Reliable and Highly Available Distributed Publish/Subscribe Service

Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen
University of Toronto
{*reza, jacobsen*}@*eecg.utoronto.ca*

Abstract

This paper develops reliable *distributed publish/subscriber algorithms with service availability in the face of concurrent crash failure of up to δ brokers. The reliability of service in our context refers to per-source in-order and exactly-once delivery of publications to matching subscribers. To handle failures, brokers maintain data structures that enable them to reconnect the topology and compute new forwarding paths on the fly. This enables fast reaction to failures and improves the system's availability. Moreover, we present a recovery procedure that recovering brokers execute in order to re-enter the system, and synchronize their routing information.*

1. Introduction

Publish/subscribe (P/S) is a message-passing many-to-many communication paradigm that provides an abstract and high-level interface for data *publishers* to publish messages and *subscribers* to receive messages that *match* their interest. Distributed P/S systems strive to achieve scalability and avoid a single point of failure by using an overlay network of *brokers* to which publishers and subscribers connect. Publication messages issued by a publisher client are forwarded through the network and delivered to subscriber clients with matching subscriptions.

There are an increasing number of enterprise-level applications [1], [2] as well as several standards-based industrial messaging platforms such as Tibco Rendezvous, Java JMS, AMQP, and IBM MQSeries that support various flavors of publish/subscribe, e.g., *topic-based* and *content-based*. In this paper, we focus on the content-based P/S model which allows for more selectivity for subscribers via specifying fine-grained filtering subscriptions. The system delivers those publications that satisfy all specified filters. Flexibility and decoupling provided in these models are the major driving force for adoption in application areas with requirements for many-to-many communication between heterogeneous systems such as enterprise application integration and business process management [2].

While typical usage scenarios of an average Internet user may tolerate loss of a few publications, enterprise applications [1], [2] often demand high service reliability and availability. We define *reliability* as the exactly once delivery of publications to matching subscribers in the same order that they were published at the source. Furthermore, *availability* in our context refers to the portion of time that the P/S service is provided to (non-faulty) clients over the system's life span.

Several existing publish/subscribe systems [3], [4], [5] have taken a best-effort approach to reliability, in the sense that it is to some extend tolerable that publications be reordered, or some publications never be delivered at all. Intermittent component failures and unreliable communication links are among the most common causes that may lead to these "unreliable" scenarios. On the other hand, time-critical applications such as those that process financial market data have stringent delivery delay demands. While a simple FIFO store-and-forward approach may improve reliability, it does not necessarily achieve high availability (i.e., while a broker is down publication delivery to a subset of subscribers is stalled). Finally, topology reconfiguration techniques have been studied [6] in an attempt to handle broker failures. These techniques improve availability of the system; however, it is not straightforward to maintain reliable publication delivery while reconfiguration is in progress.

In this paper, we develop distributed P/S algorithms that uphold service reliability and at the same time achieve high availability. The publication delivery service provided by our system relieves the subscribing clients from maintaining message forwarding state. It is thus the sole responsibility of the brokers to ensure FIFO-ordered delivery of publications to matching subscribers. For this purpose, brokers maintain subscription routing information. This information is used to establish publication forwarding paths from publishers to subscribers and is kept in a *consistent* state throughout the entire network. In our approach, we ensure that concurrent failure or recovery of up to δ brokers does not violate the system's consistent subscription state and thus reliable publication forwarding is achieved. We rely on a topology management and subscription propagation scheme that enables brokers

to readily compute new forwarding paths in the event of failure of their nearby neighbors. Our approach eliminates the need for subscription re-propagation and does not require nodes to negotiate and agree on new forwarding paths. This greatly improves the availability of the service in the face of one or more failures.

We use δ as a configuration parameter that corresponds to the maximum number of *concurrent* broker failures that our algorithms will tolerate without interrupting the P/S service. We adopt a tree-based topology network. Brokers are required to maintain a partial view of this tree that includes all brokers within distance $\delta + 1$ (Figure 1(b)). This information is stored in a local data structure called the broker's *topology map*. The topology map is updated as brokers *join* and *depart* from the system. In addition to the topology map, the brokers also maintain a *subscription table* that serves to route publications. The subscription propagation algorithm disseminates clients' subscriptions which are inserted into the brokers' subscription tables. The subscriptions contain a $from$ field that points to another broker located $\delta + 1$ hops (in the topology tree) closer to the subscriber. In the event of failure of one or more neighbors, the broker uses this information and its topology map to reconnect the network topology, and forward publications over new paths towards matching subscribers.

Moreover, we consider transient failures and devise a recovery procedure that failed brokers can use in order to re-enter the system. The recovery procedure restores the broker's lost routing information. During recovery, the broker also participates in message forwarding using special messages that contain complete information about the destinations of the message. This eliminates the chance that a message is mistakenly dropped due to insufficient routing information (during recovery). Once the recovery is complete, the broker's internal data structures are up-to-date with respect to subscription and topology information that was present in the system both before the start of recovery as well as while recovery was in progress.

The contributions of the paper are as follows: *(i)* a novel topology management and subscription propagation scheme that lies at the core of our approach (Section 2); *(ii)* a publication forwarding algorithm that is capable of handling broker failures (Section 3); *(iii)* a recovery procedure that enables previously failed nodes to recover by restoring their lost state (Section 4); *(iv)* an optimization technique that improves the overall performance of the system both in terms of message traffic and CPU utilization (Section 5). We elaborate on the implementation details, and evaluate various performance aspects of our systems in Sections 6 and 7, respectively. Finally, we review related work in Section 8 and conclude the paper in Section 9.

2. Broker Routing Information

In this section, we elaborate on the topology management protocol that updates the brokers' topology maps, as well as the subscription propagation algorithm used to populate the subscription routing tables.

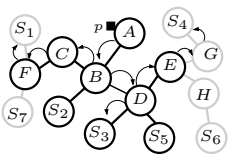
2.1. Topology Map

In our approach brokers are organized in a tree-based overlay topology. Throughout this paper we use the term *topology* and *topology path* to abstractly refer to this tree, and a path in this tree, respectively. A broker's topology map is a data structure that contains the broker's partial view of the topology. This view is in the form of a tree centred at the broker itself, and includes vertices (i.e., other brokers) and edges located within distance $\delta + 1$ of the broker. For example, Figure 1(b) illustrates A's topology map (darker lines) in a network where $\delta = 2$. The topology map is updated incrementally as brokers join the system or leave permanently.

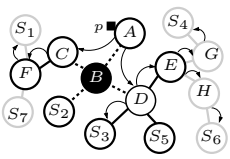
As part of the join operation, an incoming broker A initializes its own local topology map by contacting an already joined broker B, referred to as *joinpoint*. The joinpoint delivers a subset of its own topology map and subscription table to the joining broker. The transferred topology map contains brokers within distance δ of B (these brokers are within distance $\delta + 1$ of A). Furthermore, the joinpoint propagates a *join message*, j, to other neighboring brokers containing information about the newly created link, AB. Other brokers that receive j update their local topology map accordingly, and continue to send j to their downstream brokers. This process stops at the brokers that either have no additional neighbor (i.e., *edge brokers*), or are located at distance δ of the joinpoint. At this point, the brokers issue a *confirmation* message, $Conf^j$, and send it in the reverse direction towards the joining broker. Intermediate brokers collect all confirmation messages and subsequently issue a new confirmation to their neighbors located closer to the joining broker. Finally, when the joining broker receives the confirmation message from the joinpoint, the join operation is complete, and the broker is ready to accept new joining brokers (if it is a broker), or issue subscriptions/publications (if it is a subscriber/publisher). The extended version of this paper proves the correctness of the join operation for concurrently executing joins [7]. For the case of a node's permanent departure, our system is able to operate until arrival of a replacing broker. The new broker performs a similar operation to join that informs the neighboring brokers to update their topology map.

Src	Subscription Predicate
✓ S_1	$[stock = `SUN'], [quote = *]$
✗ S_2	$[stock = `SUN'], [quote > 30]$
✓ S_3	$[stock = `DELL']$
✓ S_4	$[stock = *]$

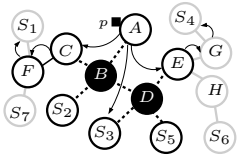
(a) Subscriptions matching publication p : $[stock, `SUN'], [quote, 55]$



(b) Darker lines are in A's topology map



(c) After failure of broker B



(d) After failure of brokers B and D

Figure 1. A sample network configured with $\delta = 2$. Publication p matches subscriptions $\{S_1, S_3, S_4\}$; brokers A and B compute $Rcpt^p_A = \{F, E, S_3\}$, $Rcpt^p_B = \{S_1, S_2, S_3\}$. Arrows indicate broker-to-broker connections over which p is sent.

2.2. Subscription Routing Tables

Brokers maintain subscription routing tables that contain entries for subscriptions inserted into the system. Each entry is in the form of $\langle preds, from, seqVect \rangle$, where $preds$ are the predicate filters specified by the subscribing client and determine the type of messages it is interested to receive; $from$ is the identifier of another broker that is updated as the subscription propagates through the network, and $seqVect$ is a vector of sequence numbers generated by the last $\delta + 1$ brokers on the propagation path of the message. The subscription propagation algorithm works in a similar fashion as the join algorithm: each broker that receives the subscription, s, stores it locally, and further forwards it to other brokers farther away from the subscriber. However, unlike join messages that propagated only to brokers within distance $\delta + 1$, the subscription messages are propagated throughout the network. Once the subscription reaches an edge broker, it is confirmed via a confirmation message, $Conf^s$, that is sent back. Intermediate brokers receive all confirmation messages, and subsequently confirm the subscription. Once $Conf^s$ arrives at the subscriber, the subscription operation is completed, and the client can reliably receive matching publications.

In the case a broker F has failed, the neighboring brokers use their topology maps to identify other non-faulty brokers downstream of F and continue propagation of s via new paths. Since the broker may bypass failed brokers, the $from$ field of the subscription is updated in accordance to the distance of the receiver which might be up to $\delta + 1$ hops away:

- If subscriber is within distance $\delta + 1$ from the receiving broker, $from$ points directly to the subscriber.

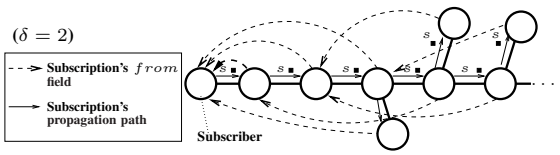


Figure 2. The $from$ fields of subscriptions point back to a broker on the connecting path to the subscriber.

- Otherwise, $from$ points to a broker on the topology path towards the subscriber that is $\delta + 1$ hops closer to the subscriber than the receiving broker.

Figure 2 illustrates the above conditions in a sample network. Note that updating $from$ in this manner ensures that it always points to a broker located in the topology map of the receiving broker.

3. Publication Forwarding Protocol

In this section, we describe how brokers use their topology maps and subscription tables to forward publications both in absence and presence of failures. We also devise mechanisms to detect and eliminate duplicate delivery of messages.

3.1. Non-Faulty Execution

For each publication message, p, a forwarding broker generates a unique monotonically increasing sequence number upon receipt of p for the first time. This sequence number is added to the $seqVect$ of the publication message when it is sent. Considering the path that p traversed to arrive at B as upstream (and the other direction as downstream), B's forwarding task is to send p to those downstream subscribers that are interested in p's content. This is done by first identifying the subscriptions whose predicate fields *match* the content of the publication message. B inserts the $from$ field of these subscription entries that are downstream from B w.r.t. p's propagation path into a set data structure called the *recipient set*, $Rcpt^p_B$:

$$Rcpt^p_B = \{from | \exists \langle preds, from \rangle \wedge matches(p, preds) \wedge from \text{ downstream of } B \text{ w.r.t. } p\}$$

B then uses its local topology map to generate a set of associated topology paths for each broker in $Rcpt^p_B$. This set is referred to as *outgoing paths*, $outPaths^p_B$. Finally, B sends a copy of p to only the closest available broker on each $path_i$ in $outPaths^p_B$, and awaits receipt of its confirmation message, $Conf^p$. The receiving (downstream) brokers follow the same procedure, ensuring that p is forwarded only on the paths that lead to a matching subscriber. Once the publication is delivered to the matching subscribers, the subscriber immediately issues a confirmation message that is sent

in the reverse direction of the publication propagation. Similar to subscriptions and join messages, upstream brokers further issue new confirmation messages once they receive all outstanding confirmations from downstream brokers. At this point, the brokers also discard their local copy of the publication, as well as all the internal data structures regarding the message.

3.2. Handling Broker Failures

When there are no failures, brokers are connected to their *immediate* neighbors in the topology, and forward publications over topology paths (Figure 1(b)). However, once a neighboring broker fails, its non-faulty brokers (say B) uses its topology map to identify and connect to the next available brokers one hop away from the failed peer (Figure 1(c)). If any of the new connections also fail, the broker proceeds with the next neighbor(s) (Figure 1(d)). Intuitively, if the number of failed brokers does not exceed δ, B will be able to maintain the network's connectivity and use newly created connections to forward messages. To achieve this behavior in our forwarding algorithm, the brokers designate some of their *sessions* (i.e., a connection between two broker) to other peers as *active*. The set of active sessions correspond to closest non-faulty neighboring brokers in every direction of the topology map. Roughly speaking, a session is active if *(i)* its remote endpoint is not failed, and *(ii)* all intermediate brokers are failed. The active sessions are updated regularly based on the broker's knowledge of the topology as well as the current state of the sessions (i.e., operational, failed, recovering).

Publications are sent only on active sessions, and only if the topology path to their remote endpoint intersects an outgoing path in $outPaths$. If this test succeeds, we say that a copy of the publication is *checked out* for the session. The broker sends the checked out messages and awaits receipt of the confirmations. This is illustrated in Figures 1(c) and 1(d) in which A establishes new active sessions to C and D and ultimately to E and S_3. This effectively enables B to bypass the failed brokers. A publication p that matches S_1 and S_2's subscriptions is forwarded over these sessions since the topology path to these endpoints intersects a topology path in $outPaths^p_B$.

3.3. Duplicate Messages

Upon detection of the failure of the remote endpoint or as part of the recovery, the broker may have to re-send the publications that have been previously forwarded, but have not yet been confirmed. These messages may arrive at the downstream brokers as duplicates. To ensure exactly-once, reliable delivery, we devise mechanisms to detect and filter out duplicate messages. For this purpose, the brokers use the sequence vectors of the messages, $seqVect$, in the following way: $seqVect$ of a message contains a list of the sequence numbers generated by the last $\delta + 1$ brokers on the topology path of its propagation. A receiving broker initially inspects the sequence vector, and keeps track of the highest sequence number it has seen in a message so far. A duplicate is detected once any of the sequence numbers in the message's $seqVect$ is equal or precedes the stored highest sequence number associated with the same broker.

Duplicate messages are not forwarded again. However, if the broker is still awaiting receipt of the corresponding confirmation messages, the identity of the new sender of the message is preserved (this sender may be different from the sender of the first copy). Once the broker is about to confirm the message, it sends a confirmation message to all senders of the message. On the other hand, if the duplicate message has previously been confirmed, the broker immediately issues the confirmation without any delay. Intuitively, this is sufficient, since having previously confirmed the message, the broker is assured that its downstream brokers have already received and processed the message.

4. Recovery Protocol

A failed broker loses all its subscription routing information and is unable to receive new subscription messages. It is thus the task of the recovery procedure to restore the broker's routing information according to the current state of the network. We assume that a recovering broker R retains the same broker identifier (and address) it used prior to failure, and that it can restore its topology map (either from stable storage or via contacting a registry service that keeps track of the brokers' location in the topology). We divide the recovery procedure in three parts: *(i)* synchronization point discovery and synchronization; *(ii)* message forwarding via guided messages; *(iii)* and finally, termination of recovery.

4.1. Synchronization Protocol

We refer to the mechanism that delivers the lost routing information as *synchronization* and refer to the peers that send this information to the recovering broker as *synchronization points*. The synchronization points correspond directly to the active sessions introduced in Section 3.2, and are discovered in the same manner as the forwarding algorithm manages the active sessions. The broker issues a request to each synchronization point in order to retrieve a portion of its lost routing information.

Upon receiving the recovery request, the synchronization point computes a subset of its own topology

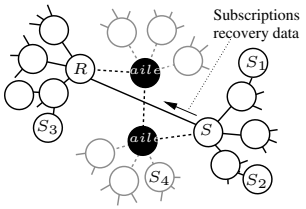


Figure 3. Synchronization point S sends subscriptions from its downstream subscribers (S_1, S_2 but not S_3, S_4) to recovering broker R.

map that falls within distance $\delta + 1$ of the recovering broker. This may include new brokers that have recently joined (missing in the initial topology map of the recovering node) or are currently in the process of joining the system. This information is sent to the recovering broker which will immediately updates its topology map and may further identify new peers to connect to.

Once the transmission of topology information is complete, the synchronization point proceeds to transfer the subscription information stored in its subscription table. For any such subscription s, the synchronization point constructs a new subscription message s' only if $s.from$ is located downstream of the synchronization point w.r.t. the recovering broker. More intuitively, had the recovering broker not failed, it would have received these subscriptions from the synchronization point.[1] Figure 3 illustrates this process. The new subscription has an identical $preds$ and $seqVect$ fields as s, but its $from$ field is modified in the same manner the subscription propagation algorithm updates the subscriptions' $from$ field.

4.2. Forwarding with Guided Messages

A guided message is a message that encapsulates a join, subscription, or publication message and includes a header that conveys information about the destination of the message. Once the synchronization point completes the transmission of the topology information, it starts to deliver the normal traffic in the system to the recovery broker by encapsulating them in a guided message and including the identifiers of downstream recipients. Having already received the topology information, the recovering broker is able to further forward the message on its own by inferring the required path information via consulting its own topology map. Guided messages improve the recovery process in two ways: First, while synchronization tries to restore the *old* routing information that was already present in the system, guided messages strive

1. This is different from the case of topology map recovery information, which may contain information edges that fall in between the recovering broker and the synchronization point, as well as other edges away from the peers.

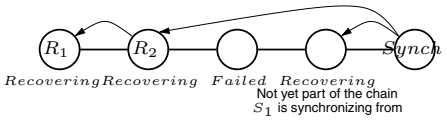


Figure 4. Chain of recovering broker that all receive part of their routing information from $Synch$.

to ensure consistency of the recovered broker's routing information w.r.t. the join and subscribe operations that are carried out concurrently with the recovery. As we described in Section 2.2, subscription and join messages need to be confirmed and respectively added to the subscription tables and topology map. For this purpose, the synchronization point, delivers these messages to the recovering broker much like the normal system operation. The receiver also processes the received messages similarly but instead of computing the recipient set on its own, it uses the information provided in the header of the guided message. Second, if the subscription routing tables are large in size their transfer may take some time. During this time, the subscription information at the recovering broker may not be sufficient to guarantee reliable delivery. Using guided messages provides an opportunity to minimize the impact of the recovery on the systems' publication delivery delay. This is achieved by prioritizing guided message over recovery messages in transit.

4.3. Termination of Recovery

Once the synchronization point has successfully transferred all its local subscription routing tables to the recovering broker, it sends a special SUB_END message to indicate this fact. Upon receipt of this signal from all synchronization points, the recovering broker completes its recovery by adding all received subscriptions to its routing tables, and notifies its synchronization points of its state change. The synchronization points subsequently update their local state and continue with normal forwarding of messages (i.e., without use of guided messages).

To allow faster recovery, recovering brokers may synchronize their routing state by contacting their nearby brokers that may also be in the recovery stage. This is illustrated in Figure 4. A recovering synchronization point (R_2 in the figure) may not initially possess all the subscription information to supply to the other recovering peer (R_1). Instead, this information arrives at R_2 as the brokers proceed through their recovery, and it may well be the case that R_2 also attempts to recover from R_1. In order to avoid deadlocks in termination of recovery, we require recovering synchronization points (e.g., S_{rec}) to signal SUB_END to other recovering peers once they have received this signal from *all other* synchronization points. Note that in this case, SUB_END may be sent

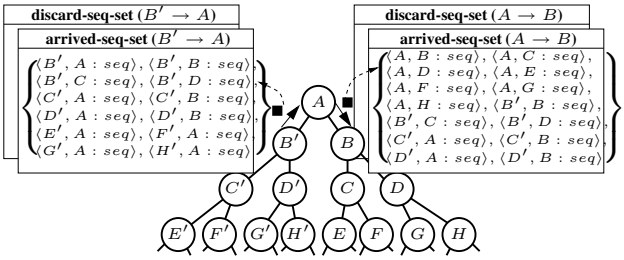


Figure 5. Content of a sample $dack$ message ($\delta = 2$)

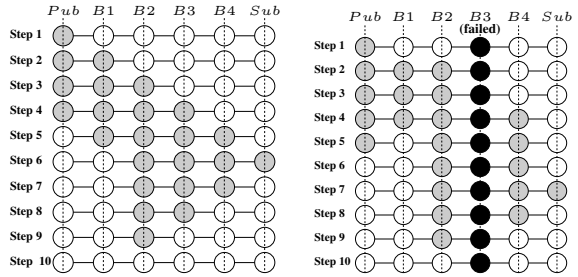
before S_2 has received *all* the subscription information it needs for full recovery. Nonetheless, since S_2 has received SUB_END from all other peers, it will receive no other subscription message that may be sent to R_1. In Section 7, we analyze the effectiveness of concurrent recovery on the recovery delay.

5. Improving Publication Propagation

In this section, we present an optimization in order to reduce the number of network messages. In our reliable forwarding algorithm, for every message (publication or subscription) that a node forwards, it expects to receive a confirmation. For the case of publication messages, confirmations only serve to indicate that the upstream node can safely discard its local copy of the message as all downstream matching subscribers have successfully received the publication. In our improved scheme, nodes periodically exchange aggregated acknowledgments in order to achieve the same purpose. Since publications are typically the dominant traffic in the system (i.e., orders of magnitude larger than subscriptions) this approach significantly reduces the network traffic.

At each node, acknowledgment messages are exchanged periodically over *all* connections and contain information with regard to the brokers within distance $\delta + 1$ downstream from the sending node. We thus refer to these messages as *depth-ack* ($dack$). A $dack$ message contains two sets of information: *(i)* $arrivalSeqSet$; and *(ii)* $discardedSeqSet$. An entry in each set is of the form: $\langle A, B : seq\rangle$, where seq is the largest sequence number that reporting broker A received or discarded from B. The sender of a $dack$ message includes entries for all A and B that are within $\delta + 1$ hops from each other (Figure 5).

Now consider any publication p that was processed and assigned seq^p_N by node N. Furthermore, let $outPaths^p$ be the set of paths from N to the computed recipient set of p. Receiving a $dack$ message, node N purges its own copy of p once one of the following safety conditions holds over all paths in $outPaths^p$: *(i)* All nodes on that path have reported arrival sequence numbers from N that succeed seq^p_N; or *(ii)* At least one node on that path has reported discarded sequence



(a) Brokers discard pubs once $\delta + 1$ downstream brokers on all $outPath^p$ received them

(b) Brokers discard pubs once at least one downstream broker on each $outPath^p$ discard them

Figure 6. Publication propagation on *one* topology path ($\delta = 2$). Highlighted brokers hold a copy of publication.

numbers from N that succeed seq^p_N. Figure 6(a) illustrates this mechanism on a single path between Pub and Sub nodes. It is easy to see that once the condition (*i*) holds either the publication has been delivered to the matching subscriber (if it is within distance $\delta+1$ from N), or at least $\delta+1$ other nodes towards the subscriber have received the publication. Thus, a copy will survive failure of up to δ of these nodes. In both cases, it is safe for N to purge p locally. On the other hand, if a node on this path fails to communicate its $dack$ information, then N relies on the second condition to decide when to discard p. The intuition is that nodes farther from N can see farther in the network and if the publication makes its way to the subscriber, they will eventually discard their own local copy of p. As illustrated in Figure 6(b) (Step 6) this subsequently signals N to purge p as well.

It is interesting to note the locality effect brought about using the $dack$ mechanism. In case of unavailability or failure of a subscriber (or any broker), the matching publications will queue up only on a chain of $\delta+1$ preceding nodes (on the path to the publisher) without affecting other nodes globally. Once failure conditions are resolved, the subscriber receives the outstanding publications reliably.

6. Implementation

In this section, we briefly elaborate on our implementation of the δ-fault-tolerant P/S system. Figure 7 illustrates the internal architecture of our brokers. Except for the *subscription manager*, our clients also share similar components. This design decision avoids reliance of the clients on their immediate neighboring broker and allows us to achieve true δ-fault-tolerance. The key components of a broker are as follows: The *message queue* stores incoming messages, provides functionality to detect duplicates, keeps track of the received/sent copies of the messages, and checks for whether messages can be confirmed; the *session manager* handles high-level broker-to-broker communica-

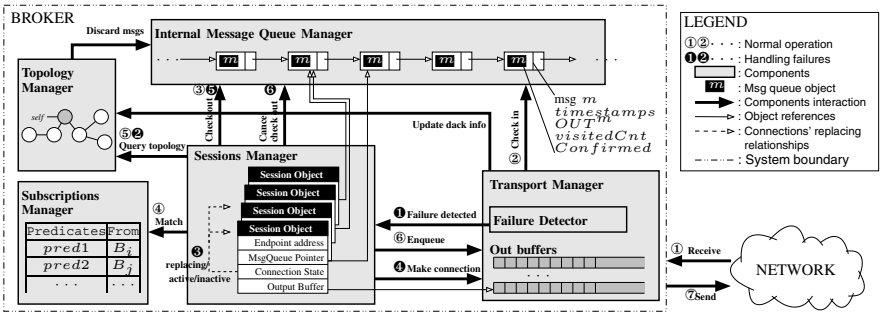


Figure 7. Internal architecture of a broker

tion, initiates or terminates sessions, keeps track of the state of remote brokers (i.e., joining, operational, or recovering), and consults with the topology manager in order to update the set of *active* sessions; the *topology manager* maintains the topology map and implements a topology cache that supports fast path look-up. In our implementation, the cached topology information proved to be an integral component that helped bridge the performance gap between different values of δ; the *subscription manager* stores the subscription routing table and includes a matching engine that given the content of a publication message returns its recipient set; the *transport manager* incorporates incoming and outgoing buffers and TCP connection handles to remote brokers and clients. It also provides means for failure detection via regular heart beat monitoring. Arrived messages are inserted into the FIFO message queue only if a prior copy of the message has not been received. If this duplicate detection test fails, a placeholder object is created and appended to the queue. The placeholder contains the original message object, the sender set, recipient set, outgoing set, as well as an integer counter ($visitedCounter$) to keep track of the number of sessions that have visited and processed the encapsulated message. A pointer to the placeholder is also inserted into a local hash table with the message's source sequence number[2] as the key in order to facilitate lookups. On the other hand, if the message is identified as a duplicate, the hash table is used to identify whether it has also been confirmed or not. If the old placeholder is still present, the sender of the new copy is added to the sender set, otherwise, a confirmation message is immediately issued.

The sessions that are designated as active (by the session manager) each possess a message queue handle that moves along the message queue and processes the messages. The processing involves incrementing the $visitedCounter$, and checking whether the path to the session's remote endpoint intersects the paths to any member of the recipient set. If so, a copy of the message is sent over the session and the session's identifier is added to the outgoing set for the message's placeholder. This process is referred to as a *checkout*. Each session has a limitted size outgoing transport buffer to temporarily hold checked-out messages until transmitted. Once full, the session stops checking out new messages. Alternatively, one could use output buffers as unlimited transmission queues which can hold any number of messages. This design can eliminate the need for a shared message queue (in our design) if arriving messages are processed and checked out immediately. However, this approach complicates dealing with failures. This is due to the fact that messages placed in different output queues do not necessarily preserve the initial arrival (FIFO) order. When a session's communication link becomes unavailable, its outstanding messages need to be merged with those on other session or split over new links that need to be created. In our design however, this is handled in a straightforward way by discarding outstanging messages in the output buffer of the failed link and having the newly created links to check out messages from the shared message queue.

When a node issues the confirmation of a message, it discards the original message placeholder from both the message queue and its hash table. Furthermore, it inserts a confirmation message into the message queue that includes the source sequence number of the original message, and has a recipient set equal to the senders set of the original message. A confirmation message is checked out and sent only over sessions whose endpoints appear in the recipient set. Once a confirmation message arrives at a broker, its sequence number is used to locate the placeholder of the original message. If the placeholder exists, the broker removes the session from the outgoing set and performs a test to identify whether to confirm the message or not. The test simply checks two conditions: the message's placeholder has an empty outgoing set and $visitedCounter$ is equal to the total number of active sessions. If both

2. Our sequence numbers are of the form: $\langle brokerId, epochNo, incNo \rangle$, where $brokerId$ is the broker identifier, $epochNo$ is the timestamp of broker's last start time, and $incNo$ is an incrementing integer counter.

conditions hold, the broker proceeds to issue a new confirmation message. Otherwise, the message remains enqueued to be checked out and confirmed.

Sessions may fail at any time and activation of new sessions may deactivate previously active ones. In this section, we only focus on the case of the failure of an active session. To handle a failure, the connection manager initiates new sessions to the neighbors of the failed broker. Activation of each session follows a handshake that identifies the last received message by the remote broker. This is used to initialize the session's handle in the message queue.

When we developed the $dack$ mechanism, we disabled normal confirmation of publication messages, and added timer tasks to periodically send $dack$ messages and purge the messages in the message queue. Brokers maintain an arrived/discarded sequence number for each remote broker in their topology map. When a new $dack$ message arrives, the sequence numbers of the associated brokers are updated only if they succeed the old values. The updated sequence numbers appear on the upcoming $dack$ messages that the broker issues. On the other hand, when the purge timer goes off, the broker first computes a *safe* sequence number for each broker in the topology map according to safety conditions (Section 5). Then, it iterates through the message queue and discards publications whose recipients ($Rcpt$) have a safe sequence number succeeding the sequence number assigned by the broker itself.

7. Experimental Evaluations

Our evaluations are carried out in a computing cluster composed of 21 machines connected via Gigabit Ethernet and with 4 GB of available memory. Each cluster node has four 32-bit, 1.8 GHz Intel Xeon cores. We exclusively assigned one core for brokers that we carried out measurements on. Remaining brokers were distributed uniformly among the remaining cores. For Sections 7.1 through 7.4, we use a network topology of 86 brokers configured with $\delta = 3$. The measurements correspond to three adjacent brokers B_1, B_2, and B_3 that form a chain and have an average fanout of 11. This setup corresponds to a network with a high fanout.

7.1. Publication Delivery Delay

We evaluate the impact of bypassing brokers on the publication delivery delay as perceived by the subscribing clients of the system. We used a publishing source to carry out the measurements. Figure 8 illustrates the results under three executions in which the publication messages bypass one, two, and three adjacent brokers. At time $50s$ failures take place. A higher spike correspond to publications that are delayed longer. The difference between the spikes closely matches the failure detector timeout (configured at $10s$). The spikes on

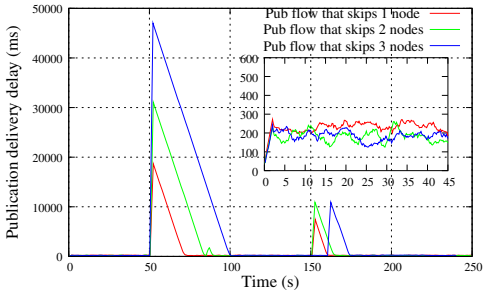
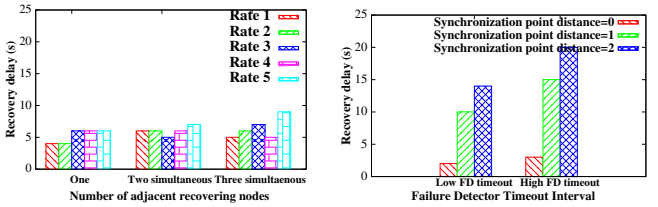


Figure 8. Publication delivery delay after one, two, and three simultaneous failures ($50s$) and recoveries ($150s$).



(a) Simultaneous recovery of adjacent brokers

(b) Impact of failure detector timeout and distance of synchronization point on recovery delay

Figure 9. Time to recover, $\delta = 3$.

the right correspond to simultaneous brokers recovery at $150s$. Furthermore, it is clear that simultaneous recovery of multiple brokers has an almost constant impact on publications delay.

7.2. Time to Recover

Potential factors that play a role in brokers recovery delay are the amount of time to transfer subscription routing entries, rate of publication traffic, failure detector timeout, availability of neighbors, and their state. We carried out measurements under various conditions, in which brokers (individually or concurrently) start the recovery procedure. We considered subscription state transfer of all 2600 subscriptions in the system in order to single out the impact of other factors.

Figure 9(a) illustrates the time to recover for multiple adjacent recovering brokers under 5 different publication traffic rates. We considered publication rates that do not overwhelm the brokers. It is interesting to see that the time to recover in all cases are almost the same. This implies that if recovering brokers are able to connect to their neighbors (even those that are recovering) they can quickly retrieve the recovery data.

On the other hand, if the recovering broker has to retry several brokers until it discovers a non-faulty synchronization point, the recovery is delayed. This extra time closely corresponds to the failure detector timeout value. Under low and high timeout values, Figure 9(b) compares the recovery time of a broker that has to connect to only its immediate neighbors (left box), brokers one hop away (middle box), and brokers two hops away (right box).

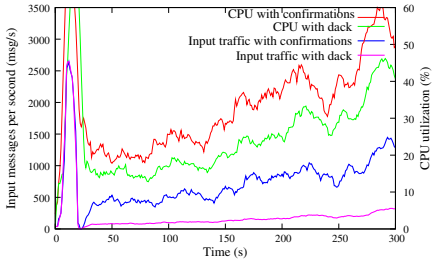


Figure 10. $dack$ mechanism improves message traffic and lowers CPU utilization (initial spikes correspond to subscription and join traffic)

7.3. Impact of Using $dack$ Mechanism

Substituting confirmation messages with the $dack$ mechanism is an important optimization technique that reduces the message traffic associated with publication flows. In this experiment, we measure a broker's input message traffic and CPU utilization, both when confirmation messages are used and when the $dack$ mechanism is enabled. Figure 10 illustrates the results in an execution where we continuously increase the publication rate. It is clear that the use of $dack$ messages greatly improves both the network traffic and CPU utilization under various load conditions.

7.4. Load on Brokers

This experiment investigates the impact of bypassing unavailable neighbors on a broker's load. We measured the input and output message rates, and the CPU utilization in presence of 0, 1, or 2 failures. Figure 11 illustrates the results. At times $150s$ and $275s$ brokers B_1 and B_2 fail, respectively. In the time interval $150-275$, broker B_2 reconnects the network partitions and bypasses its failed neighbor B_1. This is reflected in the graphs by a short drop, followed by sharp CPU and traffic spikes for broker B_2. B_3 also experienced a less tangible but similar trend. At time $275s$ broker B_2 fails too and broker B_3 initiates to bypass both B_1 and B_2. After the initial spikes that correspond to the enqueued volume of publications, B_3's output publication traffic stabilizes at about twice the amount before failures. Its input message traffic however, remains constant. Finally, its CPU load shows an increase of about 10%.

7.5. Comparison to Primary/backup Approach

We now use the network of Figure 12(left) to compare our approach ($\delta = 2$) against the primary/backup approach [8]. There are 21 brokers and 600 subscriptions are inserted through brokers designated with 'Sub'. Publishers connect to brokers designated with 'Pub' and publish at varying rates. In the primary/backup approach, R_1, R_2 and R_3 belong to the same virtual broker, and following the failure of R_2 and R_3, broker R_1 has to take over all the load from the other two (Figure 12(right)). In our approach however (Figure 12(middle)) R_1 will be

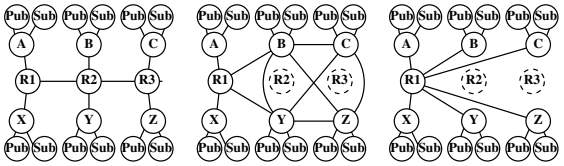


Figure 12. Left: Network used for experiment. Centre: Network configuration in our approach after failures. Right: Network configuration in primary/backup approach after failures (R_1 is backup for R_2, R_3)

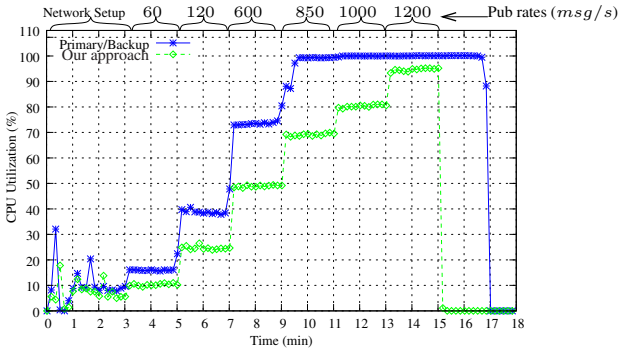


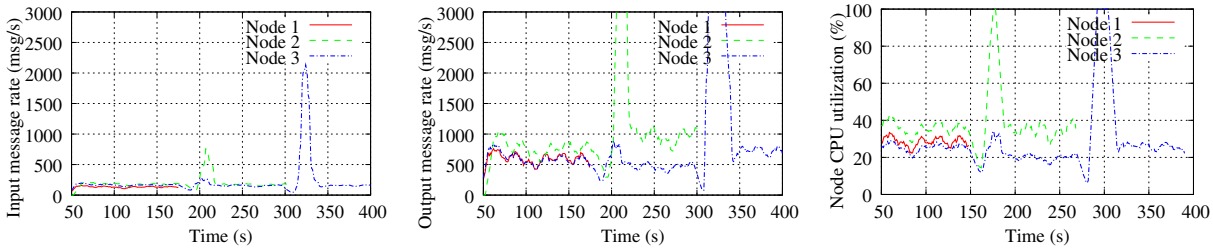
Figure 13. CPU utilization on R_1 after failures

a broker located adjacent to the failed nodes. But since there are other nearby brokers that can provide forwarding routes between the upper and lower parts of the network, R_1 experiences a lower load compared to the primary/backup approach. Figure 13 illustrates that after failures, R_1's CPU utilization is about 30% lower in our approach. The reasons for this is twofold: first after failure our network is more connected and the traffic is more effectively distributed. Also, in our approach a broker receives a publication only if there is a subscriber in its lower subtree. As a result, failure of a nearby broker does not affect its incoming traffic.

8. Related Work

Related work falls into the following two categories, reliable group multicast, and reliable P/S systems. We discuss them in turn. The problem of reliable publication delivery is related to the reliable multicast problems. However, in P/S systems with a high publication rate, it is costly to ensure properties such as virtual synchrony [9], or total ordering of publication delivery. This is mainly due to the fact that each publication is delivered based on individual subscribers' interests, and maintenance of a shared *group view* for this level of dynamism among a large number of clients is infeasible. Thus, we believe that per-source in-order delivery, as required by our reliability specification, provides a reasonable balance between application requirements and scalability of the implemented system.

Resilient Overlay Network (RON) [10] is an architecture that aims to improve the resiliency of distributed network applications against Internet path outages. A



(a) Brokers' input message rate (b) Brokers' output message rate (c) Brokers' CPU utilization

Figure 11. Load on brokers after and before failure of their neighbors ($\delta = 3$).

RON network consists of nodes at disparate locations throughout the Internet which continuously monitor each other, and forward messages via alternative routes if some paths become unavailable. This problem is relevant to ours since maintaining reachability is a key part of our approach. However, we address the problem within the context of content-based P/S systems which is completely different from IP routing.

We now review some of the most important related work in the P/S literature. The Gryphon distributed P/S system introduces the concept of virtual brokers to tolerate failures [8]. Each virtual broker is a set of physical machines which act as identical replicas. We compared with their approach and showed that upon failures, our system achieves better load stability. Additionally, our topology can be deployed transparently without the need for replica assignment.

Snoeren et al. [5] propose an approach to build a fault tolerant P/S system, by constructing redundant disjoint forwarding paths between subscribers and publishers. Publications are forwarded concurrently on all redundant paths. Since failure on all paths is unlikely, chances of message loss is reduced. This approach has the advantage of lowering the delivery delay (to the delivery delay of the first arrived publication) but incurs high bandwidth even in absence of failures. Furthermore, reliable delivery is not guaranteed.

In Hermes [11] the routing information at P/S brokers is *soft* state. This is designed to cope with failures by having clients periodically renew subscriptions. In general, this approach does not necessarily prevent publications loss. XNET [12] proposes two schemes, *crash/recover* and *crash/failover*, to deal with broker failures. However, these approaches are unable to handle multiple simultaneous failures. Cugola et al. [6] also present an algorithm to improve the efficiency of reconfiguration in highly dynamic P/S networks. The proposed *reconfiguration path* approach minimizes publication loss during a reconfiguration process, but does not completely prevent it. Epidemic (gossip) algorithms have been applied to P/S systems, to improve the availability of highly dynamic systems [13]. However, these approaches do not provide strict publication delivery guarantees. In contrast to both approaches, our proposal is suitable for more stable environments and achieves ordered delivery guarantees. In an earlier work, we also investigated use of cyclic overlays in enhacing system's fault-resiliency [14]. The approach presented in this paper however, guarantees reliability at all times by ensuring exactly-once delivery of publications.

9. Conclusions

In this paper, we proposed a reliable distributed publish/subscribe approach that promises availability of service in the face of up to δ concurrent broker failures. Our approach is suitable for large and reasonably stable environments such as that of an enterprise or a data center [1], where reliable and timely publication delivery is desired in spite of failures.

References

[1] J. Reumann, "The design of a reliable message distribution service for Google," in *ACM Middleware*, 2007.

[2] G. Li *et al.*, "Decentralized execution of event-driven scientific workflows," in *SCW*, 2006, pp. 73–82.

[3] A. Carzaniga *et al.*, "Design and evaluation of a wide-area event notification service," *ACM TOCS*, 2001.

[4] G. Cugola *et al.*, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *IEEE TSE*, 2001.

[5] A. C. Snoeren *et al.*, "Mesh-based content routing using XML," in *SOSP*, 2001.

[6] G. Cugola *et al.*, "Minimizing the reconfiguration overhead in content-based publish-subscribe," in *SAC 2004*.

[7] R. Sherafat and H.-A. Jacobsen, "δ-fault-tolerant publish/subscribe systems," 2007, Technical report, CSRG-570, University of Toronto.

[8] S. Bhola *et al.*, "Exactly-once delivery in a content-based publish-subscribe system," in *DSN*, 2002.

[9] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM TOCS*, 1987.

[10] D. G. Andersen *et al.*, "Resilient overlay networks," in *SOSP*, 2001.

[11] P. R. Pietzuch and J. Bacon, "Hermes: A distributed event-based middleware architecture," in *DEBS*, 2002.

[12] R. Chand and P. Felber, "XNET: A reliable content-based publish/subscribe system," in *SRDS 2004*.

[13] P. Costa *et al.*, "Introducing reliability in content-based publish-subscribe through epidemic algorithms," in *DEBS*, 2003.

[14] G. Li *et al.*, "Adaptive content-based routing in general overlay topologies," in *ACM Middleware*, 2008.