On-demand Replication for Failover in Content-based Publish/Subscribe Overlays

Young Yoon, Vinod Muthusamy and Hans-Arno Jacobsen
Department of Electrical and Computer Engineering, 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 a broker that misbehaves—perhaps due to failure, congestion, or periodic maintenance—with a new broker in a running system. The key to our replacement technique is distributed replication of routing information on an adaptively deployed new broker which serves reliable re-route paths for publication messages. Our technique focuses on avoiding lengthy disruption of messaging service, and supporting best-effort weak consistency on publication delivery via the replica.**

I. Introduction

Distributed content-based publish/subscribe is a powerful messaging substrate for various event-based systems including RSS filtering [14], stock-market monitoring [17], network management [11], algorithmic trading with complex event processing [13], and business process execution [16]. 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. A monitoring subsystem detects an SLA violation caused by a faulty broker (B3). The broker may have crashed permanently, be experiencing temporary congestion, or may just need to undergo planned maintenance. In any case, a resource management system provisions a new broker (B4) to replace the faulty one. 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 redirected through the replica. This paper addresses the problem of replacing a faulty broker by replicating its state. The fault detection and resource management subsystems are complementary concerns and out of the scope of this paper.

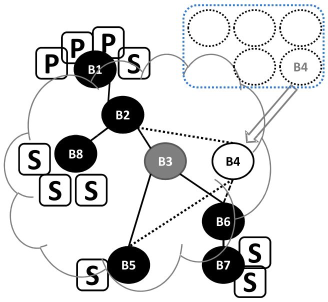


Fig. 1. On-demand replication

The replication model described here differs from traditional replication work in redundant routing overlays, distributed databases, virtual machines, operating systems, or file systems [2], [3], [5], [6], [10], [12], [15], [20]. Rather than having a set of 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 no longer available and its state cannot be directly replicated. 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 [2], [3], [20].

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 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 correctness 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, disturbing only the

Middleware Systems Reserch Group Technical Report, University of Toronto, November, 2009 available at
http://www.msrg.utoronto.ca/publications/pub_view.php?id=118

neighbors of a faulty broker.

This paper makes the following contributions: (1) In Sec. III, 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. IV then develops two additional replication protocols. A synchronous protocol satisfies the required consistency properties, and an incremental protocol relaxes one of the consistency properties in order to achieve better performance. (3) Finally, Sec. V offers an experimental evaluation of the replication protocols in a real system under a variety of workloads.

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

II. 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 are summarized.

Background: This paper builds on the PADRES distributed, content-based pub/sub platform [7] that uses advertisement-based routing [4], [8]. 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), a logical list of `[advertisement,last hop]` tuples. Subscriptions propagate in reverse direction along the advertisement trees, and are stored in the Publication Routing Table (PRT), a logical list of `[subscription,last hop]` tuples, and is used to route publications. 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.

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 [2]. 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 [20]. 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 [3] pub/sub network reserves resources and constructs paths with sufficient capacities to satisfy clients' quality-of-service requirements. Certain brokers can be replicated 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 [5], [10]. 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: There has been substantial work in replicating distributed databases for the purposes of availability or fault tolerance. The 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 [9], [18].

Database replication is a fundamentally different model of replication than is the focus of 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 replacement for an existing broker 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 it has failed.

System level replication: By replicating an entire machine no application-specific knowledge is required. The Remus system [6] 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 [12] which replicates processes at the operating system level, and Coda [15] 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.

III. On-demand broker replication

This section explores the issues with supporting on-demand broker replication. When a broker is determined to be faulty, a new broker is provisioned to replace the faulty one. 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. III-A 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. IV.

A. Basic replication protocol

A straightforward 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.

The protocols in this paper, on the other hand, 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. 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. If a neighbor is an advertisement-forwarding broker, it forwards advertisements from its routing table to the replica. The replica in turn stores the advertisements in its routing table and relays them to its other neighbors. A neighbor with subscriptions that match a relayed advertisement functions as a subscription-forwarding broker and forwards its matching subscriptions along the reverse path of the advertisement back to the replica. Again, the replica stores the subscriptions in its routing table and forwards them 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.

Algorithm 1: Basic protocol at replica broker R

```
Input: $N \leftarrow \{n | n$ is a neighbor of faulty broker $F\}$
if receive initialize message then
    send request for advertisements to all $n \in N$;
if receive advertisement $adv$ from any $n \in N$ then
    insert $adv$ into $SRT$;
    forward $adv$ to all $n_i \in N$ where $n_i \neq n$;
if receive subscription $sub$ from any $n \in N$ then
    insert $sub$ into $PRT$;
    $\mathbb{A}_n \leftarrow \{adv | adv \in SRT \wedge adv$ matches $sub\}$;
    forward $sub$ to the lasthop of each $adv \in \mathbb{A}_n$
```

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     $\mathbb{S}_n \leftarrow \{sub | sub \in PRT \wedge sub$ 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 $PRT$;
```

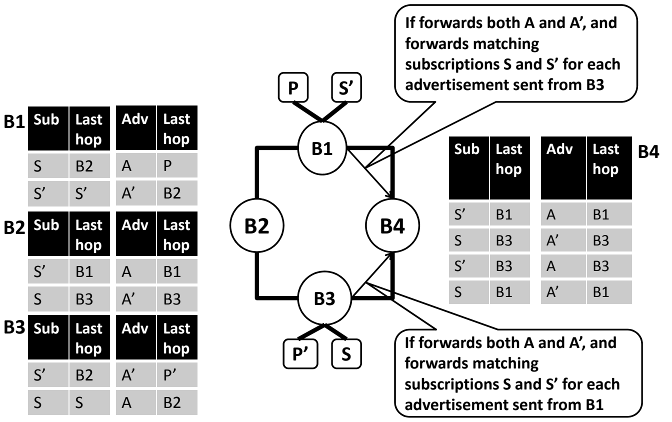
The algorithm that runs at the replica as part of the basic replication protocol is detailed in Algorithm 1, and the corresponding algorithm at the neighbors is specified in Algorithm 2. The reader will notice in Lines 2 and 6 of Algorithm 2 that the protocol avoids forwarding advertisements and subscriptions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

Not shown in the algorithms is that 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.

B. Analysis of basic replication protocol

1) Key benefits: The replication 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. In this way, regardless of the size and complexity of the broker overlay, only a small number of brokers participate in any given replication operation, ensuring minimal impact to the remainder of the overlay.

As mentioned above, the protocol also takes care to avoid duplicate routing table entries. Without such precautions, in-



B1 Sub	Last hop	Adv	Last hop
S	B2	A	P
S′	S′	A′	B2

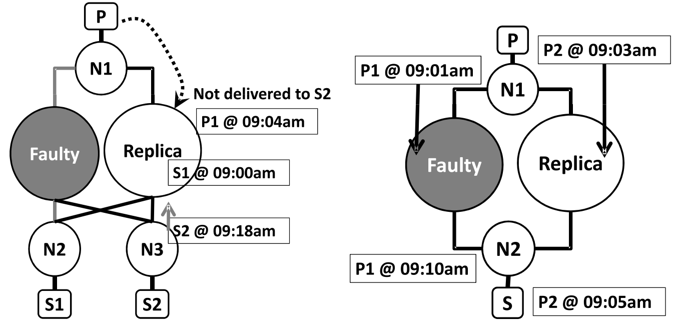
B2 Sub	Last hop	Adv	Last hop
S′	B1	A	B1
S	B3	A′	B3

B3 Sub	Last hop	Adv	Last hop
S′	B2	A′	P′
S	S	A	B2

Sub	Last hop	Adv	Last hop B4
S′	B1	A	B1
S	B3	A′	B3
S′	B3	A	B3
S	B1	A′	B1

If forwards both A and A′, and forwards matching subscriptions S and S′ for each advertisement sent from B3
P
S′
B1
B2
B4
B3
P′
S
If forwards both A and A′, and forwards matching subscriptions S and S′ for each advertisement sent from B1

Fig. 2. Example: Duplicate entries if neighbors do not filter



(a) *Undelivered publications* due to subscription ordering.
(b) *Out of order publications* when two replicas operate at different service rate.

Fig. 3. Examples of violations of correctness with replicas

correct routing behavior can occur as shown in Fig. 2. 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.

2) Complexity: The number of messages exchanged during the replication protocol is linear with the number of neighbors of the faulty broker and the routing state at this broker. More precisely, the message 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 routing tables of the faulty broker, respectively.

The derivation is as follows. Algorithm 1 requires $|N|$ messages 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). Note that since no duplicate advertisements and subscriptions 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.

3) 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. 3(a) the advertisement-forwarding broker forwards publications through the replica before all the subscription-forwarding brokers have sent their subscriptions to the replica. The publication is then forwarded only towards those brokers whose subscriptions are in the replica's routing tables; the replica does not yet know there are other interested subscribers.

Another limitation of the basic protocol 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. 3(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.

C. Replication properties

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

Property 1: 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: 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.

It is important to 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 *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, $|\mathbb{P}_{incons}|$, that are potentially delivered in an inconsistent manner—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, $|\mathbb{P}_{incons}| \geq notification\_rate \times (t_d - t_f)$, 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.

There are two ways to avoid losing publications already sent to a broker that has permanently failed. One approach would be 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 [10]. A second approach is to persist all publications in stable storage and suffer the significant overhead this would impose. While such schemes are possible, this paper opts for replication protocols that can be easily integrated into existing pub/sub systems. 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 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. 3(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.

To address the limitations of the basic replication protocol, Sec. IV develops more sophisticated replication protocols.

IV. Advanced replication protocols

A. Synchronous replication protocol

This section develops a synchronous replication protocol that satisfies the weak consistency property defined in Sec. III-C. 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.

1) 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 that the protocol can safely commence the publication delivery phase.

Loosely speaking, the replica must retrieve all the advertisements from its neighbors, and then all the matching subscriptions from its neighbors. To let the replica know when it has received all advertisements or subscriptions from its neighbors, 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 3. 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 3. (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 3.

Algorithm 3 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.

2) Synchronous publication delivery resumption: An advertisement-forwarding broker has an important role to play. In the first phase, once it is aware a neighbor is faulty and will be replaced with a replica, it immediately suspends forwarding any messages in the output queue to the faulty broker. When

Algorithm 3: Synchronous replication at replica R

```
Input: $N \leftarrow \{n | n$ is a neighbor of faulty broker $F\}$
if receive initialize message then
    send request for advertisements to all $n \in N$;
    $state \leftarrow$ WaitForAdvs;
    $PendingLastAdvs \leftarrow |N|$;
    $PendingLastSubs \leftarrow 0$;
if receive advertisement adv from any $n \in N$ then
    insert $adv$ into $SRT$;
    forward $adv$ to all $n_i \in N$ where $n_i \neq n$;
    if $state = WaitForAdvs$ then
        $PendingLastSubs \leftarrow PendingLastSubs + |N| - 1$;
if receive subscription sub from any $n \in N$ then
    insert $sub$ into $PRT$;
    $\mathbb{A}_n \leftarrow \{adv | adv \in SRT \wedge adv$ matches $sub\}$;
    forward $sub$ to the lasthop of each $adv \in \mathbb{A}_n$
if receive lastAdv control message from any $n \in N$ then
    $PendingLastAdvs \leftarrow PendingLastAdvs - 1$;
    if $PendingLastAdvs = 0$ then
        $state \leftarrow$ WaitForSubs;
        if $PendingLastSubs = 0$ then
            $state \leftarrow$ Replicated;
            send $replication\_done$ message to all $n \in N$;
if receive lastSub control message from any $n \in N$ then
    $PendingLastSubs \leftarrow PendingLastSubs - 1$;
    if $state = WaitForSubs \wedge PendingLastSubs = 0$ then
        $state \leftarrow$ Replicated;
        send $replication\_done$ message to all $n \in N$;
```

Algorithm 4: 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      $\mathbb{A}_n \leftarrow \{adv | adv \in SRT \wedge adv.lasthop \neq R\}$;
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($OutputQueue_R$);
```

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 4 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. It does so by deferring sending publications until replication is complete so traditional pub/sub forwarding properties can be relied on. The negative point of this design is 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.

B. 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.

The incremental replication protocol requires that 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 the subscription-forwarding brokers. The algorithms at the publisher-forwarding broker and the replica are outlined below.

1) Incremental publication delivery resumption at advertisement-forwarding broker: Recall in the synchronous protocol, the advertisement-forwarding broker suspended forwarding publications to the replica. In the incremental protocol, on the other hand, publications destined to a broker undergoing replication are indexed in a wait-ready queue data structure as illustrated in Fig. 4. There are three classes of queues in this structure: a priority queue for messages whose control messages should be continued to be handled during the 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. 4.

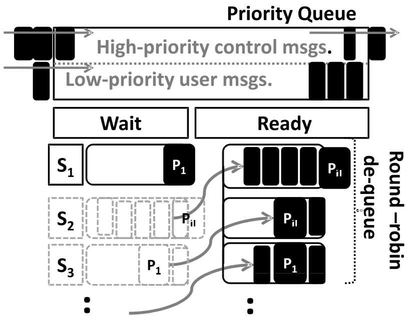


Fig. 4. Wait-ready queue data structure for incremental publication delivery resumption.

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.

Algorithm 5: Incremental publication forwarding at replica R

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

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.

Returning to duplicate publications in the wait queues, 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 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.

2) 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 these subscriptions, the replica caches publications until the replication protocol is complete.

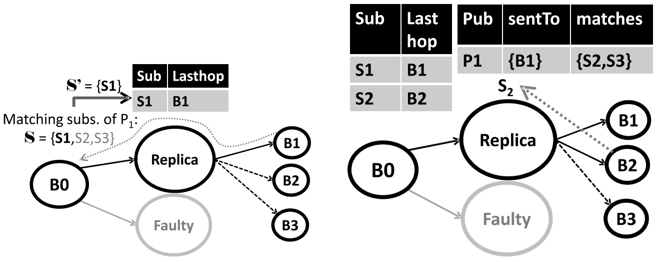
The incremental publication forwarding algorithm at the replica, shown in Algorithm 5, makes use of two data structures. The `sentTo` structure records the set of neighbors a publication has already been forwarded to, and the `matches` structure indexes the set of publications that match a given subscription 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 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. 5 illustrates a replica selectively forwarding publications. Consider subscriptions S_1, …, S_n from brokers B_1, …, B_n that all match publication P_1 that is held in a wait queue at broker B_0. Suppose in Fig. 5(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 $\mathbb{S} = \{S_1, \ldots, S_n\}$ in this case. In Fig. 5(a), the replica first forwards P_1 to B_1, but then also notices that subscriptions $S_2, \ldots, S_n$ were included as matching subscriptions in P_1 but these subscriptions are not yet in the replica's routing table. As seen in Fig. 5(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 $S_2, \ldots, S_n$. 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



Sub	Lasthop
S1	B1

(a) First matching subscription arrives at $B0$. Replica notices not all matching subscriptions have arrived.

Sub	Last hop
S1	B1
S2	B2

Pub	sentTo	matches
P1	{B1}	{S2,S3}

(b) When $S2$ arrives at the replica, it is matched against the cached publication.

Fig. 5. Example of selective publication forwarding at a replica

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. 4 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 matches subscriptions not only against advertisements in the routing table but also the cached publications. On the other hand, caching publications at the replica means they are now one hop closer to the subscription-forwarding broker saving computation and communication delays. For example, in the scenario in Fig. 5, 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.

C. Discussion

This paper has developed three replication protocols: basic, synchronous, and incremental. As well, consistency properties were formalized to define correct behavior of the system when a faulty broker is replaced with a replica.

Protocol	*Weak complete*	*Weak ordered*
Basic	×	✓
Synch.	✓	✓
Increm.	✓	×

TABLE I
PROTOCOL PROPERTIES

Table I summarizes the properties that each protocol satisfies. For those publications that do not traverse the faulty broker, 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.

The extent to which publication ordering is corrupted and delivery latency depends on the workload is experimentally analyzed in Sec. V.

V. Evaluation

This section presents an experimental evaluation of the on-demand broker replication protocols developed in Sec. IV. The protocols have been fully implemented and integrated into the PADRES[1] distributed content-based pub/sub system, and 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.

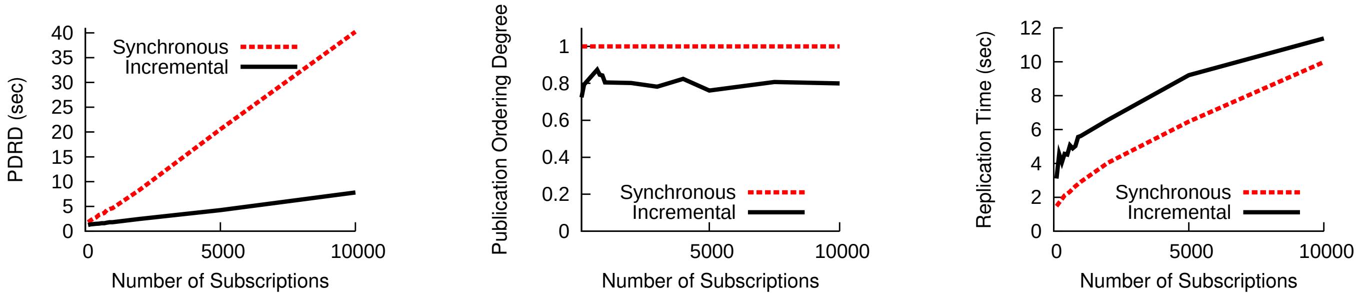
The initial topology consists of three brokers plus a replica adaptively deployed as in Fig. 3: two brokers, each representing an advertisement-forwarding and a subscription-forwarding broker, connect directly to the faulty broker. As well, some of the experiments vary the number of neighbors to up to 20 brokers. Each neighbor 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 [19], 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*. While constructing more complex messages would affect the pub/sub matching time—which plays a part in the replication protocols but is an orthogonal concern to the design of the protocols—doing so would only obscure the relationship between the workload parameters and the resulting workload characteristics making the experiments more difficult to control and the results less straightforward to analyze.

The protocols are compared using three metrics: the *publication delivery resumption delay* 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, and the *replication time* considers how long it takes the new replica broker to synchronize its routing state. Notice that the first two metrics are of concern to subscribers—the perceived disruption in service and the fidelity of the publication streams—whereas the third metric has no direct influence on the clients but is

[1] Source available for download at http://padres.msrg.org.



(a) Publication resumption delay

(b) Publication ordering

(c) Replication time

Fig. 6. Effect of subscription population: The incremental replication protocol scales better than the synchronous protocol in terms of publication delivery resumption delay. The synchronous protocol, however, satisfies *weak ordered* delivery and offers quicker replication time.

presented to better understand the behavior of the protocols. The details of how each metric is measured are outlined next.

The publication delivery resumption delay 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 subscriptions. The resumption delay indicates how quickly the protocols resume the service to subscribers during fail-over.

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 third metric, 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, and the number of neighbors.

A. Subscription population

In this experiment, the number of subscriptions varies from 100 to 10 000, 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. 6(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 7.8 s when there are 10 000 subscriptions, an improvement of 81% over the synchronous protocol. This shows that despite the overhead of the incremental protocol, there is a net benefit to resuming publication propagation as soon as an alternative subscription path is available through the replica.

Fig. 6(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.

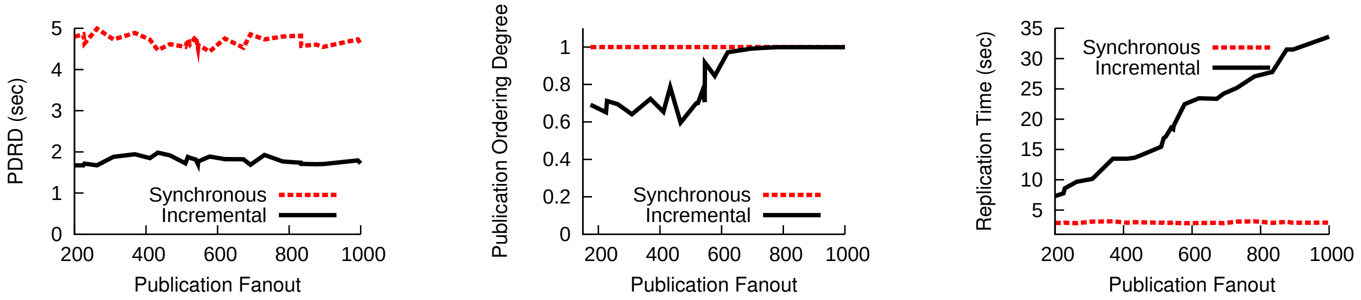
Fig. 6(c) shows that replication time increases with the number of subscriptions for both protocols. In contrast to the delay, however, the synchronous protocol now outperforms the incremental one by about 42%. The latter requires both the replica and neighbor brokers to process publications as well 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.

B. Publication fanout

In the set of experiments whose results are summarized in Fig. 7, 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. 7. The number of publications and subscriptions is fixed at 500 and 1000, respectively.

The results in Fig. 7(a) indicate that while the resumption delay of the incremental protocol is about 62% better than the synchronous one, neither protocol is particularly sensitive



(a) Publication resumption delay

(b) Publication ordering

(c) Replication time

Fig. 7. Effect of publication fanout: Publication delivery is insensitive to the fanout for both protocols, the ordering degree of the incremental protocol improves with the fanout, and replication time remains constant for the synchronous protocol but increases with the fanout for the incremental protocol.

to the publication fanout, again due to the fact that brokers forward matching subscriptions in essentially random order.

A more interesting result is in Fig. 7(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 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. 7(b) are skewed towards popular publications and general subscriptions.

The first observation from Fig. 7(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.

The variation in ordering experienced by the incremental protocol can be understood using Fig. 8, which depicts a worst-case scenario in which publications are displaced by the maximum amount. At some advertisement-forwarding broker, there are four publications waiting to be sent that were received in the order from P_1 to P_4, with P_1 being the least popular and P_4 the most popular. Furthermore, there are four subscriptions that are forwarded to this broker, with S_1 the most general and S_4 the most specific. Fig. 8 illustrates the state of the incremental protocol's data structure in which the publications are indexed by their matching subscriptions.

Suppose the broker in Fig. 8 receives subscriptions by increasing generality, from S_1 to S_4. When the broker receives S_1, it will forward P_4 (the only publication matching S_1); and when it receives S_2, it forwards P_3 (the only remaining publication matching S_2). By processing subscriptions in this order, publications are forwarded in reverse order, with P_4 sent first and P_1 last. By simply reversing the subscription processing order—from most to least general—publications are sent in the desired order.

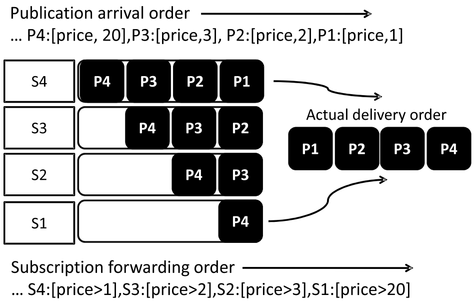

Publication arrival order
… P4:[price, 20],P3:[price,3], P2:[price,2],P1:[price,1]
S4
P4
P3
P2
P1
S3
P4
P3
P2
S2
P4
P3
S1
P4
Actual delivery order
P1
P2
P3
P4
Subscription forwarding order
… S4:[price>1],S3:[price>2],S2:[price>3],S1:[price>20]

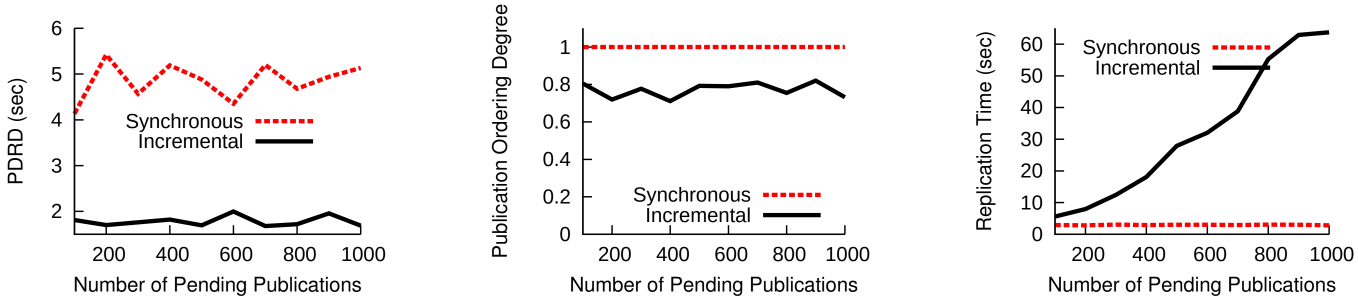
Fig. 8. Publication reordering when pubs skewed towards low fanout, and subs towards narrow interests.

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. Continuing with the example in Fig. 8, if the subscription population is biased towards more specificity (such as S_1), then assuming subscriptions are received in random order, it is likely that these more populous specific subscriptions will be processed earlier, leading to greater displacement of publications.

This phenomenon is evident in Fig. 7(b) where lower fanout workloads (which have more specific subscriptions) correlate with worse publication ordering for the incremental protocol. Also notice that the ordering metric fluctuates more with lower fanout. This is because it is only more *likely* that more specific subscriptions are processed before the more general ones in these workloads; a fortuitous early forwarding of a general subscription will improve publication ordering even for the low fanout workloads.

Moving on to the replication time, Fig. 7(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. 7, 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.



(a) Publication resumption delay (b) Publication ordering (c) Replication time

Fig. 9. Effect of pending publications: The incremental protocol's replication time increases with the number publications. Otherwise, both protocols are insensitive to pending publications for all metrics.

C. 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.

Measuring publication delivery resumption delay in Fig. 9(a), the incremental protocol outperforms the synchronous one by an average of 63%. However, neither protocol is sensitive to the number of pending publications. Combined with the insensitivity to fanout seen in Fig. 7(a), it can be inferred that the delay results in Fig. 6(a) are due to the subscription population alone.

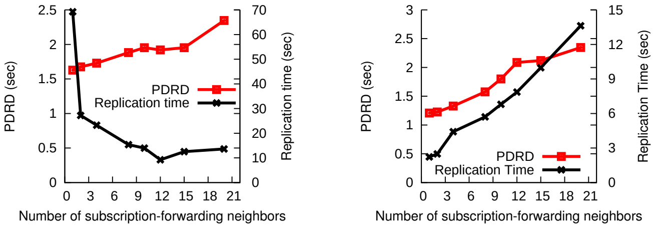
In terms of the publication ordering in Fig. 9(b), the incremental protocol only achieves an ordering degree of about 0.8, but this time it is not sensitive to the number of pending publications. Comparing results, it is evident the publication fanout, not the number of subscriptions or publications, 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.

D. Neighboring brokers

This experiment focuses on the incremental publication forwarding at the replica (described in Algorithm 5).

First, to isolate the overhead of matching subscriptions against cached publications in Algorithm 5, 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 about 44% despite there being 20 times as many neighbors.



(a) Load-balanced workload (b) Proportional workload

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. [2]

E. Discussion

The experiments above consistently show that the incremental algorithm delivers publications quicker under all conditions evaluated. The price, however, is the ordering of publications that can be corrupted to varying degrees. 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.

[2] PADRES currently uses RMI for inter-broker communication.

The development of an order- and content-aware subscription forwarding policy, however, is the subject of future work.

The results also show that the incremental protocol suffers a longer replication duration. However, as this factor has no direct effect on the quality of service experienced by the pub/sub clients in the system, the seemingly poor performance of the incremental protocol by this measure should be given less weight than the benefits observed in terms of the other metrics.

VI. Conclusion

This paper addresses on-demand replication of pub/sub brokers. Unlike traditional replication where replicas actively synchronize, the on-demand model only provisions a replica after the fault. On-demand replication conforms with on-demand grid and cloud computing architectures, but requires that replication proceed with no prior participation of the faulty broker, imposes minimal overhead to a correctly functioning system, scales to large overlays, and achieves timely, complete, and ordered delivery of publications to clients.

During replication publications may not be dropped, or delivered out of order. These violations are formalized in two properties: complete delivery and ordered delivery. In on-demand replication, it is not possible to recover messages dropped by the faulty broker without resorting to fundamental modifications to the pub/sub routing algorithms, and so weaker variants of the consistency properties are defined.

Three protocols were developed: a rudimentary basic protocol makes no attempt to guarantee delivery, a synchronous protocol guarantees the weaker consistency properties, and an incremental protocol sacrifices ordering to resume publication delivery before replication completes. The incremental protocol uses a novel wait-ready queue and caching mechanism to quickly forward publications as subscription paths are reconstructed.

In evaluations the incremental protocol achieves significant delivery delay improvements under different subscriber populations, publication fanout, number of neighbors, and publication stream lengths. For example, with 10 000 subscriptions, the average subscriber is serviced in 7.8 seconds, a 81% improvement over the synchronous protocol.

The publication ordering degree with the incremental protocol was dependent on the subscription propagated order. In experiments where subscription ordering was not controlled the ordering degree was about 0.8, but when the publication fanout was high, ordering approached perfect order.

The results in this paper are promising, and especially encouraging is the possibility of improving the ordering of the incremental protocol. We plan to develop an analytical model to determine how to better control subscription propagation so as to improve both ordering and delay metrics. Furthermore, a tunable parameter would allow trading off qualities such as the ordering, delay, and completeness of publication delivery.

References

[1] Metrics for degree of reordering in packet sequences. In *LCN*, 2002.

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

[3] N. Carvalho et al. Scalable QoS-based event routing in publish-subscribe systems. In *NCA*, 2005.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM ToCS*, 2001.

[5] G. Cugola et al. Towards dynamic reconfiguration of distributed publish-subscribe middleware. In *SEM*, 2002.

[6] B. Cully et al. Remus: high availability via asynchronous virtual machine replication. In *NSDI*, 2008.

[7] E. Fidler et al. Distributed publish/subscribe for workflow management. In *ICFI*, 2005.

[8] L. Fiege et al. Engineering event-based systems with scopes. In *ECOOP*, 2002.

[9] J. Gray et al. The dangers of replication and a solution. In *SIGMOD*, 1996.

[10] R. S. Kazemzadeh et al. Reliable and highly available distributed publish/subscribe service. In *SRDS*, 2009.

[11] B. Mukherjee et al. Network intrusion detection. *IEEE Network*, 1994.

[12] S. Osman et al. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 2002.

[13] P. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *IEEE Network*, 2004.

[14] I. Rose, R. Murty, et al. Cobra: Content-based filtering and aggregation of blogs and RSS feeds. In *NSDI*, 2007.

[15] M. Satyanarayanan et al. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. on Computers*, 1990.

[16] C. Schuler, H. Schuldt, and H.-J. Schek. Supporting reliable transactional business processes by publish/subscribe techniques. In *TES*, 2001.

[17] Y. Tock et al. Hierarchical clustering of message flows in a multicast data dissemination system. In *IASTED PDCS*, 2005.

[18] M. Wiesmann et al. Understanding replication in databases and distributed systems. *DCS*.

[19] W. Y-M. et al. Subscription partitioning and routing in content-based publish/subscribe systems. In *DISC*, 2002.

[20] Y. Zhao et al. Subscription propagation in highly-available publish/subscribe middleware. In *Middleware*, 2004.