Highly Available Distributed Publish/Subscribe Systems[1]

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

Abstract

In this paper, we design large reliable, distributed publish/subscribe (P/S) systems that are capable of coping with concurrent failure of multiple nodes and links. Our fault-tolerant algorithms exploit multi-path forwarding to ensure both liveness and safety in the face of up to δ failures (δ-fault-tolerant). Furthermore, in the case the number of failures exceed δ, the algorithms guarantee safety with a chance that liveness may be compromised. We provide in-order exactly-once publication delivery semantics as part of our reliable P/S service. It is challenging to ensure the latter since upon detection of a failure our algorithms may trigger retransmission of messages over a different path. We define and enforce a legitimate message propagation *property in a way that honors reliability and δ-fault-tolerance requirements. Based on this property, we propose a novel scheme that allows nodes to detect duplicate arrival of messages only by keeping track of the sequence numbers assigned by their nearby peers (within distance $2\delta + 1$). We prove the safety properties of our approach.*

Keywords: Publish/subscribe, reliability, Fault-tolerance, distributed systems, content-based routing

Categories: Distributed systems; Fault tolerant systems; Middleware; Reliability, availabil-

[1]A modified version of this paper has been submitted to Dependable Systems and Networks, 2009.

ity and safety; Networking and networked systems

1 Introduction

Publish/subscribe (P/S) is a message-passing communication paradigm that has gained much attention from both industry and academia in recent years. It provides an abstract and high-level interface for data producers (publishers) to publish messages and consumers (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 interconnected network of routers (brokers) that collectively perform the task of delivering publications to subscribers.

The flexibility and decoupling provided in this model is a major driving force for its adoption as the main communication substrate in an increasing number of enterprise-level applications [6, 1, 7, 2]. There are several standards-based industrial messaging platforms [3, 4, 5] supporting various flavors of publish/subscribe (e.g., *content-based* and *topic-based*). There are also several proprietary P/S solutions that are customized for particular application areas. For instance, Tibco Rendezvous [2] disseminates highly delay-sensitive financial market, and Google uses a proprietary P/S platform to disseminate advertisements to massive server farms [1]. Emerging new directions in research are also investigating P/S-based coordination and execution of business processes [15].

The reliability requirements of these new application areas are extensive and beyond the best-effort quality of service (QoS) provided by mainstream P/S systems. In our previous work [14, 17] we proposed *reliable* and fault-tolerant distributed P/S algorithms that tolerate concurrent crash failure of up to δ brokers (δ-*fault-tolerant*). Reliability in this context refers to per-source in-order and exactly-once delivery of publications to all matching subscribers, and δ denotes a system configuration parameter determined by system administrator.

In our approach, network nodes (i.e., brokers, subscriber, and publishers) are organized in a tree-based topology, and each maintain a *topology map* data structure. A node's topology map captures the layout of other peers in the topology within a certain radius ($\delta + 1$ hops away). Brokers also

maintain a *subscription routing table* data structure that stores *subscription tuples* each containing a client's subscription predicate, as well as partial information about the path towards to the issuing subscriber. Nodes use subscription predicates to *filter* unwanted publications and forward only those that *match* the clients' interests. In the face of node failure, the combination of the information in the topology map with the subscription tuple's path information help to reconnect the network, and prevent interruptions to publication flows. Our tree-based topology enables flexible content-based message routing which is otherwise hard to achieve (e.g., using Distributed Hash Tables (DHT) [19]). Moreover, since typical P/S infrastructures maintain a large volume of routing state across network brokers, organization of subscription routing tables and the topology maps in our approach allows for quick reaction to failures without the need to re-propagate subscriptions.

In this paper, we extend our δ-fault-tolerant system w.r.t. three important features: *(i)* partition-tolerant subscription propagation; *(ii)* node recovery; and *(iii)* improvements to message traffic. The *partition-tolerant subscription propagation* algorithm (Section 3) delivers clients' subscriptions to brokers and updates subscription routing tables. We properly address scenarios in which due to link failures or node crashes parts of the network become unreachable and do not receive some subscriptions. Based on the fact each node's topology map is confined to a limited radius, we characterize two cases: *small* and *large fragments*. In the former, the information in the topology map is sufficient to reconnect the overlay and deliver subscriptions to the rest of the network's nodes. Parts of the network that are unreachable will nonetheless remain unaware of the subscription and we ensure that these nodes do not violate reliable publication delivery by mistakenly dropping matching publications. Moreover, once network unreachability conditions are resolved we deliver the missed subscription tuples as part of the recovery algorithms (Section 4).

In the latter case however, the information stored in the topology map is not sufficient to reconnect the overlay. As a result large network subtrees may not receive some subscriptions. We properly identify these unreachable parts via a *partition information* data structure, $parInfo$,

and convey this information to the subscriber. Furthermore, once communication is resumed we transfer all *partially* propagated subscriptions. Compared to small fragments, large fragments take longer to recover since subscriptions propagate more hops. Before recovery concludes, we tag all publications that are received from the previously unreachable nodes with $parInfo$. This allows the subscriber to identify messages that may have been potentially dropped before the recovery procedure concludes.[1]

For the purpose of fault-tolerance, forwarding nodes maintain a copy of the publication until it is delivered to all downstream subscribers. In our original algorithm [14] this was carried out via propagating *confirmation messages* for individual publications in order to acknowledge their delivery. In this paper, we devise a novel approach (Section 8) based on aggregated acknowledgments to avoid the extra overhead of confirmation messages. We address the challenges associated with content-based P/S systems, in which the destination of publications are not known a priori and are determined by matching. Each publication can be delivered to a different subset of subscribers. This makes the nature of the problem different from that in the reliable multicast literature.

The rest of the paper is organized as follows: Section 2 gives an overview of P/S algorithms and our previous work. We present the partition-tolerant subscription propagation algorithm in Section 3, and the recovery procedure in Section 4. Section 8 provides details of our approach to use acknowledgment messages. Implementation details and experimental evaluations are presented in Sections 9 and 10. We review the related work in Section 11 and conclude the paper in Section 12.

[1] In a separate research, we are investigating historic data access in distributed P/S systems. We intend to use this mechanism to retrieve publications issued before or during recovery.

2 Background

2.1 Typical P/S Systems

Distributed P/S systems are composed of an interconnected overlay of brokers to which clients (subscribers and publishers) connect. Typically, nodes are aware of their immediate neighbors only. Subscribers issue subscription messages that are propagated throughout the network. Brokers store the subscription as well as the neighbor's identifier (address) *from* which the subscription was received. When a publication arrives, subscriptions are evaluated against the content of the publication in order to determine a matching subset. The publication is forwarded to those neighbors that previously forwarded the corresponding subscription messages. This way the publication arrives at subscribers by traversing subscriptions' propagation paths in reverse.

2.2 Reliable δ-Fault-Tolerant Algorithm

We proposed a reliable δ-fault-tolerant P/S forwarding algorithm [14] that can be seen as a general extension of typical P/S systems. In our system, nodes are organized in a tree-based topology that is used to disseminate publications. Each node maintains a *topology map* that captures a *partial* layout of the topology constrained to the nodes within distance $\delta + 1$. This information is updated as nodes join the system such that the type (broker, subscriber, or publisher), address, and the *topology path* to nodes in the topology map are known (details can be found in [14, 17]). Brokers also maintain subscription routing tables that contain entries corresponding to subscriptions inserted into the system (subscription propagation presented in Section 3). Each subscription entry is in form of a tuple, $\langle pred, from, seqVect \rangle$, where $pred$ is the predicate filter specified by the subscribing client and specifies the type of messages it is interested to receive; $seqVect$ is a vector of sequence numbers each generated by a node on the propagation path of the message; $from$ is the identifier of another node determined as follows:

- If the subscriber is at most $\delta + 1$ hops away, $from$ points directly to the subscriber;

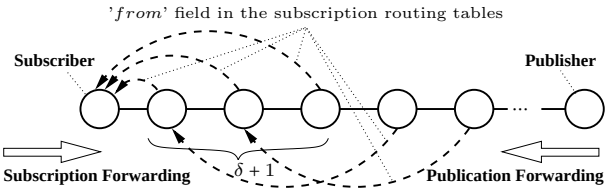


Figure 1. The $from$ field of entries in the subscription routing tables points back to a node on the connecting path to subscriber.

- Otherwise, $from$ points to a node on the connecting path to the subscriber that is $\delta + 1$ hops closer to the subscriber.

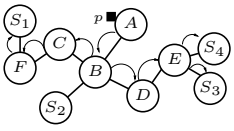
Figure 1 illustrates the above invariants for subscription routing entries. It is worthy to note that $from$ always points to a node located in the topology map.

For each publication message, p, a forwarding node generates a unique increasing sequence number upon initial receipt of p. This sequence number is added to the $seqVect$ of the publication message when it is being 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 subscription tuples whose predicate field *match* and are evaluated to *true* by 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 local data structure called the *recipient set*, $RCPT_B^p$:

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

Using its local topology map, B then generates a set of associated topology paths for each node in $RCPT_B^p$. This set is referred to as *outgoing paths*, $outPath_B^p$.

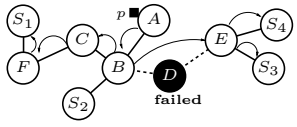
Under non-faulty conditions B is connected to its *immediate* neighbors in the topology, and forwards p to these nodes (Figure 2(a)). This effectively enables the node to deliver p to all downstream subscribers by sending only a few copies to its neighbors (multicast effect). Once an



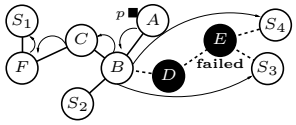
(a) No failures

Src	$SubscriptionPredicate$
✓ $\mathbf{S_1}$	$[temp, >, 10], [city, =, *]$
✗ S_2	$[temp, >, 0], [city, =, ' LA']$
✓ $\mathbf{S_3}$	$[city, =, ' NYC']$
✓ $\mathbf{S_4}$	$[temp, <, 25]$

(b) Subscriptions matching publication $p : [temp, 20], [city, ' NYC']$



(c) One failure



(d) Failure of any two nodes

Figure 2. A sample network that can tolerate two failures ($\delta = 2$). Publication p matches subscriptions $\{S_1, S_3, S_4\}$; nodes A and B compute $RCPT_A^p = \{F, E\}$, $RCPT_B^p = \{S_1, S_2, S_3\}$. Arrows between nodes indicate connections over which p is sent.

immediate neighbor becomes unavailable B uses its topology map to connect to the next nodes one hop away (from the failed peer). If any of the new connections also fail, B proceeds with the next neighbors. Roughly speaking if the number of failed attempts does not exceed δ, B will be able to maintain the network's connectivity and use its connections to forward messages. This is illustrated in Figures 2(c) and 2(d) in which B uses its new connections and the knowledge of the outgoing paths, $outPath_B^p$, to forward p bypassing failure of other nodes.

Once p arrives at a matching subscriber, it issues a confirmation message, $conf^p$, to the sender of the original message. Every forwarding node collects all confirmation messages, discards its local copy of p, and issues a new $conf^p$ message upstream.

3 Partition-Tolerant Subscription Propagation

3.1 Overview

The subscription propagation protocol serves to deliver clients' subscriptions to network nodes and establish subscription routing entries. The subscriber generates a subscription message, s,

containing its subscription predicate, $pred$, and a vector of sequence numbers, $seqVect$ of size $3\delta + 1$. Each element in this vector is of the form $N_i : seq^s_{N_i}$, where N_i is a forwarding node along the propagation path of s and $seq^s_{N_i}$ is the sequence number generated at node N_i upon arrival of s *for the first time*. A subscriber, initially fills $s.seqVect$ with its own sequence numbers, and this vector is updated properly at other forwarding nodes to reflect the sequence numbers assigned by the last $3\delta + 1$ nodes on the propagation paths. If N_i sends a copy of s bypassing (skipping) some nodes, $N_{i+1}, \cdots, N_{i+k}$, the corresponding positions in $s.seqVect$ are filled with $N_{i+1} : \perp, \cdots, N_{i+k} : \perp$, where $\perp$ designates null.

The subscription propagation algorithm is in its core similar to the publication forwarding algorithm (Section 2) with the following exceptions:

- Recipient set: As opposed to publications which are sent on selected paths (towards subscribers) based on the result of matching, subscriptions are intended for all brokers in the network. This is achieved by inclusion of all known downstream brokers (within distance $\delta + 1$) in the recipient set;
- Subscription table updates: Once all confirmation messages are received a new *subscription tuple*, $t = \langle pred, from, seqVect \rangle$, is added to the subscription routing table. $pred$ and $seqVect$ correspond to same fields in the subscription message, and $from$ is the $(i+1)^{th}$ node id on $seqVect$.

As described above, a subscription is intended for all brokers in the network. A broker crash or unreachability may lead the entire subscription propagation process to a deadlock since its corresponding confirmation is never received. In the remainder of this section, we introduce the notions of *small* and *large fragments* in order to precisely characterize these unreachability conditions, and then propose our approach to deal with them in a deadlock free manner.

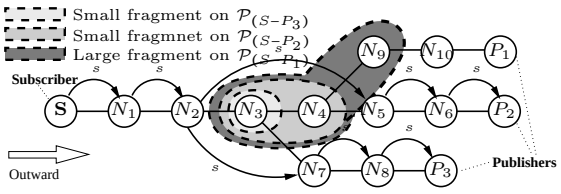


Figure 3. Intersecting small and large fragments on multiple subscription propagation paths ($\delta = 2$).

3.2 Active Connections and Fragments

Let a *sub-pub-path* between S and P, $\mathcal{P}_{(S \to P)} = \langle N_0, N_1, \cdots, N_i, \cdots, N_l \rangle$, be the topology path from subscriber, $S = N_0$, to publisher, $P = N_l$, and consider the *outward* (*inward*) direction on this path to be from S to P (P to S). At any point, each node N_i maintains a number of connections (possibly zero) to some other nodes on this path. Let $outward_{N_i}$ be the set of these connection endpoints to nodes located *within distance* δ outward from N_i on $\mathcal{P}_{(S \to P)}$. If $outward_{N_i}$ is non-empty we refer to the connection to the closest node in this set as the *active* connection. When a connection becomes active at N_i, we say that it has been *activated*. An active connection to N_j may *bypass* the sequence of intermediate nodes $\langle\langle N_{i+1}, \cdots, N_{j-1} \rangle\rangle^S$ which we refer to as a *small fragment* (on $\mathcal{P}_{(S \to P)}$). On the other hand, if $outward_{N_i} = \varnothing$ we say that N_i has no active connection on $\mathcal{P}_{(S \to P)}$, and refer to the sequence of nodes $\langle\langle N_{i+1}, \cdots, N_{i+k} \rangle\rangle^L$ as a *large fragment* (on $\mathcal{P}_{(S \to P)}$) where $k = min(\delta + 1, l - i)$. In either case, we name N_i as the *lead node* of the fragment. Figure 3 illustrates formation of possibly intersecting fragments.

3.3 Failure Detection

Our protocol manages the nodes in the *outward* set by using the output of a local failure detector which signals unavailability of a node.[2] Initially, nodes have active connections to their immediate outward neighbor (on any pub-sub-path), i.e., $N_{i+1} \in outward_{N_i}$. This node is monitored and if becomes unavailable, the protocol proceeds to connect to the next outward node, i.e., $outward_{N_i} \leftarrow outward_{N_i} \cup N_{i+2} - N_{i+1}$. This creates a small fragment of size one. If more outward nodes become

[2]The failure detector in our implementation uses heartbeats which may mistakenly identify a non-faulty node as failed. Our approach is resilient to such mistakes.

unavailable, the above process repeats (a maximum of $\delta + 1$ times) thus expanding the small fragment. Eventually, either an available node, N_{i+k}, is found, or a large fragment is formed ($outward_{N_i} = \varnothing$).

3.4 Legitimate Propagation Paths

Consider the portion of $seqVect$ that corresponds to the last $2\delta+1$ nodes along the propagation paths of a message, m, we say the propagation path of m is *legitimate*, *iff*, the number of $\perp$ values in this portion does not exceed δ. Roughly speaking legitimacy ensures that m has been forwarded by a majority of nodes on their propagation path.

To check for legitimacy, nodes simply count the number of $\perp$ values in messages' $seqVect$. Illegitimate propagations may only occur if the network suffers from extensive (at least $\delta+1$) link or node failure that are also relatively close (on a chain of $2\delta + 1$). Depending on the value of δ this is likely to be rare, and our handling of these situations may only compromise liveness, but not safety. Due to space limitation, we do not elaborate further on how to deal with illegitimate propagations here. In general, this involves retransmission of messages once previously bypassed nodes become available. We elaborate more on this in Section 5.5.

3.5 Subscription Forwarding Algorithm

We now present the details of how subscription s issued by subscriber S is forwarded by a node, N_i, on its propagation paths. N_i computes $outPath^s_{N_i}$ to be the union of all paths to downstream publishers located in the topology map, as well as paths to all downstream nodes at (exactly) distance $\delta + 1$. Each $\mathcal{P} \in outPaths^s_{N_i}$ fully intersects with possibly many sub-pub-paths between S and publishers P_i downstream from N_i. For all paths, $\mathcal{P}$ there are two possibilities:

Case 1: If N_i has an active connection, it sends a copy of s outward and N_i waits to receive the corresponding confirmation message, $conf^s$. Once all confirmations are received, N_i takes the following confirmation steps: *(STEP-I)* issues its own $conf^s$ to the sender(s) of s; *(STEP-II)* add the corresponding subscription tuple to its own subscription routing tables; and *(STEP-III)* if

it has a connection to an inward node different from the sender(s), the tuple is also sent to the closest such node. If an active connection bypasses nodes on a small fragment, it may indeed be the case that some fragment nodes remain unaware of the subscription message. We deal with such cases during recovery (Section 4).

Case 2: If N_i does not have an active connection for path, $\mathcal{P}$, a large fragment is formed. N_i issues a *partial confirmation* message, $\hat{conf}^s$, after receiving confirmations from all other paths $\mathcal{P}' \in outPaths^s_{N_i}$. A partial confirmation message contains a set of partition information tuples, $\{parInfo\}$, each conveying information about one large fragment that blocked complete propagation of the subscription. A $parInfo$ tuple is of the form: $\langle N_i, seq_{N_i}, lFrag \rangle$, where N_i is the lead node, $lFrag$ is the sequence of nodes on the large fragment, and seq_{N_i} is a sequence number generated by N_i. These tuples are also stored locally in a set called *partition information table.*

Now consider the propagation of $\hat{conf}^s$ towards the subscriber, S. Nodes between S and N_i that receive a partial confirmation must also issue a partial confirmation that carries the partition tuples that were received plus any additional tuples corresponding to new large fragments that are formed. Finally, nodes (including S) add the partition tuples to their local partition information table if lead node is closer than $2\delta + 1$.

Publication Tagging

Before a large fragment recovers (Section 4) it is possible that some fragment nodes create an active connection to a node inward from the fragment's lead node and attempt to forward publications. Any such publication, p, may match $s.pred$ and yet come from portion of the network that is still unaware of s. In particular, presence of a fully propagated subscription, s', that matches p may lead to such scenarios. It is thus unsafe to pass these publications normally, since there might be a following publication, p', that matches s but is discarded as it does not match s' (or any other fully propagated subscription).

To deal with this scenario, we tag such publications with $parInfo$. This takes place at any of the nodes located within distance $2\delta + 1$ of the fragment's lead node that have this information in its

partition information table. Subscribers with partially confirmed subscriptions that receive these publications will then be able to compare this tag with the partition information they received with $\hat{conf}^s$ and identify the publication as one that has been delivered due to overlapping effect of other subscriptions.

4 Recovery

Formation of small and large fragments as a result of nodes crash or links failure may lead to situations where some subscriptions are not delivered to parts of the network. Nodes execute the *recovery* protocol described in this section, in order to *(i)* reconnect to fragments, *(ii)* restore missing partition information and subscription tuples; and *(iii)* (re-)send messages through *recovered* fragment.

4.1 Reconnecting to Fragment Nodes

The lead node of a fragment periodically checks availability of the fragment nodes. If a new connection is successfully established it becomes active. Likewise, if any node on the fragment initiates a connection to the lead node, it immediately becomes active.

4.2 Recovery Data Transfer

Once a new connection is activated (becomes active) a hand shake is performed and nodes exchange portions of their partition information and subscription routing tables. As illustrated in Figure 4, either side of the connection may lack information about subscriptions that came from subscribers in their own or the other side's subtrees. The exchanged partition information tuples include those that correspond to lead nodes within distance $2\delta + 1$ of the receiving peer. The receiver uses the sequence number of each $parInfo$ entry to determine if it had prior knowledge of the tuple. New tuples are added to its own partition information table and also sent to other nodes to which an active connection is present.

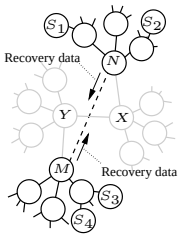


Figure 4. Recovery over active connections (dashed lines) in a network where $\delta = 2$.

4.2.1 Small Fragments

Let, N be the activating side of the connection and M be the other endpoint[3], and for now assume that N has no knowledge of a large fragment that includes M. N notifies M about the new state of their connection (i.e., activated) and requests its last sequence number, $lSeq_N$, that M has received in a message. M looks through its own subscription tuples to identify $lSeq_N$ present in $seqVect$ of a subscription tuple, and replies accordingly. N, uses $lSeq_N$ to compute a subset of its subscription tables that M *may* be missing. This subset, $bypassedSubSet^M$, contains all subscription tuples at N that were assigned a sequence number that succeeds $lSeq_N$. This set is sent to M in order to be merged with its subscription routing tables.

Some tuples in $bypassedSubSet^M$ may be already present in M's subscription routing tables. These entries might have been added as a result of a propagation path of the corresponding subscription that did not pass through N. Thus, it is necessary for M to properly check the received tuples in order to remove duplicates. For this purpose, it maintains a set, $\{X : lSeq_X\}$, of the last sequence number received from nodes within distance $3\delta + 1$. A subscription tuple, $t \in bypassedSubSet^N$ (similarly for $bypassedSubSet^M$) is duplicate if it has *any* element, $X : seq_X$, that does not succeed $lSeq_X$. After removing duplicate entries, the tuples are added to the local subscription tables. The $from$ field of the added tuples are computed appropriately using the tuples' $seqVect$ such that it points to a node δ+1 hop closer to the subscriber. Finally, the resulting tuples set is sent to any other node, to which N has an active connection. The receiving node

[3]A connection can be active only at one endpoint.

processes the tuples[4] in a similar manner, i.e., duplicate elimination, subscription table update, sending out on active connections.

4.2.2 Large Fragments

Now consider the case in which N has a set of parition tuples $\{parInfo\}$ containing large fragments, $lFrag = \langle\langle \cdots, M, \cdots \rangle\rangle^L$ (M is on the large fragment). N may have a set of partially confirmed subscriptions, $parConfSubs^{parInfo}$, that did not propagate fully outward on $lFrag$. These subscriptions need to be forwarded on the reminder of their propagation paths. This is done by sending $\{parInfo\}$ along with $\{parConfSubs^{parInfo}\}$, to M. M further forwards these sets over its active connections on the corresponding paths and updates its own partition information table accordingly. The entire subscription set is processed at each node as a whole, and the confirmations are issued likewise. For simplicity, here we consider that there is no other large fragments affecting the propagation of this information. Thus, once the confirmation messages arrive, M updates its subscription routing table, and replaces each partition tuple, $parInfo$ with $\overline{parInfo}$, indicating recovery of the large fragment. This is reflected in the partition information tuple and reported to N, which updates its own partition information table accordingly by removing $parInfo$. This change is also reported over other active connections to nodes within distance $2\delta + 1$ of $lFrag$'s lead node. The partition table updates also imply that no other subsequent publication will be tagged with $parInfo$.

4.3 Message Forwarding

Once recovery data transfer completes (for the case of large fragments, after $parConfSubs^{parInfo}$ is sent), the activating side, N, resends all non-confirmed messages (subscription or publication) via the active connection if their propagation paths passes through M (M is on a path in the message's $outPaths$). These messages are further forwarded down the network on paths that previous copies of the messages may have already propagated through. However, since there is

[4] Also as part of subscription propagation (Section 3).

a chance that old propagations have become illegitimate at some point the retransmitted copies help establish the legitimacy requirement. In Section 5.5 we elaborate on how retransmissions transforms illegitimate propagations into legitimate ones. The correctness proves for small and large fragments are provided in Sections 6 and 7.

5 Legitimate Propagation Paths

Our reliable fault-tolerant algorithms implement a distributed P/S system that provides exactly-once delivery guarantee. In our approach, messages (i.e., publications or subscriptions) produced at any source (i.e., publisher or subscriber) may be forwarded along diverse paths towards their final recipients (i.e., matching subscribers for publication messages, or any broker in the system for subscription messages). Failures and recoveries trigger retransmission of messages that are not yet confirmed. This may result in situations in which copies of a single message arrive at a node along different the propagation paths multiple times. We need to ensure that under such circumstances, the nodes are able to *(i)* detect duplicate arrival of multiple copies of the message; *(ii)* decide the faith of the duplicate copies.

In this section we first briefly review two simple duplicate detection algorithms and discuss their drawbacks. We then define the legitimate message propagation property, elaborate on how it facilitates the task of duplicate detection. Furthermore, we provide a method to check for legitimacy, and elaborate on an algorithm to enforce this property. Our algorithm is based on message retransmission and ensures safety at all times. Finally, we describe the likelihood of failure patterns that violate the legitimate message propagation property.

5.1 Simple Approaches for Duplicate Detection

Retransmission may result is a situation in which multiple copies of a message, say m, arrive at a node, say N. Based on their arrival order at N, we use superscripts $m^0, \cdots, m^i$ to identify these copies with an *arrival index*. The first arrived copy of m has an arrival index of 0 (i.e., m^0) and is referred to as the *initial copy*. Later copies of m are considered as *duplicate copies* (at node N

which is implicit in the context) and have arrival indexes greater than 0. Furthermore, we refer to the arrival index, ci, of the last index before confirming the message as the *confirmation index*. Finally, we use the notion of *post confirmation duplicate message*, $m^{>ci}$, to refer to a duplicate m^i where $i > ci$ and a *pre-confirmation duplicate message*, $m^{\leq ci}$, to refer to a duplicate m^i where $0 < i \leq ci$.

We now describe two simple approaches based on unique message identifiers, and monotonically growing sequence numbers to detect duplicate messages. We also provide some of their associated disadvantages.

i – Unique message identifiers (id): if messages have unique identifiers a receiving node can detect duplicate arrival of multiple copies of the message by comparing their the id of the copies. Based on this algorithm:

$$\forall i \neq 0, m^i \text{ is a duplicate copy of } m^0 \Leftrightarrow m^0.id = m^i.id$$

This approach has the disadvantage of requiring nodes to keep track of all message ids they have received thus far. If nodes retain the message identifiers forever, then the system is not stable as the memory required grows unboundedly with arrival of new messages. One solution can be to use a timeout timer to purge old message ids. However, the exact timeout value requires assumptions on the upperbound message retransmission delays. This is in particular tricky, since this delay is dependent not only on link delays but also the downtime period of links and nodes. The downtime period is usually arbitrary and can hardly be bounded.

ii – Monotonically increasing sequence numbers: if links are FIFO and nodes process messages in the same arrival order (from each link) then we can use monotonically increasing sequence numbers to detect duplicate arrival of the message. For this purpose, every message, say m, carries the source node's id, say S, and a source-assigned, monotonically increasing sequence number. Upon arrival of m^0 at any other network node, M, the value of the highest sequence

number received from S, $highSeq(S)$, is updated with the $m.seq$. All future copies, $m^i (i \neq 0)$ will be detected as duplicates if their sequence number does not succeed that of $highSeq(S)$. In other words:

$$m \text{ is an initial copy } \Leftrightarrow (highSeq(m.source) \text{ is not defined }) \vee (m.seq \succ highSeq(m.source))$$

This approach works, since FIFO links and in-order processing of messages at all network nodes between S and M does not change the order of consecutive messages. Thus, M never receives m^0 before any other message produced later from the same sources.

This approach has the disadvantage of requiring nodes to keep track of the highest sequence numbers from all message sources, i.e., publishers, and subscribers. In a large long-running system in which message producers come and go, this implies a potentially large set of information needs to be kept up to date. Furthermore, this burden has to be taken even when there are no failures in the system.

We now present a new approach with low overhead that also blends well with our forwarding algorithms 2. It is based on a combination of both aforementioned duplicate detection schemes without the disadvantages of either one: message identifiers are not retained forever (also there is no timeout value) and nodes require no global knowledge of all producers.

5.2 Legitimate Message Propagation

In this section, we first define the *legitimate propagations* of a message, and then present algorithms that enforce this property in order to facilitate the task of duplicate detection.

Definition 1. *A* **propagation** *of a message m, $prop(m)$, from source S to any other node, N, at distance d from each other is a sequence of* zeros *and* ones *of length $d+1$, $\langle 1, \cdots, 1 \rangle$, where there is a* one *at location i,* iff*, it has visited the i^{th} topology node on the path from S to N, and* zero *otherwise. The first element in this sequence corresponds to S and is always* one*. We say m*

skipped *a node on its propagation path,* iff*, there is a* zoro *at the node's corresponding location in* $prop(m)$.

We now define *legitimate* propagations of m that restrict the number of nodes that can be skipped by the message. We use this restriction to ensure that certain safety requirements are met. At the same time, this restriction allows for at least failure of δ links or nodes and thus honors the δ-fault-tolerance requirement.

Definition 2. *Propagation of message* m *is* **legitimate***,* iff*, all subsequences of length* $2\delta + 1$ *of* $prop(m)$ *have at least* $\delta + 1$ one *values.*

Intuitively, a message has a *legitimate propagation* if it visits the majority of nodes along its propagation path. More precisely (based on the above definition) considering a message propagated over a chain of nodes we require that the message visits at least $\delta + 1$ nodes among any chain of length $2\delta + 1$. Note that with the above definition, propagation paths shorter than $2\delta + 1$ are automatically legitimate.

5.3 Checking for Legitimate Propagations

Checking to see if message m has had a legitimate propagation is straight forward: every message carries a sequence vector, $seqVect$, that includes elements[5] in the form of $N : seq_N^m$ where seq_N^m is a monotonically increasing sequence number generated and assigned by node N upon receipt of m for the first time. Considering the outgoing copy of m that N sends to another node M while skipping nodes $N_1, \cdots, N_k$, $seqVect$ will be appended by $N_1 : \perp, \cdots, N_k : \perp, N : seq_N^m$. The null values $N_i : \perp (1 \leq i \leq k)$ correspond to the nodes, N_i, between N and M that are skipped due to node or link failures.

To check for legitimacy, it is adequate to consider the null values in the $2\delta + 1$ ending elements on $m.seqVect$. These elements also correspond to zeros on $prop(m)$. The propagation of m is

[5]We discuss the required size of the sequence vector in Section 5.7.

```
procedure isDuplicate(m)
    ret ← false
    for all N_i : seq_{N_i} ∈ m.seqVect(0 ≤ i < 2δ + 1) do
        if seq_{N_i} ≻ highSeq(N_i) ∨ highSeq(N_i) is undefined then
            highSeq(N_i) ← seq_{N_i}
        else
            ret ← true
    return ret
```

Figure 5. Duplicate detection of legitimate messages

thus legitimate, *iff*, the number of these null values does not exceed δ:

$$prop(m) \text{ is legitimate} \Leftrightarrow (\text{number of } (N_i : \perp) \in m.seqVect) \leq \delta$$

This test can be performed on the sender node, N. Also, note that the since m has arrived at the N legitimately, it may only be the case that the outgoing copy is skipping at least one node, N_1. In Section 5.5, we elaborate on how recovery of links/and nodes triggers (re)transmission of illegitimate messages ensuring propagation of messages on legitimate paths.

5.4 Duplicate Detection For Legitimate Messages

If propagation of message m is legitimate, nodes can simply use the last $2\delta+1$ sequence numbers in $m.seqVect$ for duplicate detection. Using this information, m is a duplicate copy, *iff*, any of the sequence numbers, $N_i : seq_{N_i}$, in $m.seqVect$ does not succeed the highest sequence number of the corresponding node, $highSeq(N_i)$, that N is aware of. Furthermore, all $(N_i : seq_{N_i})$ on $m.seqVect$ are used to update $highSeq(N_i)$ if seq_{N_i} succeeds $highSeq(N_i)$. Algorithm 5, illustrates this procedure.

In contrast to (Approach 5.1–ii) nodes only need to keep track of the highest sequence numbers assigned by other peers within within distance $2\delta+1$. This eliminates the need for global knowledge of all message sources.

Lemma 1. *Algorithm 5 detects duplicate arrival of all legitimate copies of a message.*

Proof. Let m^0 and m^i respectively be an initial and a duplicate copy of message m generated at

S that arrived at node N legitimately. Without loss of generality, consider N to be the closest node to S that cannot detect duplicate messages m^0 and m^i. If N is closer to S than $2\delta + 1$, then $S : seq_S^m$ is present in both m^0 and m^i. Since m^0 arrived at N first then upon arrival of m^i, $highSeq(S)$ is at least as seq_S^m. This implies that the test of $seq_S^m > highSeq(S) = seq_S^m$ fails in Algorithm 5 and m^i will be detected as a duplicate message.

Now consider the case where S and N are farther than $2\delta+1$ apart. Due to legitimacy property, both $prop(m^0)$ and $prop(m^i)$ must have non-null entries corresponding to a single node X (i.e., $X : seq_X^{m^0}$, and $X : seq_X^{m^i}$) in common where X is within distance $2\delta + 1$ from N. This is true because $prop(m^0)$ and $prop(m^i)$ each have at least $\delta+1$ non-null entries. Since the size of $seqVect$ is $2\delta+1$ then at least one node (i.e., X) has non-null entries in both vectors. If $seq_X^{m^0}$ and $seq_X^{m^i}$ are identical, then N detects m^i as a duplicate message. Otherwise, if $seq_X^{m^0}$ and $seq_X^{m^0}$ are distinct then X will be a closer node to S than N that has processed both m^0 and m^i and has not detected duplicate copies by wrongly assigning distinct sequence numbers to them ($seq_X^{m^0}$ and $seq_X^{m^i}$). This is a contradiction and the proof is complete. □

5.5 Ensuring Legitimate Propagation Paths

If messages were always legitimately propagated, then Algorithm 1 would suffice to detect duplicate arrival of messages. However, certain patterns of failures may result in message propagation paths that skip more nodes than is allowed under the legitimacy property. Once this occurs, then the propagation paths of duplicate copies of a message may not necessarily share any node. Figure 6 illustrates such propagations. In this section, we present a scheme to deal with these situations. Roughly speaking, nodes stop to propagate messages if they become illegitimate. Such messages are thus delayed until node or link recoveries trigger retransmission of the message on legitimate propagation paths. This ensures safety of the algorithms at all times.

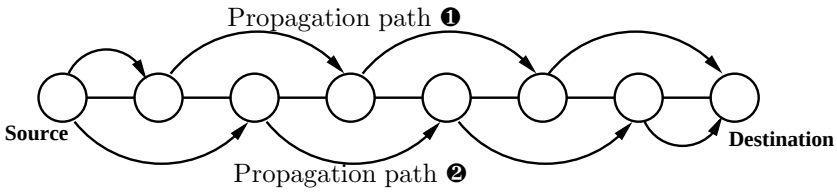


Figure 6. Illegitimate message propagations may only share the source and destination nodes. Duplicate detection on propagation paths longer than $2\delta+1$ does not have access to the source sequence number in the message's $seqVect$. On the other hand, legitimate paths intersect at least once every $2\delta+1$ hops, and enable duplicate detection algorithm to look only at the last $2\delta+1$ sequence numbers in $seqVect$.

5.6 Recovery and Retransmission of Messages

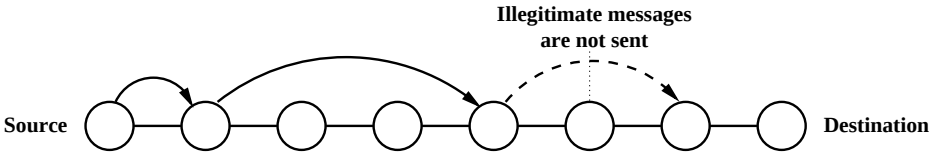
Retransmission takes place as part of recovery (upon activation of a link) and after all transmission of the partition and subscription information. All non-confirmed messages, m, are sent over the newly activated link if there is a recipient in that direction. For publication message this implies presence of a matching subscriber, and for subscription messages this takes place only if the subscription has not been received from the same direction.

As illustrated in Figure 7(a), illegitimate messages are not propagated. Both retransmission at the upper stream nodes (7(b)) or the immediate downstream node (7(c)) can result in new propagations that are legitimate. Any node receiving a (re-)transmitted message, may or may not have previously received it. Furthermore, if a previous copy has been received, it may or may not have been confirmed by that node. Depending on these circumstance, the node handles the message differently. We summarize these cases below for a retransmitted message m arriving at a node N and then describe how N distinguishes these cases:

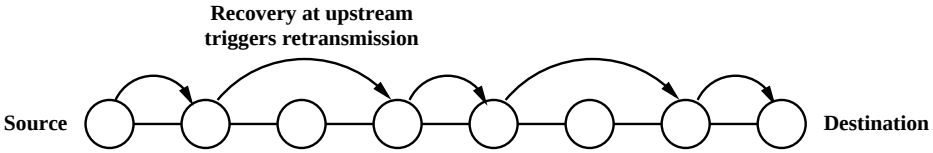
i – Node N has not yet received m: $isDuplicate(m)$ returns $false$;

ii – Node N has received m but not confirmed it yet: $isDuplicate(m)$ returns $true$ and a copy of m is in memory;

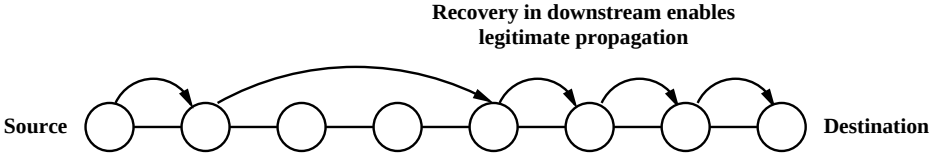
iii – Node N has received m and also confirmed it: $isDuplicate(m)$ returns $true$ but no copy of



(a) Illegitimate messages are never sent



(b) Recovery in upstream nodes triggers retransmission of messages. Newly arrived copies are propagated if legitimate



(c) Recovery of the downstream node may enable legitimate propagation of the message

Figure 7. In a network where $\delta = 2$, a message becomes illegitimate if in any chain of 5 nodes it skips more than 2 nodes

m is in memory.

5.6.1 Subscription vs. Publication Messages

Since we need to ensure that the confirmation of a legitimate subscription message (propagated in the reverse direction) is also legitimate, a receiving node has to (re-)propagate the subscription message regardless of whether it has been received or confirmed before. More specifically, a received subscription message is always propagated unless it becomes illegitimate (which stalls the propagation of the message until further retransmissions). Furthermore, once the confirmation is issued a new subscription tuple is appended to the local subscription routing tables only if it has not been applied before.

In contrast, publications messages are re-propagated only if it has not been confirmed yet.

This is because, confirmation of publication messages need not to be legitimate. For the case of subscription message however (as described above) retransmission takes place regardless of whether the message has been confirmed before. This ensures, that confirmation of a subscription message is also legitimate.

5.7 Length of Sequence Vectors

With sequence vectors of length $2\delta + 1$, Algorithm 5 is able to detect all duplicate arrivals of legitimate messages. However, besides legitimate messages, duplicate detection is also necessary as part of the recovery procedure to discard duplicate subscription entries. Subscription entires stored in the subscription routing tables store the sequence vector of the original subscription message. Upon activation of a link between N and M (N is the activating side) these entries are sent to M in order to be added to its subscription routing table. There is a potential that M has already added the subscription either by having received its original subscription message or via recovery by another node. Similar to duplicate detection of messages, the sequence vector is used for eliminating already existing entries.

As illustrated in Figure 8, nodes engaged in recovery may be up to δ hops apart. Since the legitimate propagation requirement might have been enforced without considering this gap, the receiving node, M, needs to know the assigned sequence numbers of $2\delta + 1$ nodes inward from the sending node N. This asks for a sequence vector of length $(2\delta + 1) + \delta = 3\delta + 1$. It is worthy to note, that the additional sequence numbers corresponding to nodes in range of $2\delta + 1$ to $3\delta + 1$ from M need not to be kept in memory at all times. Rather, these sequence numbers are needed during recovery only.

5.7.1 Likelihood of Violation of the Legitimacy Property

Message propagation may be stalled when legitimacy is violated. This happens in cases where there is at least $\delta + 1$ unavailbility of links or nodes. Thus our δ-fault-tolerance requirement is not compromised. Moreover, in order to violate the legitimacy property, these failures must be within

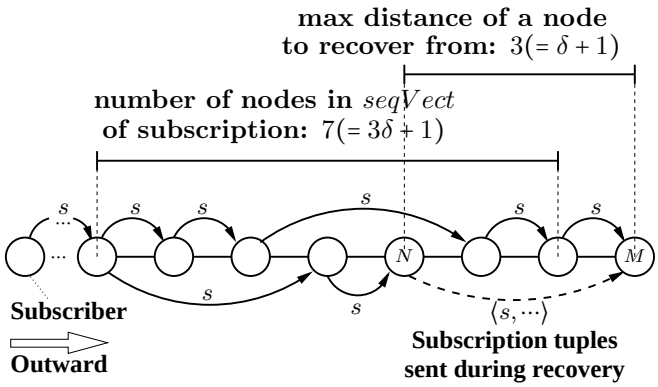


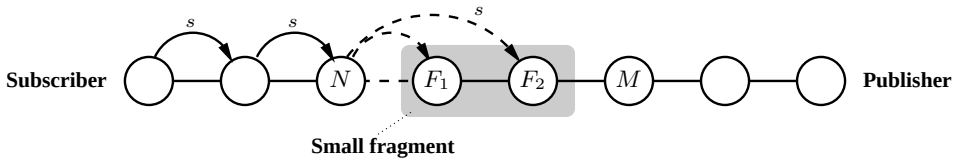
Figure 8. Recovery over active connections (dashed lines). Nodes engaged in recovery are at most $\delta+1$ hops away. By legitimacy identical sub tuples share an element in their $seqVect$ of length $3\delta+1$ ($\delta = 2$).

a $2\delta + 1$ proximity of each other. We believe that in random node and link failure patterns our approach will not incur much overhead, unless failures of more than δ links and nodes are frequent. In that case, the system would have been better configured for a higher level of fault-tolerance by increasing δ.

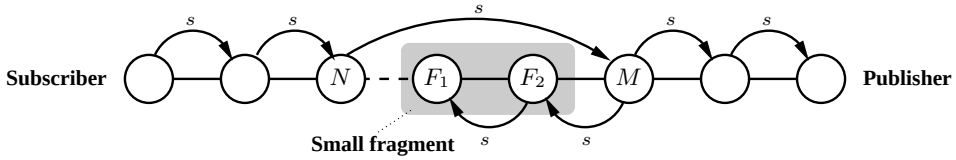
6 Small Fragments

In our distributed P/S system, subscribing clients become eligible to enjoy reliable publication delivery once they receive their subscription confirmation message. This event indicates that propagation of the subscription has been (fully, or partially) completed in the network and system brokers are aware of the subscription predicate and a forwarding path for matching publications towards the subscriber.

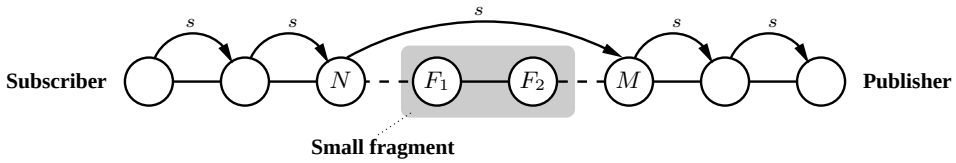
If due to fragmentation a small fragment is formed, then nodes on the fragment may not have received some subscriptions even after the confirmation message has been delivered to the subscriber. In such cases, a matching publication may be wrongly dropped as a result of this lack of knowledge about the subscription. In our algorithms we have taken steps to prevent this from happening. The general idea to deal with such scenarios is interwines subscription propagation



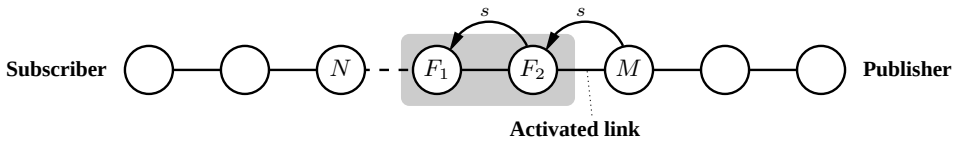
(a) Formation of a small fragment



(b) Bypassing a small fragment: M has an active connection to nodes on the small fragment



(c) Bypassing a small fragment: M does not have an active connection to nodes on the small fragment



(d) Small fragment recovery: missing confirmed subscriptions are sent before publications

Figure 9. Sub-pub propagation in presence of small fragments (δ = 2). Dashed lines represent links that are unavailable.

with the recovery procedure. These two algorithms ensure that any forwarding node has adequate subscription information that prevents publications loss. We first review the overall behavior of the algorithms and then prove its correctness in a lemma.

As illustrated in Figure 9(a), unavailability of some nodes or their interconnecting network links may prevent normal propagation of subscription messages and thus result in formation of a (small or large) fragment. In this section we focus on small fragments. Figures 9(b) illustrates the case in which, the first available node beyond the small fragment (node M in the Figures) has an active connection to the nodes of the fragment, and Figure 9(c) illustrates the case in which M does not possess such active connection. In the former, some nodes of the fragment will still be

able to receive the subscription and M waits to receive their confirmation message. In such cases, any received matching publication will be correctly forwarded towards the subcriber. On the other hand, in the latter case unavailability of the active connection between M and any fragment nodes, cause these nodes to miss the subscription even after M issues the confirmation message to N. However, no publication will ever arrive to any fragment nodes, $F_1, \cdots$, unless one such active connection is created (either from M or from some other node outward from M). This initiates the recovery procedure between the nodes which delivers the missing subscriptions to fragment nodes (F_2 in the Figure 9(d)) before forwarding of any publication. Moreover, the received subscription is forwarded on the receiver's active connections (node F_1 in the figure). Ultimately, all fragment nodes *on the forwarding path of any matching subscriptions* properly processes the subscription preventing publication loss.

Lemma 2. *No node on a small fragment will ever drop a publication that matches a confirmed subscription.*

Proof. Consider a sub-pub-propagation of a subscription s from subscriber S, to any publisher, P which was affected by a small fragment consisting of nodes $F_1, \cdots, F_l$, where $l \leq \delta$. Let p be a matching publication that was published after arrival of subscription's confirmation, c. We show that p will be delivered to S.

Consider there contrary is true, and that some nodes on the small fragment wrongly drop p or do not forward p on the path towards S. In either case, the node must be unaware of s, i.e., has not added its subscription entry to its subscription routing tables. Let F_k be the closest node on the closest small fragment to P that is unaware of s and receives p. Also, X be the node from which F_k received p. From the way the algorithms work, it is clear that the link from X to F_k is active on X's side.

Let t_p be the time X sends p to F_k, t_s be the time X added the subscription entry of s to its subscription routing table, and t_a be the last time before t_p that the link to F_k was activated. There are two possibilities:

i – If $t_s \leq t_a$: then at time t_a, and during the recovery between X and F_k, subscription entry associated with s would be forwarded to F_k which would be added to its subscription routing tables. Furthermore, this takes place before any publication (including p) is sent to F_k.

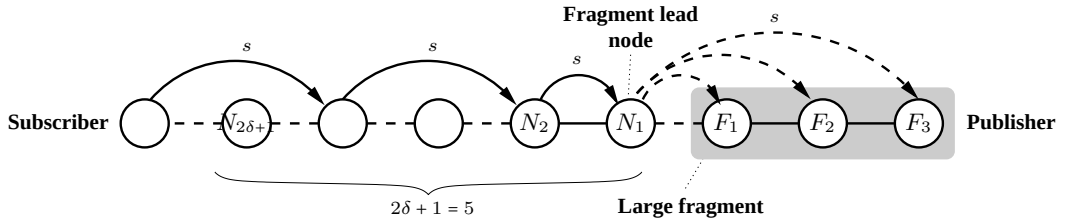
ii – If $t_s > t_a$: then the subscription message of s must have been forwarded to F_k by X according to the subscription propagation algorithm (STEP *iii*) and X must have waited for its confirmation. By that time F_k would have properly forwarded s and added its subscription entry to its local subscription routing table.

□

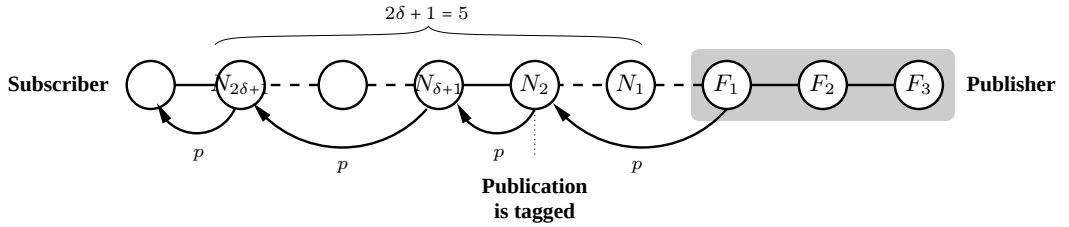
7 Large Fragments

In contrast to small fragments where no confirmed subscription's matching publications may be lost, large fragments may lead to scenarios in which some publications matching partially confirmed subscriptions are not delivered. Large fragments generally are a result of more than δ failures, and in our algorithm ensures the following safety property: once the network faulty conditions are resolved, matching publications will be delivered in-order from some point in time and no message in any publisher's publication stream will be dropped afterwards. This point in time is marked by the completion of a recovery procedure which delivers missing subscriptions to nodes outward from the fragment lead node.

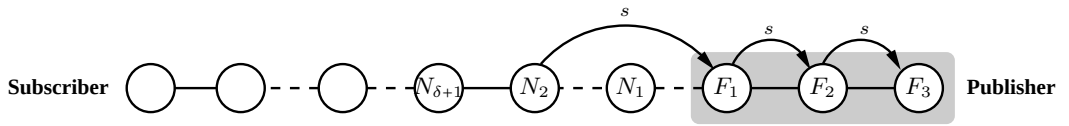
A major challenge towards this goal is dealing with complications that arise as the recovery procedure starts and publications from sources outward the large fragment start to flow. Consider three publications $p1, p2, p3$ that match a partially propagated subscription s, and assume s' is another subscription from a subscriber on the inner side of the large fragment that only matches $p1$. Now consider an execution in which $p1$ and $p2$ are published but not $p3$. and s and s' are the only subscriptions in the system and the large fragment is still not recovered. Since s' is fully propagated, its matching publication $p1$ will be forwarded but will be queued at the fragment nodes as there is no connection towards the subscriber. Furthermore, $p2$ is correctly dropped



(a) Formation of a large fragment: at least $\delta + 1$ nodes on the chain $N_0, \cdots, N_{2\delta+1}$ will receive the partial confirmation message conveying information about the large fragment.



(b) Before recovery completes, publications matching s is forwarded (and tagged) by at least one of the nodes that is aware of the large fragment.



(c) Once the recovery is complete, nodes on the large fragment tag publications with $\overline{parInfo}$ implying that the large fragment is recovered.

Figure 10. Subscription propagation in presence of large fragments ($\delta = 2$). Dashed lines represent links that are unavailable. Publisher F_3 is on the large fragment.

since nodes beyond the large fragment are not aware of any matching subscription (s is missing at these nodes, and s' does not match $p2$). Now assume that an active connection is created between nodes on the fragment towards the subscriber. Partially propagated subscription s may be forwarded outwards on this link, so may be the queued publication $p1$. This publication is intended to subscriber s' but as a result of its overlap with s it is possible that it is delivered to s as well. Once propagation of s completes, $p3$ is published and is delivered to s as well.

In the above execution, subscriber s received two publications $p1$ and $p3$ while $p2$ was dropped. We would like to prevent such scenarios by tagging publication $p1$ with information that notifies subscribers with partially propagated subscriptions (e.g., s) about such anomalous scenarios. In our future work, we consider *historic data access* mechanisms to retrieve the missing publications.

Nonetheless, the algorithms described in this section are necessary to allow subscribers identify publications that are delivered as a result of a side effect with some fully propagated subscriptions.

To achieve the aforementioned goal, partition information, $parInfo$, is included in the partial confirmation message and sent back to the subscriber s along the propagation path of the subscription. As illustrated in Figure 10(a), this path is legitimate and at least $\delta + 1$ nodes on the chain $N_1, \cdots, N_l$ ($l \leq 2\delta + 1$) towards S will receive the confirmation message and subsequently add $parInfo$ in their local partition information tables. Furthermore, if publications are sent inward from fragment nodes and before recovery of the fragment completes, they are tagged with $parInfo$ by at least one of the nodes on the chain $N_1, \cdots, N_l$ that are aware of $parInfo$. This is illustrated in Figure 10(b) and will allow partially confirmed subscribers with the same $parInfo$ to detect the above situation. Finally, the recovery of the large fragment will complete at some point and future publications will be tagged with $\overline{parInfo}$ which effectively cancels out the $parInfo$ tags (Figure 10(c)).

Lemma 3. *A subscriber with a partially confirmed subscription tagged with a partition info, $parInfo$, will receive publications from publishers outward from $parInfo.lFrag$ that are tagged with $parInfo$, or $\overline{parInfo}$. Furthermore, once a matching publication with $\overline{parInfo}$ is received, all future publications are also tagged with $\overline{parInfo}$.*

Proof. Let S and P be any subscriber and publisher which are separated by a large fragment, $lFrag$, with nodes $F_1, \cdots, F_f$ where $f \leq \delta + 1$. Furthermore, let $N_1, \cdots, N_l$ ($l \leq 2\delta + 1$) be the chain of nodes from the fragment's lead node N towards S, and assume s (issued by S) is partially confirmed and is tagged by $parInfo$ of the large fragment, $lFrag$. Finally, assume p and p' are publications issued (in order) from P that matches s and arrive at S after receipt of its confirmation. We split the proof into two parts:

Part I. p is tagged by either $parInfo$ or $\overline{parInfo}$. Consider the contrary is true and that p is neither tagged by $parInfo$ nor $\overline{parInfo}$. Since p comes from P located beyond the large fragment, it must have passed through some nodes on the chain $N_1, \cdots, N_l$. If $l < 2\delta + 1$ then $S = N_l$ which

will tag p itself. On the other hand, if $l = 2\delta + 1$, then at least $\delta + 1$ of these nodes have previously forwarded confirmation of s which was tagged by $parInfo$ and have subsequently added this information to their local partition information tables. Since p is not tagged by $parInfo$ then it must have been forwarded by neither of these nodes. This is not possible since due to the legitimacy property, p cannot skip more than δ nodes on the chain $N_1, \cdots, N_1$.

Part II. If p is tagged by $\overline{parInfo}$, then p' is also tagged with $\overline{parInfo}$. p can only be tagged with $\overline{parInfo}$ by any of $F_1, \cdots, F_f$. Furthermore, since it has arrived at S, it is clear that it has visited at least $\delta + 1$ nodes on the $N_1, \cdots, N_l$. If $l < 2\delta + 1$, then $S = N_l$ and it will replace $parInfo$ with $\overline{parInfo}$ locally. Thus, tagging any subsequent publication from F_i (including p') by $\overline{parInfo}$. On the other hand, if $l = 2\delta + 1$, then p has been forwarded by at least $\delta + 1$ nodes on $N_1, \cdots, N_l$ which have subsequently replaced $parInfo$ with $\overline{parInfo}$. In any legitimate propagation, p' will also be forwarded by at least one of these nodes which tags it with $\overline{parInfo}$.

This concludes the proof.

□

8 Improving Publications Propagation

In this section, we present an important 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 serves to indicate that the upstream node can safely discard its own copy as all downstream matching subscribers have successfully received the publication. In our new 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 depth of $\delta+1$ of the network (downstream from the sending

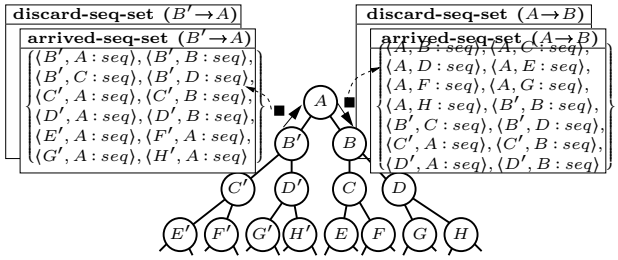


Figure 11. Content of sample *dack* messages ($\delta = 2$).

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 node 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 11).

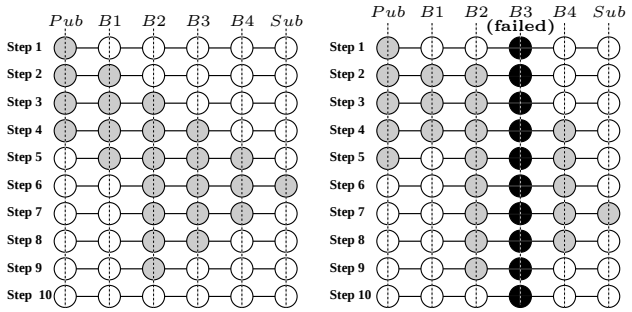
Now consider any publication p that was processed and assigned seq_N^p by node N. Furthermore, let $outPaths^p$ be the set of paths from N to the computed recipient set of p.[6] Receiving a *dack* message, node N purges its own copy of p once for all paths in $outPaths^p$ either of the following hold:

(i) All nodes on that path have reported arrival sequence numbers from N that succeed seq_N^p; or

(ii) At least one node on that path has reported discarded sequence numbers from N that succeed seq_N^p.

Figure 12(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

[6] The length of these paths is at most $\delta + 1$.



(a) If there are no failures, a forwarding node discards its copy once at least $\delta + 1$ nodes (downstream) received it

(b) Having failures in distance $\delta + 1$, a forwarding node discards its copy when any node downstream discards it (steps 5/6)

Figure 12. Propagation of a publication into network ($\delta = 2$). Highlighted nodes hold a copy of the message.

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 12(b) (Step 6) this subsequently signals N to purge p as well.

It is interesting to make note of a 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.

9 Implementation

In this section, we briefly elaborate on our implementation of the δ-fault-tolerant P/S system. Figure 13 illustrates the internal architecture of our brokers.[7] The core of the system is the *connection manager* which maintains the node's abstract *connection objects* each reflecting the state of one connection to another node. The connection manager consults with the topology

[7]Except for the *subscription manager*, our clients also have similar components.

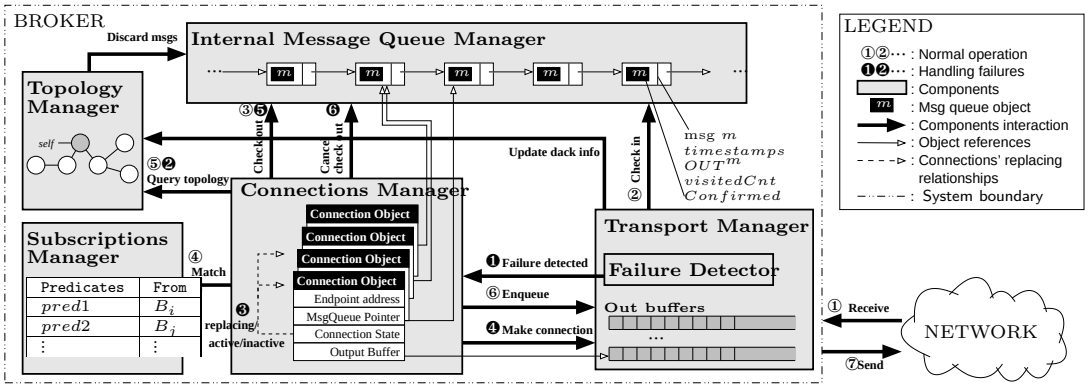
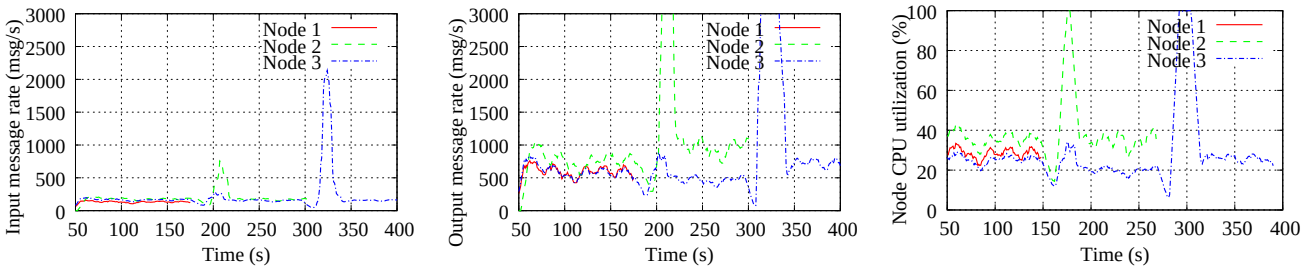


Figure 13. Internal architecture of a broker.

manager to retrieve topology paths to other nodes, and interacts with the failure detector to be notified of unavailability of a remote node via its corresponding connection. Furthermore, each connection object possesses a handler to a TCP connection in the transport layer, and its associated output buffers. At any point, the state of a connection object may be one of *active*, *inactive*, *replacing*, and *beingReplaced*. A connection to remote node X is *active* if it is not failed and there is no other available connection to a closer node on the path to X. Otherwise, the connection is *inactive*. Upon detection of the failure of an *active* connection, its state changes to *beingReplaced* and the connection manager proceeds to create new *replacing* connections to X's farther neighbors. The *beingReplacing* connections correspond to nodes on a fragment.

Publication and subscription messages are placed on a FIFO queue in the *message queue manager* component. Active connection objects have pointers that move along this queue and *check out* messages if they correspond to a node on the $outPath$ of the message. The check out process also updates the $seqVect$ of the outgoing message to reflect any nodes that are bypassed (using $\perp$ values). The remote node's identifier is also added to a *checkedOut* set associated with the message. Once a confirmation arrives, its sender's identifier is removed from the *checkedOut* set. If the set becomes empty, and all active connections have tried to check out the message the message is discarded, and the subscription routing table is updated (for subscription messages only). A confirmation is also placed at the end of the message queue to be checked out by the



(a) Nodes input message rate (b) Nodes output message rate (c) Nodes CPU utilization

Figure 14. Load on nodes after and before failure of their peers (δ = 3).

connection to the sender of the original message.

If the *dack* mechanism is enabled, the nodes periodically send *dack* messages. A *dack* message is processed by updating a topology cache that contains sets of reported arrived and discarded sequence numbers from each node in the topology map. Based on this information new *dack* messages are generated and sent to other nodes. Nodes also use this cache to maintain safe sequence thresholds, $safeSeq$, for each node in their topology map. The $safeSeq$ of a node, X, is the minimum of the reported arrived and discarded *locally generated* sequence numbers from nodes on the topology path to X (Section 8). A cleaner thread at nodes periodically purges publication messages from the message queue if the safe sequence threshold of all nodes in their recipient set succeeds the message's locally assigned sequence number. Subscription messages on the other hand use the normal confirmation mechanism.

10 Evaluation

In this section, we present the results of real world and large scale experimental evaluations related to various aspects of our system. The topology network is configured with δ = 3 and is composed of 86 nodes deployed on a computing cluster with 21 quad-core machines. We dedicated an individual CPU to each node in order to get accurate timing measurements. The measurements correspond to three core nodes in the network with an average node degree of 11 and are carried out under the following four categories.

10.1 Load on Nodes

This experiment investigates the impact of bypassing unavailable neighbors on a node's load. The metrics that we measured are input and output message rates, and the CPU utilization. Figure 14 illustrates the results. Nodes 1, 2, and 3 form a chain in the core of the network and at time $150s$ and 275 nodes 1 and 2 fail respectively. In the time interval $150-275$ node 2 reconnects the network and bypasses node 1 by connecting to its neighbors. This is reflected in the graphs by a short drop, followed by sharp CPU and traffic spikes for node 2. Node 2 has also experienced a less tangible but similar trend. At time 275 node 2 fails too and node 3 reconnects the network shortly after, by bypassing both node 1 and 2. The spikes in the graphs is similar to the first failure but higher and more lasting.

It is interesting to note that after the network is stabilized, the *input traffic rates come back to their levels before the failure* (Figure 14(a)). This is due to the fact that in our approach regardless of the network fault conditions, a node receives a message only if there is a subscriber in its lower subtree. However, the output traffic increases (Figure 14(b)) as more copies need to be sent out. This is also reflected by a *slight* increase in the node's CPU utilization (Figure 14(c)). This is significant since non-faulty nodes do not experience a large increase in their load after failure of their neighbors.

10.2 Publication Delivery Delay

In this section, we present the impact of bypassing nodes 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 15 illustrates the results under three executions in which the publication messages bypass one, two, and three adjacent nodes. At time 50 failures take place and at time 150 nodes start to recover all at once. The higher spikes correspond to publications that were delayed more. The difference between the spikes also closely match the failure detector timeout. Furthermore, it is clear that simultaneous recovery of multiple nodes has an almost constant

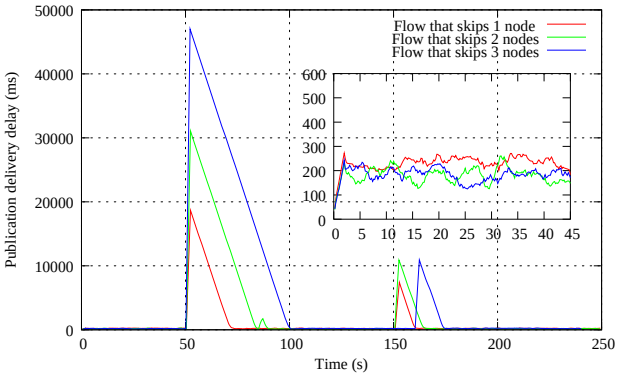


Figure 15. Deliver delay for publications that bypass one, two, or three nodes. Nodes fail at time $50s$ and recover simultaneously at time $150s$ ($\delta = 3$).

impact on publications.

10.3 Impact of Using *dack* Messages

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 node's input message traffic and CPU utilization, both when confirmation messages are used and when the *dack* mechanism is enabled. Figure 16 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.

10.4 Time to Recover

Potential factors that play a role in nodes 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 nodes (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.

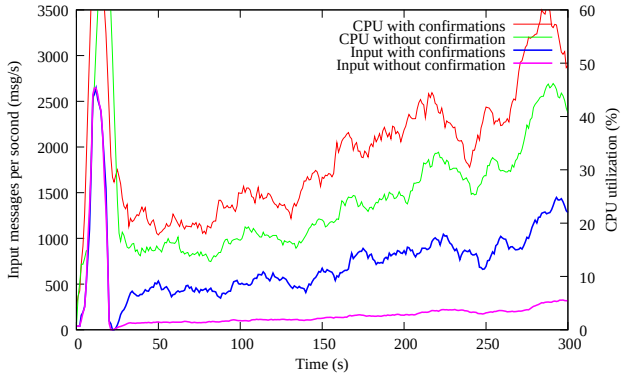


Figure 16. Improvements to message traffic and CPU utilization by substituting confirmation messages with the $dack$ mechanism.

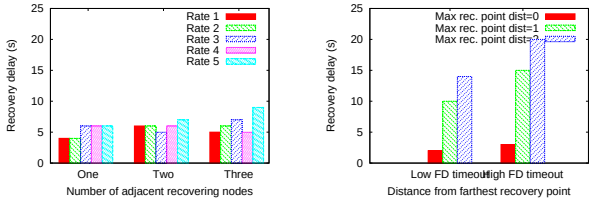


Figure 17. Time to recover, $\delta = 3$.

Figure 17(a) illustrates the time to recover for multiple adjacent recovering nodes under 5 different publication traffic rates. We considered publication rates that do not overwhelm the nodes. It is interesting to see that the time to recover in all cases are almost the same. This implies that if recovering nodes 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 node has to retry by connecting to a farther neighbor, the recovery is delayed. This extra time closely corresponds to the failure detector timeout value. Under low and high time out values, Figure 17(b) compares the recovery time of a node that has to connect to only its immediate neighbors (red box), nodes one hop away (dashed green), and nodes two hops away (dashed blue).

11 Related Work

Related work falls into the following two categories, reliable group multicast, and reliable P/S systems. We discuss them in turn below. The problem of reliable publication delivery is to some extent relevant to the traditional reliable group multicast problems. However, in practical P/S systems with a high publication rate, it is costly to ensure properties such as virtual synchrony [10], or total ordering of publication delivery. This is mainly due to the fact that each publication is delivered based on individual subscriber's interest, 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) [8] is an architecture that improves the resiliency of distributed network applications to Internet path outages. For this purpose, a RON network is deployed consisting of nodes at disparate locations throughout the Internet. Nodes monitor each other, and forward messages via alternative routes if some paths become unavailable. This problem is relevant to ours since reachability is a key part of our approach. However, we address the problem within the context of the content-based P/S systems which is considerably distinct from IP routing.

We now review some of the most important related work in the P/S literature. The Gryphon distributed P/S system [9] introduces the concept of virtual brokers to tolerate failures. Each virtual broker is a set of physical machines which act as identical replicas. In [14], 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. [18] propose an approach to build a δ-fault tolerant P/S system, by constructing $\delta + 1$ disjoint forwarding paths between subscribers and publishers, and forwarding publications concurrently on all redundant paths. Thus, in presence of δ failures, at least one forwarding path remains unaffected. However, this approach incurs high bandwidth consumption. In contrast,

when there are no failures our approach sends publications along *a single* shortest path towards each subscriber. Furthermore, messages are only re-sent when a failure is initially detected (and not afterwards).

In Hermes [16] 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 [11] 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. [13] also present an algorithm to improve the efficiency of reconfiguration in highly dynamic P/S networks. However, the proposed *reconfiguration path* approach only strives to minimize the publication loss during a reconfiguration process, rather than preventing it. Epidemic (gossip) algorithms have been applied to P/S systems [12], in an attempt to improve the reliability of highly dynamic systems. However, these approaches do not provide strict publication delivery guarantees.

12 Conclusions

In this paper, we extended our δ-fault-tolerant distributed publish/subscribe approach [14] in order to tolerate network partitions, support node recovery, and improve network's message traffic. We believe, that our approach is suitable for large and reasonably stable environments such as that of an enterprise or a data center [1], where reliable publication delivery is desired in spite of failures. As future work, we would like to exploit our scheme to allow for multi-path load balancing, and support some of P/S optimization techniques such as subscription covering. We are also investigating to adapt our scheme to tolerate byzantine failures, e.g., malicious attacks or software bugs that may lead to arbitrary node behavior.

13 Broker vs. Client Functionalities

In our approach we have enabled system clients (publishers and subscribers) to connect to brokers other than the one that they initially joined to. In fact in our approach, upon failure or unavailability of a client's immediate broker the disconnected client may create connections to the failed node's farther neighbors which may include both brokers and other clients.

We support this design by the following argument: consider a 2-fault-tolerant system in which the publish/subscribe system is to be available in the face of two failures. If we restrict a subscriber to connect to only *one* broker (likewise, only one broker can connect to the subscriber) then upon failure of that broker the subscriber may suffer from interruptions in its incoming publication flows.

On the other hand, in our design the publication delivery can only be interrupted upon failure of at least three brokers or failure of the client itself. Creation of these connections are temporary, and last for as long as a closer broker recovers. Meanwhile, the publications flow over various paths and the client's publication delivery service is not interrupted.

References

[1] The Design of a Reliable Message Distribution Service for Google, 2007. Tutorial at the ACM/IFIP/USENIX 8th International Middleware Conference.

[2] Tibco Enterprise Service Bus. `http://www.tibco.com/resources/software/esb/tibco_esb_datasheet.pdf`.

[3] Java Message Service (JMS). `http://java.sun.com/products/jms/`.

[4] Elvin Distributed Publish/Subscribe System. `http://elvin.org/specs/index.html`.

[5] MQSeries Publish/Subscribe Applications. `http://www.redbooks.ibm.com/abstracts/sg246282.html`, 2001.

[6] Healthcare Collaborative Network Solution Planning and Implementation. `http://www.redbooks.ibm.com/Redbooks.nsf/9445fa5b416f6e32852569ae006bb65f/34cc1a37088754ce85257051004b1f19?OpenDocument`.

[7] US Patent 7103680 -Publish/subscribe data processing with publication points for customized message processing. `http://www.patentstorm.us/patents/7103680/description.html`.

[8] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Symposium on Operating Systems Principles*, pages 131–145, 2001.

[9] S. Bhola, R. E. Strom, S. Bagchi, Y. Zhao, and J. S. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *DSN*, 2002.

[10] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[11] R. Chand and P. Felber. XNET: a reliable content-based publish/subscribe system. In *SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*. IEEE Computer Society, 2004.

[12] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Introducing reliability in content-based publish-subscribe through epidemic algorithms. In *DEBS*, 2003.

[13] G. Cugola, D. Frey, A. Murphy, and G. Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe, 2003.

[14] R. S. Kazemzadeh and H.-A. Jacobsen. Reliable and Fault-Tolerant Distributed Publish/Subscribe Systems. In *INFOCOM*, 2008 (submitted).

[15] G. Li, V. Muthusamy, H.-A. Jacobsen, and S. Mankovski. Decentralized execution of event-driven scientific workflows. In *SCW*, pages 73–82. IEEE Computer Society, 2006.

[16] P. R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCS Workshops*, 2002.

[17] R. Sherafat and H.-A. Jacobsen. δ-fault-tolerant publish/subscribe systems, 2007. Available online at: `http://research.msrg.utoronto.ca/pub/Padres/WebHome/tech-rep07.pdf`.

[18] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using xml. In *SOSP*, 2001.

[19] D. Tam, R. Azimi, and H.-A. Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In K. Aberer, V. Kalogeraki, and M. Koubarakis, editors, *DBISP2P*, volume 2944 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2003.