Partition-Tolerant Distributed Publish/Subscribe Systems

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

***Abstract*— In this paper, we develop *reliable* distributed publish/subscribe algorithms that can tolerate concurrent failure of up to δ brokers or links. In our approach, δ is a configuration parameter which determines the level of fault-tolerance of the system, and reliability refers to exactly-once and per-source in-order delivery of publications to clients with matching subscriptions. We propose protocols to address three problems in presence of broker or link failures: *(i)* subscription propagation; *(ii)* event forwarding; and *(iii)* broker recovery. Finally, we study the effectiveness of our approach when the number of concurrent failures exceed delta. Via experimental evaluations, we demonstrate that a system configured with a modest value of $\delta = 3$ is able to reliably deliver 97% of publications in presence of failure of up to 17% of brokers.**

Keywords: Publish/subscribe; reliability; fault-tolerance
Categories: Distributed systems; Fault-tolerant systems; Middleware; Reliability; Networked systems

I. Introduction

Many of today's large-scale distributed systems require reliable many-to-many communication capabilities that go beyond the provisions of underlying network protocols. Examples of such applications include news dissemination services, push-based RSS feeds [1], [2], job tracking and monitoring applications [3], and financial market data distribution [4] or realtime processing systems for algorithmic trading [5]. Since TCP's focus and design objectives are mainly to facilitate reliable point-to-point communication, developers of distributed applications with many-to-many communication requirements often need to custom-build additional infrastructure to ensure scalable and reliable operation. A minimal effort to address this problem needs to carefully account for message loss, reordering, node failures and network partitioning.

A *reliable Publish/Subscribe (P/S) middleware* is well positioned to bridge the gap between low-level networking protocols and high-level communication needs of many distributed applications and thus help relieve developers of much of the hassle associated with reliable messaging at scale. The P/S model provides a simple and powerful abstraction enabling information sources to *publish* messages and information sinks to *subscribe* to messages of interest (without explicit knowledge of the source). This can be done in two ways by subscribing to specific channels (also known as topics) or via specifying filtering constraints over the published content. The former is called channel-based (or topic-based) and the latter is referred to as content-based P/S.

In this paper, we focus on the reliability aspects of content-based distributed P/S systems that are composed of a set of dedicated message routers (called brokers) forming an *application-level* overlay network [6], [7], [8], [9]. Reliability in our context refers to "*exactly-once per-source ordered delivery of publications to matching subscribers*". This definition allows us to establish an abstract notion of an ordered *gap-less* message flows between each source and sink. Note that in content-based P/S only a subset of publications from a publisher may be present in the flow that is delivered to a subscriber, and our notion of gap-less delivery must be interpreted over this sequence of matching publications. More specifically, we require that the order of published messages must be preserved, and for every two consecutive publications that are delivered to a client no intermediate matching publications must have been published.

Packet loss, and broker and link failures (which are common in large distributed systems) may hinder reliability and introduce reordering or gaps in the publication flows. Furthermore, link failures may cause even more challenges when they prevent delivery of subscriptions to network brokers. In such scenarios, brokers that remain unaware of a subscription may discard matching publications and introduce gaps in the flow of messages delivered to the client. We now use a simple application scenario to first demonstrate the applicability of reliable P/S model for network applications and highlight the challenge of dealing with link failures.

Motivating example: Consider a Content Distribution Network (CDN) with content servers around the globe. To feed new content internally to the servers, each server connects to a P/S broker and subscribes to the updates published by the content-provider (as shown in Figure 1). This way the CDN server acts as a subscribing client of the internal P/S network and receives a reliable stream of content updates which it will serve externally to Internet users. For example, subscription s_1 =`[prov:bbc, loc:nAmerica]`, allows a server to subscribe to updates from BBC news for North America, or s_2 =`[prov:bbc, type:video, extra:mostViewed]` matches most-viewed video content from BBC News. Published content comes with a content descriptor that is used to determined which servers must receive the message. For instance, p_1 =`[prov:bbc, loc:nAmerica, type:video, subj:oilSlick, extra:mostViewed]` describes a most-viewed video file of the oil slick disaster in the Gulf of Mexico, or p_2 =`[prov:bbc, loc:nAmerica, type:HTML, extra:frontPage]` describes the BBC news front page. Now let's assume that p_1 and p_2 are recently published content (in-order) and that p_2's HTML content has a link to the video file of p_1 that must be stored on the same server. The per-source in-order requirement in our reliability specification ensures that servers will receive these messages in-order, and thus prevents scenarios such as when an HTML page has a dangling reference to a non-existing video file.

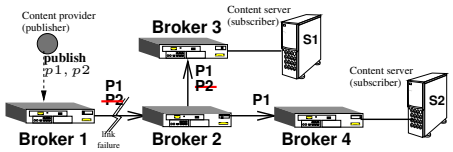


Figure 1. Partitions may lead to publication loss.

Challenge of dealing with link failures: Routing paths in the internal P/S network are constructed by propagating subscriptions between brokers. Each broker stores the previous hop that the subscription was received from and uses this information to forward matching content in the reverse direction towards the subscriber. Now consider servers S_1 and S_2 in Figure 1 subscribe to s_1 and s_2, respectively. We describe an unreliable delivery scenario in which due to transient failure of the link between B_1 and B_2, server S_2 only receives p_1 and misses p_2 and thus its publication flow will contain a gap (note that p_1 and p_2 both match s_2): Assume that s_1 and s_2 are issued prior to and during the link failure, respectively. As a result s_2 will not reach Broker B_1. If p_1 and p_2 are published at this point, Broker B_1 only keeps p_1 (which matches s_1) and filters out p_2 (which matches none of its received subscriptions). Later, once the link is re-established, B_1 sends outstanding messages including p_1 towards B_2 and since B_2 has already received both s_1 and s_2 it will send p_1 to both B_3 and B_4 which will deliver the message to S_1 and S_2. Furthermore, it is evident that server S_2 will never receive p_2 and thus its publication flow from the BBC publisher will have a gap. Finally, note that due to the content-based nature of messaging, S_2 has no way to say that p_2 is the next message that it must have received.

Overview of the approach: We introduce a fault-tolerance configuration parameter which we denote as δ and require brokers to maintain knowledge of their (δ + 1)-neighborhood. The idea is that a broker can use this knowledge to bypass up to δ of its failed (or unreachable) neighbors. This simple approach improves the network connectivity and enables any formation of up to δ concurrent failures to be tolerated (δ-fault-tolerant). Apart from pure network connectivity, we also need to ensure that link failures do not contribute to scenarios in which incomplete subscription routing information at brokers results in gaps in publication flows. A simple scenario in which this can happen was described above. Our approach to prevent such cases relies on ensuring that "*a publication is delivered to a matching subscriber only if it is forwarded by brokers that are all aware of the client's subscription*". Maintaining this invariant when brokers and links may go through cycles of failures and recoveries is challenging. To overcome this challenge, we devise three algorithms that work in synergy and ensure reliable publication delivery at all times. The contributions of this paper can be summarized as follows: *(i)* our subscription propagation algorithm is responsible to deliver subscriptions to parts of the network that are reachable from the issuing subscriber while bypassing disconnected or failed brokers; *(ii)* our publication forwarding algorithm uses this routing information in order to forward matching publications and to detect and tag those publications that may compromise reliability when delivered to certain subscribers; *(iii)* our recovery procedure ensures that when a failed broker restarts or a link is re-established the routing tables of endpoint brokers are synchronized as if the failure had never happened. Finally, we experimentally study scenarios in which the number of failures exceed the guaranteed system limit of δ. We show a modest value of δ = 3 is able to ensure reliable delivery of 97% of publications in presence of failure of up to 17% of brokers.

II. System Model

We assume asynchronous communication links with unknown delivery delays. In this model, a failed link can be thought of as a link that can become infinitely slow, thus delaying some messages forever, i.e., lose messages. We also assume that each broker is equipped with a local failure detector (FD) with eventually strong and eventually accurate properties [10] (perhaps implemented by a ping mechanism) that outputs the set of neighbors that are currently *unreachable* from the broker. Theoretical results have shown that for two processes A and B, crash of B or failure of the communication link between A and B is indistinguishable from A's point of view.[1] As a result, our notion of unreachability inevitably encompasses *both* link failure and crash of a neighbor. During the interval that A's FD does not indicate failure of B, we say that A maintains an *established* session to B denoted by $\mathcal{S}_{A,B}$. Finally, we assume that A's messages sent to B during a session is delivered either with FIFO ordering or are never delivered (i.e., in the latter case A will eventually detect failure of B).[2]

Upon joining the system brokers form an initial tree-based application layer overlay network[3] and maintain knowledge of their neighbors located within distance $\Delta = \delta + 1$. This tree is referred to as the *primary tree* and its edges are called *primary links* which correspond to the communication sessions established when brokers join the system. A *primary path* is a sequence of pairwise adjacent brokers in the primary tree. In absence of failures, publications and subscriptions are forwarded between brokers over primary links. However, occurrence of failures may necessitate a broker to bypass its unreachable neighbor(s) by establishing new sessions to neighbors of the unreachable broker(s). We say that A's session to B becomes *active* when $\mathcal{S}_{A,B}$ is established and A has no other established session to another Broker $C \in \mathbf{P}(A, B)$. We use Act_A to denote the set of A's active sessions which may only include sessions to brokers in A's Δ-neighborhood. Note that in general, activation of session $\mathcal{S}_{A,B}$ does not imply that Broker B also views $\mathcal{S}_{B,A}$ as active. Finally, the definition of active sessions also implies that for any arbitrary Broker X, Broker A *may* only have *one unique* active session on the primary path $\mathbf{P}(X, A)$. We say that this session is A's active session for this path.

III. Network Partitions

While our notion of unreachability concerns inability of two brokers to communicate over a *direct* link, we use the notion of *network partitions* to capture the inability to *route* messages over paths comprised of multiple brokers. Consider a primary path $\mathbf{P}(src, dst)$. Once a broker on this path detects a subsequent broker to be unreachable, it attempts to reach out and activate a session to the broker(s) further down the path. This creates sequences of unreachable brokers along $\mathbf{P}(src, dst)$ that are bypassed by the newly activated sessions. We say that these brokers form a partition *island*. For example, Figure 2 illustrates that Brokers B_3 and B_4 are bypassed on $\mathbf{P}(B_0, B_7)$ when Broker B_2 activates $\mathcal{S}_{B_2,B_4}$. As a result, we say that Brokers B_3 and B_4 are *nodes* on the *partition island* and B_2 is the *partition detector* (note that B_2 has detected unreachability of partition nodes).

It is possible that due to multiple failures, a partition detector cannot reach out to any broker further down the path. This is shown in Figure 2 where B_2 on $\mathbf{P}(B_0, B'_7)$ is unaware of Broker B'_6

[1] This may lead to what is referred to in the literature as a FD mistake.
[2] These assumptions are consistent with the way TCP works.
[3] A registry service may assist brokers to identify the best broker to join.

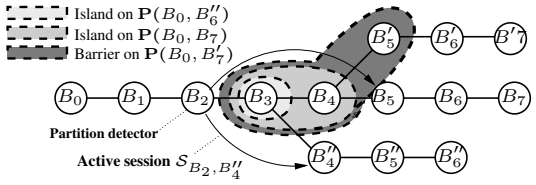


Figure 2. Primary paths (solid lines) and network partitions (highlighted areas) in network with δ = 2.

(since Δ = 3) and cannot activate any new sessions. In this case, we say that a partition *barrier* is formed. More concretely, a partition is a barrier if no active session bypasses the partition nodes.

Regardless of its type, a partition is identified by a unique *partition identifier* generated by the partition detector. More formally pid is a tuple $(B, i, pnodes)$, where B is the partition detector, i is a unique incremental value assigned by B and $pnodes = \langle B_i, \cdots \rangle$ is the primary path consisting of partition nodes. The brokers in subtrees of the last broker on $pnodes$ are said to be *beyond* the partition. For example, the partitions shown in Figure 2 are $\{(B_2, 1, \langle B_3 \rangle), (B_2, 2, \langle B_3, B_4 \rangle), (B_2, 3, \langle B_3, B_4, B'_5 \rangle), \}$, and are created when B_3's active sessions to Brokers B_3, B_4 and B'_5 fails. Brokers store the partition identifiers in a local set data structure called the Partition Table (PT). Furthermore, if communication over a failed link is restored, old partitions may recover and thus be removed from PT. Whenever a partition id is added to or removed from PT, the broker notifies its neighbors over all its *established* communication sessions using partition information messages that propagate to brokers within distance D_{PT} of the partition detector (we will determine the value of D_{PT} in Section VI). Receiving brokers update their own PT accordingly and notify their neighbors.

Message types: There are four types of messages in our system: *(i)* partition information messages carry partition identifiers; *(ii)* publications are data messages generated by publishers; *(iii)* subscription specify the subscriber's interest; and *(iv)* confirmation messages acknowledge delivery of publications or subscriptions to subtrees in the primary tree. Except for the first type, the other three message types may be tagged by brokers with a set of partition ids, $T = \{pids\}$, to indicate that forwarding of the tagged message was prematurely affected by the partitions in T.

IV. Subscription Propagation Protocol

Subscribers connect to a broker of choice[4] and issue subscriptions that will propagate throughout the network. We say that a broker *accepts* a subscription when it is added to the Subscription Routing Table (SRT). Only accepted subscriptions are used for publication forwarding. In our system, acceptance of a subscription does not immediately follow its arrival at a broker and there are several steps in between that we elaborate on in this section.

Assume s is a subscription issued by local subscribers of source broker S. Copies of s that arrive at a Broker B contain the following information: *(i)* subscription predicates, $pred$; *(ii)* a trailing portion of propagation path, $\mathbf{SP}_s(B)$ along $\mathbf{P}(S, B)$; and *(iii)* a vector of sequence numbers, $\mathbf{SeqVec}$. Brokers use $pred$ to *evaluate* against a publication's content and determine whether subscription matches the publication; $\mathbf{SP}_s(B)$ is a sub-path of $\mathbf{P}(S, B)$ of length D_{SRT} that is updated as s propagates through the network; and finally, $\mathbf{SeqVec}$ a vector of length D_{SRT} that is used for message identification and ordering. We now describe subscription propagation in absence and in presence of failures.

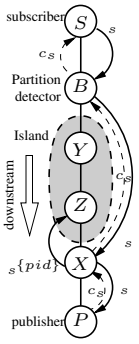
[4]Load balancing algorithms [11] may consider different network parameters to assign a client to a broker.

A. Subscription Propagation When There are no Failures

A network broker, B, processes arrived subscriptions in order, and for each subscription, s, B first forwards s over its active sessions to all brokers downstream of B (downstream is the direction away from S). Since there are no failures B has an active session to all its immediate neighbors in the primary tree. B inserts the id of these brokers into a set, Out_s (called the outgoing set) before sending s to these brokers. Next, B waits to receives acknowledgements from brokers in Out_s. The processing of s at downstream brokers takes place in a similar fashion until s arrives at an edge broker (with no downstream brokers). At edge brokers, Out_s is empty and s is accepted immediately. Furthermore, a special form of acknowledgement called a confirmation message, c_s, is issued upstream. Once c_s arrives at a broker, say B, its sender is removed from $Out_s(B)$ and B checks whether $Out_s(B)$ became empty. If so, B accepts s and proceeds to send c_s upstream, otherwise it waits for the remaining confirmations to arrive.

In its current form, the subscription propagation scheme described thus far resembles a simple tree propagation algorithm with acknowledgements. However, the possibility of broker and link failures have significant implications on the way the algorithm works in reality. For instance, if B's session to its downstream broker Y becomes disconnected and $Y \in Out_s$, then B has to decide whether to wait for a confirmation or to proceed to confirm s in some way. The first choice may compromise liveness: if Y is crashed permanently then the propagation process will be blocked indefinitely. On the other hand, the second choice may compromise reliability: if Y is not crashed it will remain unaware of s and may discard publications that match s (thus introduce gaps in the publication flow – similar to the scenario in Section I). In what follows we describe our approach that allows Broker B to make progress in a way that does not compromise reliability. We further break down the discussion into two parts that correspond to partition islands and partition barriers.

B. Subscription Propagation over Islands

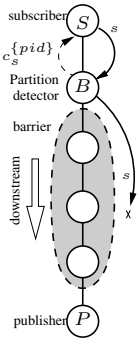


As mentioned earlier, if B's active session to Broker Y fails a partition (pid) is formed. Let $\mathbf{P}(S, P)$ be the primary path from S to an arbitrary publisher, P, that is located beyond the partition and downstream of B. If B successfully activates a new session on $\mathbf{P}(B, P)$, then pid is by definition a partition island on this path. In this situation, B continues by sending s to X (instead of Y) thus replacing Y with X in Out_s. Then B awaits to receive c_s from X and subsequently remove X from Out_s. In case, $\mathcal{S}_{B,X}$ fails in the meantime before c_s arrives, B replaces X in Out_s with any newly activated session that bypass X. However, if $\mathcal{S}_{B,X}$ remains active long enough, X should be able to complete propagation of s and send c_s to B. At this point B removes X from Out_s and accepts s once $Out_s(B) = \varnothing$.

Note that in the above scenario brokers on the island have not received (nor accepted) s. However, while B was unable to reach the island brokers, a broker, say X, beyond the island may possibly be able to communicate with them. In this case, X must have an active session to Broker $Z \in pid.pnodes$. Upon accepting s Broker X uses $\mathcal{S}_{X,Z}$ to send a copy of the subscription *tagged* with pid (s^T s.t. $pid \in T$). A tagged subscription message is only of interest to brokers on the partition islands that it was tagged with (i.e., $pid.pnodes$) and thus will only be sent to these brokers. Upon receipt of a tagged subscription, it is immediately accepted and sent over active sessions to other brokers on the island (for example, over $\mathcal{S}_{Z,Y}$ in the figure). The intuition behind this *upstream* subscription propagation is simply to ensure that if an active session from a broker beyond the island exists to brokers on the island, then this session will not be used to send a matching publication to the island broker prior to the subscription being accepted by the island broker.

C. Subscription Propagation over Barriers

For the case of barriers, no bypassing session exists to deliver s to brokers on or beyond the partition. As a result, all brokers downstream of the partition detector, B, will not receive and accept the subscription. In this scenario, Broker B removes barrier nodes from $Out_s(B)$ and accepts s once the remaining confirmations arrive and Out_s becomes empty. Furthermore, B tags the confirmation message that it sends upstream with pid of the barrier whose nodes have not confirmed s (c_s^T s.t. $pid \in T$). As upstream brokers accept s and issue their own confirmations, they pass these tags along and possibly add new pids to T. Once the source broker, S accepts s, it stores all the tags in the received confimrations along with the subscription in its SRT. The purpose of these tags (carried in confirmation messages) is quite different from the tags in a subscription message (described in Section IV-B). pid tags carried in confirmation messages indicate that publications generated by brokers on or beyond the partition ids are not readily safe to be delivered to the subscriber even though they match its predicates. This is due to the fact that the publisher's source broker, P, has not accepted s. The forwarding algorithm (Section V) uses these tags to detect publications (e.g., p_1 in sample scenario described in Section I) whose delivery to the client is unsafe.



V. Publication Forwarding

In short, the publication forwarding algorithm delivers a publication, p, to a subscriber with subscription s only if *(i)* p matches s; and *(ii)* p's source broker and all other brokers along its propagation path had already accepted s prior to forwarding p (*safety condition*). We now describe forwarding of p as it arrives at Broker B. Forwarding constitutes of five steps: queuing, barrier checking, matching, routing, and cleanup. The queueing step determines whether p has been received before (duplicate detection is described in Section VI) and appends p to a local FIFO queue in order to preserve the order of messages in a flow. The barrier checking step involves determining whether p's sender is on any of the barriers currently known to B (stored in its PT). If such a barrier exists, B tags the publication with the corresponding

```
1: procedure PROCESS_SUB(s)
2:     ActiveUpstream ← Map(SP_s(B))
3:     Out_s(B) ← Act_B − ActiveUpstream
4:     for all X ∈ Out_s(B) do
5:         SEND_SUB_TO(s, X)
6: procedure SEND_SUB_TO(s, X)
7:     s ← s.clone()
8:     for all pid ∈ T do
9:         if X is beyond pid.pnodes then
10:            Tag s with pid
11:    UPDATE_SEQ_VEC_SENT_TO(s, X)
12:    Send s to X
13: procedure ON_SUB_CONFIRMATION_RECEIVE(c_s)
14:    for all pid tag in c_s do
15:        s.Tags ← s.Tags ∪ pid
16:    Out_s(B) ← Out_s(B) − c_s.sender
17:    if Out_s(B) == ∅ then
18:        ACCEPT_AND_CONFIRM(s)
19: procedure ACCEPT_AND_CONFIRM(s)
20:    Add s to SRT
21:    Tag c_s with s.Tags
22:    for all pid ∈ s.tags do
23:        Send s to Mapping(pid.pnodes)
24:    for all pid ∈ PT do
25:        if B == pid.detector && pid.pnodes ∌ SP_s(B) then
26:            Tag c_s with pid
27:    for all X ∈ s.senders do
28:        Send c_s^pids
29: procedure ON_CONFIRMATION_RECEIVE(c_s)
30:    Out_s ← Out_s − c_s.sender
31:    if Out_s == ∅ then ACCEPT_AND_CONFIRM(s)
```

Figure 3. Subscription propagation algorithms at broker B

pids: p^T s.t. $pid \in T$. This tag will convey special meaning for subscribers whose subscriptions were confirmed with an identical pid. Next, the set of subscriptions accepted at B that match p are computed, $Match_p(B)$. Finally, for each subscription s in this set, B determines its active session on $\mathbf{SP}_s(B)$ and adds the corresponding broker to the message's outgoing set ($Out_p(B)$) in a similar manner done in subscription propagation. For each copy of p that is sent out to brokers in $Out_p(B)$, B awaits to receive a confirmation, c_p, indicating successful delivery of p to matching subscribers in the corresponding subtree. When a confirmation arrives, B removes the sender from $Out_p(B)$, and checks whether the set has become empty. If so, B discards p is its queue (the cleanup step) and sends a confirmation to the broker(s) that copies of p had arrived from. Figure 4 illustrates this algorithm.

If p matches B's local subscriber, s, then B must first determine whether delivery of p is safe. For this purpose, p's tags are examined against pids seen in s's confirmation tags. If a shared pid exists, then p has been issued by a broker, P, located beyond a barrier. Since P is likely not to have accepted s prior to issuing p, delivery of p to the client is not safe (it may leads to gaps). Otherwise, if p and c_s have no pid tags in common, then p has been forwarded reliably and thus is delivered to the subscriber.

VI. Duplicate Detection

To illustrate the need for duplicate detection, consider a simple case in which brokers A, B and C form a chain and publication p is first sent from A to C first via Broker B. If session $\mathcal{S}_{A,B}$ fails

```
1: procedure ON_PUB_RECEIVE(p)
2:     if Is_Duplicate(p) then
3:         p' ← Queue.Find(p)
4:         if p' == null then return
5:         else
6:             p'.senders ← p'.senders ∪ p.sender
7:     else
8:         Queue.append(p)
9:     for all pid ∈ PT s.t. p.source on pid.pnodes do
10:        Tag p with pid
11:    Match_p(B) ← match p against subs in SRT
12:    CHECK_LOCAL_SUBS(P)
13:    for all s ∈ Match_p(B) do
14:        X ← ActiveMapping(SP_s(B))
15:        Out_p(B) ← Out_p(B) ∪ X
16:        SEND_PUB(p, X)
17: procedure ON_PUB_CONFIRMATION_RECEIVE(c_p)
18:    p ← Queue.Find(c_p)
19:    Out_p(B) ← Out_p(B) − c_p.sender
20:    if Out_p(B) == ∅ then
21:        DELETE_AND_CONFIRM_PUB(P)
22: procedure DELETE_AND_CONFIRM_PUB(p)
23:    Queue.delete(p)
24:    for all X ∈ p.senders do
25:        Send c_p to X
26: procedure SEND_PUB(p, X)
27:    p ← p.clone()
28:    p.updateSeqVector()
29:    UPDATE_SEQ_VEC_SENT_TO(p), X
30:    Send p to X
31: procedure CHECK_LOCAL_SUBS(p)
32:    if Is_Duplicate(p) then return
33:    for all s ∈ Match_p(B) do
34:        if s.source = B then
35:            if s.Tags ∩ p.Tags = ∅ then
36:                Deliver p to subscriber – reliably
```

Figure 4. Publication forwarding algorithms at broker B

before B sends c_p to A, then A will try to re-send p directly to C over $\mathcal{S}_{A,C}$. Since C may have already received p from B, the second copy of p will be a duplicate.

In a trivial duplicate detection scheme, brokers can use source-assigned sequence numbers and keep track of highest sequence received from *all* other brokers. Since message transmissions during each session and processing at brokers are FIFO, a duplicate message will have a sequence number that does not succeed the highest sequence number previously received from the same source. In this section, we develop an alternative approach that allows brokers to use sequence numbers assigned to forwarded messages by nearby brokers within a certain distance. We use D_{SEQ} to denote this distance. Compared to the approach that uses source-assigned sequence numbers, this has the advantage of not requiring brokers to maintain state for every other source broker.

Before we introduce our scheme, we would like to draw an analogy for a special case of $\delta = 0$. Such a system lacks fault-tolerance and since no broker can be bypassed a simple sender-assigned sequence number schemes can be used to detect duplicates: a receiving broker only needs to keep track of the highest sequence number received from its *immediate neighbors* and a message, m, is a duplicate if seq_m does not succeed $highestSeq(m.sender)$.

If $\delta > 0$ and brokers can be bypassed, the above approach cannot be readily used. Figure 5 illustrates a hazardous scenario for $\delta = 1$ in which due to occurrence of failures in a certain order, publication p is forwarded to the destination broker via two completely disjoint paths. As a result, the only way for the destination broker to detect p's second copy as a duplicate is to use the message's original source-assigned sequence number. Since the source can be any arbitrary broker, this implies that any destination broker has to keep track of sequence numbers from all source brokers. Depending on the system size, this amount of state can be prohibitively large.

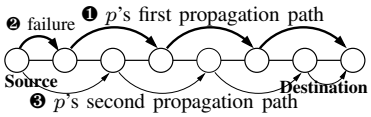


Figure 5. In general, messages may be propagated over paths that are disjoint.

The question that we face at this point is whether we can use an approach that relies on sequence numbers of nearby brokers (similar to the case of $\delta = 0$) for the general case of $\delta > 0$. The answer is positive, however, message propagations must satisfy certain restrictions as required by the *legitimate* propagation property.

Legitimate propagations: Propagation of publication m from source broker, S, to a destination broker, B, is *legitimate* if in all primary sub-paths of $\mathbf{P}(S, B)$ of length $2\delta + 1$, m bypasses no more than δ brokers. This definition has two important properties: first, a publication can still bypass δ failed (or unreachable) brokers and thus the system can remain δ-fault tolerance; and second, m's legitimate propagation ensures that m is forwarded by a *majority* of brokers along the primary path between S and B. Intuitively, our duplicate detection algorithm exploits the latter property in order to ensure that the propagation paths of the first copy of m that arrives at B and that of its duplicate, m', overlap over every sub-path of length $2\delta + 1$ (i.e., m can at most bypass δ brokers and m' can bypass a different set of δ brokers, thus leaving one broker in common). As a result, a variation of the sender-assigned detection scheme can be developed that requires brokers to track highest sequence numbers assigned only by their neighbors within distance D_{SEQ} (such that $D_{SEQ} \geq 2\delta + 1$).

Algorithm: We use sequence numbers that are composed of two parts: $seq = \langle seqBrkr : seqVal \rangle$, where $seqVal$ is an incrementally updated integer value assigned by Broker $seqBrkr$ to a message m upon arrival of its *first* copy. Furthermore, m carries a vector of sequences of length D_{SEQ} assigned by brokers along its propagation path. Our duplicate detection algorithm is shown in Figure 6 and can be broken down into three main procedures:

- $UPDATE\_SEQ\_VEC\_SENT\_TO(m, X)$ is called when m is being sent from Broker B to Broker X and updates $m.\mathbf{SeqVec}$ by inserting B' generated sequence number $seq_m(B)$ and adding $\langle \perp : B_i \rangle$ for every broker B_i in between B and X that is being bypassed;
- $UPDATE\_HIGHEST\_SEQUENCES(m)$ updates the highest sequence numbers of neighboring brokers according to the values in $m.\mathbf{SeqVec}$;
- Finally, $IS\_DUPLICATE(m)$ is called upon arrival of every message, m, and determines if a copy of m has been received before: m is a duplicate *iff* at least one of its sequence numbers, $seq_m(B_i)$, does not succeed $highest(B_i)$. If no previous copies were received, B generates $seq_m(B)$ for m.

```
1: function IS_DUPLICATE(m)                          ▷ m is a pub or sub
2:     for all seq ∈ m.SeqVec && seq ≠ ⊥ do
3:         B ← seq.broker
4:         if highestSeqVal(B) ≥ seq.val then
5:             dup ← true
6:             break
7:     UPDATE_HIGHEST_SEQUENCES(m.SeqVec)
8:     if dup == false then
9:         seq_m(B) ← New_seq()    ▷ non-duplicate, generate new sequence
10:    return dup
11: procedure UPDATE_HIGHEST_SEQUENCES(SeqVec)
12:    for all seq ∈ SeqVec && seq ≠ ⊥ do
13:        B ← seq.broker
14:        if seq.val > highestSeqVal(B) then
15:            highestSeqVal(B) ← seqVal
16: procedure UPDATE_SEQ_VEC_SENT_TO(m, X)
17:    for all Y ∈ P(B, X) do
18:        if Y == B then
19:            Insert seq_m(B) into m.SeqVec
20:        else
21:            if Y ≠ X then          ▷ Broker Y is being bypassed
22:                Insert ⟨Y : ⊥⟩ into m.SeqVec
```

Figure 6. Duplicate detection algorithms at broker B

VII. Recovery Procedure

Recovery is the process of delivering missed subscriptions to brokers on or beyond a partition and has two forms: The first form is called *full recovery* and involves a broker that has previously experienced a transient crash failure and has lost its SRT in its entirety. The second form however, is called a *partial recovery* and involves a non-crashed recovering broker that has merely became unreachable from another broker(s). In this case, the broker's SRT is likely to be only partially out of sync from its neighboring brokers. In what follows we elaborate on both types of recovery.

A. Partial Recovery

Partial recovery is initiated upon activation of a new session. More specifically, once Broker B activates $\mathcal{S}_{B,R}$ to Broker R, a synchronization process is triggered during which B, called the *synch-point*, sends subscriptions in its SRT that are not accepted by R, called the *recovering broker*. Note that recovery is asymmetric, i.e., R does not act as synch-point for B, unless $\mathcal{S}_{R,B}$ is activated.

Partial recovery has five steps: *(i)* B notifies R that its session to R is now active; *(ii)* R replies to B by sending a summary of the subscriptions it has already accepted; *(iii)* B uses R's reply to transfer those subscriptions that it has accepted, but are missing at R; *(iv)* R receives the subscriptions from B and propagates them through parts of the network; *(v)* B removes pid of partitions that it is their detector from its PT and notifies its neighbors over all its established sessions. Description of each step is as follows: Step *(i)* is obvious. In step *(ii)* in the above process, Broker R uses sequence numbers to construct $\mathbf{SEQ}_{srt}(R)$ in order to summarizes the subscriptions currently present in its SRT. For this purpose, R first identifies set of all brokers, X_i, that are downstream from broker B (i.e., $B \in \mathbf{P}(R, X_i)$) and R has seen X_i's sequence number in a message. For all such brokers, R inserts the highest sequence number seen from X_i into $\mathbf{SEQ}_{srt}(R) = \{highestSeq(X_i) | B \in \mathbf{P}(R, X)\}$. In step *(iii)*, Broker B receives $\mathbf{SEQ}_{srt}(R)$ and uses a procedure similar to $IS\_DUPLICATE()$ in order to identify missing subscriptions at R. More specifically, this is done by substituting sequence numbers in $highestSeq$ with the ones in $\mathbf{SEQ}_{srt}(R)$ and running $IS\_DUPLICATE()$ as before. However, for this to work the subscriptions must be propagated over a stronger form of legitimate propagation that allows δ bypassed brokers in sub-paths of length $3\delta + 1$. To clarify the need for the stronger form we use Figure 7 that illustrates arrival of two copies of a single subscription to a recovering Boker R first via normal propagation (s_1 in the figure) and later during recovery via the synchronization point S (s_2 in the figure). We would like to enable the Broker R to detect s_2 as a copy of s_1 in much the same way as before.

Brokers R and S may be located up to $\delta + 1$ hops away and as a result all sequence numbers corresponding to the intermediate brokers that may be present in s_1 will not be useful (since it is not present in s_2). Furhtermore, since s_2 is already accepted at S week legitimacy is only enforced over the primary path from the subscriber to S. The figure illustrates the case that the set of visited brokers by s_1 and $_2$ over this path can be empty within distance $2\delta - 1$ of the synchronization point. However, this set is certainly non-empty within distance 2δ of S. Adding the maximum distance between S and R (which is $\delta + 1$) we arrive at $D_{SEQ} = 3\delta + 1$ which is the required length of sequence vectors that allows Broker R to identify a common (non-suceeding) sequence number in both messages. It is important to note that only subscription messages need strong form of legitimacy (with $D_{SEQ} = 3\delta + 1$) and for publications the weaker form is sufficient (with $D_{SEQ} = 2\delta + 1$).

At the end of step *(iii)*, B transfers the set of subscriptions identified to be missing at R. In step *(iv)*, Broker R first eliminates any potential duplicates (duplicates may still arrive as a result of concurrent recoveries) and then processes each received subscription, s_i, issued by source Broker S_i as follows:

- For each of R's active sessions, $\mathcal{S}_{R,X_j}$, to Broker X_j downstream of R (i.e., R on $\mathbf{P}(S_i, X_j)$), s_i is sent to X_j and R waits to receive confirmation c_{s_i} from X_j;
- When c_{s_i}s arrive at B they might be tagged by a (possibly empty) set of partition ids, T', corresponding to new partition barriers downstream of R. Furthermore, T' might be different from the s_i's tags stored at B. At this point, *(a)* R accepts s_i with tags T'; *(b)* sends $c_{s_i}^{T'}$ to B and $s_i^{T'}$ is sent over any

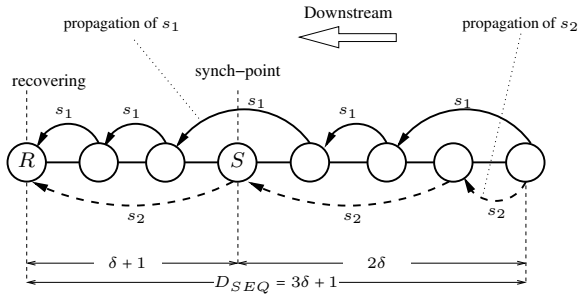


Figure 7. Strong legitimacy enables duplicate detection during recovery: s_1 and s_2 are two copies of the same subscription propagated over disjoiont paths - Note that while propagation of s_2 to S is legitimate, once accepted by S it can be sent during recovery to R bypassing up to δ additional brokers. Strong legitimatcy ensures that R can detect duplicates by using sequence numbers assigned to the subscription prior to arrival at S.

active session to broker Y on $\mathbf{P}(R, B)$ (similar to upstream subscription propagation). Processing at Brokers B and Y continues as follows:

- Broker B that receives $c_{s_i}^{T'}$ compares T' with the original set of tags, T, associated with s_i in its SRT. If $T' \not\subset T$, then B replaces $pid_k \in T$ that R was on with the $pids \in T'$. Furthermore, B generates a special type of publication that has T' as its payload and matches all subscriptions that are tagged with pid_k. This publication is called a *partition recovery notifier* and is forwarded reliably in the network. Subscription source brokers, S_i, that receive this publication replace the pid_k with those in T'.
- When Y receives $s_i^{T'}$, it is processed as if it has been received as part of a recovery procedure where R acts as a synch-point and Y is a recovering broker.

B. Full Recovery

Full recovery takes place when a crashed broker, R, restarts. We require a restarted broker to be able to restore its Δ-neighborhood from stable storage or by querying a network management service aware of this information. The broker then proceeds to activate sessions to its immediate neighbors, N_i, in the primary tree. If successfull, activation of each such link will cause N_i to also activate a session to R and initiate a partial recovery to R, i.e., N_i acts as a synch-point. Once all parital recoveries over all active sessions of Broker R end, the recovery is finished.

We now prove that our P/S system maintains reliable publication delivery in presence of δ concurrent failures.

Lemma 1. *If P is the source broker of a publication p that is delivered to a matching subscriber s at Broker S, then P has accepted s prior to generating p.*

Proof: Due to the way that the publication forwarding algorithm works, it is clear that delivery of p to the subscriber requires s to be accepted at S. Let T be the set of tags carried by c_s messages that lead to s being accepted at S. We can have two possibilities: If P is not on or beyond a barrier whose tag appears in T then s must have successfully propagated to P and P must have subsequently accepted and confirmed s. The alternative is when P was on or beyond a barrier and thus may not have received (nor confirmed) s during subscription propagation. In this case, the corresponding pid must be included in one of the c_s received at S. Since P is downstream of $B = pid.detector$, and it is only broker B that can tag c_s with pid, we can conclude that at the time c_s was sent from B, pid must have already been present in B's PT. This implies that all upstream brokers of B within distance D_{PT} must have received and added pid to their PT prior to receiving c_s (Property 1). Now consider the forwarding of publication p from P to S. Since p's arrival at S follows s's acceptance at S, and processing of subscriptions, publications and confirmations take place in FIFO order at all brokers, we can conclude that at no broker along $\mathbf{P}(P, S)$ publication p was processed before c_s (Property 2). Let $X \neq B$ be the broker on $\mathbf{P}(P, B)$ that sends p to a broker on $\mathbf{P}(B, S)$. Since X can bypass at most δ brokers, p must arrive at a broker Y that is within distance 2δ of B. By Property 1 and Property 2, we know that Y is already aware of pid and thus tags p (based on the publication forwarding algorithm). From this point on, the forwarding of p carries the pid tag. At Broker S, the tag is checked against tags in c_s^T and since pid is a common tag, p is not included in the reliable publication flow that is delivered to the subscriber.

So far, we only considered the case that the partition corresponding to pid has not recovered. To consider such scenario, we examine two cases: In the first case, the pid tag of the subscription is removed at S due to receipt of a recovery publication that actually delivered s to P (i.e., P accepted s). Prior to the time that P accepts s however, the previous argument still holds and thus all publications from P are tagged and excluded from the reliable publication flow. In the second case, the pid tag of the subscription at S is replaced by pid' since re-propagation of s during a recovery reached a new barrier identified by pid'. In this case, publications generated from P will be tagged with the new partition id, pid', and will be excluded at S. ■

Lemma 2. *If p_1 and p_2 are two publications from P that are consecutively delivered to local matching subscriber of S, then: there is no intermediate publication p' that was generated at P after p_1 and before p_2 that matched s.*

Proof: The proof is by contradiction: Assume such p' exists. By Lemma 1, P must have accepted s prior to issuing p_1. As a result, P must have forwarded p' to another broker B_1 on $\mathbf{P}(P, S)$. Now assume B_i is the first broker on this path that receives p' but has not yet accepted s. Such broker must exist, otherwise, p' would have been forwarded hop-by-hop to S. If B_{i-1} is the broker that sends p' to B_i at time t_p, let t_s be the time B_{i-1} accepts s, and t_p be the time that p is sent from B_{i-1} to B_i. The session $\mathcal{S}_{B_i,B_{i-1}}$ must have last activated at a prior time, $t_a < t_p$. From the assumption, we know that $t_a < t_p$ and $t_s < t_p$. Now consider the following timing possibilities: (i) *If $t_a < t_s$, then B_i must have been on a partition island. Due to the way subscription propagation algorithm works, Broker B_{i-1} must send s upstream towards the B_i on the island at t_s. Furthermore, B_i must have immediately accepted s prior to receiving t_p;* (ii) *If $t_a > t_s$ then the session $\mathcal{S}_{B_i,B_{i-1}}$ is activated while s is in B_{i-1}'s SRT. Activation of the session at t_a must initiate the recovery procedure in which B_{i-1} sends s as a missing subscription to B_i. Since B_i is on a partition island, it immediately accepts s. Finally, since recovery takes place prior to transmission of publications, then Broker B_i must already have accepted s before t_p. This concludes the proof.* ■

Theorem 1. *The P/S algorithms presented uphold the reliable delivery specification.*

Proof: We need to show that for any two publications p_1 and p_2 delivered consecutively to a local subscriber of Broker S, the delivery order is the same as generation order and no intermediate publication p' is generated by the same source. The proof of ordered delivery follows directly from the fact that link transmission and all processing at brokers take place in FIFO order. The proof of gap-less delivery comes directly from Lemma 2. ■

VIII. Other Considerations

Upholding propagation legitimacy: If there are more than δ concurrent failures, the legitimate propagation property may be violated. Such a scenario can be detected at a broker simply by checking the **SeqVec** of messages. If the number of $\perp$ values in the *outgoing* copy of a message is more than δ then the message

Parameter	Value	Description
δ	-	Configuration parameter
Δ	$\delta+1$	Knowledge of brokers neighborhood
D_{PT}	2δ	Partition info msgs propagation distance
D_{SEQ}	$3\delta+1$	Size of sequence vectors

Table I
SYSTEM PARAMETERS

violates legitimate propagation requirements. Assume Broker B detects such a condition when sending message m generated by source Broker S over an active session $\mathcal{S}_{B,X}$. In order to maintain *hard* reliability guarantees, B must stop sending m and subsequent messages destined to Broker X and wait until one of the following conditions take place: *(i)* If a new session, say $\mathcal{S}_{B,Y}$, becomes active such that $Y \in \mathbf{P}(B,X)$ then X becomes deactive and the legitimacy condition must be re-evaluated for messages sent to Y; *(ii)* If a recovery takes place on $\mathbf{P}(S,B)$, new copies of m that arrive may help resolve the situation if they have bypassed fewer brokers; *(iii)* Finally, if X is an edge broker and $\mathcal{S}_{B,X}$ fails then the situation is also automatically resolved.

Determining D_{PT}: In order to determine the value of D_{PT} we use the legitimacy property. Remember that the role of the brokers within distance D_{PT} of a partition detector is to tag publications that arrive from $pid.pnodes$ with pid. If s is a subscription that was confirmed by c_s^{pid} then consider the *last* $2\delta+1$ brokers on $\mathbf{P}(S, pid.detector)$. Since s was propagated legitimately, then at least $\delta+1$ of these brokers have accepted s with tags that include pid. Since forwarding of a publication, p, that matches s must also be legitimate (in the reverse direction), we can conclude that p will pass through at least one broker that has previously accepted s along with pid. This implies that $D_{PT} = 2\delta$ is a sufficiently large value (note that the partition detector is the $(2\delta+1)$-th broker). In other words, if every broker within distance 2δ of a partition detector stores the partition id in its PT, then all publications that match s and are sent from a broker on $pid.pnodes$ will be correctly tagged. Table I summarizes all system parameters and their values.

P2P settings: The approach presented in this paper provides the algorithmic framework for a P/S system that can tolerate up to δ concurrent broker or link failures while maintaining service reliability at all times. This is a hard guarantee and is most applicable for large-scale enterprise-grade or datacenter level dedicated messaging infrastructures in which the demand for high throughput and highly reliable messaging is a primary concern and broker churn is very low (mainly due to failures). On the other hand, a peer-to-peer P/S deployment over the Internet lacks a dedicated broker infrastructure and may rely on clients to act as network brokers. Furthermore, clients may leave the system at any time thus incurring high churn that is likely to exceed δ. In these circumstances, although our approach can still tolerate departure of large number of peers (Section IX), the long-term operation of the system requires a special mechanism to *replace* brokers that have permanently left the system with the ones that join. Consideration of such techniques is part of our future work.

IX. EVALUATION

In this section, we focus our analysis on the performance of the system once more than δ brokers have failed (or left the system).

A. Network connectivity after failures

In our approach brokers in a network configured with parameter δ are aware of their $(\delta+1)$-neighborhood. As a result, occurrence of more than δ adjacent failures in a chain cannot be bypassed and the brokers downstream of each end of the chain become disconnected. Figure 8 illustrates how many concurrent failures impact broker-to-broker connectivity for various values of δ. Each data point is the average of 100 simulation runs in a network of 1000 brokers. In each run the number of brokers shown on the x-axis have been randomly chosen to fail. The y-axis shows the percentage of broker pairs that will be disconnected in networks with primary tree fanout of 3 (left graph) and 7 (right graph). It can be seen that in both cases increasing δ improves network connectivity by reducing the probability of occurrence of $\delta+1$ failures in a chain.

B. Impact of failures on network throughput

Next, we investigated how the increased network connectivity translates to actual delivery of publications in presence of failures in a real execution scenario. For this purpose, we carried out real experiments using our implementation of the fault-tolerant publication forwarding algorithm on University of Toronto's Scinet cluster computer. We deployed a large network of 500 brokers such that each broker is assigned a dedicated CPU core (Intel Xeon) at 2.66 GHz and has access to 800 MB of memory. Since our aim is to examine all pair-wise broker path interconnections, we chose publication and subscription workloads with a 100% matching distribution, i.e., each publication is broadcast to all other brokers. Figure 9 illustrates the publication delivery count (y-axis) in a 120s measurement interval after the specified number of brokers have failed (x-axis). It can be seen that the system throughput for $\delta = 3$ is just within $2-3\%$ of expected deliveries had there be no disconnections. Furthermore, lowering δ from 3 to 1 degrades system's throughput by steps of about 4%. Finally, for $\delta = 0$, the system suffers from vast throughput degradation due to widespread disconnections.

C. Size of brokers' Δ-neighborhoods

Increasing δ improves the network's resilience to failures, but it comes at the cost of increased amount of state to be maintained. Figure 10 shows this effect by illustrating the distribution of the size of brokers' $\delta+1$-neighborhoods in a network of 1000 brokers. The left diagram corresponds to a network with primary tree fanout of 3, and it can be seen that almost all brokers' neighborhoods include less than 100 brokers. On the other hand, in the right diagram that corresponds to network with a primary tree fanout of 7, a few brokers (located in the center) may have large neighborhoods for the case of $\delta = 3$. This suggests that a smaller δ may be more feasible for networks with a high fanout primary tree.

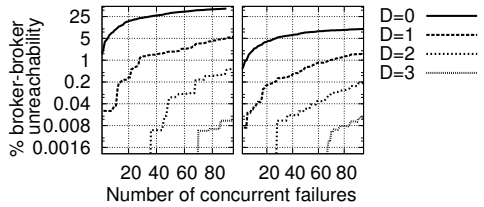


Figure 8. The network connectivity after failures for a network of size 1000, and primary tree fanout of 3 (left) and 7 (right).

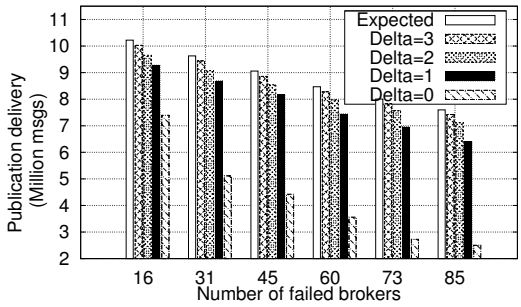


Figure 9. Publication delivery during a 120s measurement interval in a 500 broker network with different number of failures.

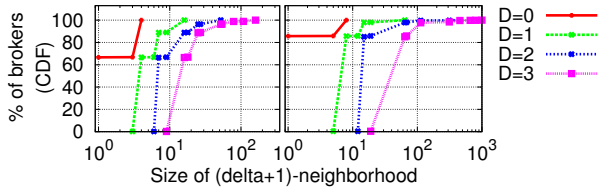


Figure 10. Distribution of brokers' Δ-neighborhood size for a network of 1000 brokers with primary tree fanout of 3 (left) and 7 (right).

D. Number of Active Sessions

The number of active communication sessions that a broker maintains may increase as neighboring brokers fail and need to be bypassed. To analyze this effect we performed simulation runs for networks of size 1000. Figure 11 illustrates the distribution of the number of brokers' active sessions after occurrence of 16 and 60 failures in networks configured with $\delta = 3$. The top and bottom diagram corresponds to the case where the brokers fanout in the primary tree is 3 and 7, respectively. It can be seen that the number of active sessions remains very low for the majority of brokers.

X. Related Work

In this section, we discuss the related work. The basic IP multicast protocols provide only a best-effort quality of service (QoS). There have been proposals to implement reliable versions of IP multicast protocol. For instance, OTERS [12] proposes *subcasting* to improve reliability via retransmissions. Their network management relies *multicast route backtracking* which is closely related to our notion of brokers' Δ-neighborhoods, and how the subscriptions routing information is used for forwarding. However, their approach is not immediately applicable to a content-based P/S system with selective publication delivery.

Authors in [13] propose to overcome this challenge by mapping publications onto multicast groups as the underlying routing scheme. While routing between brokers uses the mapped publication, delivery at final brokers uses the original client subscriptions in a content-based manner to determine whether the publication matches the subscription. However, group communication techniques provide stronger reliability and ordering guarantees than our approach and thus incur more overhead.

Overlay-based routing techniques provide another body of work that can be applied to build fault-resilient P/S systems. For instance, *Resilient Overlay Network (RON)* [14] is an architecture that improves the network resiliency to Internet path outages via deploying overlay nodes at various locations throughout the Internet. RON networks can act as the underlying communication substrate of a distributed P/S middleware and improve its resilience to network path failures. However, it is not clear how strict reliability which is the focus of this paper can be implemented atop RON.

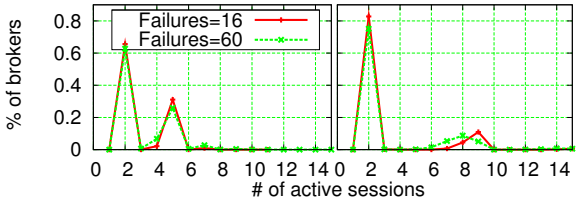


Figure 11. Distribution of the number of active sessions for a network of 1000 brokers with a primary tree fanout of 3 (left) and 7 (right).

Another approach [15] attempts to reconfigure the broker network after a failure. This enables the network to recover from failures by excluding the failed broker from the system. The reconfiguration process however, provides best-effort delivery as it may lead to publication loss, and re-ordering.

Snoeren et al. [16] propose another approach to build a fault-tolerant P/S system, by constructing multiple disjoint forwarding paths between subscribers and publishers in a mesh network. Publications are redundantly forwarded on all available paths. In the face of concurrent failures, publication loss is avoided even if one forwarding path remains unaffected. However, this approach incurs additional bandwidth due to redundant publication forwarding.

XNET [17] proposes two schemes to deal with broker failures. Operation of their crash/failover scheme is similar to our system configured with $\delta = 1$ and allows brokers to establish direct connections to downstream brokers of a failed neighbor. In this regard, our approach can be viewed as a generalized extension which has the additional advantage of ensuring reliable publication delivery in presence of concurrent failures.

The Gryphon P/S system [18], [19] provides similar publication delivery guarantees (i.e., FIFO ordered and gap-less) to our system by introducing the concept of virtual brokers which consist of a set of physical replicas. In the face of failures, replicas act as primary/backups. However, the approach to ensure gap-less publication delivery in [19] requires global knowledge. Furthermore, in [20], we compared a crash-tolerant version of our approach against the replica-based technique of [18] and showed that after failures, the remaining brokers in our approach experience better load stability.

XI. Conclusions

In this paper, we developed reliable distributed P/S algorithms that tolerate concurrent failure of brokers and links. Our approach address the problems of subscription propagation, publication forwarding, and broker recovery. In presence of up to δ failures, our system can maintain hard delivery guarantees. We also evaluated the performance of our system when the number of concurrent failures exceeds δ. Our results showed that after relaxing the legitimacy requirement, the system can still reliably deliver 97% of publications in presence of failure of as much as 17% of brokers.

References

[1] "PubSubHubbub," http://code.google.com/p/pubsubhubbub/.
[2] I. Rose *et al.*, "Cobra: Content-based filtering and aggregation of blogs and rss feeds." in *NSDI*. USENIX, 2007.

[3] G. Li *et al.*, "A distributed service-oriented architecture for business process execution," *TWEB*, vol. 4, 2010.
[4] "Tibco Enterprise Service Bus," http://www.tibco.com/resources/software/esb/tibco_esb_datasheet.pdf.
[5] M. Sadoghi *et al.*, "Efficient event processing through reconfigurable hardware for algorithmic trading," in *VLDB*, 2010.
[6] E. Fidler *et al.*, "The PADRES distributed publish/subscribe system," *ICFI*, 2005.
[7] A. Carzaniga *et al.*, "Design and evaluation of a wide-area event notification service," *Trans on Comp Sys*, 2001.
[8] P. R. Pietzuch and J. Bacon, "Hermes: A distributed event-based middleware architecture," in *ICDCS*, 2002.
[9] G. Cugola *et al.*, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *Trans on Software Eng*, 2001.
[10] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, pp. 225–267, 1996.
[11] A. Cheung and H.-A. Jacobsen, "Dynamic load balancing in distributed content-based publish/subscribe," in *Middleware*, 2006.
[12] D. Li and D. R. Cheriton, "Oters (on-tree efficient recovery using subcasting): A reliable multicast protocol," in *ICNP'98*.
[13] L. Opyrchal *et al.*, "Exploiting IP multicast in content-based publish-subscribe systems," in *ACM Middleware '00*.
[14] D. Andersen *et al.*, "Resilient overlay networks," in *SOSP*, 2001.
[15] G. Cugola *et al.*, "Minimizing the reconfiguration overhead in content-based publish-subscribe," in *SAC '04*.
[16] A. C. Snoeren *et al.*, "Mesh-based content routing using xml," in *SOSP*, 2001.
[17] R. Chand and P. Felber, "XNET: a reliable content-based publish/subscribe system," in *SRDS '04*.
[18] S. Bhola *et al.*, "Exactly-once delivery in a content-based publish-subscribe system," in *DSN*, 2002.
[19] Y. Zhao *et al.*, "A general algorithmic model for subscription propagation and content-based routing with delivery guarantees," 2005.
[20] R. S. Kazemzadeh and H.-A. Jacobsen, "Reliable and highly available distributed publish/subscribe service," in *SRDS '09*.