tick  $t$ , and additionally, that every downstream subscriber who needed that data message has delivered it and no longer needs it.
- S: silence, meaning either no message was sent during tick  $t$ , or else a message was sent but was filtered out en route to this stream.
- F: final (or “don’t-care”): the greatest lower bound of D\* and S, meaning that no data message is needed downstream, either because none was ever sent to this stream (the tick was once S) or else because the data is no longer needed (the tick was once D\*).
- E: error — this state should never be reached in a proper implementation even under failure conditions.

States change values in two ways: (1) becoming monotonically higher by *knowledge accumulation*, and (2) becoming monotonically lower by *forgetting*. Knowledge accumulation models what happens when messages are received. Forgetting models the loss of information due to

failure or deliberate discarding. For example, each knowledge tick can drop from its current value to Q. In the current algorithm, any S or D\* tick is automatically lowered to F.

Knowledge accumulates when an upstream node outputs a value to a downstream node via some edge operation. The node changes its state to the lattice least upper bound between its previous state and the new value.

Any stream except the pubend can “forget” information. This models the fact that only the pubends store information stably; all other state is “soft”. Of course, the pubend remains as a single point of failure in this model, but that can be addressed by employing orthogonal techniques such as shared and replicated logging disks.

An output computed by a transformation at some edge operation can be lost or delayed on the way to the downstream stream. This models the fact that links can either fail, or can deliver values of ticks out of order. Our algorithm assumes that there will eventually be a “long enough” failure free period along the path between the source and each sink. More precisely, if a tick is infinitely often sent through the graph, it will eventually arrive without failure.

### 2.1.1 Curiosity

Each curiosity value  $C_t$  can be one of the following:

- C: curious, meaning that some downstream subscriber has an increased need to know  $K_t$ ,
- A: anti-curious/ack, meaning that no downstream subscriber needs to know  $K_t$ , either because the data has already been delivered to the subscriber ( $K_t$  is or was D\*), or because  $K_t$  is or was S,
- N: neutral — in this algorithm, that implies that a knowledge value of D may be sent but need not be resent, and a knowledge value of S need not be sent.

The default curiosity is N.

Curiosity and knowledge are linked in that a tick whose knowledge state becomes F is assigned a curiosity of A and vice-versa.

Under normal (failure-free) operation, messages are not lost, and gaps in the time-ordering (due to links reordering messages) will be transient. In that case, subscribers will never need to be curious. Subscribers will receive D messages in time order, and will ack the interval from tick zero to the tick of their latest received D message. The ack will be propagated upstream. When all child nodes of a stream become anti-curious for a tick, then the parent stream can also become anti-curious for that tick. Hence, ack information is consolidated as it travels upstream. Eventually, all messages will be delivered to all subscribers that need the messages, and an ever-growing prefix of all streams will become both F and A.

If there are failures, either the fact that the message stream is not advancing, or the fact that there is a persisting gap of Q ticks between non-Q ticks will trigger certain nodes to become curious about particular ticks. This curiosity will flow upstream, and some upstream node will answer the curiosity by indicating which Q ticks are D, and which F (in this algorithm F and S are treated the same), eventually filling the gaps and advancing the stream.

## 2.2 Pubends

Pubends are the source nodes in the knowledge graph. An implementation of a pubend can represent either a single publisher, or (more typically), a consolidation of multiple publishers sharing a single pubend. The knowledge stream at a pubend has the form  $F * [D|F] * Q * ^1$  — that is, a range of “past” ticks whose values are no longer needed because they have all been delivered, a range of “present” but unacknowledged ticks, and an unknown “future”.

The implementation of a pubend assigns a unique tick number to each received message, assigns D to that tick, assigns F to all older ticks since the previous D, and logs the message to stable storage. Messages that are successfully logged are considered published and will be delivered, while those that are not logged are considered *not* published and will *not* be delivered.

A simple implementation leaves all future ticks in state Q. When no new message has been published for long enough, the pubend changes a range of Q ticks to F without assigning any D tick. In a previous paper [1] we showed that if a pubend is aware of its expected publication frequency relative to other pubends, it can improve the performance of downstream merge operations by pre-assigning F to some number of future ticks after the latest D.

We assume that some mechanism is in place to assure that no two pubend streams that will ever be merged place different data messages on the same tick, e.g. by requiring each pubend to assign its D messages to ticks whose low-order bits correspond to its pubend number.

## 2.3 Subends

The knowledge stream of a subend is used to deliver messages to subscribing clients. We allow for either publisher order or total order.

If the subscriber requires *publisher order*, it is acceptable for the message streams of multiple publishers to be interleaved arbitrarily. In practice, we consolidate multiple publishers in a single pubend, so it is actually the pubend streams that are interleaved. This is illustrated in Figure 1, where client C1 has requested a subset of Topic 1 (satisfying  $Loc = NY \wedge p > 3$ ) in publisher order, and hence receives

<sup>1</sup>\* represents 0 or more occurrences

an interleaving of the data messages associated with its two subend streams. Obviously, any protocol that implements pubend order *a fortiori* implements publisher order.

If, on the other hand, the subscriber requires *total order*, then at some point, the interleaving must be performed by merge hyperedges. These merges are deterministic, the order being determined by the D tick times. For example, clients C2 and C3 have requested a subset of Topic 2 (satisfying  $i = 1$ ) in total order. Each client is associated with a single subend receiving a single merged stream originating from the two pubends. In this case C2 and C3 will receive the same stream.

The service specification for the subscriber is the following:

1. *Safety*: If a D message is delivered, then (a) it meets the subscription constituting the OR of each path predicate, where each path predicate is the AND of the filter predicates along the path; (b) its associated tick is later than those of other D messages previously delivered to this subscriber from the same subend.
2. *Liveness*: If a message is published (meaning that it corresponds to a D tick of some pubend) and it satisfies the subscription, it will eventually be delivered.

Because of lossy propagation, a subend’s knowledge stream may contain gaps (ticks that are Q when subsequent ticks are not Q), but all gaps are eventually resolved. We associate with each subend a monotonic *doubt horizon*  $t_D$ , defined as the latest time such that all ticks  $t < t_D$  where  $C_t \neq A$  satisfy  $K_t = D$ . If there are D-ticks with times later than  $t_D$ , we delay delivering their data until the Q ticks have been resolved either to D or F and  $t_D$  advances. This guarantees that we never deliver a message out of its correct order in the subend’s stream. Whenever  $t_D$  advances, we deliver all previously undelivered D-messages in order.

Besides delivering messages in tick order to the clients, subends are responsible for initiating the upstream flow of curiosity, marking delivered message ticks as A, silences as A, and (eventually) gaps as C.

## 2.4 Filter and Merge Operations

Whenever knowledge changes at a stream, the change is propagated through the filter or merge edges towards the stream or streams downstream of that stream.

The knowledge propagation rules are straightforward: a filter passes a D tick unchanged if it matches the filter predicate, otherwise it converts it to an F, and it passes an F tick unchanged; a merge passes a D tick to its output, but passes F only if all inputs are F. Recall that the output may be arbitrarily delayed before arriving at and being accumulated into the downstream knowledge stream. As discussed in the next section, silence propagates more lazily than data.

Curiosity propagates upstream as follows: A tick becomes anti-curious only if *all* downstream streams propagate A to it. The A tick can then be propagated up to the predecessor node (if a filter) or to all predecessor nodes (if a merge). A C tick propagates to the predecessor node of a filter edge. It propagates to those predecessors of a merge edge that have Q ticks.

### 3 Algorithm and Implementation

In this section we describe the guaranteed delivery (GD) protocol as it is implemented in Gryphon. The GD protocol follows the abstract model of the previous section and is used in the studies of the next section. It simplifies the model slightly, in that there are no merges, and therefore the knowledge graph is a forest consisting of a collection of spanning trees each rooted at one of the pubends.

Virtual brokers are mapped onto collections of broker machines called *cells*. Virtual brokers that host pubends (pubend hosting brokers or PHBs), and subend hosting brokers (SHBs) each map to a cell consisting of a single physical broker machine. Intermediate virtual brokers may map to a cell consisting of multiple physical brokers. For instance, Figure 3 shows the virtual and the physical topology for the failure injection tests of Section 4, in which there is a single PHB, two intermediate virtual brokers, and five SHBs. Intermediate virtual brokers IB1 and IB2 correspond to cells containing the physical broker pairs b1-b2, and b3-b4, respectively. A virtual link in the virtual topology corresponds to a *link-bundle* or *fat link*, containing multiple connections. The multiple brokers and multiple connections are used both to share load and to provide rapid backup on failure. The physical brokers belonging to a cell maintain connections with one another, so that if one physical broker loses connectivity to a downstream cell, it may be able to route messages to that cell via another broker in its cell that has not lost connectivity.

#### 3.1 Knowledge and Curiosity Propagation in a Broker

Since there are no merges, it suffices for each broker to maintain for each pubend  $P$  an input stream data structure  $istream[P]$ , and for each downstream cell  $c$  receiving messages from  $P$  a set of output stream data structures  $ostream[P, c]$  connected to  $istream[P]$  by filter edges. Each physical broker in a cell contains a replica of these structures, although the different brokers may have different states of knowledge about the different ticks.

**Propagating Knowledge:** Knowledge is propagated downstream using knowledge messages. A knowledge message either has the form  $F*Q*F*DF*Q*$  (data message),

or  $F*Q*F*Q*$  (silence message). In both cases, the message encodes (using timestamp ranges) a prefix of “past” ticks whose values are known to be not needed. Data messages include a D tick “payload” bracketed by silence; silence messages represent a range of F ticks.

We distinguish between two kinds of knowledge messages: (1) first-time messages, and (2) retransmitted messages. First-time data messages are sent to all downstream nodes that match the filters. First-time silence messages are only sent to a downstream stream if there is at least one C tick value in the corresponding  $ostream$  that overlaps with the silence ticks. A first-time data message can be transformed into a first-time silence message, if the D tick is filtered out. Retransmitted messages can originate at the pubend or at an intermediate broker. They are only sent down on paths that have expressed curiosity for some ticks in the message (that is, where the  $ostream$  curiosity for that tick is specifically C). A D tick in a retransmitted message is transformed into a Q (i.e. removed from the message) if the downstream cell is not curious for the D tick (but is curious for some of the F ticks in the message).

**Propagation and Interaction of A, F ticks:** Ticks with value A are propagated upstream using ack messages. These messages contain a single timestamp T to encode a range of ticks. An ack message sent from  $s2$  to  $b1$  with timestamp T changes all ticks  $[0, T]$  in  $ostream[P, s2]$  into A. When the tick value, at tick  $t$ , in all  $ostream[P, c]$  turns into A, the corresponding tick value in  $istream[P]$  is also turned into A, triggering an ack message to be propagated upstream towards P.

Whenever a knowledge tick in any stream is F, its curiosity is changed to A. Hence D ticks that have been filtered (turned into F) at an intermediate broker can be immediately acked by it, without waiting for the F to propagate to downstream subends.

**Propagation and Consolidation of C ticks:** Curiosity is propagated upstream using nack messages originating at the subend. Each nack message encodes a contiguous range of C ticks. When a nack message is received at a broker, it tries to *satisfy* the nack using information in its output stream. For instance, suppose  $b1$  receives a nack from stream  $istream[P]$  in cell  $s2$ . The broker first checks which ticks in the nack range are F or D in  $ostream[P, s2]$ . These ticks can be satisfied locally. Knowledge messages corresponding to the satisfied ticks are then sent downstream. All unsatisfied ticks are changed to C in both  $ostream[s2]$ , and  $istream$ . Nacks are consolidated by this process because a nack message is propagated upstream only if some C tick accumulated in  $istream$  was not already C (meaning that a nack must have been sent earlier).

Nacks are repeated if not satisfied within a certain time

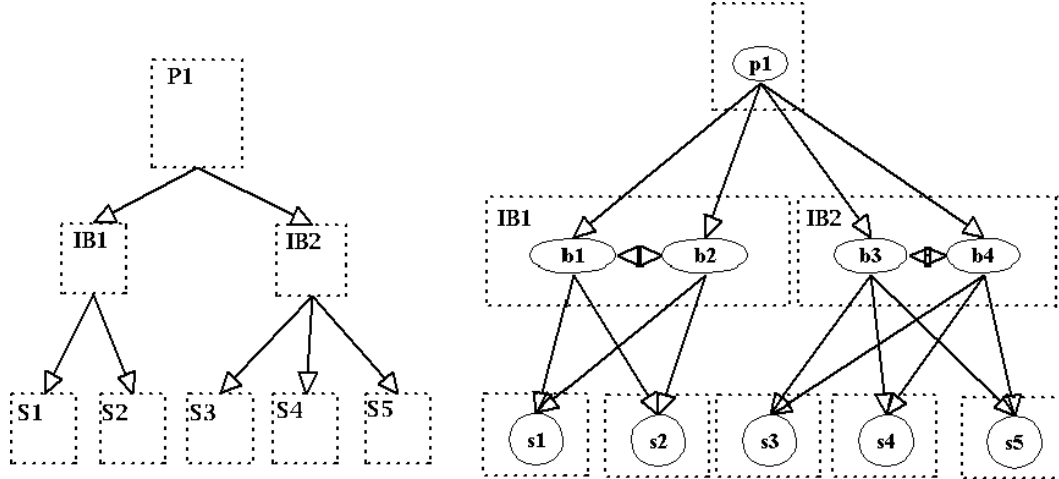


Figure 3. A virtual broker tree and its physical realization

interval. The repetition interval (nack repetition threshold) is estimated in a manner similar to how TCP estimates the retransmission timeout value (RTO) [7]. The system is configured with a minimum repetition interval. To ensure that  $C$  ticks repeated by the same subend are not blocked by the nack consolidation behavior described above, the  $C$  ticks in an *istream* are forgotten after the minimum repetition interval so that nacks appear “fresh.”

**Propagation through Link Bundles:** Let us consider a message being routed from  $p1$  towards its downstream neighbor  $IB1$ . The link is chosen by hashing the pubend id associated with the source pubend of this knowledge tree, onto one of the available links. In this example, whenever both the links  $p1-b1$  and  $p1-b2$  are operational, messages from about half the pubends hosted by  $p1$  will flow to  $IB1$  along  $p1-b1$ , and half along  $p1-b2$ . Suppose the path  $p1-b1$  is chosen, and then the path  $b1-s1$  is broken. In that case,  $b1$  will route messages towards  $s1$  “sideways” via its cell neighbor  $b2$ . Periodic link status messages are exchanged between brokers so that this sideways routing is only transient, after which the messages will switch from the  $p1-b1$  to the  $p1-b2$  path. Note that during an interval of no failures or recoveries, successive messages from the same pubend in  $p1$  will flow on the same path towards the downstream subscribers. Ack and nack messages sent upstream through a link bundle are sent to whichever physical broker in the cell last sent a downstream message from the relevant pubend; if this information is lost, the messages are sent to all physical brokers in the upstream cell.

### 3.2 Failures and Liveness

Broker and link failures lead to message loss, which cause subends to see a gap in their knowledge stream. A

gap is a sequence of  $Q$  ticks between non- $Q$  ticks. There are two extreme approaches to resolving these gaps, both of which involve retransmission of messages, (1) subend driven liveness, and (2) pubend driven liveness. We provide tuning parameters such that the system can be run with one of these approaches or anything in between.

**Subend-driven Liveness:** This is mainly based on two liveness parameters, *gap curiosity threshold* (GCT) and *nack repetition threshold* (NRT). The NRT parameter is estimated by the subend based on the round-trip response to previous nacks, and exponential backoff is used to handle pubends that are down. When a gap is created in the knowledge stream, the subend starts the GCT timer for the  $Q$  ticks in the gap, and sends a nack after expiry of the timer, if the  $Q$  ticks have not already been satisfied. Nacks for these  $Q$  ticks are repeated every NRT interval, until they are satisfied. An additional parameter, *delay curiosity threshold* (DCT), is important to guard against the loss of the latest message, when no pubend-driven liveness is in use. The subend initiates a nack if its doubt horizon trails real time by more than the DCT.

**Pubend-driven Liveness:** This is based on one parameter, the *ack expected threshold* (AET). The pubend expects all ticks that are more than AET interval before the current time to be acked. If they are not, it sends an AckExpected message with a timestamp  $T$  equal to the current time minus AET. The message flows down on all paths that have not acked up to  $T$ . A subend receiving this message will immediately nack all ticks up to  $T$  that are  $Q$  in its knowledge stream.

In our experiments, we typically run with low GCT and NRT values, a higher AET, and a infinite DCT. Hence we are using a mixture of both liveness approaches, with

subend-driven liveness dominating.

## 4 Experimental Results

Two sets of experiments were performed to demonstrate the overhead of the guaranteed delivery (GD) protocol and its behavior in the presence of faults. The machines used for the experiments are dedicated 6-processor Power-PC machines running AIX 4.3 with 3072 MB of memory. There are two network interfaces attached to each machine - a gigabit ethernet PCI adapter and a 100 Mbps ethernet PCI adapter. The broker code is written in Java, and was run using IBM's JRE 1.3.

### 4.1 Comparison of Guaranteed Delivery and Best-effort protocols

These experiments measure the failure-free overhead of the GD protocol using two metrics: (1) mean CPU utilization, and (2) median latency from publishers to the subscribers. The best-effort delivery protocol used for comparison does not perform any knowledge accumulation, curiosity propagation, message logging or retransmission, and only sends downstream D tick messages. A two broker asymmetric configuration is used for the experiments. Publishing clients are connected to one broker, the pubend hosting broker (PHB), while the subscribing clients are connected to the second broker, the subend hosting broker (SHB). The latency seen by the subscribing clients connected to the SHB is called remote latency. For measuring local latency, a subscribing client is connected to the PHB. The input message rate is 2000 messages/s and each published message is 250 bytes long. Each subscriber receives 2 msgs/s on a dedicated TCP connection to the SHB. The subscribers connect to the SHB through the gigabit network since the fan-out out of the SHB is quite high (up to 32000 msgs/s on 16000 connections). The broker-to-broker connection is on the 100 Mbps link.

Figure 4 shows the variation in CPU utilization at the brokers while varying the number of subscribing clients. The CPU utilization at the subend hosting broker expectedly increases with increasing number of subscribing clients and the utilization when running the guaranteed delivery (GD) protocol is higher than running best-effort delivery of messages. The difference between the GD protocol cost and the best-effort protocol cost does not increase with the number of subscribers and stays constant at less than 4%. This is because our implementation optimizes the GD stream state needed by each subend, by consolidating it across all subends at the same SHB. The figure also shows that the CPU utilization at the PHB does not increase with the number of subscribers connected to the SHB. The CPU

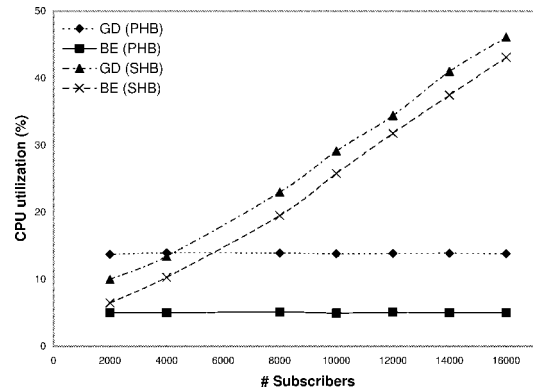


Figure 4. Variation of CPU utilization with Number of Subscribers (Input Rate = 2000 msgs/s)

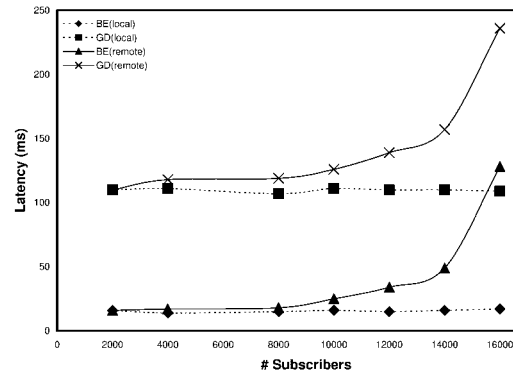


Figure 5. Variation of Latency with Number of Subscribers (Input Rate = 2000 msgs/s)

overhead for GD (wrt best-effort) at the PHB, about 8%, is more than that at the SHB, due to the overhead of logging.

Figure 5 shows how the local and remote latency vary with the number of subscribing clients. The local latency does not show an increasing trend with the number of subscribers, since the subscribers are at the SHB, while the endpoints for the local latency are on the PHB. Expectedly, the remote latency increases with the number of subscribers for both GD and best-effort. However, the difference between GD and best-effort remains approximately constant with increasing number of subscribers. This latency difference of about 100ms is due to the delay introduced by the logging of guaranteed messages at the PHB. This constant difference is observed in both the local and the remote latencies.

## 4.2 Failure Injection Results

These tests measure the system fluctuation and recovery in the presence of faults. We measure the system dynamics under two types of faults, broker crash and link failure.

**Experimental Setup:** We configured a network of 10 brokers in 8 cells as shown in figure 3. Broker p1 is designated as the pubend hosting broker. There are 4 intermediate brokers, b1-4, with b1-2 in one intermediate cell and b3-4 in another. Each of the intermediate brokers has a direct link to the pubend hosting broker. There are 5 subend hosting brokers s1-5, with s1 and s2 linked to cell IB1 and s3-5 linked to cell IB2. Broker p1 hosts 4 pubends, each of which is receiving and logging 25 msgs/s (each message is 100 bytes) from a publisher, for a total input rate of 100 msgs/s. This low rate is used since we want to observe system dynamics without hitting any processing capacity constraints at brokers. The filters at intermediate brokers allow all messages through. All these tests use the following liveness parameters: GCT=200ms, NRT=600ms, AET=10s, DCT=infinity.

**Metrics:** Three metrics are used: (1) end to end message latency, (2) number of nacks sent, and (3) *nack range* sent. The *nack range* metric counts the number of time ticks (in milliseconds) that are nacked. For instance, since each pubend is publishing about 25 msgs/s, consecutive messages from a pubend are about 40ms apart. A *nack range* of 800ms would mean that about  $800/40=20$  messages are nacked. The *nack* metrics are measured at subend and intermediate brokers to check for *nack consolidation*.

**Failures Injected:** We present results from three kinds of failures: (1) link failure of b1-s1, (2) crash failure of b1, (3) crash failure of p1. We repeatedly injected each kind of failure. We did not average data over multiple instances. Instead, we chose to display a modal instance, in order to preserve details of dynamic behavior.

Crash failures were injected by killing the broker process, and link failures by closing the TCP connection. We observed that since queuing of messages inside a broker was rare, failures injected in this simple manner were immediately detected by adjacent brokers, and caused messages to be immediately switched to a different path. Therefore many such failures did not result in even a single message loss! In practice, it is likely that many failures are not detected immediately by adjacent brokers. We therefore revised our failure injection to include two steps: (1) the link or broker to be failed was stalled for about 2-3 seconds during which it accepted data but did not forward it, (2) then it was failed. This caused about 2-3 seconds of data messages to be lost.

**Results for b1-s1 failure:** Figure 6 shows the behavior of the system when the link b1-s1 is failed for 10 seconds, for a pubend whose messages were flowing on p1-b1 before

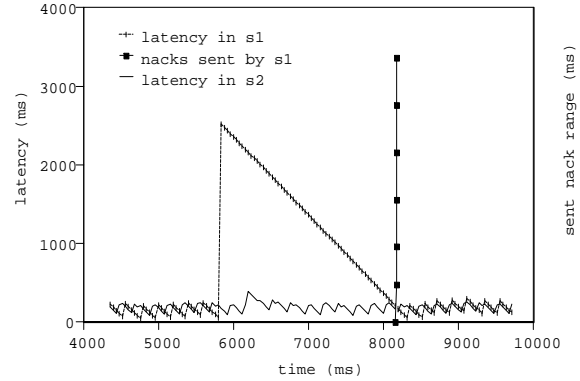


Figure 6. Latency and Nacks for b1-s1 failure

the failure. For both message latency plots and *nack* message plots, the X axis is the send time of the message. Each point on the *nack range* plot is a single *nack* message, and the *nack range* value is cumulative. As the connection is first stalled, s1 does not notice the failure till more than 2 seconds later (just after time 8000). Then it sends nacks (to b2, since the b1-s1 link is down), and receives the messages lost on the link. Our implementation chops a large tick range that need to be nacked into smaller parts, so that the effect of a *nack* message being lost is small, hence we see multiple *nack* messages sent by s1. Note that the last message published, before the failure, is received after the nacks are sent, and hence experiences a latency of 2.5 seconds. Since the lost messages are received in a burst after the nacks are sent, the latency values for s1 show a saw-tooth form. After all routing switches to the path b2-s1, the latency returns to normal. Note that the latency at s2 was unaffected by the failure. The other tests also showed similar behavior for subscribers not on any failure path, so we only show affected subscribers in the remaining plots.

**Results for b1 crash:** In this test, broker b1 was crashed and restarted 30 seconds later. Before the crash, b1 and b2 were splitting the input message load, i.e., each was handling messages from 2 of the 4 pubends. Figure 7 shows two plots, one for the latency and the other zooms in on the time when the nacks are sent, to show the nacks sent by s1, s2 and b2. The latency is shown for a subscriber that is subscribing to messages from a pubend whose messages were flowing through broker b1. The first peak in the latency is due to stalling broker b1 before crashing it. As soon as the b1 crashes, its neighbors detect the failure and all messages start flowing through b2. The latency plot for s2 is similar to s1 and is not shown. The second latency peak occurs when b1 recovers, which results in messages from 2 of the 4 pubends to again start flowing through b1. The main reason for this transient increase in latency is the extra computation in the broker machine just when it starts

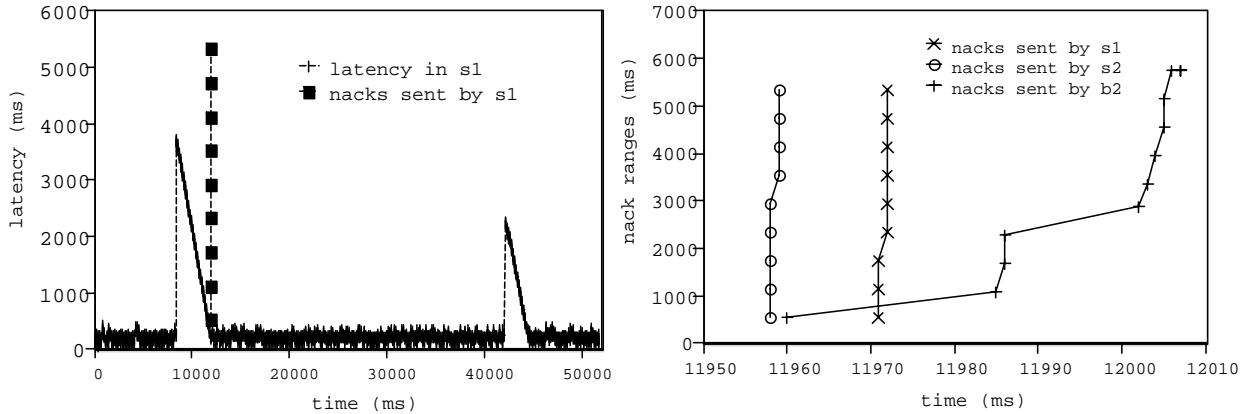


Figure 7. Latency and Nack Range for b1 crash

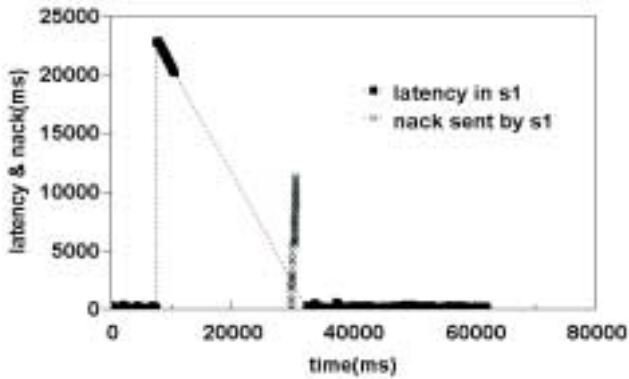


Figure 8. Latency and nacks for p1 crash

up, such as to run the Java JIT compiler. Note that no nacks are sent at this time, since messages are not lost, just delayed.

The second plot in figure 7 shows the nacks and nack range sent by s1, s2 and b2. As s1 and s2 lost about the same messages due to b1's failure, the nacks sent by them are almost identical in number and range. Nacks sent by b2 are in response to nacks by s1 or s2. Since b2 does not have any of the data requested by the nacks (that data was flowing through b1), all the nacks are forwarded to p1. Note that the nack consolidation by b2 is almost perfect, in that the nack range sent by b2 is about half that of s1 and s2 combined. Another thing to note is that the nack range of s1 and s2 is about 5500ms. This corresponds to 2 different pubends, both of whose messages were flowing through b1, so about 2750ms of data was lost for each pubend. This agrees with the time interval for which b1 was stalled before the crash.

**Results for p1 crash:** In this test, the PHB p1 was crashed and restarted after about 20 seconds. This affected all the

subscribers at s1-s5 in a similar manner. Figure 8 shows the latency seen by a subscriber connected to s1, and the nacks initiated by s1. Unlike the earlier tests, where the publisher kept publishing despite the failure, here the publisher was down and unable to publish for the duration of p1's crash. Any messages that the pubends had logged but were unable to send out before the crash show a high latency, as can be seen by the partial sawtooth form of the latency. Before the pubend crash, messages are being received in order at s1. When the pubend crashes, no new messages are received at s1, but no gaps are created either. As delay curiosity threshold (DCT) is infinity, s1 does not initiate any nacks while p1 is down. When p1 recovers, a time interval longer than the ack expected threshold (AET) has already elapsed, causing it to first send an AckExpected message that contains the timestamp of the last message it logged (at each pubend) before it crashed. This results in nacks from s1-s5, and the latency quickly returns to normal.

## 5 Related Work

**Guaranteed Delivery in Pub/Sub Systems:** Most Internet scale pub/sub systems, such as SIENA [5], offer best-effort delivery. Guaranteed delivery is offered in messaging systems such as IBM's MQseries, but such systems are message queueing systems that use a store and forward approach to ensure reliability. The store and forward approach incurs high latency since messages need to be logged at each stage, and cannot support high throughput due to the high per message overhead.

The work that is most closely related to ours is the Diversity Control Protocol (DCP) [11], used to route and filter XML documents on an overlay mesh network that filters at intermediate nodes. DCP runs on a replicated n-resilient mesh network, unlike our network in which messages are

not replicated on redundant paths. However, our protocol can be easily adapted to a DCP like network. DCP guarantees delivery by running a hop-by-hop reliable protocol, where the receiving broker becomes the reliable sender for the next hop. Since a gapless stream is reconstructed at each hop, the entire stream is delayed when a single gap is found.

**Reliable Multicast Protocols:** There is a large body of work on reliable multicast in the networking literature. Most of it, such as SRM [6] and RMTP [8], deals with building an end-to-end reliable protocol using the underlying best-effort IP multicast service. The ordering offered by these protocols is publisher order. Most of the details of such protocols deal with solving the ack or nack-explosion problem in the absence of router assist, by either arranging the receivers in a hierarchy of groups or by using receiver backoff.

Two exceptions to a purely end-to-end approach for reliable multicast are Active Reliable Multicast (ARM) [13] and the Breadcrumb Forwarding Service (BCFS) [14, 15]. ARM uses 'active networking' routers to efficiently consolidate nacks flowing to the source. The consolidation is done by caching nack requests for a certain time interval. Retransmissions are forwarded only to downstream routers that have nacked, again by examining the cached nack requests. The BCFS [14] network service, is similar to what is implemented in the active routers for ARM. The curiosity stream described in this paper is a technique for doing such nack consolidation and forwarding of retransmissions, and can be applied to ARM and BCFS.

**Atomic Multicast:** There are a large number of systems that provide atomic multicast/broadcast primitives, such as Isis [4] and Horus [12]. They use virtual synchrony or one of its variants to provide this atomicity guarantee. These synchrony protocols typically require synchronizing the complete system on each membership change, and hence are hard to scale to more than a few hundred participants [9].

Both the reliable multicast and the atomic multicast protocols are group-based, and do not support selective filtering.

## 6 Conclusion

In this paper we have presented a new model and algorithm for providing exactly-once message delivery to subscribers in a content-based publish-subscribe system. The problem is challenging due to filtering at intermediate nodes, and previous algorithms have adopted a simple but restrictive hop-by-hop store and forward approach. In comparison, our solution maintains only soft-state at intermediate brokers, and does not need to stall message forwarding in the presence of message loss. This allows the system to sustain high throughput despite failures, and makes it sim-

ple to dynamically replace brokers in the overlay network. Our implementation has an overhead of only 4% CPU utilization, compared to best-effort delivery, in the absence of failures. We have also demonstrated that our implementation rapidly switches around failures, and does effective nack consolidation.

## References

- [1] M. K. Aguilera and R. E. Strom. Efficient atomic broadcast using deterministic merge. In *Proceedings of the 19th ACM symposium on Principles of distributed computing*, 2000.
- [2] 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.
- [3] 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.
- [4] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.
- [5] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, December 1998.
- [6] 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.
- [7] V. Jacobson. Congestion avoidance and control. *Computer Communication Review*, 18(4):314–329, 1988.
- [8] J. C. Lin and S. Paul. Rmtp: A reliable multicast transport protocol. In *Proceedings of IEEE Infocom'96*, pages 1414–1424, 1996.
- [9] R. Piantoni and C. Stanescu. Implementing the swiss exchange trading system. In *Symposium on Fault-Tolerant Computing (FTCS)*, pages 309–313, 1997.
- [10] S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *Proceedings of ACM SIGCOMM*, pages 15–25, 1999.
- [11] A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using xml. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, 2001.
- [12] R. van Renesse, K. Birman, and S. Maffei. Horus: a flexible group communication system. *Communications of the ACM*, 39(4), April 1996.
- [13] L. wei H. Lehman, S. J. Garland, and D. L. Tennenhouse. Active reliable multicast. In *Proceedings of IEEE INFOCOM'98*, 1998.
- [14] K. Yano and S. McCanne. The breadcrumb forwarding service: A synthesis of pgm and express to improve and simplify global ip multicast, 2000.
- [15] K. Yano and S. McCanne. A window-based congestion control for reliable multicast based on tcp dynamics. In *Proceedings of ACM Multimedia*, pages 249–258, 2000.