

# On-demand Replication for Highly Available Content-based Publish/Subscribe Messaging

Young Yoon, Vinod Muthusamy, and Hans-Arno Jacobsen

Middleware Systems Research Group, University of Toronto

**Abstract.** Content-based publish/subscribe overlays offer a scalable messaging substrate for various event-based distributed systems. In an enterprise environment where service level agreements are strictly enforced, maintaining high availability of the broker overlay is critical. To meet this requirement, protocols are developed to replace or load-balance a broker that misbehaves—perhaps due to failure, congestion, or periodic maintenance—with a new broker in a running system. The key to our techniques is distributed replication of routing information on an adaptively deployed new broker which serves reliable re-route or alternative paths for publication messages. Our techniques focus on avoiding lengthy disruption of messaging service, and supporting consistency properties on publication delivery via the replica.

## 1 Introduction

Distributed content-based publish/subscribe is a powerful messaging substrate for various event-based systems including RSS filtering [21], stock-market monitoring [25], algorithmic trading with complex event processing [18], and business process execution [23]. While scalable and efficient messaging remains important, it is also critical to maintain high availability of the broker overlay especially in an enterprise environment where Service Level Agreements (SLA) are enforced. To meet high-availability requirements, this paper develops practical techniques to replace or offload a pub/sub broker that causes an SLA violation with a new broker in the running system *on-demand*. Fig. 1 illustrates a typical scenario with two non-intersecting overlays from different application domains: *A* and *B*. A monitoring subsystem detects an SLA violation in *A* caused by a faulty broker (B3). The broker may have crashed permanently, may just need to undergo planned maintenance, or may be experiencing temporary congestion. In any case, a resource management system provisions a new broker (B4) either from a pool of stand-by resources or from another application domain (*B*) by *lend-and-borrow* [20]. The new broker then replicates the faulty broker’s routing state not by contacting the faulty broker but by conversing with the faulty broker’s immediate neighbor brokers (B2, B5, B6). Once alternate paths through the replica are established, messages may be exclusively redirected through the replica, or split among the alternative paths.

The replication model described here differs from traditional replication work in redundant routing overlays, distributed databases, virtual machines, operating systems, or file systems [3, 9, 10, 14, 16, 22, 28]. Rather than having a set of

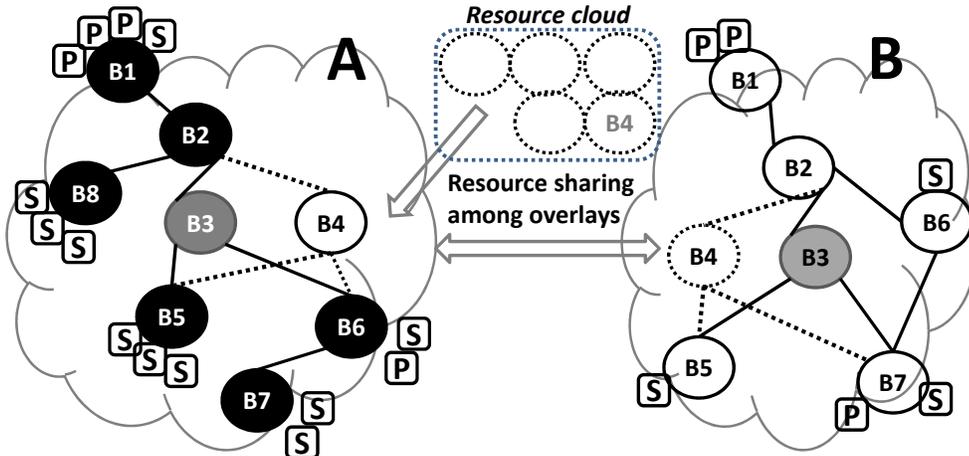


Fig. 1. On-demand replication

replica brokers that continuously synchronize their states *in anticipation* of failure, the problem in this paper is to deploy a replacement for a faulty broker only when the need arises. Such a model can be more cost-effective than having an over-provisioned system of redundant nodes, and comports with on-demand computing architectures such as grid and cloud computing.

The on-demand replication model brings with it several challenges that impede the direct adoption of traditional replication approaches. For example, by the time replication is deemed necessary, the broker being replaced is not responsive enough, thus its state is not feasible for direct replication. Instead, the protocols in this paper exploit the soft state nature of the broker routing tables and reconstruct a faulty broker's state from that of its neighbors.

Another goal is that to be truly on-demand, the replication solutions should impose minimal overhead until replication is required. The protocols proposed in this paper, for instance, only require knowledge of a broker's overlay neighbors be maintained in an external directory service. This information is relatively static and cheap to collect compared to the much more frequent routing state updates and data messages flowing through the network. Other than communicating topology information, the solutions developed here impose *zero* overhead to a normally functioning system both before replication is required and after it completes, and so existing routing algorithms can operate at full speed. This is in contrast to replication models that require active synchronization of broker state or other overhead even when the system is not experiencing any faults [3, 28].

As replication is occurring in a live production system, on-demand replication protocols should avoid disrupting the system, and minimize the effect of broker failure and replacement on end-users. Disruption can be defined in several ways and this paper formalizes consistency properties that define invariants on message delivery and ordering. Another disruption concern is the timely delivery of messages. Sometimes timeliness and consistencies can be opposing goals, and this paper proposes protocols with novel queue management techniques that can tradeoff one goal for the other. Finally, replication protocols should scale to large

broker overlays as it is precisely large systems that are likely to experience more faults. The protocols in this paper are distributed and localized, involving only the neighbors of a faulty broker.

Fig. 1 shows that a broker exhibits different roles in the context of *replication* in a pub/sub overlay. Specifically, we classify according to an advertisement-based pub/sub system [6, 11, 12], as follows:

- *Replication Source Broker* - A broker that is being replicated by the replication target broker, *e.g.*, B4 in Fig. 1. For brevity we use *faulty broker* as an equivalent term.
- *Replication Target Broker* - A broker that replicates the routing information on the replication source broker, *e.g.*, B3 in Fig. 1. For brevity, we use *replica* as an equivalent term.
- *Advertisement-Forwarding Broker* - An immediate neighbor of the faulty broker that forwards advertisements to it, *e.g.*, B2 in Fig. 1.
- *Subscription-Forwarding Broker* - An immediate neighbor of the faulty broker that forwards subscription along the reverse path of matching advertisements to the faulty broker, *e.g.*, B5 and B6 in Fig. 1.

Note that the advertisement-forwarding and subscription-forwarding brokers can be either edge or internal brokers in the overlay. Also, a broker may play the roles of both both an advertisement-forwarding and subscription-forwarding broker, *e.g.*, B1 and B2 in Fig. 1.

This paper makes the following contributions: (1) In Sec. 3, an on-demand broker replication model is introduced, and a basic replication protocol is used to highlight some of the unique challenges and to help define formal consistency properties required of a replication protocol. (2) Sec. 4 then additionally develops advanced replication protocols that satisfy various levels of requirement on performance and consistencies. Sec. 4 also explains novel methods to redeliver and schedule publications among replicas for the purpose of replacement or load-balancing in case permanent or transient failure occurs. (3) Finally, Sec. 5 offers an experimental evaluation of the replication protocols in a real system under a variety of workloads.

Before presenting the above contributions, Sec. 2 presents some background concepts and surveys related work.

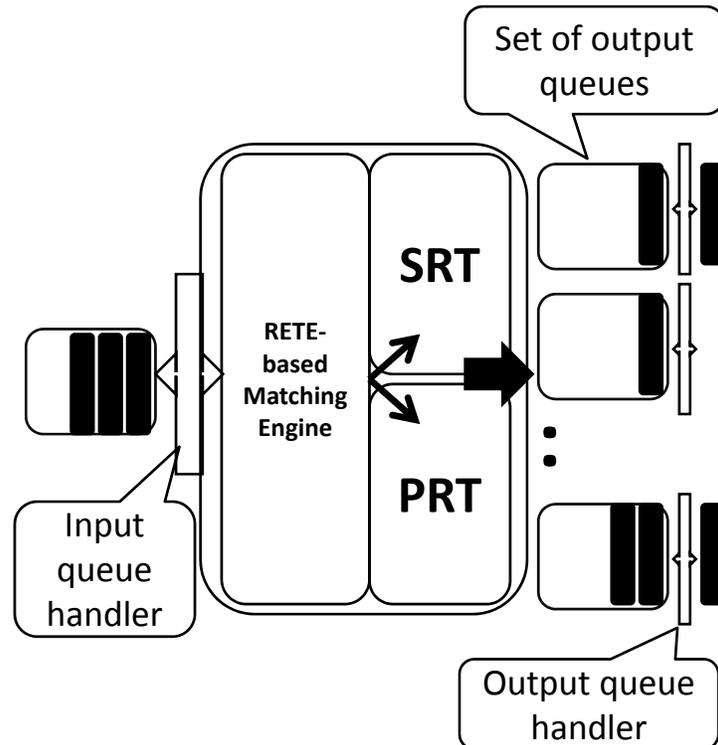
## 2 Background and related work

This paper is put in context of existing replication techniques in overlay networks, distributed databases, and lower level systems are discussed. But first, essential background pub/sub concepts relevant to the understanding of this paper are summarized.

*Background:* This paper builds on the PADRES distributed, content-based pub/sub platform [11] It uses a predicate-based subscription language, in which an atomic subscription is a conjunction of attribute-operator-value tuples and a composite subscription consists of atomic subscriptions connected by logical

operators. In addition, variables allow correlation of attribute values across publications. PADRES adopts the standard pub/suboverlay architecture shown in Fig. ???. The overlay network connecting the brokers forms the basis for message routing. A PADRES broker uses a rule-based engine to perform scalable message routing and matching. Clients connect to the broker overlay and issue messages into the broker overlay. that uses advertisement-based routing [6, 12]. Each broker records its overlay neighbors, and advertisements are broadcast through the overlay forming an advertisement tree. Advertisements are stored in the Subscription Routing Table (SRT). Subscriptions propagate in reverse direction along the advertisement trees, and are stored in the Publication Routing Table (PRT). If a publication matches a subscription in the PRT, it is forwarded to the last hop of that subscription until it reaches the subscriber.

Each PADRES broker consists of an input queue, a rule-based matching engine, and a set of output queues. The matching engine takes messages from the input queue and routes it to proper output queues after matching the message content against the SRT and PRT. An output queue represents a destination which could be a broker, a client, a database or a JMS broker.



**Fig. 2.** Architecture of PADRES for content-based routing and messaging.

*Overlay replication:* The Gryphon pub/sub system supports exactly-once delivery semantics in a network where each logical broker is in reality represented by a set of redundant physical brokers [3]. A variety of techniques, including acknowledgments and negative-acknowledgments from subscribers, and periodic ticks from publishers (even when the publisher is silent) are used to detect failures. This paper, however, supports on-demand replication as opposed to over-provisioning a redundant network.

Subsequent work supports a subscription propagation scheme that ensures in-order delivery without any missing messages [28]. The solution employs distributed virtual time vectors to detect whether a broker's routing state is sufficiently updated to achieve correct delivery. Like the earlier work, this too requires redundancy in the network. Furthermore, the time vectors include elements for each subscribing broker, and thus may not be appropriate for large loosely-coupled broker networks. The protocols developed in this paper, on the other hand, require knowledge only of the faulty broker and its neighbors, and therefore their performance is independent of the size of the overlay.

The IndiQoS [5] pub/sub network reserves resources and constructs paths with sufficient capacities to satisfy clients' quality-of-service requirements. Certain brokers can be replicated for redundancy or load balancing, but these replicas must be kept synchronized, and there is no dynamic mechanism to add replicas. By contrast, this paper supports an on-demand replication scheme that can be used to replicate brokers in an existing pub/sub system. Moreover, it is lightweight, and imposes virtually no overhead to the existing system when replication is not taking place.

Another approach is to support dynamic reconfigurations in the overlay routing paths in order to bypass faulty brokers [8, 17], including systems built on a P2P network [2, 7, 19]. These approaches may be useful in some contexts but are orthogonal to the problem addressed in this paper. The overlay topology may be fixed for any number of reasons, including capacity planning requirements, security constraints, or business agreements among the owners of the overlay nodes. In such cases, a faulty broker cannot be routed around and must be replaced.

*Database replication:* Distributed database replication solutions are generally classified as passive schemes where a designated master handles all client requests and the backups synchronize their state with the master, and active techniques in which all replicas can process client requests [13, 26]. Database replication is a fundamentally different model of replication from that in this paper. In distributed databases, replicas must continuously coordinate as they process user requests and update their state. The problem in this paper though is to provision a replica only when the need arises. As a consequence, the design goals are different: broker replication in this paper must be fast, ought to minimize the disruption to the system, and should be able to recreate a faulty broker's state without ever having communicated with it, i.e., after or during a failure. Some database replication works are QoS-aware. Serrano *et al.* [24] trade off performance and consistency by the placement of a replica close to high-demand clients. While sharing objectives, this paper differs in its focus on the replication

of operational state, i.e., routing state and consistencies on publication delivery. Another useful technique is to proactively select a running replica based on the probabilistic distribution of response time to the clients [15]. Such techniques can be applied to this paper in case a new broker needs particular capacity specification of a resource, e.g., network i/o, process power.

*System level replication:* By replicating an entire machine no application-specific knowledge is required. The Remus system [10] replicates virtual machine state across physical machines, so physical host failures are masked by transparently continuing execution using the state on another physical host. Other system level replication work include Zap [16] which replicates processes at the operating system level, and Coda [22] which offers a distributed file system with file-level replication across servers. The key benefit of low level replication—no knowledge of or modifications to applications is required—also leaves them unable to exploit application knowledge. For example, virtual machine replication requires every state change to the machine to be synchronized, which can impose significant overhead. In the pub/sub overlay, however, a broker’s routing state can be regenerated from information available at other brokers, a property that the protocols in this paper take advantage of. Moreover, as with database replication, these replication models are not on-demand.

To recap, existing replication solutions assume a model where replication occurs continuously, and require mechanisms to actively replicate state in anticipation of a failure, whereas the protocols in this paper operate on-demand only when a fault is detected and a replica is required. Such over-provisioning is expensive and runs contrary to the emerging grid and cloud architectures that promote on-demand resource utilization. Another benefit of the on-demand model is there is virtually no overhead when the system is behaving normally. One challenge to this approach, however, is that there is no residual information of the faulty component’s state, and instead application level properties must be exploited to recover this state.

### 3 On-demand broker replication

This section explores the issues with supporting on-demand broker replication. When a faulty broker is detected, a new broker is provisioned to replace it. The replication can be invoked at anytime on brokers in a running system, and should disrupt the system as little as possible. This disruption can be defined in various ways including the number of brokers that must participate in the protocol, the time it takes to complete the replication, and any inconsistencies in the messages that subscribers receive as a result of the protocol. Each of these properties will be discussed in full detail along with the presentation of the protocols below. A simple replication protocol is first presented in Sec. 3.1 and serves to highlight some of the subtle issues that arise in an on-demand replication protocol. These issues are captured in formal properties that any replication protocols should support. These properties motivate the need for more sophisticated replication protocols developed in Sec. 4.

---

**Algorithm 1:** Basic protocol at replica broker  $R$ 

---

**Input:**  $N \leftarrow \{n | n \text{ is a neighbor of faulty broker } F\}$

- 1 **if** *receive initialize message* **then**
- 2     send request for advertisements to all  $n \in N$ ;
- 3 **if** *receive advertisement*  $adv$  *from any*  $n \in N$  **then**
- 4     insert  $adv$  into  $SRT$ ;
- 5     forward  $adv$  to all  $n_i \in N$  where  $n_i \neq n$ ;
- 6 **if** *receive subscription*  $sub$  *from any*  $n \in N$  **then**
- 7     insert  $sub$  into  $PRT$ ;
- 8      $\mathbb{A}_n \leftarrow \{adv | adv \in SRT \wedge adv \text{ matches } sub\}$ ;
- 9     forward  $sub$  to the lasthop of each  $adv \in \mathbb{A}_n$

---

### 3.1 Basic replication protocol

A naive broker replication approach is to initialize the replica's state from a central repository of global routing information. Maintaining this repository, however, may be infeasible, especially when routing information is updated frequently due to high churn rate of advertisements and subscriptions. Moreover, the central repository becomes a single point of failure, and potential performance bottleneck. But it may limit the scalability. Thus the protocols in this paper, instead, exploit the observation that the routing state of a broker can be recreated from the routing state of its neighbors. Therefore, only the broker topology, which is relatively stable and cheap, needs to be maintained by a distributed directory service [4]. In this way the replication protocols are distributed and more scalable.

The basic replication protocol consists of a conversation between the replica broker and the immediate neighbors of the faulty broker. In the following discussion, neighbors of a faulty broker that forward advertisements and publications to the faulty broker (or replica) are referred to as *advertisement-forwarding brokers*. Likewise, those that forward subscriptions are *subscription-forwarding brokers*. Of course, a given neighbor may play both roles.

The replica begins by contacting each neighbor of the faulty broker to initiate distributed replication. An advertisement-forwarding neighbor forwards its advertisements to the replica, which stores the advertisements in its routing table and relays them to its other neighbors. A subscription-forwarding broker forwards its matching subscriptions along the reverse path of the advertisement back to the replica, which the replica stores in its routing table and forwards towards the appropriate advertisement-forwarding neighbors. Once all the advertisements and subscriptions have been disseminated, the routing state has been replicated and the protocol is complete.

The algorithm that runs at the replica as part of the basic replication protocol is detailed in Algorithm 1, and the corresponding algorithm at the neighbors is specified in Algorithm 2. The reader will notice in lines 2 and 6 of Algorithm 2 that the protocol avoids forwarding advertisements and subscriptions in cycles by discarding messages whose previous hop was the faulty broker.

---

**Algorithm 2:** Basic protocol at neighbor broker  $n$  of replica  $R$  and faulty broker  $F$

---

```

1 if receive request for advertisement from  $R$  then
2    $\mathbb{A}_n \leftarrow \{adv | adv \in SRT \wedge adv.lasthop \neq F\}$ ;
3   forward every  $adv \in \mathbb{A}_n$  to  $R$ ;
4 if receive advertisement  $adv$  from  $R$  then
5   insert  $adv$  into  $SRT$ ;
6    $\mathbb{S}_n \leftarrow \{sub | sub \in PRT \wedge sub \text{ matches } adv \wedge sub.lasthop \neq F\}$ ;
7   forward every  $sub \in \mathbb{S}_n$  to  $R$ ;
8 if receive subscription  $sub$  from  $R$  then
9   insert  $sub$  into  $PRT$ ;

```

---

Any publications received during the protocol are forwarded in the usual manner, that is, they are forwarded to the last hop of any matching subscriptions. The advertisement-forwarding broker is a special case since its routing state includes subscriptions that will direct publications to both the faulty and the replica brokers. Depending on whether the faulty broker is unavailable or simply overloaded, the advertisement-forwarding broker can direct publications only to the replica broker or load balance publications among replica and faulty brokers.

### 3.2 Analysis of basic replication protocol

**Key benefits:** The protocol only involves the replica and direct neighbors of the faulty broker. The neighbors accomplish this by indicating in the advertisements and subscriptions they forward a time-to-live value of two, so these messages traverse no more than two hops to another neighbor. Therefore, regardless of the size of the broker overlay, only a small number of brokers participate in any given replication operation.

As mentioned above, the protocol also takes care to avoid duplicate routing table entries. Without such precautions, incorrect routing behavior can occur as shown in Fig. 3. Suppose brokers  $B1$  and  $B3$  host both publishers and subscribers; that advertisements  $A$  and  $A'$  are issued by publishers  $P$  and  $P'$ , respectively; and that both advertisements match both subscription  $S$  and  $S'$ . Blindly forwarding advertisements from both edge brokers causes duplicate entries in both their routing tables, which will result in publication messages being routed multiple times in the cyclic routing path that forms.

**Complexity:** The messages complexity is linear with the neighbor set and routing state. More precisely, the complexity is  $O(|N| + |\mathbb{A}| + |\mathbb{S}|)$ , where  $N$  is the set of neighbors of the faulty broker, and  $\mathbb{A}$  and  $\mathbb{S}$  are the set of advertisements and subscriptions in the faulty broker's routing tables, respectively.

The derivation is as follows based on Algorithm 1. First,  $|N|$  messages are required to establish connections with each neighbor in  $N$ . The replica broker receives advertisements  $\mathbb{A}_n$  from each  $n \in N$  (lines 3–5), and subscriptions  $\mathbb{S}_n$  from each neighbor  $n$  (lines 6–9). Since no duplicate advertisements and subscriptions

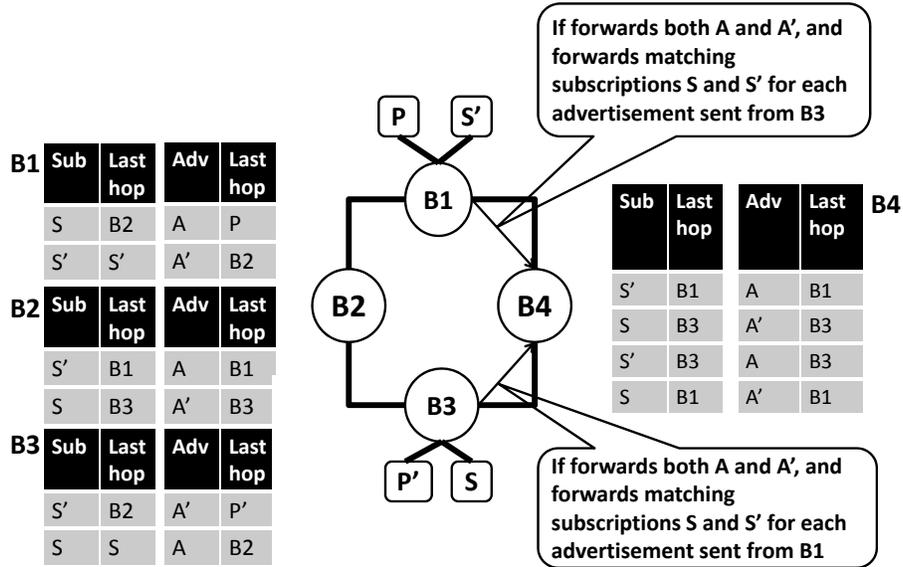


Fig. 3. Example: Duplicate entries if neighbors do not filter

are forwarded,  $|\mathbb{A}| + |\mathbb{S}| = \sum_{n \in N} |\mathbb{A}_n| + |\mathbb{S}_n|$ . Similarly, for each neighbor  $n \in N$ ,  $|\mathbb{A}_n| + |\mathbb{S}_n| = |\mathbb{A}_r| + |\mathbb{S}_r|$ , where  $\mathbb{A}_r$  and  $\mathbb{S}_r$  are the set of advertisements and the set of subscriptions, respectively sent from the replica,  $r$ .

**Limitations:** The basic replication protocol is simple but suffers from a number of flaws, including not delivering publications to all interested subscribers, corrupting the order of publication delivery, and dropping publications altogether.

The protocol may not deliver publications to all interested subscribers. For example, in Fig. 4(a) the advertisement-forwarding broker forwards publications through the replica before all the subscription-forwarding brokers have sent their subscriptions to the replica. The replica, unaware that there are other interested subscribers, only forwards the publications towards those brokers whose subscriptions are already in its routing tables.

Another limitation is that publications may be delivered out of order. The PADRES pub/sub routing protocols guarantee per-source ordering: publications from a publisher are delivered to interested subscribers in the order they were published. In the example in Fig. 4(b), there are two publications  $p_1$  and  $p_2$ , where  $p_1 \prec p_2$  indicates that the same publisher issued  $p_1$  and  $p_2$  and that it published  $p_1$  before  $p_2$ . The faulty broker is overloaded and received  $p_1$  before the overload was detected and the replica was provisioned. Publication  $p_2$ , on the other hand, arrives after the replica is active, and propagates to the subscription-forwarding broker through the replica. After some time, the overloaded broker finally sends  $p_1$  to the subscription-forwarding broker, but by this point the order of the two publications has been reversed, violating per-source ordering.

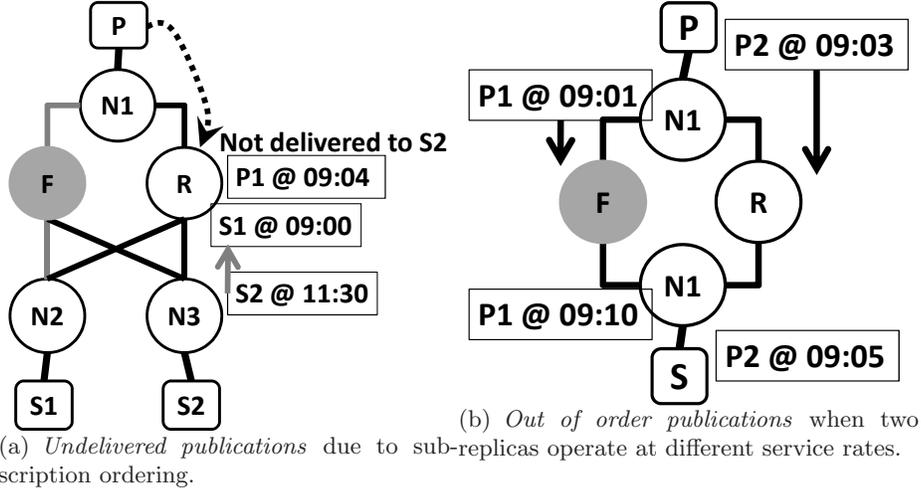


Fig. 4. Examples of violations of correctness with replicas

### 3.3 Replication properties

To formally capture violations such as those above, Properties 1 and 2 formulate consistency in terms of the publications delivered to a pair of subscribers.

**Property 1** *Strong Complete Delivery*: Consider any pair of subscribers  $s_1$  and  $s_2$  that have issued identical subscriptions (at any time) and have received matching publication sets  $P_1$  and  $P_2$ , respectively, and that publications  $p'$  and  $p$  appear in both  $P_1$  and  $P_2$ , where  $p' \prec p$ . If there exists a  $p''$  where  $p' \prec p'' \prec p$ , then  $p''$  must either belong to both  $P_1$  and  $P_2$  or neither  $P_1$  nor  $P_2$ .

Informally, the complete delivery property requires that publications be delivered to all interested subscribers. The property is economical in the sense that it makes no assumptions about time synchrony or network characteristics—so subscribers subscribing at different times may receive different sets of publications—and it refrains from defining what a match is. In the context of this paper, it can be used to ensure that the replication protocol does not alter the matching semantics of an existing pub/sub routing protocol.

**Property 2** *Strong Ordered Delivery*: For any pair of publications,  $p'$  and  $p$ , received by a subscriber, where  $p' \prec p$ , the subscriber should receive  $p'$  before  $p$ .

Informally, the ordered delivery property ensures publications from any given publisher are not delivered out of order to interested subscribers.

Note that both delivery properties should hold for any subscribers including those for whom the publications in question must traverse the faulty broker, the replica broker, or neither broker. This is what allows the properties to be defined in a protocol agnostic manner. This paper will refer to Properties 1 and 2 as *strong consistency* properties. The individual properties are, however, orthogonal in that a system may support either property, both, or neither.

Now, to ensure consistency, the system must not lose any publications already sent to the faulty broker. Otherwise, a subscriber reachable only through the faulty broker would receive a different set of publications from one reachable without traversing the faulty broker.

The number of publications,  $|\mathbb{P}_{incons}|$ , potentially delivered inconsistently—whether delivered out of order or not delivered to all interested subscribers—can be bounded to those that traverse the faulty broker after it has failed. If the faulty broker has permanently failed, and there are no other copies of these publications in the system, then there is no way to avoid losing such publications. Therefore,  $|\mathbb{P}_{incons}| \geq \text{notification.rate} \times (t_d - t_f)$ , where  $t_f$  is when the faulty broker actually failed, and  $t_d$  is when the failure is detected. Notice that  $|\mathbb{P}_{incons}|$  considers *potentially* inconsistent publications. It is entirely possible that if the faulty broker is only temporarily overloaded, that all the publications sent to it will eventually be delivered in a consistent manner.

To avoid losing publications already sent to a permanently faulty broker, one approach is to fundamentally alter the typical one-hop pub/sub forwarding algorithms so that sufficient brokers maintain copies of the publications in order to recover from an expected number of concurrent failures [14]. A second approach is to persist publications in stable storage and suffer the overhead this would impose. This paper opts for replication protocols that can be easily integrated into existing pub/sub systems, and it becomes necessary, therefore, to accept inconsistencies arising from publications sent to the faulty broker before its failure was detected. This relaxed requirement is expressed in Properties 3 and 4, which are weaker forms of Properties 1 and 2, respectively. As before, Properties 3 and 4 are together referred to as *weak consistency* properties.

**Property 3** *Weak Complete Delivery: The subset of publications that do not traverse a faulty broker should satisfy complete delivery (Property 1).*

**Property 4** *Weak Ordered Delivery: The subset of publications that do not traverse a faulty broker should satisfy ordered delivery (Property 2).*

While the weak consistency properties above ignore publications propagating through a faulty broker, they still apply where the paths to one or both subscribers contains the replica broker.

The basic protocol violates weak complete delivery (Property 3) but satisfies weak ordering (Property 4). In Fig. 4(a), the subscriber reachable through the replica does not receive a publication whereas another subscriber directly connected to the advertisement-forwarding broker would receive the publication, violating the weak complete property. As for weak ordering, nothing in the replication protocol affects the order of messages in either the input or output queues at the replica or any of the advertisement-forwarding and subscription-forwarding brokers. Therefore, as long as the conventional pub/sub routing protocols ensure ordered delivery, the basic replication protocol will satisfy Property 4.

---

**Algorithm 3:** Synchronous publication delivery recommencement at advertisement-forwarding neighbor broker  $n$  of replica  $R$  and faulty broker  $F$

---

```

1 if receive request for advertisement from  $R$  then
2   SuspendUserMessageHandling( $OutputQueue_R$ );
3   move messages in  $OutputQueue_F$  to  $OutputQueue_R$ ;
4   Close( $OutputQueue_F$ );
5   foreach  $adv \in SRT$  where  $adv.lasthop = F$  do
6      $adv.lasthop \leftarrow R$ ;
7   foreach  $sub \in PRT$  where  $sub.lasthop = F$  do
8      $sub.lasthop \leftarrow R$ ;
9    $\mathbb{A}_n \leftarrow \{adv \mid adv \in SRT \wedge adv.lasthop \neq R\}$ ;
10  forward every  $adv \in \mathbb{A}_n$  to  $R$ ;
11  send lastAdv message to  $R$ ;
12 if receive replication_done message from  $R$  then
13  ResumeUserMessageHandling( $OutputQueue_R$ );

```

---

## 4 Advanced replication protocols

To address the limitations of the basic replication protocol from Sec. 3.1, this section develops more sophisticated replication protocols. These protocols achieve various combinations of the properties defined in Sec. 4, by incremental and synchronous

### 4.1 Synchronous replication protocol

This section develops a synchronous replication protocol that satisfies the weak consistency property defined in Sec. 3.3. In particular, all publications not already forwarded to the faulty broker are delivered to all interested subscribers, and per-source publication ordering is maintained.

The strategy underlying the synchronous protocol is to operate in two phases. First, the replica fully constructs its routing state, during which phase the processing of user-generated messages destined to the faulty broker is suspended. When the routing state has been replicated, the second phase begins where pending and newly arriving user-generated messages are redirected to the replica. The subsections below describe each phase of the protocol in turn.

**Synchronous replication of routing state:** When constructing the routing state at the replica broker, it is important to know when the routing information is complete so the protocol can safely commence the publication delivery phase.

The replica must retrieve all advertisements, and then all matching subscriptions, from its neighbors. For the replica to know it has received all advertisements or subscriptions, each neighbor sends a *lastAdv* or *lastSub* control message after it has sent all the requested messages to the replica. This control message can be piggybacked onto the last message from a neighbor, but must

be sent even in the case where the neighbor has no advertisement to send to the replica, or the neighbor has no subscription matching an advertisement.

This synchronous replication of a faulty broker’s state at the replica is detailed in Algorithm 4. The key difference from the basic replication protocol in Algorithm 1 is that the replica now maintains counters to know how many more advertisements or subscriptions are pending. For example, the algorithm expects advertisements from each neighbor in  $N$  and sets the *PendingLastAdvs* counter accordingly in line 4 of Algorithm 4. (More precisely it expects a *lastAdv* control message from each neighbor.) Furthermore, for every advertisement, the replica expects subscriptions from every neighbor but the one that sent the advertisement. Since the number of advertisements is not known *a priori*, the *PendingLastSubs* counter is updated as advertisements are received as shown in line 10 of Algorithm 4.

Algorithm 4 uses a local state machine whose current state is recorded in the *state* variable. The replica begins in the `WaitingForAdvs` state until it has received all advertisements at which point it transitions into the `WaitingForSubs` state (line 18). When the replica has received all subscriptions it transitions into the `Replicated` state (line 25), and notifies its neighbors that the replication phase is complete so they may begin the second phase of the protocol where user-generated messages are processed.

**Synchronous publication delivery resumption:** In the first phase, once the advertisement-forwarding broker is aware a neighbor is faulty, it suspends forwarding any messages in the output queue to the faulty broker. When replication is complete, in the second phase of the protocol the broker redirects publications originally destined for the faulty broker to the replica instead. Algorithm 3 sketches the key portions of the algorithm at an advertisement-forwarding broker. When the broker learns about the replica, it internally replaces the faulty broker with the replica by moving outstanding messages for the faulty broker to the replica’s output queue (Line 3), and then reconfigures its routing tables to act as though advertisements and subscriptions from the faulty broker were from the replica instead (Lines 5–8). The latter ensures new user-generated messages bound for the faulty broker are directed to the replica. The broker also defers sending user-generated messages until replication completes (Lines 2 and 13). This ensures the replica’s routing tables are complete so no messages are dropped. Be aware that only user-generated messages are suspended; advertisements, subscriptions and control messages involved in the replication protocol continue to propagate.

Subscription-forwarding brokers have relatively straightforward behavior. They reply to each advertisement from the replica with the matching subscriptions not already forwarded to the replica, followed by a *lastSub* control message.

The synchronous replication protocol achieves weak consistency, including delivering publications not already sent to the faulty broker to all interested subscribers and preserving per-source publication ordering, by deferring sending publications until replication is complete so traditional pub/sub forwarding

---

**Algorithm 4:** Synchronous replication at replica  $R$ 

---

**Input:**  $N \leftarrow \{n | n \text{ is a neighbor of faulty broker } F\}$

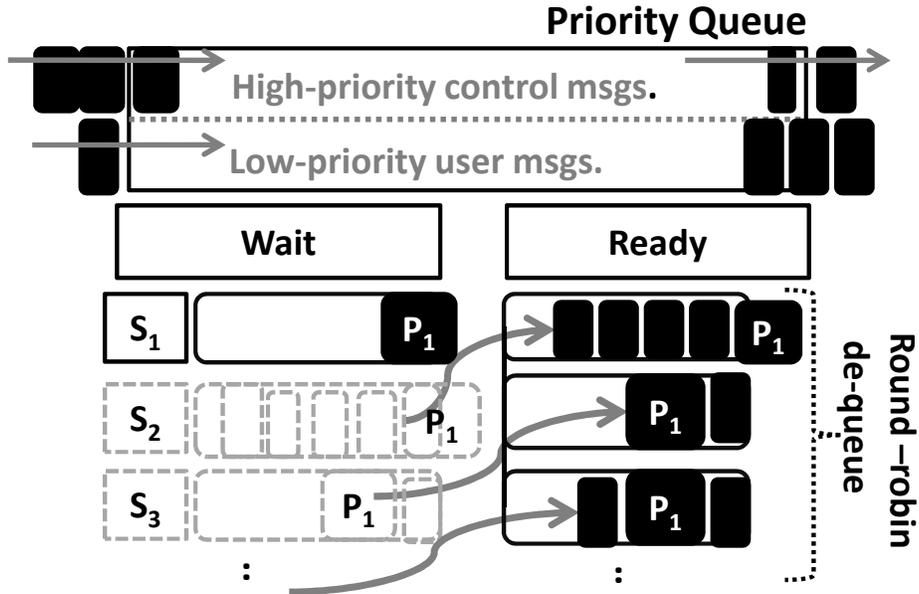
- 1 **if** *receive initialize message* **then**
- 2     send request for advertisements to all  $n \in N$ ;
- 3      $state \leftarrow \text{WaitForAdvs}$ ;
- 4      $PendingLastAdvs \leftarrow |N|$ ;
- 5      $PendingLastSubs \leftarrow 0$ ;
- 6 **if** *receive advertisement  $adv$  from any  $n \in N$*  **then**
- 7     insert  $adv$  into  $SRT$ ;
- 8     forward  $adv$  to all  $n_i \in N$  where  $n_i \neq n$ ;
- 9     **if**  $state = \text{WaitForAdvs}$  **then**
- 10          $PendingLastSubs \leftarrow PendingLastSubs + |N| - 1$ ;
- 11 **if** *receive subscription  $sub$  from any  $n \in N$*  **then**
- 12     insert  $sub$  into  $PRT$ ;
- 13      $\mathbb{A}_n \leftarrow \{adv | adv \in SRT \wedge adv \text{ matches } sub\}$ ;
- 14     forward  $sub$  to the lasthop of each  $adv \in \mathbb{A}_n$
- 15 **if** *receive lastAdv control message from any  $n \in N$*  **then**
- 16      $PendingLastAdvs \leftarrow PendingLastAdvs - 1$ ;
- 17     **if**  $PendingLastAdvs = 0$  **then**
- 18          $state \leftarrow \text{WaitForSubs}$ ;
- 19         **if**  $PendingLastSubs = 0$  **then**
- 20              $state \leftarrow \text{Replicated}$ ;
- 21             send *replication\_done* message to all  $n \in N$ ;
- 22 **if** *receive lastSub control message from any  $n \in N$*  **then**
- 23      $PendingLastSubs \leftarrow PendingLastSubs - 1$ ;
- 24     **if**  $state = \text{WaitForSubs} \wedge PendingLastSubs = 0$  **then**
- 25          $state \leftarrow \text{Replicated}$ ;
- 26         send *replication\_done* message to all  $n \in N$ ;

---

properties can be relied on. The weakness is the lengthy service disruption, as subscribers receive no publications until replication is complete, which means that a single slow neighbor can prolong replication and disrupt the system for an excessively long time. To address this, the next section develops an incremental replication protocol that can support time-sensitive pub/sub applications.

## 4.2 Incremental replication protocol

This section presents an incremental replication protocol that reduces service disruption by resuming publication streams as soon as possible, while sacrificing per-source publication ordering. Contrast this to the synchronous protocol which can resume publication delivery only after the entire protocol is complete but does ensure publication ordering. Incremental replication requires advertisement-forwarding brokers forward their publications to the replica once they receive a matching subscription from the replica, and the replica to cache publications for late arriving subscriptions from subscription-forwarding brokers. The algorithms at the publisher-forwarding broker and the replica are outlined below.



**Fig. 5.** Wait-ready queue data structure for incremental publication delivery resumption.

**Incremental publication delivery resumption:** Recall in the synchronous protocol, the advertisement-forwarding broker suspended forwarding publications to the replica. The incremental protocol, on the other hand, can process these publications sooner. Publications destined to a broker undergoing replication are indexed in a wait-ready data structure as illustrated in Fig. 5 consisting of three queue classes: a priority queue for control messages that should be processed even during replication, a set of wait queues per subscription, and a set of ready queues per subscription.

Matched publications sent to the replica are indexed in a wait queue indexed by their matching subscriptions. A publication that matches multiple subscriptions is indexed multiple times, as in the example of publication  $p_1$  in Fig. 5. When a subscription arrives at the advertisement-forwarding broker from the replica, the publications in the wait queue associated with the subscription are moved to a ready queue also indexed by the subscription. Therefore, publications known to match a subscription are scheduled to be forwarded as soon as the subscription path through the replica is established. Publications in the ready queues are forwarded to the replica in a round robin fashion. Note again that messages in the priority queue are given higher priority so publication forwarding does not impede the replication protocol. The advertisement-forwarding broker avoids duplicates by checking against a log of already forwarded publications. The memory for this log, as well as the wait and ready queues can be reclaimed once the replica indicates replication is complete.

The memory complexity of the protocol at the advertisement-forwarding broker is  $\Theta(f|P|)$  where  $f$  is an average fanout, *i.e.*, the number of subscriptions

---

**Algorithm 5:** Incremental publication forwarding at replica  $R$ 

---

```
Input:  $N \leftarrow \{n | n \text{ is a neighbor of faulty broker } F\}$ 
1 /* Init. and handle advs as in Algorithm 4. */;
2 if receive subscription  $sub$  from any  $n \in N$  then
3   /* Match and forward  $sub$  as in Algorithm 4 then do the following. */;
4    $\mathbb{P} \leftarrow \text{matches}[sub]$ ;
5    $\text{matches}[sub] \leftarrow \emptyset$ ;
6   foreach  $p \in \mathbb{P}$  do
7     if  $sub.lasthop \notin \text{sentTo}[p]$  then
8       forward  $p$  to  $sub.lasthop$ ;
9        $\text{sentTo}[p] \leftarrow \text{sentTo}[p] \cup sub.lasthop$ ;
10 if receive publication  $pub$  from any  $n \in N$  then
11    $\mathbb{S} \leftarrow \{sub \in PRT | sub \text{ matches } pub \wedge sub.lasthop \neq n\}$ ;
12   foreach  $s \in \mathbb{S}$  do
13     forward  $pub$  to  $s.lasthop$ ;
14      $\text{sentTo}[pub] \leftarrow \text{sentTo}[pub] \cup s.lasthop$ ;
15    $\mathbb{S}' \leftarrow \text{extractMatchingSubs}(pub)$ ;
16   foreach  $s \in \mathbb{S}'$  where  $s.lasthop \notin \text{sentTo}[pub]$  do
17      $\text{matches}[s] \leftarrow \text{matches}[s] \cup pub$ ;
```

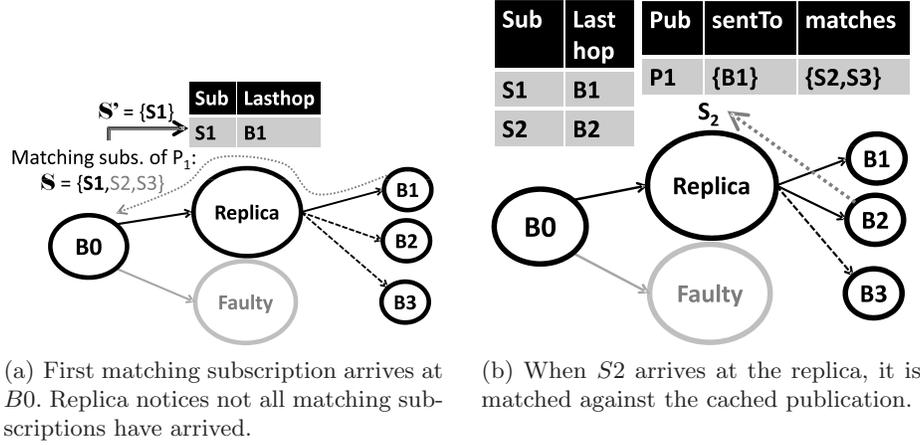
---

a publication matches, and  $P$  is the set of pending publications queued in the wait-ready queue. The worst case occurs when every publication matches every subscription, in which case an optimization would be to simply forward the publication upon receiving the first subscription and altogether avoid indexing the publication in multiple wait queues. More space-efficient data structures with smart indexing based on actively monitored fanout trends is left for future study.

**Incremental publication forwarding at replica:** The second piece of the incremental replication protocol requires the replica to correctly send the incrementally forwarded publications to all interested subscribers. The replica can of course immediately forward the publication towards the matching subscriptions in its routing table. However, because the replication protocol is ongoing there may be more matching subscriptions arriving from subscription-forwarding brokers that the replica is currently unaware of. To ensure publications are correctly forwarded towards subscriptions that arrive late, the replica caches publications until the replication protocol is complete.

The incremental publication forwarding algorithm at the replica, shown in Algorithm 5, makes use of two data structures. The `sentTo` structure records the set of neighbors a publication has already been forwarded to, and the `matches` structure indexes the set of publications that match a given subscription that is not already in the replica's routing table. The algorithm also assumes that publications incrementally forwarded by advertisement-forwarding brokers include the identifiers of subscriptions known by the advertisement-forwarding broker to match the publication.

When an incrementally forwarded publication arrives at the replica, as Algorithm 5 shows, the replica forwards it to matching subscriptions in its routing



**Fig. 6.** Example of selective publication forwarding at a replica

tables (Lines 11–14), then records the remaining subscriptions that will match (Lines 15–17). When a subscription arrives, cached publications are matched and forwarded (Lines 4–9). The algorithm also avoids sending duplicate publications even if multiple matching subscriptions arrive from a broker.

The number of cached publications at the replica increases with the number of subscription-forwarding brokers since it is safe to remove the publication *iff* it received either a matching publication or a *lastSub* from all subscription-forwarding brokers. However, as with the advertisement-forwarding broker, the memory overhead is only imposed while replication is taking place.

Fig. 6 illustrates a replica selectively forwarding publications. Consider subscriptions  $S_1, \dots, S_n$  from brokers  $B_1, \dots, B_n$  that all match publication  $P_1$  that is held in a wait queue at broker  $B_0$ . Suppose in Fig. 6(a) that subscription  $S_1$  arrives first at the replica and is forwarded to  $B_0$ , adding a routing entry at the replica along the way. Broker  $B_0$  then forwards matching publication  $P_1$  to the replica, including the IDs of the subscriptions that it knows to match  $P_1$ , which is the set  $\mathcal{S} = \{S_1, \dots, S_n\}$  in this case. In Fig. 6(a), the replica first forwards  $P_1$  to  $B_1$ , but then also notices that subscriptions  $S_2, \dots, S_n$  were included as matching subscriptions in  $P_1$  but these subscriptions are not yet in the replica’s routing table. As seen in Fig. 6(b), the replica then caches  $P_1$  and records that  $P_1$  has already been forwarded to broker  $B_1$  and that it is awaiting matching subscriptions  $S_2, \dots, S_n$ . When the next subscription, say  $S_2$  from broker  $B_2$ , arrives,  $P_1$  is matched in the cache and forwarded to  $B_2$ . In this way, the incremental replication protocol delivers publications as soon as a matching subscription path is established, but also ensures delivery to late-arriving subscriptions.

The incremental replication protocol satisfies weak complete delivery (Property 3) but violates weak ordering (Property 4). Weak complete delivery is achieved because advertisement-forwarding brokers forward exactly the same publications to the replica they would have to the faulty broker had it not

failed; the replication protocol does not add or remove entries from their routing tables, and they do not drop any publications in their queues. Furthermore, the replica broker caches publications until they are forwarded to all known interested subscription-forwarding brokers, so it too forwards publications to all the neighbors the faulty broker would have (although perhaps in a different order). As for violation of weak ordering, for the wait-ready queue in Fig. 5 the order that publications are moved from a wait queue to a ready queue is a function of the order that subscriptions arrive, and may not preserve the order in which publication arrived. Furthermore, the ready queues are served in round-robin fashion leading to further shuffling of publications.

One drawback with incremental replication is the replica must match incoming subscriptions against advertisements in the routing table as well as the cached publications. However, publications cached at the replica are now one hop closer to the subscription-forwarding broker saving computation and communication delays. For example, in the scenario in Fig. 6, subscription  $S_2$  from broker  $B_2$  retrieves publication  $P_1$  from the replica avoiding the two additional hops to otherwise fetch the publication from broker  $B_0$ . As to which of these two factors—additional processing at the replica and caching publications closer to their destination—will dominate will depend on the workload. We evaluate this aspect empirically in the next section.

### 4.3 Advanced load-balance protocols

The basic, synchronous, and incremental protocols are primarily concerned with provisioning a replica broker to use in place of a faulty one. It is possible that the faulty broker is still functional, but simply operating at a rate that fails to fulfill its service level agreement. This may be due to an increase in the matching overhead the incoming workload incurs, or a growth in the output queue lengths resulting from a large fanout at the broker. In such cases, the workload can be divided among the faulty and replica brokers. Key techniques to the load-balancing protocols are scheduling of the messages among alternative replicas and re-ordering messages if strong consistency is required.

**Message scheduling at advertisement-forwarding broker:** Upon receipt of the subscription via the newly established path through the replica, the advertisement forwarding broker adds the replica to the list of candidate next hops. When the publication delivery resumes, the advertisement broker routes the publication to the output queue of the next hop with the shortest queue length.

**Message re-ordering at subscription-forwarding broker:** To avoid out-of-order messages as shown in Fig. 4(b), the subscription forwarding broker can re-order messages to ensure the *strong ordered* property is maintained. It is assumed that publications are tagged with a publisher-specific sequence number, defining a per-source partial order on publications. Once the subscription forwarding broker receives at least one message from each replica queue, all pending

messages in the inputs queues except the last received messages are re-ordered and forwarded.<sup>1</sup> To ensure progress even when the publisher is idle, the advertisement forwarding broker periodically signals the subscription forwarding broker. We assume that messages are delivered to each input queue in-order. Therefore, we do not consider the incremental replication protocol as additional re-ordering at the input queue for the messages delivered through the replica is necessary. Apparently, the congestion rate at the brokers will further delay the ordering of the messages and thus disrupts the delivery of the publication messages to the subscribers. We evaluate the impact of the congestion rate in Sec. 5

In the scenario where the faulty broker does not drop any messages, both the synchronous and incremental protocols can ensure strong complete delivery as defined in Sec. 3.3. Furthermore, the message re-ordering algorithm can provide strong ordered delivery.

Note that message scheduling and re-ordering can be used in conjunction with a replica broker running any of the replication protocols. The combinations of protocols that result are discussed in Sec. 4.4.

#### 4.4 Discussion

This paper has developed replication protocols for replacement and load-balancing. As well, consistency properties were formalized to define correct behavior of the system.

Table 1 summarizes the properties satisfied by the protocols developed in this paper. Consider first a permanent failure of the faulty broker. For those publications that do not traverse the faulty broker for the replacement, the basic protocol may drop messages, but ensures per-source ordering. The synchronous protocol, on the other hand, satisfies both these requirements at the expense of requiring a blocking protocol that cannot resume forwarding publications until the entire replication protocol is complete. To address this issue, the incremental protocol sacrifices ordered delivery to more quickly forward publications.

For the transient failure case, publications can be load balanced among the faulty and replica brokers, and the message scheduling and re-ordering techniques can be applied to the above replication protocols. For those publications that do traverse either the faulty broker or the replica, the non-re-ordering protocols opt for quick resumption of publication delivery, while re-ordering protocols support the strong consistency properties. Note that in Table 1, the weak consistency properties for the transient failure protocols are omitted to emphasize that the strong properties are more relevant here since publications can traverse both faulty and replica brokers.

Replacement protocols are subject to comparison against structured pub/sub overlays with inherent failover mechanism. However, the comparison between

---

<sup>1</sup> If the replica broker is provisioned using the basic or synchronous protocols, each input queue is already ordered, and the publications in the queues can be re-ordered in linear time by greedily ordering the publications at the head of each queue.

<i>Failure Type</i>	<i>Protocol</i>	<i>Weak complete</i>	<i>Weak ordered</i>	<i>Strong complete</i>	<i>Strong ordered</i>
Permanent	Basic-Failover	×	✓		
	Synch-Failover	✓	✓	n/a	n/a
	Increm-Failover	✓	×		
Transient	Basic-MS	-	-	×	×
	Synch-MS	-	-	✓	×
	Increm-MS	-	-	✓	×
	Basic-MS-RO	-	-	×	✓
	Synch-MS-RO	-	-	✓	✓
	Increm-MS-RO	-	-	✓	✓

**Table 1.** Protocol properties (MS = message scheduling, RO = re-ordering)

structured and unstructured pub/sub overlays required evaluation at many different angles, which is not the scope of this paper. As we envision the pub/sub overlay leveraging from the subsystem managing resources from the resource cloud, one can argue for the naive approach of relying on live migration of virtual machines and letting the TCP to handle lost and out-of-order messages. Performance implication of VM-based failover is an interesting topic for future measurement study. But note that for the load-balancing case, even the VM-based framework must be aware of the application logic which coordinates brokers to schedule messages among the alternative paths. Moreover, unless the VM runs on a bare-bone physical machine, pub/sub overlays deployed on VMs can exhibit delays in messaging.

The extent to which publication ordering and delivery latency are affected depends on workload characteristics that are thoroughly analyzed in Sec. 5.

## 5 Evaluation

This section presents an experimental evaluation of our on-demand broker replication protocols. The analysis will focus on the scalability of the protocols with respect to different workloads, and the relative performance of the protocols in terms of metrics that measure the user-perceived service and system overhead of the protocols.

The protocols have been fully implemented and integrated into the PADRES<sup>2</sup> distributed content-based pub/sub system. All experiments were performed on a cluster of IBM x3550 machines. The machines in the cluster communicate over a 1Gbps switched Ethernet connection, and each machine contains two Intel Xeon 5120 dual-core 1.86GHz processors and 4GB of RAM. The initial topology consists of three brokers plus a replica adaptively deployed as in Fig. 4: two brokers, each representing an advertisement-forwarding and a subscription-forwarding broker, connect directly to the faulty broker. As well, some of the experiments vary the number of neighbors to up to 20 brokers. Each neighbor

<sup>2</sup> Source available for download at <http://padres.msrg.org>.

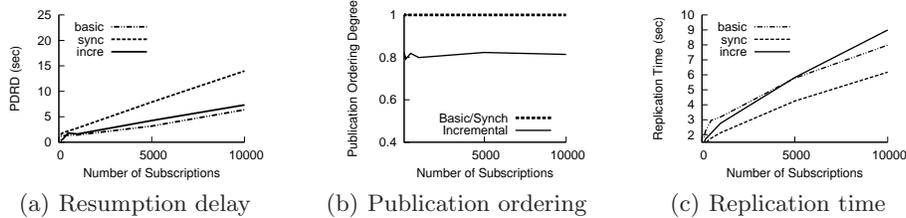
hosts between 100 and 1000 publishers, and between 100 and 10 000 subscribers. These topologies are sufficient for the evaluation because the replication protocols are localized, such that regardless of the size of the network they only involve the replica and the immediate neighbors of the faulty broker. The initial topology is updated dynamically through an emulated resource provisioner that launches a new broker on a reserved node in the cluster for execution of the replication protocols. The provisioning is triggered by a simulated failure: the handler of the output queue to the faulty broker is stopped and publications accumulate in the queue up to a configurable length. The growing queue length represents the delay in detecting the failure.

Publications and subscriptions are synthesized to reflect real-world subscription popularity [27], with subscriptions consisting of  $(attribute > value)$  predicates where  $value$  follows a Zipf distribution of degree between 0.1 and 4.0 over a default value range of [1, 20]. Similarly, publications are  $(attribute, value)$  pairs where  $value$  is Zipf distributed.

Using the above message templates, and by simply varying the skewness of the value distributions and deciding whether values are skewed towards the upper or lower portions of the range under consideration, it is possible to control a variety of workload characteristics, such as the generality of subscriptions and popularity of publications. For example, subscriptions whose values are biased towards lower values are more general, that is, they express broader interests and will match more publications. Similarly, publications with higher values will match more subscriptions, that is, they are more popular and are said to have a larger *fanout*. Constructing more complex messages would affect the pub/sub matching time and plays a part in the replication protocols. But this is an orthogonal concern to the design of the protocols. The construction would obscure the relationship between the workload parameters and the resulting workload characteristics, thus experiments become less controllable and analyzable.

The protocols are compared using four metrics: the *publication delivery re-summption delay (PDRD)* measures the delay experienced by subscribers in receiving publications due to the provisioning of the new broker; the *publication ordering degree* quantifies the extent to which publications are delivered to subscribers in the expected order; the *last message processed time (LMPT)* evaluates the delay in delivering all pending messages; and the *replication time* considers how long it takes the new replica broker to synchronize its routing state. Notice that the first three metrics are of concern to subscribers—the perceived disruption in service and the fidelity of the publication streams—whereas the final metric has no direct influence on the clients but is presented to better understand the behavior of the protocols. The details of how each metric is measured are outlined next.

The PDRD per subscription is measured as the elapsed time since the subscriber neighbor broker gets the connection request from the replica broker until this neighbor broker receives the first publication matching the subscription via the replica. The experiments typically present the average delay across all sub-



**Fig. 7.** Effect of subscription population.

scriptions. The resumption delay indicates how quickly the protocols resume the service to subscribers during replication.

The order in which publications are delivered to subscribers is quantified using a histogram which counts the frequency with which messages are displaced by various lengths [1]. More precisely, the publication ordering degree is a weighted sum of the message displacement frequencies computed as  $\sum_{D=0}^S (S - D)f(D)$ , where  $S$  is the length of the sequence of messages,  $D$  is the displacement of a message, and  $f(D)$  is the number of messages displaced by  $D$ . In the results ordering degree is normalized by dividing it by maximum ordering degree,  $S^2$ , to allow comparisons across experiments with different sequence lengths.

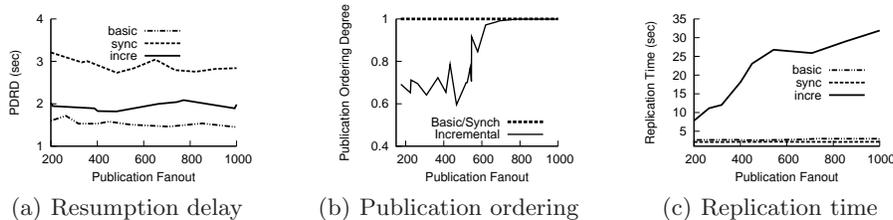
The LMPT per subscription is measured as the elapsed time since the subscriber neighbor broker gets the first publication matching the subscription via the replica until it receives the last message pending at the advertisement-forwarding broker since the fault occurred.

Replication time, measures how long it takes to replicate the routing state of the faulty broker at the newly deployed replica broker. The replication time is the elapsed time since the replica broker sends advertisement forwarding requests to its neighbors until it receives all the subscriptions from its neighbors.

In the following subsections, the replication protocols are evaluated under varying workloads in order to understand how the protocols scale with the subscription population, the publication fanout, the number of pending publications, the number of neighbors, and the congestion rate.

*Subscription population:* In this experiment, the number of subscriptions varies from 100 to 10000, with 100 pending publications. Subscriptions and publications follow a Zipf distribution with degree 0.5, and advertisements are issued to ensure all subscriptions propagate across the faulty broker and replica.

Fig. 7(a) plots how the publication delivery resumption delay varies with the number of subscriptions. The results show that the average resumption delay grows proportionally with the number of subscriptions to be forwarded. Synchronous resumption is especially sensitive to the number of subscriptions, as all the publications have to wait until the replication completes. Incremental resumption scales better, with sub-linear growth in the average delay. Moreover, the difference widens for larger subscription populations, with the incremental protocol achieving delays of 7.3 s when there are 10000 subscriptions, which is 47% less than the delays of the synchronous protocol. This shows that despite the overhead of the incremental protocol, there is a net benefit to resuming



**Fig. 8.** Effect of publication fanout

publication propagation as soon as an alternative subscription path is available through the replica. The basic replication protocol slightly performs better than the incremental protocol. However, 2% of the publications were dropped and this was consistently exhibited throughout the rest of the experiments.

Fig. 7(b) shows that the synchronous protocol provides perfect publication ordering while the incremental protocol only achieves a normalized ordering degree of about 0.8. Note that the ordering is not sensitive to the number of subscriptions but does fluctuate. This is because PADRES brokers forward matching subscriptions in the order retrieved from the routing table data structures, which for the purposes of this experiment are essentially random.

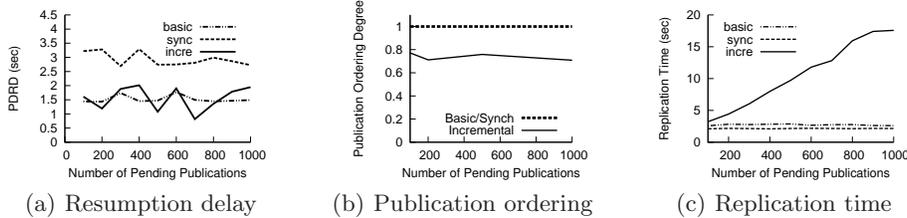
Fig. 7(c) shows that replication time increases with the number of subscriptions for both protocols. In contrast to the delay, however, the synchronous protocol spends 31% less time than the incremental. The latter requires both the replica and neighbor brokers to process publications as well as subscriptions during replication, whereas the synchronous protocol avoids processing publications until after replication is complete. Furthermore, reorganizing pending publications in the wait output queues is an expensive operation that the synchronous protocol does not perform.

A side effect of increasing subscriptions is increasing publication fanout. The next experiment isolates fanout.

*Publication fanout:* In the set of experiments whose results are summarized in Fig. 8, the skewness of the publication and subscription distributions are varied by controlling their Zipf degrees from 0.1 to 4.0. One effect of altering these parameters is on the popularity of a given publication, that is, the number of subscriptions it matches, and is represented by the average publication fanout in Fig. 8. The number of publications and subscriptions is fixed at 500 and 1000, respectively.

The results in Fig. 8(a) indicate that while the resumption delay of the incremental protocol is about 62% better than the synchronous one, neither protocol is particularly sensitive to the publication fanout, again due to the fact that brokers forward matching subscriptions in random order.

A more interesting result is in Fig. 8(b) which measures how ordering varies with publication fanout. Fanout is controlled in two ways: by varying the skewness of the publication and subscription Zipf distribution values, and selecting whether to bias towards high or low values in the range. In particular, for the lower fanout workloads publications are biased towards being less popular (their



**Fig. 9.** Effect of pending publications.

values are smaller, and hence fewer subscriptions match them), and subscriptions are biased in favor of less generality (their predicate values are larger, and hence their range of interest is narrower). Conversely, the larger fanout workloads in Fig. 8(b) are skewed towards popular publications and general subscriptions.

The first observation from Fig. 8(b) is that, as expected, the synchronous protocol achieves perfect ordering regardless of the publication fanout. The incremental protocol, on the other hand, displays more interesting behavior: although the ordering degree is as low as about 0.6, the ordering improves and becomes more stable with higher fanout workloads, eventually approaching the performance of the synchronous protocol.

Clearly, the order in which subscriptions are received by an advertisement-forwarding broker affects the ordering of the publications. As the protocols in this paper do not explicitly reorder subscriptions, it is the subscription distributions that determine the likelihood with which more general subscriptions are processed before more specific ones. in

Moving on to the replication time, Fig. 8(c) shows a roughly linear relationship between publication fanout and replication time when incremental resumption is used. With synchronous resumption, on the other hand, there is no overhead of maintaining wait-ready queues at the neighbor brokers or monitoring publication matching state at the replica broker, and thus publication fanout has no effect on replication time.

To summarize the results in Fig. 8, regardless of the popularity of publications (i.e., their fanout), the incremental protocol delivers publications sooner than the synchronous one at the expense of a more unordered publication stream. However, the publication ordering under the incremental protocol becomes more stable and approaches that of the synchronous protocol under workloads that exhibit high publication fanout.

*Pending publications:* In this experiment, the number of subscriptions is kept constant at 1000, and the number of pending publications ranges from 100 to 1000. Both subscriptions and publications follow Zipf distributions with degree 0.5 over the default content value range. The intent is to study the effects of publications that are queued while the faulty broker is being replaced with the replica. This may occur, for example, if the failure of the faulty broker is not detected quickly.

Similar to the results from Fig. 8, in Fig. 9(a) the incremental protocol outperforms the synchronous one in terms of PDRD. However, neither protocol is

sensitive to the number of pending publications. Combined with the insensitivity to fanout seen in Fig. 8(a), it can be inferred that the delay results in Fig. 7(a) are due to the subscription population alone. In terms of the publication ordering in Fig. 9(b), the incremental protocol only achieves an ordering degree of about 0.8, but this time it is not sensitive to the number of pending publications. Comparing results, it is evident the publication fanout, not the number of subscriptions or publications, dictates the ordering degree. Obviously, the replication time of the synchronous protocol is unaffected by the number of pending publications, but Fig. 9(c) shows that the replication time of the incremental protocol grows proportionally with the number of pending publications. This is because the processing and transmission of these publications is interleaved with the replication of the subscription routing state. In addition to the contention of publications and subscriptions in the queues, there is the overhead at the advertisement-forwarding broker of copying pending publications from the output queue destined to the faulty broker, and the overhead at the replica broker of recording the matching subscriptions of each received publication in order to avoid duplicates and dropped messages.

*Neighboring brokers:* This experiment focuses on the incremental publication forwarding at the replica (see Algorithm 5). First, to isolate the overhead of matching subscriptions against cached publications, 1000 subscriptions are uniformly distributed among 1 to 20 subscription-forwarding neighbors. Contrary to expectation, the incremental protocol does not suffer much in terms of resumption delay as shown in Fig. 10(a). The delay increases by only 44% despite a 20 fold increase in neighbors.

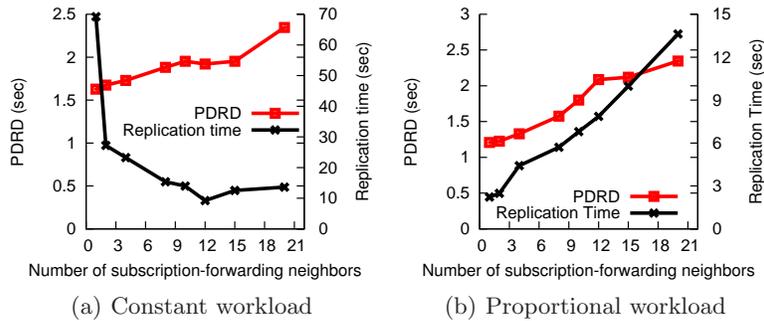
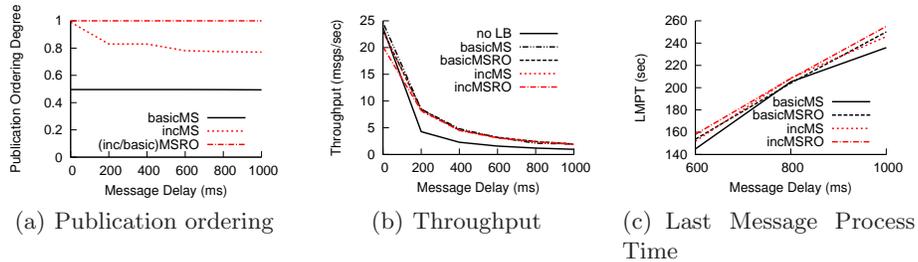


Fig. 10. Neighboring Brokers

To further stress the protocol, in Fig. 10(b) the workload is increased in proportion to the number of subscription-forwarding brokers, with 500 subscriptions per neighbor. Even in this case, the incremental protocol scales well. Going from 1 to 20 neighbors barely doubles the delay and increases the replication time by only a factor of five.

Further analysis found that the benefit of parallelized I/O operations by the brokers downplays the overhead of incremental forwarding, as the replication

time improves substantially with the number of subscription-forwarding brokers in Fig. 10(a). Better insight can be obtained by refactoring the underlying communication driver used by the brokers.<sup>3</sup>



**Fig. 11.** Effect of congestion rate.

*Congestion Rate:* This experiment highlights the benefit of message scheduling in the case of transient failures. Load-balancing the publication stream to replicas caused significant message displacement as shown on Figure 11(a). With re-ordering disabled, the basic and incremental protocols with message scheduling yielded ordering degrees as low as 0.77 and 0.49, respectively. Despite the severe message displacement, re-ordering added little processing overhead and only showed a 5% increase in the average LMPT. Most importantly, the throughput, in terms of the messages per second processed by the system, doubles when load-balancing is enabled. The results show that the adaptive load-balancing algorithms effectively manage congestion.

*Discussion:* The experiments above consistently show that the incremental and basic protocols deliver publications relatively quicker. The price, however, is corrupted publication ordering and message drops for the incremental and the basic algorithms, respectively. The analysis of the incremental protocol, confirmed in the experiments, reveals that publication ordering depends on at least the following factors: (1) content-distribution in publications and subscriptions; and (2) the order subscriptions are forwarded towards the advertisement-forwarding brokers via a replica. While (1) is a characteristic of the workload, (2) is under the control of the incremental forwarding protocol. This argues for a more intelligent protocol that reorders subscriptions based on the pattern of publication arrival order and workload distribution. Such a protocol may provide both the benefits of fast publication delivery resumption delay and good publication ordering. The development of an order- and content-aware subscription forwarding policy, however, is the subject of future work.

<sup>3</sup> PADRES currently uses RMI for inter-broker communication.

## 6 Conclusion

This paper addresses provisioning of pub/sub brokers through on-demand adaptive replication for highly-available content-based pub/sub streaming service. Our framework conforms with emerging grid and cloud computing infrastructure but isolates the replication processes to the neighbors of the faulty broker and achieves timely, complete and ordered delivery of publications to clients in the events of permanent and transient failures. We developed a total of nine protocols to deal with trade-offs between performance and consistencies specific to content-based pub/sub depending on the various needs of clients. Our replication protocols are crisp and simple with reliable coordination among the participants. Our publication re-delivery methods based on a novel wait-ready and caching mechanism effectively handles issues with content-based routing on replicas. Our framework provide clients flexible options by supporting various levels of consistencies: strong and weak variants of ordered and complete delivery of publications. Evaluations of the fully implemented protocols running on a real system with various content distributions promises support their suitability to large-scale and time-sensitive distributed pub/sub applications.

## References

1. Metrics for degree of reordering in packet sequences. In *LCN*, 2002.
2. I. Aekaterinidis and P. Triantafillou. Pastrystrings: A comprehensive content-based publish/subscribe dht network. In *ICDCS*, 2006.
3. S. Bholra et al. Exactly-once delivery in a content-based publish-subscribe system. *DSN*, 2002.
4. M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.
5. N. Carvalho et al. Scalable QoS-based event routing in publish-subscribe systems. In *NCA*, 2005.
6. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM ToCS*, 2001.
7. S. Castelli, P. Costa, and G. P. Picco. Hypercbr: Large-scale content-based routing in a multidimensional space. In *INFOCOM*, 2008.
8. G. Cugola, D. Frey, A. L. Murphy, and G. P. Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe. In *SAC*, 2004.
9. G. Cugola, P. G. P., and M. A. L. Towards dynamic reconfiguration of distributed publish-subscribe middleware. In *SEM*, 2002.
10. B. Cully et al. Remus: high availability via asynchronous virtual machine replication. In *NSDI*, 2008.
11. E. Fidler et al. Distributed publish/subscribe for workflow management. In *ICFI*, 2005.
12. L. Fiege et al. Engineering event-based systems with scopes. In *ECOOP*, 2002.
13. J. Gray et al. The dangers of replication and a solution. In *SIGMOD*, 1996.
14. R. S. Kazemzadeh et al. Reliable and highly available distributed publish/subscribe service. In *SRDS*, 2009.

15. S. Krishnamurthy, W. H. Sanders, W. H. S, and M. Cukier. A dynamic replica selection algorithm for tolerating timing faults. In *DSN*, 2001.
16. S. Osman et al. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 2002.
17. G. P. Picco et al. Efficient content-based event dispatching in the presence of topological reconfiguration. In *ICDCS*, 2003.
18. P. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *IEEE Network*, 2004.
19. P. R. Pietzuch and J. Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In *DEBS*, 2003.
20. Platform Computing. IT Modernization: From Grid to Cloud in Financial Services, 2009.
21. I. Rose, R. Murty, et al. Cobra: Content-based filtering and aggregation of blogs and RSS feeds. In *NSDI*, 2007.
22. M. Satyanarayanan et al. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. on Computers*, 1990.
23. C. Schuler, H. Schuldt, and H.-J. Schek. Supporting reliable transactional business processes by publish/subscribe techniques. In *TES*, 2001.
24. D. Serrano, M. Pati no-Martínez, R. Jiménez-Peris, and B. Kemme. An autonomic approach for replication of internet-based services. In *SRDS*, 2008.
25. Y. Tock et al. Hierarchical clustering of message flows in a multicast data dissemination system. In *IASTED PDCS*, 2005.
26. M. Wiesmann et al. Understanding replication in databases and distributed systems. *ICDCS*, 2000.
27. W. Y-M. et al. Subscription partitioning and routing in content-based publish/subscribe systems. In *DISC*, 2002.
28. Y. Zhao et al. Subscription propagation in highly-available publish/subscribe middleware. In *Middleware*, 2004.