Low-Overhead Message Tracking for Distributed Messaging

Seung Jun[1] and Mark Astley[2]

[1] College of Computing, Georgia Institute of Technology, Atlanta, GA 30339, USA
[2] IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA

Abstract. As enterprise applications rely increasingly on commodity messaging middleware, message tracking has become instrumental in testing and run-time monitoring. However, building an effective message tracking system is challenging because of the large scale and high message rate of enterprise-wide applications. To address this challenge, we consider the case of message tracking for distributed messaging middleware. We desire to record the origin, path, and destination of every application message while imposing low overhead with respect to latency, memory and storage. To achieve our goal, we propose a tunable approximation approach based on Bloom filter "histories." Our approach is tunable in the sense that more accurate audit trails may be provided at the expense of storage space, or, conversely, storage overhead is reduced for applications requiring less accurate audit trails. We describe the design of the system and demonstrate its utility by analyzing the performance of a prototype implementation.

1 Introduction

The development of enterprise business applications has increasingly relied on commodity messaging middleware for component connectivity and interaction. Because the enterprise no longer implements the underlying communication framework, message traceback and audit trails will become an important administrative capability during development and for monitoring and problem determination at run-time. Past exploration in messaging middleware [5] suggests message rates on the order of thousands of messages per second, distributed to tens of thousands of clients. On this scale, a naive logging approach may impose unacceptable overhead and may rapidly exhaust persistent resources such as disk. Thus, we need an efficient message tracking system to accommodate high volume of traffic.

In this paper, we consider the specific case of message tracking for publish-subscribe messaging middleware based on the Java Messaging Service API [4]. We choose to focus on publish-subscribe, rather than more general messaging (*e.g.* queuing), because we believe that the large scale and high message rate of publish-subscribe middleware presents the greatest challenge for message tracking. Given a network of publish-subscribe message servers we wish to:

- Record the origin, path, and destination of every message routed by the system;
- Support traceback queries for previously sent messages; and

– Impose low runtime overhead with respect to latency, memory footprint, and disk storage.

To support tracking for publish-subscribe messaging, we have developed a tunable approximation approach based on Bloom filters [6], which trades accuracy for low overhead and efficient use of persistent resources. Our approach is tunable in the sense that more accurate audit trails may be provided at the expense of more frequent high-overhead operations (*e.g.* storing buffers to disk). Conversely, overhead can be reduced for applications which require less accurate audit trails.

While Bloom filters are often used as an efficient, approximate cache [8, 15], a novel feature of our approach is the organization of Bloom filters into "histories" which record an indefinite record of message paths for later query. We present theoretical results which specify clock synchronization and message jitter limits in order to maintain accurate histories. Similarly, we evaluate a prototype implementation in order to highlight trade-offs between performance, accuracy, and resource utilization.

The remainder of the paper is organized as follows. In Section 2, we define a tracking facility in the context of publish-subscribe messaging. In Section 3, we describe the design and implementation of our system, and analyze the accuracy of our design in Section 4. In Section 5, we evaluate performance. In Section 6, we describe related work. We summarize our results and discuss future work in Section 7.

2 Tracking for Distributed Messaging

Messaging middleware is a popular "glue" technology which provides the means for applications to interact with one another using a variety of interaction patterns. In addition to proprietary interfaces, most messaging middleware products support the Java Message Service (JMS) API, which defines messaging services within the Java 2 Enterprise Edition specification [3]. We focus on the publish-subscribe portion of JMS although our techniques are readily extended to other distribution patterns and/or other messaging APIs.

We define a tracking facility for publish-subscribe messaging as follows:

> *Given the unique ID of a message, the tracking facility will report the origin of the message, the messaging servers which routed the message, and the set of clients to which the message was delivered. The message may have been accepted for delivery at an arbitrary time in the past.*

We assume that the tracking facility is queried after a message has been completely routed. However, our techniques may be used to provide partial tracking information if earlier queries are necessary.

In the context of JMS, each message has a vendor specific unique ID which is assigned when the message has been accepted for delivery. A message has been accepted for delivery when the "publish" call returns at the publisher. Thus, the JMS message ID is a valid input to the tracking facility once the "publish" call completes. Although the message ID is first known to the publisher, we assume that any entity may issue a traceback query with an appropriate message ID.

The implementation of the tracking facility is divided into two distinct components. The *message tracking* component records the routes of messages as they are distributed through the messaging system. The *path reconstruction* component uses tracking records to reconstruct the complete path of a routed message. While we focus on efficient message tracking in this paper, we provide a brief description of how path reconstruction may be implemented in Section 3.3.

3 System Design

We develop our tracking facility as a component in the Gryphon system [1], which is a robust, highly-scalable messaging system that implements the publish-subscribe portion of JMS. Gryphon is representative of a large class of middleware routing infrastructures and consists of an overlay network of messaging *brokers*, each of which may host one or more network-connected *clients* (*i.e.* publishers and subscribers). A typical publish-subscribe message is routed as follows:

1. The message is created by the publisher and submitted to the associated broker for delivery.
2. The broker receives the message and determines (a) which locally attached subscribers (if any) should receive the message, and (b) which neighboring brokers (if any) should receive the message. It then routes the message to the appropriate local subscribers and neighboring brokers.
3. Neighboring brokers repeat this process until all appropriate brokers and subscribers have received the message.

In Gryphon, the network topology forms an arbitrarily connected graph. Routes are determined by spanning trees so that each broker receives a message from at most one neighbor. In addition, Gryphon provides ordered links between brokers and we assume that brokers do not arbitrarily reorder in-flight messages. Failures may require retransmission and hence introduce duplicate messages. For the moment, we assume there are no duplicate messages. We discuss modifications for handling duplicates in Section 4.3. Thus, the current discussion assumes ordered, tree-based routing without duplicate messages.

To facilitate tracking, the JMS ID of each message is augmented with a tuple $r = (p, b, t)$, where p is the unique ID of the publisher of the message, b is the unique ID of the broker to which the publisher is attached, and t is a monotonically increasing time stamp. The time stamp is generated locally by the publisher and may include a "skew adjustment" as described in Section 3.2. The values for p and b are stored at the publisher when it connects to a broker. The JMS ID is set by the publisher-side implementation of the JMS "publish" method.

3.1 Bloom Filter Histories

Each broker maintains one or more local Bloom filter "histories" which are a compressed record of message traffic. Histories consist of both in-memory and persisted (*i.e.* stored to disk) data, and are further partitioned according to message sender (as described in the next section).

A Bloom filter [6] is a data structure for representing a set of n elements called *keys*, comprised of an m-bit vector, v, (initialized to zeros) and associated with k hash functions whose range is $\{1, \ldots, m\}$. Let $v[i]$ $(1 \leq i \leq m)$ denote the i^{th} element of the bit vector v. Given k hash functions $f_1, ..., f_k$, a Bloom filter supports two operations:

- $\textsc{Add}(r)$ adds the key r to the set of elements stored in the filter. That is, $\textsc{Add}(r)$ sets $v[f_j(r)] = 1$ for $j = 1, ..., k$.
- $\textsc{Contains}(r)$ returns *true* if the key r is stored in the filter or *false* otherwise. That is, $\textsc{Contains}(r)$ returns *true* if and only if $v[f_j(r)] = 1$ for each $j = 1, ..., k$.

Bloom filters are efficient in both speed and size because hash functions typically execute in constant time (assuming constant key size) and because the set of stored keys requires no more than m bits of storage. On the other hand, as hash functions may collide, Bloom filters are subject to *false positives*. That is, $\textsc{Contains}(r)$ may return *true* even if r is not actually stored in the filter. False positives affect the accuracy of the tracking facility and are discussed in Section 4.

Given a particular accuracy requirement, the capacity, n, of a Bloom filter is a function of m and k. Adding more keys beyond the capacity will degrade the accuracy of the filter. Therefore, when a filter reaches its capacity it is stored to disk, and a new filter is created to record subsequent keys. If a system failure occurs before a filter is stored, then the contents of the filter are lost. Therefore, to avoid excessive loss when message rates are low, filters are also periodically stored to disk according to the *persistence interval*, T_p, which specifies the maximum delay before which the current filter must be stored.

The in-memory filters and the set of filters stored to disk are paired with indexing information to form *filter histories*. That is, a filter history, H, is defined as a sequence $(B_1, R_1), ..., (B_j, R_j)$, where each B_i is a Bloom filter, and each R_i is a range of timestamps of the form $[t_{s,i}, t_{e,i}]$. The time range associated with each Bloom filter is used as a query index by the *path reconstruction* component. A history has at most one in-memory pair (B_j, R_j), called the *current filter*, with all other pairs being stored to disk.

Histories have a single operation: $\textsc{Add}(r, t)$. Given a history H with current filter (B_j, R_j), the $\textsc{Add}(r, t)$ operation invokes $\textsc{Add}(r)$ on B_j and updates R_j to include t as follows:

$$R_j = \begin{cases} [t, t] & \text{if previous } R_j \text{ is } \emptyset, \\ [t, t_b] & \text{if previous } R_j = [t_a, t_b] \text{ and } t < t_a, \\ [t_a, t] & \text{if previous } R_j = [t_a, t_b] \text{ and } t > t_b, \\ [t_a, t_b] & \text{otherwise.} \end{cases}$$

Given a filter history H and a time stamp t , we define the *matching set* as the set $M(H, t)$ of filters such that

$$M(H, t) = \{B_i \mid (B_i, R_i) \in H \text{ and } t \in R_i\}.$$

A matching set determines which filters should be queried when reconstructing the path of a message. We achieve the required accuracy and efficiency of our system by carefully managing the size of matching sets as explained in Section 4.

3.2 Message Tracking Component

Message tracking requires that we record JMS messages IDs at various points in the system. Messages are tracked in three phases. In the *publishing phase*, a publisher generates and sends a message to a local broker. This broker, called the publishing broker, is responsible for recording the originating publisher for each message. In the *routing phase*, the message is tracked at each intermediate broker to which it is forwarded. Finally, in the *delivery phase* the message is delivered by one or more brokers to interested local subscribers. The delivering brokers are required to record the set of subscribers which receive the message.

State Initialization A publisher acquires its ID, p, and the broker ID, b, at connection time. Each publisher also maintains a skew adjustment, δ, which is used to adjust locally generated timestamps so that they remain within a certain bound relative to the broker's clock. The skew adjustment is necessary to bound matching set size, as explained in Section 4.2. At publisher connection time, δ is initialized to the value $t_b - t_p$, where t_b is the current local time at the broker, and t_p is the current time at the publishing client. To correct for clock drift, the value of δ is updated as described below.

Tracking information is only stored at brokers. Specifically, each broker is initialized with the following state:

- **Skew tolerance**, T_s, determines the maximum separation between the time stamp of a message submitted by a local publisher and the broker's local clock.
- **Persistence interval**, T_p, determines the persistence interval of each local filter history.
- **Publisher history**, H_P, is a filter history that is used to record the set of messages sent by local publishers.
- **Routing history**, H_b, is a filter history that is used to record the set of messages originated by each broker b. Thus, a broker maintains multiple routing histories as discussed later.
- **Subscriber history**, $H_{b,s}$, is a filter history that is used to record messages originated by broker b and delivered to local subscriber s. A broker maintains multiple subscriber histories depending on the number of brokers and subscribers.
- **Subscriber attachment map**, S_m, is a data structure which maintains the local subscriber membership and a sequence of local timestamps indicating when the membership last changed. This map is used to reconstruct the set of subscribers which may have received a message. Although subscribers may arrive or depart frequently, the set of changes at a particular point in time are assumed to be small. We assume membership changes (and the time stamp of the change) are stored reliably to disk.

The skew tolerance, T_s, and the persistence interval, T_p, are local values which may be different at each broker. However, some care is necessary when choosing these values as they affect the overall accuracy of the tracking system. We discuss how to determine the values for T_s and T_p to achieve the desired accuracy in Section 4. In each of the phases below, we assume that filters are automatically stored to disk when full or when the persistence interval expires as described in Section 3.1. We also assume that the Bloom filter parameters m, k, and n are global settings.

Publishing Phase In the publishing phase, the publishing client creates a message, assigns an ID, and delivers the message to the broker. The ID is constructed as a tuple $r = (p, b, t+\delta)$, where δ is the skew adjustment. The broker verifies $|t_b - (t+\delta)| \leq T_s$. If the message is out of tolerance, then δ is recomputed (e.g. $\delta = t_b - t$) and sent back to the publisher. The broker records the message in the local publisher history by invoking $\textsc{Add}(r, t+\delta)$ on H_P.

Routing Phase In the routing phase, the local broker extracts the message ID $r = (p, b, t')$ and creates the routing history H_b for the originating broker b if it does not already exist. Then, the broker records the message in H_b by invoking $\textsc{Add}(r, t')$ on H_b.

Delivery Phase In the delivery phase, the local broker extracts the message ID $r = (p, b, t')$ and determines the unique IDs of the set of local subscribers $s_1, \ldots, s_j$ which should receive the message. For each subscriber s_i, the broker instantiates the subscriber history H_{b,s_i}, if necessary, and stores r in the subscriber history by invoking $\textsc{Add}(r, t')$ on H_{b,s_i}. Once each subscriber has been recorded, the message may be delivered.

3.3 Path Reconstruction Component

The path for a given message can be reconstructed by searching the broker network in a depth- or breadth-first manner. We first describe a basic algorithm for this process, and then consider optimizations and complexity.

Let $r = (p, b, t)$ be the ID of the message for which we wish to reconstruct a path. Note that b denotes the ID of the publishing broker where the message originated. We first initialize the following query state:

- K_r, initially $\emptyset$, is the set of brokers that routed the message.
- K_s, initially $\emptyset$, is the set of brokers which delivered the message to a subscriber.
- S_r is the set of subscriber IDs to which the message was delivered.
- K_a is the set of brokers to explore and is initialized to $\{b\}$.
- K_e is the set of brokers already explored and is initialized to $\emptyset$.

Starting from the originating broker b, the algorithm fills K_r, K_s, and S_r using K_a and K_e as follows:

- While $K_a \neq \emptyset$:

1. Choose $i \in K_a$ and set $K_a \leftarrow K_a - \{i\}$. Set $K_e \leftarrow K_e \cup \{i\}$.
2. If broker i contains the routing history H_b, it searches for the queried message in this history. Let $M(H_b, t)$ be the matching set for routing history H_b and time stamp t. If there exists $B_j \in M(H, t)$ such that $\text{CONTAINS}(r)$ on B_j is true, then set $K_r \leftarrow K_r \cup \{i\}$.
3. Let S_m be the set of subscribers retrieved from the subscriber attachment map which were attached to broker i at time t. For each subscriber s in S_m:
   (a) Let $H_{b,s}$ be the subscriber history with source broker b and subscriber s.
   (b) If there exists $B_j \in M(H_{b,s}, t)$ such that $\text{CONTAINS}(r)$ on B_j is true, then set $S_r \leftarrow S_r \cup \{s\}$ and $K_s \leftarrow K_s \cup \{i\}$.
4. For each neighbor j of broker i, such that $j \notin K_e$, set $K_a \leftarrow K_a \cup \{j\}$.

Upon termination, b, K_r and K_s can be used (along with the broker topology) to reconstruct the route of the message. Similarly, S_r gives the set of subscribers to which the message was delivered.

Reconstruction Complexity and Optimization Assuming no false positives (we consider the effect of false positives in Section 4.3), the time required to reconstruct a path for a given message is proportional, namely $O(N)$, to the number of brokers, N, which routed the message. It is difficult to improve on this basic complexity result under the current design where filters act as membership tests and provide little information to simplify path reconstruction. However, the size of the constant term can be reduced as discussed below.

The constant search cost at each broker depends on the number of histories which must be queried, and the number of neighboring brokers which must be queried. At each broker (except the originating broker), an efficient search is accomplished by first sampling the local routing history, and then sampling the local subscriber histories if the routing history indicates that the local broker routed the message. Thus, in the worst case, if M is the matching set size, and the maximum number of subscribers a broker may host (at any time) is S, then at each step at most $S + 1$ histories will be queried or $(S + 1) \times M$ invocations of the CONTAINS operation. Since CONTAINS takes constant time (with constant key size) and S and M are constants, the history query cost is constant but tunable by adjusting S and M.

During the path reconstruction, a broker must query every neighboring broker about the traced message, which may incur unnecessary communication cost particularly when the message was delivered to only a small number of neighbors (*i.e.* when the message was sparsely routed). If communication cost is roughly constant, then let W be the cost of communicating with a neighboring broker, verifying that its local history did not track a message, and receiving the reply. If D is the maximum number of neighbors for any broker, then there is a constant communication cost no greater than $D \times W$ incurred during path reconstruction at each broker which routed a message.

There are at least two methods for reducing communication overhead. The first is to require brokers to store routing histories of the form $H_{b,d}$ where a message is recorded in $H_{b,d}$ if it originated from broker b and was forwarded to neighboring broker d. This eliminates unnecessary communication during path reconstruction (assuming no false positives) at the cost of complicating the configuration of T_s and T_p, which must be sensitive to the rate at which filters are filled (see Section 4.2). In particular, sparsely routed messages may yield drastically different fill rates for two given histories $H_{b,d}$ and $H_{b,d'}$.

A second approach is to store "routing set" information as part of the key recorded in routing histories. For example, if maximum out degree is D and neighboring brokers are numbered 0 through $D-1$, then we can store a modified message ID $r' = (p, b, t', d)$ where d is a D-bit value with bit i set if neighbor i was forwarded the message. At path reconstruction time, we must now perform 2^D queries of the local routing history in order to determine where to continue the search. This approach is feasible when $O(\textsc{Contains}) \times M \times 2^D$ is less than $W \times D$.

3.4 Discussion

As described in Section 4, applying skew adjustments to message timestamps is critical to ensuring tracking accuracy. However, skew adjustments could be eliminated if timestamps were assigned at the broker rather than the publisher. The decision to assign timestamps at the publisher is a deliberate choice to avoid a performance penalty. If timestamps were assigned at the broker, then each call to "publish" could not complete until a reply was received from the broker since message IDs include the time stamp and the ID must be valid before returning from "publish". This may not be significant when publishing reliably since the time stamp could be piggy-backed on the acknowledgment we expect for each message, but best-effort publishing does not require any acknowledgment. Therefore, we allow publishers to assign timestamps and use skew adjustments to manage disparate publisher clocks.

At a given broker, the current implementation uses separate router histories for each broker from which messages are originated. One may wonder if it is possible to simplify the design, for example by aggregating routing histories into a single history which records traces for all non-local brokers. In the current design, aggregation is undesirable for two reasons. First, aggregating incoming messages causes filters to be filled at a higher rate. Second, without careful clock synchronization, consecutive filters may have a larger than expected overlap in their range components. Both of these effects increase the size of the matching set which results in lower tracking accuracy as discussed in Section 4. The same reasoning applies to aggregation of subscriber histories.

Nonetheless, an aggregate history would be feasible under certain conditions. Namely, the filter capacity would need to be large enough to accommodate the higher aggregate input rate; and, T_p would need to be at least as large as the T_p of any broker for which messages are being aggregated. Such a history is essentially equivalent to the sum of the individual histories we use in the current design. In particular, the net effect on overall storage consumption (memory or disk) is identical since both approaches have the same capacity and the same total number of messages are recorded in either case.

4 Analysis

Bloom filters provide efficient space utilization at the expense of reduced accuracy. In this section, we describe how to compute the accuracy of our tracking facility in terms

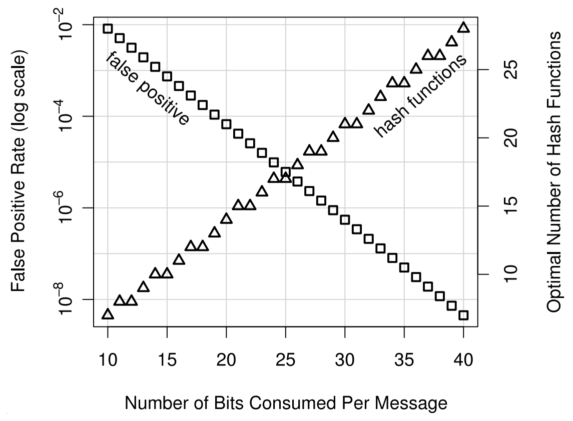

Fig. 1. False positive rate.

of Bloom filter accuracy and the size of history matching sets. We then describe how the tracking facility bounds matching set size in order to guarantee particular accuracy requirements.

4.1 Bloom Filter Accuracy

The *false positive probability* (*fpp*) of a Bloom filter is the probability that an invocation of CONTAINS will return *true* for a message that is not stored. Assuming that the output from the k hash functions is independent and uniformly distributed, *fpp* is determined by the expression [7]:

$$\mathit{fpp} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \tag{1}$$

where m is the size of the Bloom filter in bits, and n is the number of entries stored in the filter. Conversely, given a desired *fpp*, we can determine k, m, and n such that the required accuracy is met. We give an example of this process in Section 5.

Figure 1 helps determine the appropriate parameters. It shows that false positive rate decreases exponentially as the number of bits consumed per message (that is, m/n) grows with the optimal number of hash functions (that is, k) used for the given x-axis value.

We refer to the accuracy of the entire tracking system as the *expected false positive probability* (*efpp*). If all filters are identical (and hence have the same *fpp*), the *efpp* is determined by the size of the matching set:

$$\mathit{efpp} = 1 - (1 - \mathit{fpp})^{|M|} \tag{2}$$

where $|M|$ is the maximum size of the matching set. If $|M|$ is known and bounded, it is possible to guarantee a particular *efpp*.

The choice of *efpp* affects both tracking and path reconstruction performance. During tracking, *efpp* constrains Bloom filter parameters and the required matching set size.

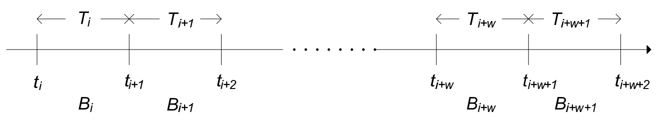

Fig. 2. Persistence of Bloom filter

These two values in turn affect the allowable skew among messages. If clocks can drift substantially, then more stringent *efpp* settings will require more frequent skew correction messages. During path reconstruction, the size of the matching set determines the number of filters which may need to be queried. Similarly, *efpp* reflects the likelihood of generating erroneous search results. Erroneous search results both increase path reconstruction overhead, by forcing searches down incorrect paths, and may corrupt results, by including incorrect brokers and subscribers on the path of a message. In the next section, we describe how matching set size is bounded by *efpp*. We discuss the effect of *efpp* on path reconstruction in Section 4.3.

4.2 Bounding Matching Sets

In this section, we derive a bound on matching set size in order to ensure a desired tracking accuracy (*i.e. efpp*). The size of matching sets is determined by the skew in timestamps of messages and the frequency of filter replacement (*i.e.* the rate at which filters in a history are stored).

Recall that the time stamp of a message ID is adjusted according to a skew adjustment, δ, for each publisher. Given a skew tolerance, T_s, the skew adjustment in the publishing phase ensures that timestamps generated simultaneously by different publishers of the same broker never differ by more than $2\,T_s$.

A filter is stored to disk either when it has reached its capacity or when the persistence interval has expired. Let T_f represent the time required to exhaust filter capacity at the peak message rate. That is, if c denotes the filter capacity (number of messages), and r denotes the maximum aggregate publishing rate (messages per unit time) from all publishers attached to a broker, then $T_f = c/r$. Likewise, let T_p indicate the time after which a filter must be stored to disk to reduce the data loss from a failure.

As a filter is replaced for the two reasons described above, the minimum filter lifetime T_n is defined as the minimum of T_f and T_p. On the other hand, the maximum filter lifetime is T_p by definition.

Given a T_n defined as above, we first consider the bounds on the size of matching sets for the publisher history and then expand the result into the cases of the routing histories and the subscriber histories. Theorem 1 formalizes the intuition that more accurate clock synchronization results in smaller matching set.

Theorem 1. *If $2\,T_s \leq w\,T_n$, where w is a non-negative integer, the size of matching set does not exceed $w + 1$.*

Proof. Figure 2 illustrates the time line of Bloom filter persistence. Let B_i be the ith persisted Bloom filter for a publishing history. The filter B_i is used from time t_i, inclusive, to time t_{i+1}, exclusive, with respect to the broker's clock. The time interval T_i is defined as $t_{i+1} - t_i$. From the definition of T_n, it follows $T_n \leq T_i$ for all i. The function $\textsc{MaxTS}[B_i]$ returns the maximum of timestamps contained in B_i (*i.e.* the second element of the time stamp range R_i), and $\textsc{MinTS}[B_i]$ returns the minimum (*i.e.* the first element of R_i). It suffices to prove that the time stamp range of B_i never overlaps with that of B_{i+w+1}.

$$\begin{aligned}
\textsc{MaxTS}[B_i] &< t_{i+1} + T_s \\
\textsc{MinTS}[B_{i+w+1}] &\geq t_{i+w+1} - T_s \\
&= t_{i+1} + \left(\sum_{j=1}^{w} T_{i+j}\right) - T_s \\
&\geq t_{i+1} + wT_n - T_s \\
&\geq t_{i+1} + T_s
\end{aligned}$$

Since the maximum time stamp in B_i is less than the minimum time stamp in B_{i+w+1}, the two ranges never overlap.

We would like to derive a similar result in order to bound matching set size for routing and subscriber histories. However, before we can do so, there are three complications that must be considered. First, since JMS subscriptions support content-based filtering, many messaging systems, including Gryphon, avoid routing messages down paths where there are no matching subscribers. As a result, a downstream broker b_2 may not see all the messages originated at an upstream broker b_1. However, this reduced fill rate does not increase matching set size and therefore does not reduce accuracy.

Second, the relative T_p values may be different at the source and downstream brokers. This is only a concern if some downstream broker has a T_p less than T_n at the source. In that case, the downstream broker may store filters at a higher rate and potentially violate the matching set bounds. For the moment, we assume this is not the case, and describe modifications for $T_p < T_n$ in the discussion below.

Finally, during forwarding, messages are subject to jitter, the time between the shortest message latency and the longest. Message jitter can increase the size of the matching set unless we account for it when determining the skew constraints. The following corollary formalizes this intuition:

Corollary 2 *Suppose T_j is the maximum jitter of messages. If $2\,T_s + T_j \leq w\,T_n$, where w is a non-negative integer, the size of matching set does not exceed $w + 1$.*

The proof is similar to that of Theorem 1 and is omitted. If T_j includes the expected skew among brokers, then corollary 2 can be used to determine the required clock synchronization among brokers.

As a guide to configuration, the persistence interval, T_p, at a given broker should be slightly larger than T_f, the time required to fill a filter at the maximum aggregate publish rate. This ensures that filters are nearly full when stored. The value for *efpp* is determined according to administrative requirements. This value can be used to guide tradeoffs between desired Bloom filter parameters (and indirectly the maximum T_f that can be supported), and desired matching set size. Once matching set size is known, the skew tolerance, T_s, at a given broker should be set to maintain the bound provided by Theorem 1.

4.3 Discussion

Besides determining matching set size requirements, the choice of *efpp* also affects path reconstruction as described in Section 3.3. In particular, a false positive may cause an unnecessary search during path reconstruction, which occurs when a neighbor finds a false positive in a local routing history, causing all of the local subscriber histories to be erroneously searched, and causing further unnecessary queries to the next set of downstream brokers.

For typical *efpp* values, the expected number of unnecessary searches is quite small. If a given broker has at most D' neighbors which did **not** observe a message, then the expected number of unnecessary searches (of one hop) that this broker will conduct is just the expected value of a binomial expression with parameters D' and *efpp*:

$$\sum_{i=1}^{D'} i \binom{D'}{i} \mathit{efpp}^i (1 - \mathit{efpp})^{D'-i} = D' \cdot \mathit{efpp}$$

Using the example *efpp* of 0.00001 from Section 5.1 and a D' of 5, this expected value is only 0.00005. At the highest *efpp* we tested, 0.05, the expected value is 0.25 (same D') or about one unnecessary one-hop search for every four brokers in the network. Note that if it is only critical to properly reconstruct the set of subscribers which received a message, then the probability of a false positive harmfully affecting the search results is reduced to $\mathit{efpp}^2 < \mathit{efpp}$ since both the local routing history and a subscriber history must report a false positive.

When applying Theorem 1 to the distributed case, we assumed that T_n at a source broker was less than the T_p at any downstream broker. In the event that there is some downstream broker b with T_p such that $T_p < T_n$, then b will store (partially full) filters more frequently than the source broker and will violate matching set size bounds. One way to avoid this problem is to retain the current filter even though it has been stored to disk, and overwrite the previous copy when filter capacity is finally reached. With this approach, only full filters are written and never at a rate higher than the source broker. The resulting filter is equivalent to the union [7] of the partially full filters that would normally have been stored separately to disk.

For space reasons, we have ignored the issue of retransmitted messages (*e.g.* retransmission due to failure), which are an unfortunate reality in large distributed systems. Retransmitted messages are only an issue when their time stamp is older than the oldest time permitted by the skew bounds in the current filter. We handle this case by adding a "singleton" set, S, to each filter history element, *e.g.* (B, R, S). The singleton set stores individual timestamps rather than a range. Retransmitted messages are stored in the current filter as usual except that their time stamp updates S rather than R. The notion of matching set is updated to include S, and filter capacity must now incorporate the maximum number of retransmissions which a filter may accommodate. This approach is tenable if the number of "old" messages per filter has a reasonable bound.

Finally, while the subscriber attachment map, S_m, is recorded using the clock of the broker hosting subscribers, the map is accessed during path reconstruction using the time stamp attached to the message, which may be skewed from the local broker. Thus, the time stamp should be adjusted by the clock skew between the two brokers. Since pairwise skews are difficult to maintain, the maximum possible skew among brokers is used for skew adjustment at the expense of potentially larger matching set. However, since this inaccuracy occurs only around join and departure, it should not affect the overall tracking accuracy in any significant way.

5 Implementation and Evaluation

We have implemented our tracking facility as an extension to the Gryphon messaging system. We refer to this implementation as the BF strategy. To evaluate our approach, we also implemented two other tracking strategies. The NULL strategy does not provide any tracking capability and serves as a baseline for performance comparisons. The ASYNC strategy is a "naive" logger which simply buffers each tracking event and periodically flushes the buffer to disk. This strategy is naive in the sense that in-memory records are not compressed and therefore the buffer must be flushed more frequently than a comparably sized BF buffer.

5.1 Implementation

In order to create message IDs with the proper tracking information, we have extended the Gryphon message ID implementation as illustrated in Table 1. The publisher is configured with a publisher ID and broker ID at connection time. The time stamp is recorded in milliseconds relative to the publisher's local time. A counter field is used to disambiguate messages which happen to be generated within the same millisecond. Our implementation synchronizes publisher clocks with local brokers so that $2\,T_s < T_n$ where T_n is either the time required to fill a filter at the peak messaging rate or T_p, whichever is smaller (see Section 4.2). This ensures that the size of the matching set is no greater than two.

Field	*Length*
Publisher ID	4 Bytes
Broker ID	4 Bytes
Time stamp	6 Bytes
Counter	2 Bytes

Table 1. Message ID

For a given *efpp* and the requirement that at most two filters overlap, we can generate corresponding Bloom filter parameters m, n and k. For example, an *efpp* of 0.00001 (five nines accuracy) corresponds to an *fpp* of less than $1 - \sqrt{0.99999} \approx 5 \times 10^{-6}$ for each filter. To satisfy this level of accuracy, we might choose $\frac{m}{n}$ to be 26 and k to be 16, so that the capacity of the each filter, n, is limited by the size of the filter m.

The number and range of the hash functions are determined by k and m, respectively. For our implementation, we derive hash functions using one or more applications of MD5 [13], with each application providing 128 hashed bits. To generate independent hashes, we prepend a random, unique prefix byte for each application. The output bits are then divided into k equal segments each addressing a range of m. For example, if $k = 16$ and $m = 65536$, we might use two applications of MD5 divided into 16 segments of 16 bits each.

5.2 Experimental Setup

We evaluated our approach by measuring experimental performance for each of the three strategies (BF, NULL and ASYNC). We consider two performance metrics for evaluation:

End-to-End Latency: The elapsed time from the publication of a message and its delivery to one matching client. This measure is an indirect indication of overhead which includes processor overhead (*i.e.* the per-message computation cost imposed by tracking) and I/O overhead (*i.e.* the per-message cost to store tracking records to disk).

Storage: The total disk storage used by each strategy. For the BF strategy, this is the size of the stored history of filters. For the ASYNC strategy, this is the size of the stored buffers.

For evaluation purposes we only considered measurements for a single broker. Since the various tracking strategies only impose processing and disk overhead (*i.e.* they do not alter routing protocols or introduce network traffic), the single broker case allows for more accurate measurements without loss of generality.

Each strategy was tested on a dedicated gigabit LAN consisting of 5 6-way 500 MHz PowerPC servers each with 4 GB of memory running AIX. The test setup consisted of one Gryphon broker hosting 200 publishers and 500 subscribers. A total of 500 topics were used, with each subscriber subscribing to a single topic, and each publisher publishing to ten topics. Subscribers were randomly distributed among the topics but a small portion of the topics had no subscribers. The aggregate input rate was 5000 messages per second (25 messages per second per publisher) divided evenly among all topics so that each subscriber received approximately ten messages per second (*i.e.* the aggregate output rate was 5000 messages per second). The broker ran on a dedicated machine, the publishers were spread evenly over two machines, and the subscribers were spread evenly over two machines. Each strategy was tested separately at a run length of 30 minutes.

We constructed two test scenarios which differed in the limits they placed on in-memory buffers. The first scenario limited buffers (*e.g.* publisher, routing or subscriber tracking records) to 10000 bytes. The second scenario limited buffers to 135000 bytes. For each scenario, we varied T_p so that we could observe the effects of storing full or partially full buffers. Finally, we used three different accuracy requirements when configuring the BF strategy. Table 5.2 summarizes the various test configurations.

Strategy	**Buffer Limit**	**Accuracy**	T_p **(seconds)**	**Stores Full Buffers?**
NULL	N/A	N/A	N/A	N/A (baseline)
ASYNC	10K	100%	1	Yes
ASYNC	135K	100%	1	No
ASYNC	135K	100%	2	Yes
BF	10K	99.999% - n=2521, m=64K, k=16	1	Yes
BF	10K	99% - n=5461, m=64K, k=8	1	No
BF	10K	95% - n=7281, m=64K, k=8	1	No
BF	10K	99% - n=5461, m=64K, k=8	2	Yes
BF	10K	95% - n=7281, m=64K, k=8	2	Yes
BF	135K	99.999% - n=38836, m=1M, k=12	1	No
BF	135K	99% - n=87381, m=1M, k=6	1	No
BF	135K	95% - n=131072, m=1M, k=6	1	No
BF	135K	99.999% - n=38836, m=1M, k=12	2	No
BF	135K	99% - n=87381, m=1M, k=6	2	No
BF	135K	95% - n=131072, m=1M, k=6	2	No
BF	135K	99.999% - n=38836, m=1M, k=12	8	Yes
BF	135K	99% - n=87381, m=1M, k=6	8	No
BF	135K	95% - n=131072, m=1M, k=6	8	No
BF	135K	99% - n=87381, m=1M, k=6	18	Yes
BF	135K	95% - n=131072, m=1M, k=6	18	No
BF	135K	95% - n=131072, m=1M, k=6	27	Yes

Table 2. Test Configurations. If "stores full buffers" is "yes", then buffers were filled before T_p expired. For the BF strategy, we show the corresponding configurations for n (filter capacity), m (filter size in bits), and k (number of hash functions).

5.3 Results

Latency was measured as the round trip time for a special latency message and includes both the tracking overhead for the message, as well as the overhead of competing with tracking for other in-flight messages. Figure 3 gives the average latency for our test configurations. The NULL strategy was also measured as a reference point.

In the first scenario, the BF strategy performs significantly better than the ASYNC strategy and imposes only slight overhead as compared to the NULL strategy. This behavior is easily understood by considering the frequency of disk operations. With a 10000 byte in-memory limit and message ID size of 16 bytes, the ASYNC strategy can record at most 625 messages between disk forces. With an aggregate message rate of 5000 messages per second, this corresponds to approximately eight disk forces per second. In contrast, the highest reliability BF strategy can accommodate 2521 messages

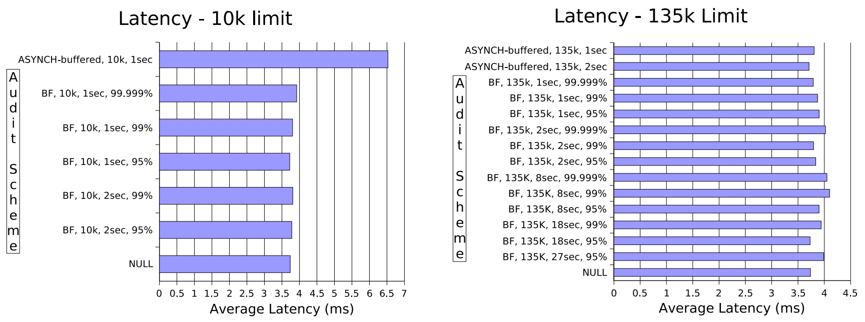

Fig. 3. Latency Comparison.

in the given memory limit, or slightly less than two writes a second. With four times as many disk forces per second, the ASYNC strategy is at a severe disadvantage when in-memory constraints are tight. The second scenario shows that, not surprisingly, reducing the frequency of disk forces is the key to controlling latency. With the higher memory limit, ASYNC can accommodate about 8437 messages between disk forces and shows performance comparable with NULL. For the BF strategy, disk forces are never a critical factor and larger buffers do not impose any significant latency overhead.

Even with larger buffers, however, the lack of in-memory compression in the ASYNC strategy makes it highly sensitive to message rate and ID size. Thus, while ASYNC achieves acceptable performance for the message rate we tested, the BF strategy has more tolerance for significant increases in rate, and is not dependent on ID size. For example, quadrupling the message rate (*i.e.* 20000 messages per second) would cause slightly less than three disk forces per second for ASYNC whereas even the highest accuracy BF configuration would still be storing nearly half-empty filters. Similarly, doubling the ID size halves the storage capacity of ASYNC without any effect on BF.

Neither ASYNC nor BF attempt further compression before forcing buffers to disk. It is reasonable to assume, however, that standard compression techniques might be applied to further reduce storage requirements. We simulate this effect by applying `gzip` to each buffer once it has been forced to disk. For evaluation purposes, we report the average number of bytes per message which is just the total disk log size divided by the number of messages routed for a particular test. Figures 4 and 5 give the average storage use (uncompressed and compressed) for our test configurations. Because our experiments use a single broker, the storage footprint is due to the publisher and subscriber histories. In a multi-broker setting, local routing history storage is bounded by the size of the publisher history at the source broker. Thus, the storage footprint of the publisher history approximates the added footprint in a multi-broker setting.

As to be expected, the ASYNC strategy is at a storage disadvantage since audit records are not compressed in memory. However, the simple structure of ASYNC logs allows for significant compression as can be seen in the figures. Conversely, the structure of filters used in the BF strategy does not allow significant compression. In fact, since the full Bloom filters must have random bit patterns with equal number of ze-

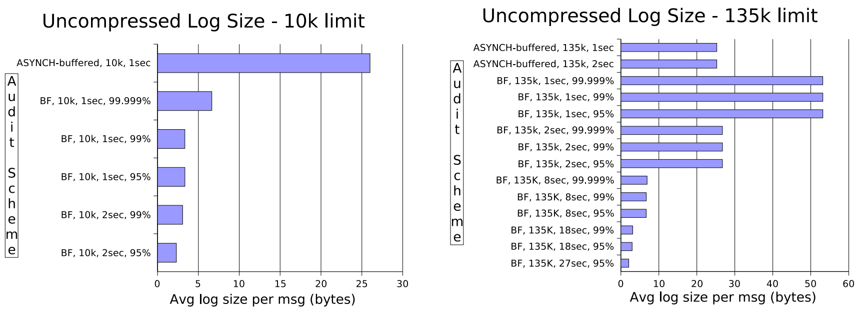

Fig. 4. Uncompressed storage use.

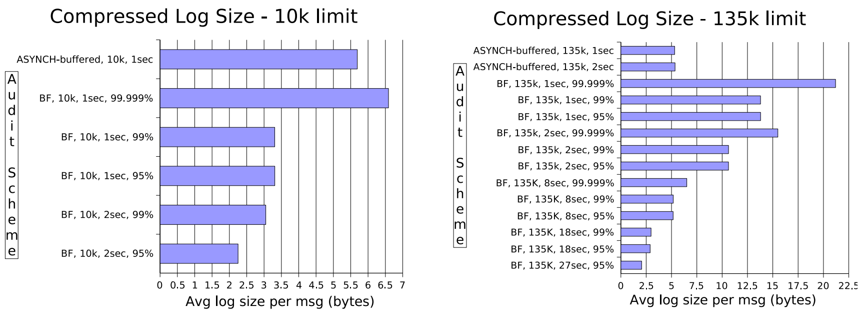

Fig. 5. Compressed storage use.

ros and ones, compression is impossible in principle. Moreover, the BF strategy is at a significant disadvantage when only partially full filters are written to disk, particularly when high accuracy is required. In these cases, the underlying filters are sufficiently random to defy straightforward compression techniques. In cases where both ASYNC and BF are forced to persist partially full filters of roughly the same size at roughly the same rate (*e.g.* 135K buffers when $T_p = 1$), the ASYNC strategy will utilize disk space more efficiently. With larger buffers, the BF strategy does not offer an advantage unless disk forces can be delayed for several seconds.

Although the compressed ASYNC strategy consumes less storage than some of BF, the BF strategy still has advantage over ASYNC for two reasons. First, the ASYNC strategy requires more complex and larger indexing to avoid otherwise inefficient sequential search. By contrast, in the BF strategy, such indexing is constructed at the filter level, as opposed to at the message level, because each Bloom filter answers a membership query efficiently by applying k hash functions. Note that a filter has a capacity of at least thousands of messages. Second, since false positives are not correlated among brokers, many false positives can be deduced as such by comparing the results from neighboring brokers. For example, if only one broker, and none of its neighboring ones, claims to have seen a message, it is highly likely to be a false positive. Thus, low accuracy (that is, high *efpp*) at a single broker does not necessarily result in the overall low accuracy, which implies that it is rather safe to take low accuracy level.

6 Related Work

Our work is most similar to traceback facilities proposed at the IP layer. These systems are designed to help identify the source of distributed denial-of-service attacks (DDoS). There are two basic approaches: packet marking [14, 16], and route auditing as in SPIE [15]. Although developed independently, SPIE is very similar to our traceback facility in that both approaches use Bloom filters to track recent traffic. The main difference is that SPIE is intended to detect recent or in-progress DDoS attacks. As a result, only a single Bloom filter (per node) is required and stale data is periodically purged. In contrast, our traceback facility is designed to allow historical queries of message routes. Thus, Bloom filters are regularly persisted and we have developed new techniques to manage queries across a history of filters.

In the context of monitoring for middleware applications, various systems have been developed starting with CORBA monitors such as JEWEL [9], and more recently Enterprise Java Bean monitors [10]. These systems focus on component level interactions, rather than middleware messaging. In the context of messaging, recent work has focused on novel routing architectures [12] and various performance enhancements [11]. However, tracking facilities which specifically address messaging middleware do not appear in the literature. Nonetheless, commercial products such as WebSphere MQ [2] provide basic auditing facilities which log message information to disk. This solution works well for low message rates but is difficult to scale because of overhead. Our tracking facility attempts to provide scalability by reducing the cost of per-message operations while compressing persistent records.

Finally, Bloom filters [6] have enjoyed wide application in distributed systems, for example as a technique for caching web pages [8]. Broder and Mitzenmacher provide an interesting survey of Bloom filters as applied to networking in [7]. Typically, Bloom filters are used as a volatile cache for various types of data. In contrast, our message tracking facility persists and retains multiple Bloom filters. Thus, our main contribution is a scheme for indexing multiple Bloom filters for the purpose of managing false positive probability.

7 Conclusion

We believe that efficient mechanisms for monitoring and auditing middleware messaging will become increasingly important as enterprise business applications are more widely deployed. In this paper, we have defined a message tracking facility for distributed messaging middleware, and presented a low overhead system design which realizes such a facility. We expect future enterprise applications to require large scale deployments with tens of nodes and tens of thousands of clients. Thus, we believe that a key feature of our approach is the ability to tune accuracy (and hence overhead and resource requirements) to the needs of the application.

We have evaluated our design by measuring overhead as compared to a system with no tracking, and by illustrating trade-offs between absolute accuracy and high overhead. A key advantage of our approach is that it is tolerant of high message rates, and insensitive to the size of IDs being tracked. In particular, we have shown that low latency overhead is possible if accuracy can be relaxed to as little as 99.999% of a "perfect" system. Likewise, if only minimal memory is available for tracking, then low disk utilization is possible even with high-accuracy. If memory is not a constraining factor, then low disk utilization requires that disk writes be delayed until filters are full. We note that it is possible to remove this limitation by either merging partially full filters offline, or storing but retaining partially full filters in memory until they have reached capacity. We intend to implement these improvements as well as seek further optimizations as part of our future work.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments.

References

1. Gryphon research project. `http://www.research.ibm.com/gryphon/`.
2. IBM WebSphere MQ. `http://www.ibm.com/software/integration/wmq/`.
3. Java 2 Platform Enterprise Edition. `http://java.sun.com/j2ee/1.4/`.
4. Java Message Service (JMS). `http://java.sun.com/products/jms`.
5. Sumeer Bhola, Robert E. Strom, Saurabh Bagchi, Yuanyuan Zhao, and Joshua S. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *DSN*, pages 7–16. IEEE Computer Society, 2002.
6. Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.
7. Andrei Broder and Michael Mitzenmacher. Network applications of Bloom filters: A survey. In *Proceedings of the 40th Annual Allerton Conference on Communication, Control and Computing*, pages 636–646, 2002.
8. Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of ACM SIGCOMM*, 1998.
9. F. Lange, R. Kroeger, and M. Gergeleit. Jewel: Design and implementation of a distributed measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):657–672, 1992.
10. Adrian Mos and John Murphy. A framework for performance monitoring, modelling and prediction of component oriented distributed systems. In *WOSP '02: Proceedings of the Third International Workshop on Software and Performance*, pages 235–236. ACM Press, 2002.
11. Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Robert Strom, and Daniel Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *Middleware '00: IFIP/ACM International Conference on Distributed Systems Platforms*, pages 185–207. Springer-Verlag New York, Inc., 2000.
12. Peter R. Pietzuch and Jean Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In *DEBS '03: Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8. ACM Press, 2003.

13. R. Rivest. RFC 1321. the MD5 message-digest algorithm, April 1992.
14. Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Practical network support for IP traceback. In *Proceedings of ACM SIGCOMM*, pages 295–306, 2000.
15. Alex C. Snoeren, Craig Patridge, Luis A. Sanchez, Christine E. Jones, Fabrice Tchakountio, Beverly Schwartz, Stephen T. Kent, and W. Timothy Strayer. Single-packet IP traceback. *IEEE/ACM Transactions on Networking (TON)*, 10(6):721–734, December 2002.
16. Dawn X. Song and Adrian Perrig. Advanced and authenticated marking schemes for IP traceback. In *Proceedings IEEE Infocomm 2001*, 2001.