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 broker machines or communication 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)* publication forwarding; and *(iii)* broker recovery. Finally, we study the effectiveness of our approach when the number of concurrent failures exceeds δ. Through large-scale experimental evaluations with up to 500 brokers, 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 its brokers.**

I. Introduction

Many of today's large-scale distributed systems require reliable many-to-many communication capabilities that go well beyond the basic provisions of underlying network protocols. Examples of such applications include news dissemination services, push-based RSS feed processing [1], [2], job tracking and monitoring applications [3], financial market data distribution [4] and realtime processing systems for algorithmic trading [5]. Developers of these distributed applications often face the challenging task to custom-build scalable and reliable message dissemination platforms.

A *reliable Publish/Subscribe (P/S) system* is well positioned to address this need and relieve developers of much of the hassle of reliable messaging at scale. The P/S model provides a simple and powerful abstraction for information sources to *publish* messages and information sinks to *subscribe* to messages of interest. This is done in a *loosely coupled* manner, via subscribing to pre-defined channels (*a.k.a.* topic-based P/S) or by specifying filtering constraints for published messages (*a.k.a.* content-based P/S).

Reliability in our context refers to *exactly-once and per-source, in-order delivery of publications to matching subscribers*. This establishes an abstract notion of ordered and *gap-less* publication flow between each source and sink. Note that due to the selectivity of subscriptions not all publications from a publisher should be delivered to a subscriber. In other words, our notion of gap-less delivery must be interpreted over the sequence of *matching publications only*. More specifically, the order of published messages must be preserved and for any consecutive publications from a source that are delivered to a subscriber no intermediate matching publication must be published by the source.

In this paper, we focus on fault-tolerance and reliability aspects of content-based distributed P/S systems composed of dedicated *application-level* message routers (called *brokers*) that form an overlay network [6], [7], [8], [9]. To tolerate failures in this architecture, Cugola et al. use overlay reconfiguration techniques [10], Snoeren et al. exploit redundant forwarding paths [11], Gryphon [12] takes advantage of replication, and our earlier work uses brokers' neighborhood knowledge [13]. Among these systems, only the latter two ensure reliable delivery (others aim for best-effort delivery).

In this paper, we extend our previous work on crash-tolerant P/S systems [13] to the general case of node crash and link failures. This extension is not trivial since a *live* but partitioned broker is unaware of the dynamically inserted subscriptions and may perform actions that are seemingly correct but violate reliability. In what follows, we first use a simple use case to motivate applicability of reliable P/S systems and then highlight the challenges of preserving reliability in the presence of link failures.

Motivating example: A Content Distribution Network (CDN) can use *reliable* P/S dissemination to feed fresh content to hundreds of its content servers. For this purpose, content providers publish updated content into a P/S network and each CDN server subscribes to the updates that it must receive and serve to Internet users (Figure 1). This way CDN servers act as subscribing clients of an internal P/S system and receive a gap-less, ordered stream of content updates. For example, subscription s_1`:[provider=BBC, location=USA]`, allows a server to receive updates about United States from BBC News, or s_2`:[provider=BBC, type=video, extra=mostViewed]` matches most-viewed videos from BBC News. Published content comes with content descriptors that must match the subscription of the interested servers and payloads that contain the published data. For instance, p_1 `:[provider=BBC, location=USA, 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 `:[provider=BBC, location=USA, type=HTML, extra=frontPage]` describes the BBC News front page. Now assume that p_1 and p_2 are recently published publications (in-order) and that p_2's HTML payload has a hyperlink to p_1's video file that must be stored on the same server. The per-source, in-order requirement in our reliability specification ensures that servers receive these messages in-order and thus prevents

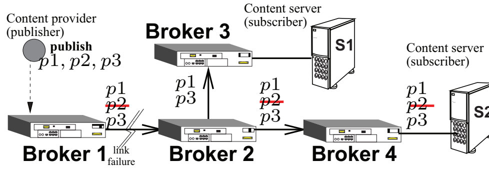


Figure 1. Network partitions may lead to publication loss.

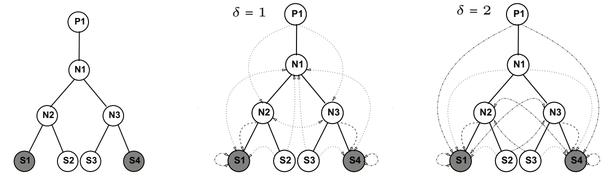
scenarios in which the HTML page has a dangling reference to a non-existing video file on the server.

Challenge of dealing with link failures: Reliable publication forwarding in the broker overlay relies on proper dissemination of client subscriptions. A link failure hinders this process by creating network partitions and preventing live brokers in these partitions from receiving dynamically inserted subscriptions. A partitioned broker that lacks this knowledge has an inconsistent routing state and may *unknowingly* discard matching publications and introduce gaps. As we elaborate below, the decentralized nature of the network and unavailability of global knowledge about the status and reachability of all brokers makes detection and prevention of such scenarios a challenging task.

To construct routing paths in the P/S overlay, subscriptions are propagated throughout the network and each broker stores each subscription and records the previous hop that it was received from. This information is used to forward future publications in the reverse direction towards the matching subscribers. To illustrate the challenges of dealing with link failures, we describe a sequence of link failures and recoveries in the network of Figure 1 which lead to violation of the gap-less delivery requirement: Consider server S_1 issues subscription s_1 which fully propagates throughout the network. However, just before server S_2 issues its subscription s_2, the link between B_1 and B_2 experiences a transient failure. As a result, Broker B_1 does not receive the new subscription. At this point, consider publisher P to publish p_1, p_2 and p_3 (in-order) which all match s_2 while only p_1 and p_3 match s_1. In this situation, Broker B_1 who is only aware of s_1 preserves p_1 and p_3 for future transfer after the link is reconnected. B_1, however, filters out p_2 which matches none of the subscriptions it has received thus far. Later, once the link is re-established, B_1 sends its outstanding messages including p_1 and p_3 towards B_2. Furthermore, since B_2 has already received both s_1 and s_2, it forwards both publications towards B_3 and B_4. Subsequently, p_1 and p_3 are delivered to both subscribers, while server S_2 never receives p_2. As a result, the publication flow delivered to S_2 contains a gap.

Overview of the approach: We introduce δ as a configuration parameter that denotes the maximum number of concurrent failures that the system must be able to tolerate. In our approach, brokers form an initial tree-based application layer overlay upon joining the system.[1] The decision

[1]A registry service may assist brokers to identify the best broker to join.



(a) Routing layer based on the primary tree (b) Forwarding links bypassing one broker (c) Forwarding links bypassing two brokers

Figure 2. Routing layer (solid lines) and forwarding layer (dashed lines).

to adopt a tree overlay is mainly motivated by the in-order publication delivery requirement. More specifically, allowing publications from a source to traverse along multiple arbitrary forwarding paths towards a matching subscriber in a general mesh makes it difficult to re-arrange the messages in their original FIFO order at the receiver. This is especially true in our content-based P/S system in which due to the selectivity of subscriptions, successive publications from a source that match a given subscription may have an arbitrary number of non-matching publications in between.

A pure overlay tree has obvious disadvantages particularly with respect to fault-resiliency. We address these concerns by introducing the notion of neighborhoods and conceptually separating the architecture into a static routing layer and a dynamic forwarding layer (depicted in Figure 2). The routing layer is entirely built upon the initial tree overlay referred hereafter as the *primary tree*. This tree is constructed as brokers join the system and establish links to an existing broker in the system. Moreover, as part of the join operation, a newcoming broker obtains knowledge of its neighbors within distance $\Delta = \delta + 1$ in the primary tree and notifies them of its arrival.

Routing tables at each broker contain references that construct end-to-end forwarding paths within the primary tree structure. To tolerate failures (caused by node crashes or link outages) we allow *controlled reconfigurations* within the primary tree that affect how messages are forwarded. These reconfigurations take place in the forwarding layer and enables a broker to establish new links that *bypass* up to δ of its unreachable neighbors by connecting to other brokers within its Δ-neighborhood. As such, the notion of neighborhoods provides a powerful tool that helps boost network connectivity even in presence of multiple concurrent failures. Figures 2(b) and 2(c) illustrate that the set of potential links between brokers constructs a highly-connected mesh.

Preventing messages from falling into loops in this graph and avoiding forwarding ambiguities requires careful consideration. In our approach, brokers rely on the original primary tree in order to make forwarding decisions after occurrence of failures. More concretely, our algorithms constructs end-to-end routing paths in accordance to the primary tree (routing layer). As failures occur and the network transforms, brokers refer back to the original end-to-end routing paths created over the primary tree to determine how to forward

the messages. This transition is smooth in the sense that no publications are lost.

Finally, we need to ensure that link failures do not result in routing inconsistencies in which incomplete subscription routing information at brokers introduce gaps in publication flows. Our approach to prevent such cases is based on ensuring an important *safety condition*: "*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 safety condition when brokers and links go through cycles of transient failures and subsequent recoveries is challenging. To overcome this challenge, our contribution in this paper is three-fold: *(i)* our subscription propagation algorithm delivers subscriptions to parts of the network that are reachable from the issuing subscriber while bypassing failed or unreachable brokers; *(ii)* our publication forwarding algorithm uses this routing information in order to forward matching publications and to detect and exclude those publications that may compromise reliability; and *(iii)* our recovery procedure ensures that when a failed broker restarts or a broken link is re-established the routing tables of endpoint brokers are in sync. We also experimentally study scenarios in which the number of failures exceed the guaranteed system limit of δ. Using real deployments with up to 500 brokers, our results show that a modest value of $\delta = 3$ is able to ensure reliable delivery of 97% of publications in presence of failure of 17% of brokers.

II. System Model

We assume asynchronous communication links with unknown delivery delays and brokers that may crash and subsequently recover. In this model, a failed link can be thought of as a link that becomes infinitely slow, thus delaying some messages forever, i.e., lose messages. We assume that each broker is equipped with a local failure detector (FD) with eventually strong and eventually accurate properties [14]. The FD can be implemented by a ping mechanism and 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. 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 each session are delivered either with FIFO ordering or are never delivered. In the latter case, A eventually detects unreachability of B.

The primary tree is constructed as brokers join the system one at a time and create a new link to an already existing broker in the system. These initial links are referred to as *primary links*. A *primary path*, $\mathbf{P}(A, B)$, is a sequence of pair-wise, adjacent brokers in the primary tree between A and B. Routing paths are constructed along primary paths and in absence of failures, publications are forwarded between brokers over primary links only. However, occurrence of failures may necessitate live brokers to *bypass* failed links or neighbors and reconnect the overlay using their neighborhood knowledge. Figure 3 illustrates a simple case in which after Broker B becomes unreachable, A establishes a new communication session to C. We say that A's session to C becomes *active* when $\mathcal{S}(A, C)$ is established and A has no other established session to another Broker $B \in \mathbf{P}(A, C)$. We use Act_A to denote the set of A's active sessions. Note that in general, active sessions are not symmetric and as shown in Figure 3, activation of session $\mathcal{S}(A, C)$ does not imply that Broker C also views $\mathcal{S}(C, A)$ as active. Finally, the definition of active sessions also implies that for an arbitrary Broker X, Broker A *may* only have *one unique* active session on the primary path $\mathbf{P}(A, X)$.

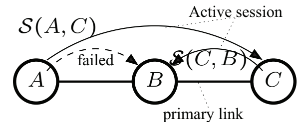


Figure 3. Asymmetry of active sessions between A and C.

III. Overlay Partitions

While the notion of unreachability described above concerns the inability of two brokers to communicate over a *direct* link, we define the notion of *overlay partitions* (or simply partitions for brevity) to capture the inability to *route* messages over paths comprised of multiple brokers. Consider a primary path $\mathbf{P}(src, dst)$. When a broker on this path detects its subsequent neighbor to be unreachable, it attempts to reach out, establish and activate sessions to the brokers 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 unreachable brokers form a partition *island* between src and dst. For example, Figure 4 illustrates that Brokers B_3 and B_4 are bypassed on $\mathbf{P}(B_0, B_7)$ when Broker B_2 activates $\mathcal{S}(B_2, B_5)$. We refer to B_3 and B_4 that are located "*on*" the partition as *partition nodes* denoted with $pnodes$, and refer to Broker B_2 whose FD module has detected the unreachability of the neighboring brokers as the *partition detector*, *pd*. Finally, we say that brokers in subtrees of the last broker on $pnodes$ are "*beyond*" the partition (e.g., Brokers B_5, B_6, B_7).

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 4 where B_2 on $\mathbf{P}(B_0, C_3)$ is unaware of Broker C_2 (since $\Delta = 3$) and thus cannot activate any new sessions to bypass C_1. In this case, we say that a partition *barrier* is formed. More concretely, a partition on $\mathbf{P}(src, dst)$ is a barrier if no active session bypasses the partition's $pnodes$ and thus dst becomes unreachable from src. Note that depending on the location of the pair of src and dst, a given partition can be both an island for some pairs while a barrier for others. For example in Figure 4, the

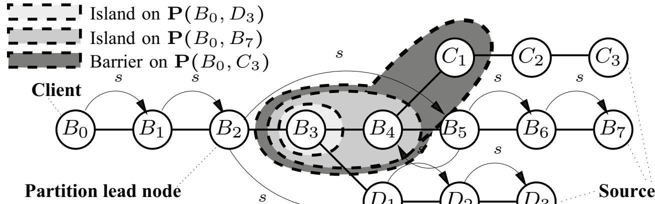


Figure 4. Primary paths (solid lines), active sessions (arrows) and network partitions (highlighted areas) in network with $\Delta = 3$.

partition consisting of B_3 alone is a barrier over primary path $\mathbf{P}(B_0, B_3)$ while it is an island on $\mathbf{P}(B_0, D_3)$.

Regardless of its type, a partition is identified by a unique tuple identifier, $pid = (pd, i, pnodes)$, where pd is the partition detector, i is a monotonically increasing value assigned by pd and $pnodes = \langle B_j, \cdots \rangle$ is the primary path consisting of partition nodes. For example, the partitions shown in Figure 4 are $\{(B_2, 1, \langle B_3 \rangle), (B_2, 2, \langle B_3, B_4 \rangle), (B_2, 3, \langle B_3, B_4, C_1 \rangle)\}$. These partitions are created when B_3's active sessions to Brokers B_3, B_4 and C_1 fail, respectively.

Message types: There are four types of messages in our system: *(i)* publications are data messages generated by publishers; *(ii)* subscriptions specify subscribers' interests; *(iii) confirmation messages* acknowledge delivery of publications or subscriptions to subtrees in the primary tree; and *(iv) partition information messages* carry pids of recently formed or recovered partitions. The first three types may be tagged by brokers with a set of partition ids, $Tags = \{pid_i, \cdots\}$, to indicate that forwarding of these messages was prematurely affected by the partitions indicated in $Tags$.

IV. Overlay Link Management

Each Broker, X, monitors the status of its active sessions. Whenever, an active session, say to Broker F, breaks down, X tries to establish and activate new sessions to neighbors of F. This process takes place in the background and in parallel to the broker's routing and forwarding tasks. Furthermore, unreachability of F creates new partitions whose partition detector is X. We require brokers to keep track of pids whose partition detectors are within distance D_{PT} (we determine this value in Section VII) in a local set data structure called the *Partition Table* (PT). For this purpose, once a partition is formed, the partition detector adds the new pid to its PT and notifies neighbors via its *established* sessions using partition information messages. Receiving brokers similarly update their PT and notify their neighbors.

V. Subscription Propagation Protocol

In this section, we elaborate on the subscription propagation protocol that distributes clients' subscriptions among brokers. Subscribers connect to a broker of choice[2] and issue subscriptions that are propagated throughout the network.

[2]A load balancer [15] may consider different network parameters to assign clients to brokers. This discussion is outside the scope of this paper.

A subscription s (issued by subscribers of source Broker S), that arrives at Broker B contains the following information: *(i)* a unique subscription id (sId) assigned by the subscriber; *(ii)* subscription predicates, $pred$; *(iii)* a trailing portion of the subscription propagation path, $\mathbf{SPath}_s(B)$; and *(iv)* a vector of sequence numbers, $\mathbf{SeqVec}$. Broker B uses $pred$ to determine what publications match s; $\mathbf{SPath}_s(B)$ is a trailing sub-path of $\mathbf{P}(S, B)$ of length D_{SEQ} that is updated as s propagates through the network; and finally, $\mathbf{SeqVec}$ is a vector of length D_{SEQ} that is used for message identification and ordering. We say that a broker *accepts* a subscription when it is added to the broker's Subscription Routing Table (SRT). *Only accepted subscriptions are used for making publication forwarding decisions*. In our system, acceptance of a subscription does not immediately follow its arrival at a broker and there are several steps in between. We now describe these steps in absence and in presence of failures, separately.

A. Subscription Propagation When There are no Failures

Figure 5 illustrates the subscription propagation algorithm and we use line numbers throughout this description to refer to it. 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 (in Lines 2–10) – downstream is the direction away from source S. Since there are no failures B has active sessions to all its immediate neighbors in the primary tree. B inserts the id of these brokers into a set, $Out_s(B)$, called the *outgoing set* and then proceeds to send s to these brokers. $Out_s(B)$ represents the set of neighboring brokers who must first confirm processing of s before B can accept the subscription. For this purpose, B waits to receive acknowledgements from these brokers. 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 (Line 5). Furthermore, a special form of acknowledgement called a *confirmation* message, c_s, is issued upstream to indicate that processing of s is complete in the subtree. c_s contains s's identifier (sId) and in Lines 12-17, upon receipt of c_s, B removes the sender from $Out_s(B)$ and checks whether the set became empty. If so, B accepts s and proceeds to send c_s upstream (Lines 14–17). Otherwise, B 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, upon failure of a communication link to a downstream node $Y \in OUT_s(B)$, Broker B must decide whether to wait for a confirmation or proceed to accept s without Y's confirmation. The former choice may compromise liveness: if Y has crashed permanently then the propagation process is blocked indefinitely. On the other hand, the latter choice may compromise reli-

ability: if Y is live, the it remains unaware of s and may discard publications that match s in the future.

In what follows, we describe how Broker B can make progress in a way that does not compromise reliability. To simplify presentation, we break down the description into two parts that correspond to partition islands and barriers.

B. Subscription Propagation over Islands

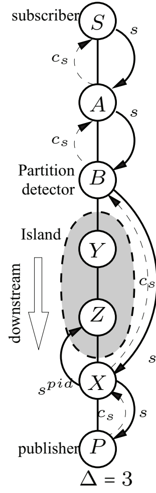


Figure 6. Case of an island.

As mentioned earlier and shown in 6, if B's active session to neighboring Broker Y fails a partition, pid, is formed. Let $\mathbf{P}(S, P)$ be the primary path from S to an arbitrary publisher P located beyond pid. 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 the subscription propagation process by sending s to X (the first reachable broker beyond the partition) thus replacing Y with X in Out_s (Lines 51–55). Then B awaits to receive c_s from X in order to remove X from Out_s. In this case, if $\mathcal{S}(B, X)$ also fails in the meantime before c_s arrives, B replaces X in Out_s with any newly activated session(s) that bypass X. However, if $\mathcal{S}(B, X)$ remains active long enough, X is able to complete propagation of s and send c_s to B. At this point, B removes X from Out_s and accepts s when $Out_s(B) = \varnothing$.

Note that in the above approach, live brokers on the island ($pid.pnodes$) may have neither received nor accepted s. In this situation, however, if a Broker X beyond pid possesses an active session to Broker Z on the partition, it can use this link to propagate s in the partition island. For this purpose, upon accepting s Broker X uses $\mathcal{S}(X, Z)$ to send a copy of the subscription *tagged* with pid. We use the notation s^{pid} to indicate $pid \in s.Tags$. A tagged subscription message is only of interest to brokers located on the partitions that it was tagged with (i.e., $pid.pnodes$) and thus is only sent to these brokers. Upon receipt of a tagged subscription, the subscription is immediately accepted and sent over active sessions to other brokers on the island (Lines 41–45). For example, in Figure 6 s is sent over $\mathcal{S}(X, Z)$. The purpose of this *upstream* subscription propagation is to uphold our safety condition (see Section I). More concretely, we would like to ensure that if an active session from a broker beyond the island to brokers on the island exists, then this session is not used to send a matching publication to the island broker prior to s being accepted by those brokers.

C. Subscription Propagation over Barriers

As shown in Figure 7, no bypassing session exists to deliver s to brokers on or beyond a partition barrier. As a result, all brokers downstream of the partition detector, B, do not receive and accept the subscription. In this scenario, Broker B removes barrier's $pnodes$ from $Out_s(B)$ and accepts s once other outstanding confirmations arrive and Out_s becomes empty. Furthermore, B tags the confirmation message that it sends upstream with

1: **procedure** PROCESS_ARRIVED_SUB(s)
2: **if** CHECK_SUB_TAGS(s) **then return**
3: $Out_s(B)$ ←Active sessions downstream of $\mathbf{SPath}_s(B)$
4: ▷ If an edge broker, accept and confirm immediately
5: **if** $Out_s(B) == \varnothing$ **then**
6: $ACCEPT_AND_CONFIRM(s)$
7: **return**
8: ▷ Send s to downstream brokers
9: **for all** $X \in Out_s(B)$ **do**
10: SEND_SUB_TO($s.clone(), X$)
11:
12: **procedure** ON_SUB_CONFIRMATION_RECEIVE(c_s)
13: $s.Tags \leftarrow s.Tags \cup c_s.Tags$
14: $Out_s(B) \leftarrow Out_s(B) - c_s.sender$
15: ▷ Accept s once all confirmations arrive
16: **if** $Out_s(B) == \varnothing$ **then**
17: $ACCEPT_AND_CONFIRM(s)$
18:
19: **procedure** SEND_SUB_TO(s, X)
20: ▷ If X is beyond a known pid, tag s with pid
21: **for all** $pid \in B$'s PT **do**
22: **if** X is beyond $pid.pnodes$ **then**
23: $s.Tags \leftarrow s.Tags \cup pid$
24: ▷ Update s's seq vector and send to X (see [16])
25: $UPDATE_SEQ_VEC_AND_SENT_TO(s, X)$
26:
27: **procedure** ACCEPT_AND_CONFIRM(s)
28: ▷ Accept s; send s to reachable tagged parition nodes
29: Add s to SRT
30: **for all** $pid \in s.tags$ **do**
31: **if** $\exists X \in Act_B \wedge X \in pid.pnodes$ **then** Send s to X
32: ▷ send confirmations with new barrier ids
33: **for all** $pid \in PT \wedge B == pid.detector$ **do**
34: **if** $pid.pnodes \cap \mathbf{SPath}_s(B) == \varnothing$ **then**
35: $c_s.Tags \leftarrow c_s.Tags \cup pid$
36: **for all** $X \in s.senders$ **do**
37: Send c_s
38:
39: **procedure** CHECK_SUB_TAGS(s)
40: $returnValue \leftarrow false$
41: **for all** $pid \in s.Tags$ **do**
42: **if** $B \in pid.pnodes$ **then**
43: Accept s
44: Send s over any active session to $pid.pnodes$
45: $returnValue \leftarrow true$
46: **return** $returnValue$
47:
48: **procedure** ON_SESSION_ACTIVATED(X)
49: ▷ Executed upon activation of a session to X
50: **for all** unconfirmed s (in-order) **do**
51: **if** $\exists Y \in Out_s(B) \wedge X \in \mathbf{P}(B, Y)$ **then**
52: $Out_s(B) \leftarrow Out_s(B) - \{Y\} \cup \{X\}$
53: **if** $\exists Y \in Out_s(B) \wedge Y \in \mathbf{P}(B, X)$ **then**
54: $Out_s(B) \leftarrow Out_s(B) \cup \{X\}$
55: **if** $\nexists Z \in Out_s(B) \wedge Y \in \mathbf{P}(B, Z) \wedge \mathcal{S}(B, Z) \notin Act_B$ **then** $Out_s(B) \leftarrow Out_s(B) - \{Y\}$

Figure 5. Subscription propagation algorithm executed at Broker B

the pid of the barrier(s) whose node(s) have not confirmed s. We use the notation c_s^{pid} to denote $pid \in c_s.Tags$. When upstream brokers accept s along with tags in c_s and issue their own confirmations, they propagate the pids that were received in confirmation messages from downstream brokers. Once all confirmations arrive at source Broker S, it accepts s and stores $c_s.Tags$ together with the subscription message in its SRT. The purpose of the tags in SRT is to indicate that those publishers whose source Brokers are on or beyond the barrier have not accepted s and thus may have unknowingly discarded other matching publications. As a result, publications from brokers on or beyond these barriers are *not* safe to be delivered to the subscriber even though they match the subscription predicates.

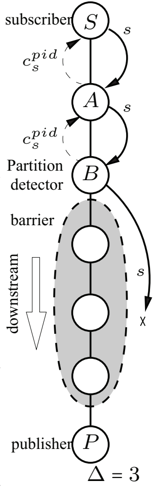


Figure 7. Case of a barrier.

VI. Publication Forwarding

The goal of the publication forwarding algorithm described in this section is to deliver a publication, p, to a subscriber with subscription s only if the following conditions are satisfied: *(i)* p matches $s.preds$ (*matching condition*); and *(ii)* p's source broker and all other brokers along its propagation path had already accepted s prior to forwarding p (*safety condition*). Our approach in this paper allows brokers to evaluate this safety condition by examining the locally available parition information (see Section IV).

We now elaborate on the forwarding algorithm using Figure 8 and use line numbers to refer to it. Forwarding comprises of five steps: queueing, barrier checking, matching, routing, and cleanup. The queueing step (Lines 2–10) first determines whether p has been received before (duplicate detection is described in Section VII) and appends p to a local FIFO message queue in order to preserve the order of messages. *This copy is kept until B ensures successful delivery of* p *to all downstream subscribers* whose subscriptions are currently accepted by B. The barrier checking step involves determining whether p's source broker is on or beyond any of the barriers currently known to B (stored in its PT). If this is the case, B tags p with the corresponding pid(s) (Lines 11–12). We use the notation p^{pid} to denote $pid \in p.Tags$. For subscriptions whose propagation was previously blocked by the same partitions (i.e., subscription confirmations were tagged with *identical* pids), the publication tags indicate possibility of violation of the safety condition. This is due to the fact that we cannot assure that p has only been forwarded by brokers who had accepted the corresponding subscriptions. As a result, we choose to exclude p from the reliable publication flow delivered to those subscribers. This exclusion takes place at the final brokers that the corresponding subscriber are connected to and not at interim brokers along the path.

After barrier checking, B computes $Match_p(B)$ and identifies subset of its *accepted subscriptions* that match p. For each matching subscription s in this set, B determines its active session on $\mathbf{SPath}_{s.src}(B)$ and adds the corresponding broker to p's outgoing set, $Out_p(B)$ (Lines 13–18). Similar to subscription propagation, p is sent to the brokers in the outgoing set and Broker B awaits to receive a confirmation, c_p, indicating successful delivery of p to matching subscribers in the corresponding subtree. For each confirmation that arrives at B, the sender of the confirmation is removed from $Out_p(B)$, and B checks whether the set has become empty (Lines 19–23). If so, B discards p from its message queue (cleanup step) and sends a confirmation to the broker(s) where copies of p had come from (Lines 25–27). In the event of a failure, B activates new sessions to bypass its unreachable neighbor in a similar way to subscription propagation. The only difference is that brokers with local matching subscribers are *not* removed from $Out_p(B)$ (Lines 42–48). This is crucial to prevent publication loss.

If p arrives at a broker with a local matching subscription, then it must first be determined whether delivery of p is safe (note that the subscription must have already been accepted at the source broker). For this purpose, $p.Tags$ is examined against the pids that were received in $c_s.Tags$. If p and c_s have no pid tags in common, then p has crossed no barriers along its path and can be safely delivered to the matching subscribers (Lines 32–38). Otherwise, if there is an identical pid in both sets, then p has been published by a source broker, P, located on or beyond a barrier. Since P is likely not to have accepted s prior to sending p, delivery of p to the client is not safe and may lead to gaps (see Section I for case scenario where delivery of publication violates reliability).

VII. Duplicate Detection

To ensure *exactly-once* delivery, brokers must identify and elimite duplicate messages. To illustrate this need, consider a simple case in which brokers A, B and C form a chain in the primary tree and publication p is sent first from A to C via B. If session $\mathcal{S}(A, B)$ fails before B sends c_p to A, then A tries to re-send p directly to C over its newly activated session $\mathcal{S}(A, C)$. Since C may have already received p from B, retransmission of p leads to duplicate messages at C.

There are a number of possible approaches for duplicate detection. Since links are FIFO, a simple scheme can use a standard source-assigned message sequence numbers and require brokers to keep track of the highest sequence assigned by each source. As an alternative, we developed a new duplicate detection scheme that relies on message sequence numbers assigned by nearby brokers only. This reduces the amount of state maintained at each broker and relies on an important property that we define below:

Legitimate propagations: Propagation of message m from source S, to a destination D, is *legitimate* if in all primary sub-paths of $\mathbf{P}(S, D)$ of length $2\delta + 1$, m bypasses no more

```
1:  procedure ON_PUB_RECEIVE(p)
2:      if Is_Duplicate(p) then
3:          p' ← Queue.Find(p)
4:          if p' == null then
5:                  ▷ Processing of previously received p' has been
    completed. send confirmation to the sender.
6:              Send c_p to X;   return
7:          else
8:              p'.senders ← p'.senders ∪ p.sender
9:      else
10:         Queue.append(p)
11:     for all pid ∈ PT s.t. p.source on/beyond pid.pnodes do
12:         Tag p with pid
13:     Match_p(B) ← match p against accepted subs in SRT
14:     CHECK_LOCAL_SUBS(P)
15:     for all s ∈ Match_p(B) do
16:         if ∃S(B,X) ∈ Act_A and X ∈ SPath_s(B) then
17:             Out_p(B) ← Out_p(B) ∪ X
18:             SEND_PUB(p.clone(), X)
19: procedure ON_PUB_CONFIRMATION_RECEIVE(c_p)
20:     p ← Queue.Find(c_p)
21:     Out_p(B) ← Out_p(B) − c_p.sender
22:     if Out_p(B) == ∅ then
23:         CLEANUP_AND_CONFIRM_PUB(P)
24: procedure CLEANUP_AND_CONFIRM_PUB(p)
25:     Queue.remove(p)
26:     for all X ∈ p.senders do
27:         Send c_p to X
28: procedure SEND_PUB(p, X)
29:      ▷ Update seq vector before sending (see [16] for details)
30:     UPDATE_SEQ_VEC_AND_SENT_TO(p, X)
31:     Send p to X
32: procedure CHECK_LOCAL_SUBS(p)
33:     for all s ∈ Match_p(B) do
34:         if s.source == B then
35:             if c_s.Tags ∩ p.Tags == ∅ then
36:                 Deliver p to local subscriber(s)
37:                 if Delivery to local subscribes successful then
38:                     Match_p(B) ← Match_p(B) − {B}
39: procedure ON_SESSION_ACTIVATED(X)
40:      ▷ Upon activation of a session to X and after recovery
41:     for all unconfirmed p in Queue do
42:         for all Y ∈ Out_p(B) ∧ X ∈ P(B,Y) do
43:             Out_p(B) ← Out_p(B) ∪ {X} − {Y}
44:         for all Y ∈ Out_p(B) ∧ Y ∈ P(B,X) do
45:             if ∃Z ∈ Match_p(B) ∧ X ∈ P(Y,Z) then
46:                 Out_p(B) ← Out_p(B) ∪ {X}
47:             if ∀Z ∈ Match_p(B) s.t. Y ∈ P(B,Z) ∧
    ∃S(B,W) ∈ Act_B s.t. Y ∈ P(B,W) then
48:                 Out_p(B) ← Out_p(B) − {Y}
```

Figure 8. Publication forwarding algorithm executed at Broker B

than δ brokers. This definition has two important properties: First, m can still bypass δ failed or unreachable brokers and thus the system remains δ-fault-tolerant; and second, m's legitimate propagation ensures that m is forwarded by a *majority of brokers along the primary path* between S and D. Intuitively, our duplicate detection algorithm (described in detail in the extended version of the paper [16]) exploits the latter property in order to ensure that propagation path of a first copy of m that arrives at destination Broker D and that of its duplicate, m', intersect 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 keep track of the highest sequence numbers assigned only by their neighbors within distance $D_{SEQ} = 2\delta + 1$.

VIII. Recovery Procedure

Recovery is the process of delivering missing subscriptions to brokers on or beyond a partition and has two forms: *Full recovery* involves a broker that has previously experienced a transient crash failure and as a result has lost its entire SRT; and *partial recovery* involves a recovering broker that has merely became temporarily unreachable from other neighboring broker(s). In the latter case, the broker's SRT is likely to be only partially out of sync. 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 S activates $\mathcal{S}(S,R)$ to Broker R, a synchronization process is triggered during which S transfers a subset of subscriptions in its SRT that are not accepted by R. In this process, S is called the *sync-point* and R is referred to as the *recovering broker*. Note that since session activation is not necessarily symmetric, the recovery procedure may take place only in one direction.

Partial recovery has five steps: *(i)* S notifies R that its session to R is now active; *(ii)* R replies to S by sending a *summary* of the subscriptions it has already accepted; *(iii)* S uses R's summary to identify and transfer those subscriptions that it has accepted but are missing at R; *(iv)* R receives the subscriptions from S and accepts them after propagating them to its downstream network; *(v)* partition information is updated within distance D_{PT} of S.

We now elaborate on these steps: Step *(i)* is obvious. In step *(ii)*, Broker R summarizes its previously accepted subscriptions by constructing a set of subscription identifiers ($\{sId\}$), which are currently present in R's SRT. In step *(iii)*, this set is transferred to S which examines its own accepted subscriptions and identifies the subset that are currently missing at R's SRT. The subscriptions corresponding to these identifiers are then transferred from S to R. In step *(iv)*, R processes each such received subscription s_i as follows (assume s_i originated by source Broker S_i):

- For each of R's active sessions, to a downstream Broker X_j (i.e., $R \in \mathbf{P}(S_i, X_j)$), s_i is sent to X_j which propagates the subscription in the same way as described in Section V. Furthermore, R waits to receive confirmation $c^j_{s_i}$ from X_j;
- Confirmation message $c^j_{s_i}$ that arrives at R may have been tagged by some pids that correspond to new barriers downstream of R. Once all confirmations arrive, R first accepts s_i with tags $T' = \bigcup(c_{s_i}.Tags)$ and then:

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 (details in [16])

Table I
SYSTEM PARAMETERS

1) R sends $c_{s_i}^{T'}$ to S; and S compares T' with the original set of tags, T, associated with s_i in its SRT. S replaces $pid_k \in T$ such that $R \in pid_k.pnodes$ and adds $pids \in T'$. Furthermore, S issues a special publication message, called a *partition recovery notifier*, p_{rec}, that contains both $\{pid_k\}$ and T'. Publication p_{rec} has a special matching semantic and *matches all subscriptions whose confirmation were tagged with* pid_k. As a result, the publication recovery notifier is forwarded in the network and once it arrives at a subscription source brokers, S_i, the broker replaces pid_k stored in its SRT with those in T'.
2) R sends $s_i^{T'}$ over any active session towards broker Y on $\mathbf{P}(R, S)$ (similar to upstream subscription propagation described in Section V-B). When Y receives $s_i^{T'}$, it is processed as if it had been received as part of a recovery procedure where R acts as a sync-point and Y is a recovering broker.

Finally, in step *(v)* S removes pid_k from its PT, and notifies neighbors within distance D_{PT} about recovery of pid_k so that they also remove pid_k from their PTs. After these updates take place, the state of the network along the forwarding paths that pass through the recovered partition returns back to normal. In other words, brokers within distance D_{PT} of $pid.pd$ have removed pid from their PTs and do not tag future publications. Furthermore, the subscription source brokers have also removed pid from their SRTs and thus will not suppress delivery of future matching publications.

B. Full Recovery

A crashed broker, R, that restarts must go through full recovery. 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 successful, activation of each such link causes N_i to also activate a session to R and initiate a partial recovery to R, i.e., N_i acts as a sync-point. Once all partial recoveries over all active sessions of Broker R end, the recovery is complete.

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

The proof of Theorem 1 can be found in [16].

Determining D_{PT}: To determine the value of D_{PT} we use the legitimacy property. Remember that the role of brokers within distance D_{PT} of a partition detector is to tag publications that arrive from brokers on the partition ($pid.pnodes$) with pid. If s is a subscription accepted at source Broker S with confirmation tag 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 any matching publication p must also be legitimate (in the reverse direction of subscription propagation), we conclude that p passes through at least one broker that has previously accepted s tagged 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, and tags publications that arrive from partition brokers accordingly, then no untagged copy of p can arrive at S and thus p is safely excluded from the reliable publication flow delivered to the subscriber. Table I summarizes all system parameters and their values.

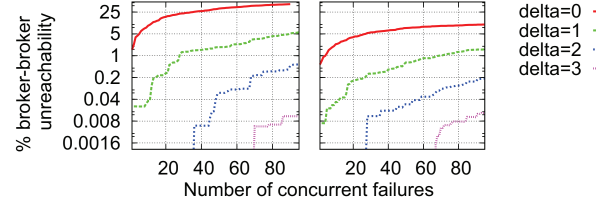


Figure 9. Network connectivity after failures for a network of size 1000 and primary tree fanout of 3 (left) and 7 (right).

IX. EVALUATION

A. Network connectivity after failures

Occurrence of more than δ failures at *adjacent* neighbors can result in disconnection of brokers downstream from each end of the failed chain. While larger values for δ decreases the likelihood of such disconnections, what we seek to study here is a comparative analysis of the expected number of disconnections resulting after concurrent failure of more than δ brokers. We used a graph simulator to construct the overlay network and inject failures at random nodes. We then counted the number for chains of $\delta+1$ brokers that were formed. Figure 9 illustrates the results. Each data point is the average of 100 simulation runs in which a designated number of brokers shown on the x-axis have been randomly chosen to fail. The y-axis shows the percentage of end-to-end brokers that are disconnected in a network of 1000 brokers with primary tree fanout of 3 (left graph) and 7 (right graph). In both cases, increasing δ improves network connectivity as it reduces the possibility of occurrence of $\delta+1$ failures in a chain. Furthermore, for a fanout of 3 there is a higher number of disconnections compared to when fanout is 7. This is due to the fact that a large number of brokers in the network are edge brokers which have *no downstream brokers*. As a result, failure of these edge brokers has no adverse impact on connectivity of other brokers.

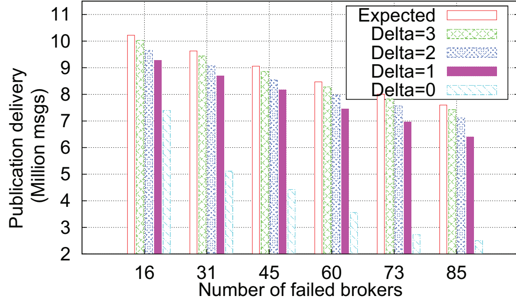


Figure 10. Publication delivery during a 120s measurement interval in a 500 broker network with different number of failures.

B. Impact of failures on publication delivery.

To investigate how increased network connectivity improves end-to-end publication delivery in presence of failures, we carried out experiments using our implementation of the fault-tolerant publication forwarding algorithm. We deployed a large network of 500 brokers on the SciNet cluster [17] such that each broker is assigned a dedicated CPU core (Intel Xeon at 2.53 GHz) and has access to 800 MB RAM. We chose publication and subscription workloads with 100% matching distribution in order to analyze ene-to-end delivery between all brokers. Figure 10 illustrates the number of publications delivered over 120s measurement interval (y-axis) after the specified number of brokers have failed (x-axis). It can be seen that the total number of deliveries for $\delta = 3$ is about 3% lower than the expected deliveries had there been no disconnections. Furthermore, lowering δ from 3 to 1 lowers publication delivery by steps of about 4%. Finally, the system with $\delta = 0$ suffers from vast publication loss due to widespread disconnections. These results demonstrate that a modest value of $\delta = 2$ or 3 provides statistically acceptable fault-tolerance against a large number of concurrrent failures.

C. Size of brokers' Δ-neighborhoods

Networks configured with larger values of δ require brokers to maintain larger amounts of state. Figure 11 shows this trend by illustrating the distribution of the size of brokers' Δ-neighborhoods in a network of 1000 brokers. In the left diagram that corresponds to a network with fanout of 3, all brokers' neighborhoods contain fewer than 100 brokers. On the other hand, the right diagram shows that when $\delta = 3$ and fanout is 7, the size of Δ-neighborhoods grow as large as 1000. This suggests that a smaller δ may be more feasible for networks with a high fanout primary tree.

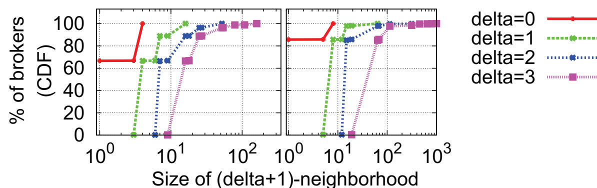


Figure 11. Distribution of brokers' Δ-neighborhood size for a network of 1000 brokers with primary tree fanout of 3 (left) and 7 (right).

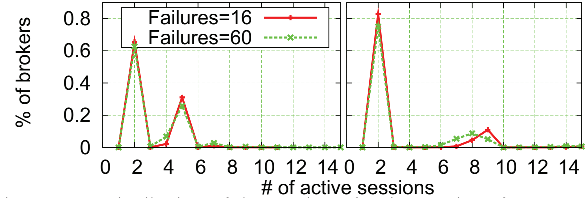


Figure 12. Distribution of the number of active sessions for a network of 1000 brokers and a primary tree fanout of 3 (left) and 7 (right).

D. Number of Active Sessions

An important factor affecting brokers performance is the number of active communication sessions established at each point in time. This number generally increases as more neighboring brokers fail and need to be bypassed. To analyze this trend we performed simulations with networks of 1000 brokers. Figure 12 illustrates the distribution of the number of brokers' active sessions after occurrence of 16 and 60 failures when δ is set to 3. The left and right diagrams corresponds to cases 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. IP multicast provides a dissemination mechanism that is closely related to P/S systems. The basic multicast protocols, however, only provide best-effort QoS and lack reliability. In the area of reliable musticast protocols, OTERS [18] proposes *subcasting* to improve reliability via retransmissions. Its network management relies on *multicast route backtracking* which is closely related to our notion of brokers' Δ-neighborhoods. However, being built upon IP multicast, OTERS is not immediately applicable to a content-based P/S paradigm which supports selective content-based publication delivery.

Opyrchal et al. [19] propose to overcome this challenge by first mapping publications into multicast groups. Dissemination in the network uses reliable group communication, and the subscribers' source brokers apply a final filtering stage based on the *content-based* matching criteria. While such group communication-based approaches provide stronger reliability and ordering guarantees than our approach, they incur substantially more overhead. We believe that our reliability specification provides a good balance for the needs of a wide-range of distributed applications while keeping the communication substrate as light-weight as possible.

Overlay-based routing techniques represent another body of work that can be applied to build fault-resilient P/S systems. In this area, *Resilient Overlay Network (RON)* [20] 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 system and improve 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.

Overlay reconfiguring techniques are also used to handle failures by excluding an unreachable broker from the network [10]. This process, however, is expensive as it requires propagation of subscriptions among non-faulty brokers to reconstruct routing paths in the new overlay.

Snoeren et al. [11] use a mesh network to construct multiple *disjoint* forwarding paths between subscribers and publishers. Publications are forwarded redundantly on all these paths. In the face of failures, message loss is avoided as long as one forwarding path remains available. This scheme is likely to incur high bandwidth due to redundant forwarding of publications – even in absence of failures.

XNET [21] proposes a *crash/failover* scheme, which is similar to our system configured with $\delta = 1$ and allows brokers to establish direct connections to downstream brokers of a failed neighbor. However, unlike our approach, XNET does not tolerate concurrent failures and does not guarantee publication delivery.

Our previous work on crash-resilient P/S systems provides a baseline for the approach presented in this paper [13]. However, our earlier approach is targeted at LAN settings and enterprise-grade networked environments where path outages are relatively uncommon. The significance of our present work on the other hand is highlighted in a wide-area environment such as the Internet where network disconnections and path failures are commonplace, and yet applications require reliable and guaranteed message delivery.

Gryphon [12] provides publication delivery guarantees similar to our system by taking advantage of a replication-based scheme in which the routing information at each broker is replicated across multiple physical machines. In the face of failures, replicas act as primary/backups. We compared a crash-tolerant version of our approach against the replica-based technique of Gryphon and showed that after failures, live replicas in Gryphon may experience large load imbalances that are proportionally higher than their original load limits in absence of failures. Furthermore, Gryphon's approach [22] to ensure gap-less publication delivery requires global knowledge while our scheme uses localized partition information.

XI. Conclusions

Provisioning of reliability as part of the messaging system facilitates the task of developing large-scale distributed applications. To this end, we developed reliable distributed P/S algorithms capable of tolerating concurrent failure of brokers and communication links. Our solution is composed of three main algorithms that address the problems of subscription propagation, publication forwarding, and broker recovery. Our scheme exploits brokers' limited and localized knowledge of nearby partitions to ensure in-order and exactly-once publication delivery. We evaluated our approach in scenarios where the number of concurrent failures exceeds δ. We demonstrated that a network of 500 brokers can maintain 97% of its publication delivery throughput despite concurrent failure of as many as 17% of the 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," *ACM Transactions on Computer Systems*, vol. 19, 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," *TSE'01*.

[10] G. P. Picco *et al.*, "Efficient content-based event dispatching in the presence of topological reconfiguration," in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[11] A. C. Snoeren *et al.*, "Mesh-based content routing using XML," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[12] S. Bhola *et al.*, "Exactly-once delivery in a content-based publish-subscribe system," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 2002.

[13] R. S. Kazemzadeh and H.-A. Jacobsen, "Reliable and highly available distributed publish/subscribe service," in *SRDS*, 2009.

[14] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, 1996.

[15] A. K. Y. Cheung and H.-A. Jacobsen, "Load balancing content-based publish/subscribe systems," *ACM Transactions on Computer Systems*, 2010.

[16] R. Sherafat and H.-A. Jacobsen, "Partition-tolerant publish/subscribe systems," 2011, http://msrg.org/papers/12183213.

[17] "SciNet HPC Consortium," http://www.scinet.utoronto.ca/.

[18] D. Li and D. R. Cheriton, "OTERS (on-tree efficient recovery using subcasting): A reliable multicast protocol," in *ICNP'98*.

[19] L. Opyrchal *et al.*, "Exploiting IP multicast in content-based publish-subscribe systems," in *IFIP/ACM International Conference on Distributed systems platforms*, 2000.

[20] D. Andersen *et al.*, "Resilient overlay networks," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[21] R. Chand and P. Felber, "XNET: A reliable content-based publish/subscribe system," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, 2004.

[22] Y. Zhao *et al.*, "A general algorithmic model for subscription propagation and content-based routing with delivery guarantees," 2005, technical report, RC23669, IBM Research.