Foundations for Highly Available 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 (SLAs) are strictly enforced, maintaining high availability and efficiency of the broker overlay is critical. To support these requirements, a set of three primitive operations are proposed to allow arbitrary transformations of an overlay to an optimal one, and two additional primitives are developed to enable on-demand adjustments when there are permanent or transient failures. Both sets of primitive operations minimize disruption by preserving message delivery guarantees even as the overlay topology changes, requiring no overhead when the overlay is not being modified, operating on a fixed neighborhood of brokers regardless of the size of the overlay, and completing quickly under a variety of conditions.**

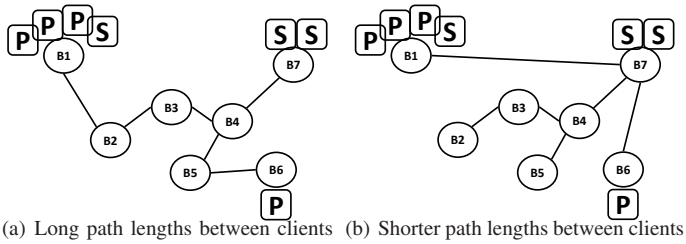
I. Introduction

Distributed content-based publish/subscribe overlays provide a powerful messaging substrate that are employed in a wide variety of applications including RSS filtering [26], stock-market monitoring [28], network management and monitoring [12], [21], algorithmic trading with complex event processing [18], [25], distributed business process execution [19], and business activity monitoring [12]. In many of the aforementioned enterprise applications, high-availability is a crucial requirement as service disruptions can be costly to the business.

High availability needs to be considered at every layer of the application stack, including the physical network and application architecture, but in the context of this paper, high availability refers to concerns related to the overlay network. Both functional requirements, such as message delivery guarantees and ordering semantics, as well as non-functional properties, such as short message delivery delays are relevant when it comes to the design of the messaging overlay.

Moreover, availability requirements must be maintained despite variations to the application workloads or system resources. A common approach to accommodate these changes is to adjust or reconfigure the overlay network in some way. For example, a broker in the overlay that crashes unexpectedly can be substituted with a replacement broker. Similarly, when a broker becomes overloaded due to an unexpected message burst, a replica of the broker can be quickly provisioned to share some of the load. Later, when the load subsides, one of these replicas can be removed. In this paper, such localized replication and consolidation of brokers are referred to as *on-demand* changes to the overlay.

The overlay may also be adjusted in a more deliberative manner. For example, an organization's policies may require a broker to be removed from the overlay for maintenance purposes, such as to apply security patches. This requires the overlay network to be seamlessly rewired around the broker to be removed without affecting the operation of the overlay. More drastic changes to the overlay may also be necessary to optimize the network for enduring changes to the application workload. For instance, consider the overlay in Fig. 1(a) in which an application workload has evolved such that there is now heavy traffic among clients at brokers B_1, B_6, and B_7. Paths between these brokers are between 3 and 5 hops in the current overlay, but an overlay optimization algorithm could determine that the overlay in Fig. 1(b) would result in a lower average message paths for this workload [2]. This paper classifies such purposeful transformations of the broker topology as *planned* changes to the overlay.



(a) Long path lengths between clients (b) Shorter path lengths between clients

Fig. 1. Example of a planned transformation of the overlay

This paper develops a set of foundational primitive operations to carry out both planned and on-demand modifications of the broker overlay. The primitives for the former are sufficient to realize arbitrary reconfigurations of the topology, and those for the latter are flexible and agile enough to address permanent broker failures as well as transient processing (matching) overloads, or input and output link bottlenecks.

Regarding the planned overlay transformation, existing works have focused on algorithms to devise a new overlay with various optimization criteria including minimizing the number of brokers in the overlay to reduce cost, optimizing path lengths or network latencies, controlling node degrees, or providing sufficient processing and network capacities [2], [9], [15], [20]. A significant limitation, however, is that these works have largely neglected the practical issues with migrating an existing deployed overlay to the new one while minimizing the disruption to the service even as the overlay is being transformed. The primitives proposed in this paper therefore now provide the practical means to make use of existing optimal overlay design algorithms on real, running systems.

As for adjustments to the overlay to repair permanent or transient failures, the traditional approach is to deploy a set of replicas that continuously synchronize their states

in anticipation of failure [4], [11], [17], [23], [27], [32]. We provide primitives to support a fundamentally different model whereby a replica for a faulty broker is deployed only on-demand. Such a model can be more cost-effective than having an over-provisioned system. Fig. 2 shows how the on-demand primitives can be used to grow or shrink overlays adaptively. This on-demand model, however, brings

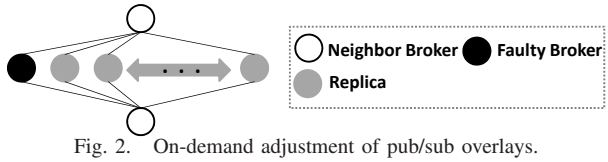


Fig. 2. On-demand adjustment of pub/sub overlays.

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, and thus its state is not available for direct replication. Instead, our protocols exploit the soft state nature of the broker routing tables and reconstruct a faulty broker's state from that of its neighbors. Another challenge is that the replication solutions should impose minimal overhead until replication or consolidation 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 negligible overhead to a normally functioning system both before replication is required and after it completes, so that existing routing algorithms can operate at full speed.

To better understand the disruptions both the planned and on-demand primitives need to avoid, this paper formalizes consistency properties that specify invariants on message delivery and ordering. Another disruption concern is the timely delivery of messages. Sometimes timeliness and consistency can be opposing goals, and this paper proposes protocols with novel queue management techniques that can tradeoff one goal for the other. Finally, the protocols should scale to large broker overlays and so all the primitives are designed to always involve a small, fixed neighborhood of brokers so any disruption is short and localized to small portion of the network.

This paper makes the following contributions: (1) Sec. II first formalizes both functional and non-functional notions of disruptions in a pub/sub overlay. The primitives developed are designed to satisfy these properties. (2) Then in Sec. III, a set of three primitives are designed to carry out arbitrarily complex transformations of an overlay by incrementally altering a small neighborhood of broker in the overlay. This incremental approach minimizes disruptions to the system, and is largely seamless to most of the overlay. (3) In Sec. IV, two on-demand overlay adjustment primitives are presented: replication and consolidation. A number of implementations of these primitives are developed that provide tradeoffs among the disruption properties from Sec. II. (4) Finally, Sec. V offers an experimental evaluation of the planned and on-demand protocols in a real system under a variety of workloads.

II. Definition of Disruption

The disruption of service caused by the overlay transformation is defined as violation of functional and non-functional requirements. This section defines the properties in order to guide the design of the transformation primitives for a non-disruptive pub/sub service.

A. Functional requirements

To formally capture functional properties that can be violated by the overlay transformation, this section formulates consistency in terms of the publications delivered to a pair of subscribers.

In the definitions, the notation $p \prec p'$ refers to the case where two publications p and p' were published by the same publisher, and that p was published before p'.

Property 1. Strong Complete Delivery: *Consider a subscriber s_1 that has issued subscriptions (at any time) and has received matching publication set P_1, and that publications p and p' appear in P_1, where $p \prec p'$. If there exists a p'' where $p \prec p'' \prec p'$, then p'' must belong to P_1.*

Informally, the complete delivery property requires that publications be delivered to all interested subscribers. In the context of this paper, it can be used to ensure that the transformation protocols do 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.

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.

This paper seeks to develop transformation protocols that can be easily integrated into existing pub/sub systems. Specifically, there is no assumption that the routing protocol tolerates failures, and so publications can be lost, violating the strong delivery property. It becomes necessary then to relax the requirements for replacing a faulty broker through on-demand replication. Our on-demand replication primitives, therefore, accept inconsistencies arising from publications sent to the faulty broker before its failure was detected. The relaxed requirements are expressed in Properties 3 and 4, which are weaker forms of Properties 1 and 2, respectively.

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 contain the replica broker.

B. Non-functional requirements

In addition to the functional requirements defined above, it is also important to consider non-functional notions of disruption. Thus, we introduce metrics to capture the performance of the system as it undergoes modifications to the overlay.

The *publication delivery resumption delay* measures the delay experienced by subscribers in receiving publications due to the reconfiguration of the topology. The *last message processed time* evaluates the delay in delivering all pending messages. The *operation time* is the overall duration of the overlay transformation.

Notice that these metrics are of concern to subscribers—the perceived delay in service can potentially lead to violations of SLAs. The metrics also reflect the cost of the overlay transformation.

It can be a conflicting goal to satisfy both functional and non-functional requirements at the same time. Various trade-off scenarios are supported by our configurable primitive operations that are explained more in detail in the following sections.

III. Planned Transformation

Over time, a given overlay G may become sub-optimal for the current workload, and existing techniques can be used to determine a new optimal overlay G' [2], [9], [15], [20]. What is missing is a mechanism to incrementally migrate the existing overlay G to the optimal one G' without disrupting the operation of the running system. We refer to this overlay redesign as a *planned transformation*.

To support such planned transformations, this section develops a set of primitive operations that allow any overlay to be incrementally transformed to another overlay with minimal disruption. Sec. III-A first presents the primitive operations and proves that they are sufficient to perform any general transformation of an overlay. The proof makes no assumption on the overlay routing algorithms and so is applicable to any connected acyclic overlay. Then, Sec. III-B provides an implementation of these primitives for a class of content-based routing overlays. The implementation focuses on minimizing the disruption to the system by constraining the operation to a constant set of brokers in an isolated region of the network.

A. Primitive Operations

This section proposes three primitive operations that can be used to perform arbitrary transformations of a broker overlay. The discussion is agnostic to the routing protocols used by the brokers, and so the broker overlay is abstracted with a graph G. Note that G represents the broker overlay, and does not include the clients in the topology.

A large set of distributed pub/sub protocols assume an acyclic topology, and we capture this by saying a graph G is *valid* iff it is a connected, undirected, acyclic graph. An operation is *safe* if it transforms any valid graph G to another valid graph G'. A sequence of operations is referred to as a *plan*, and the result of applying a plan P to a graph G is another graph denoted as $P(G)$. Also, a plan P is safe if all the operations in it are safe. In the discussion below, the immediate neighbors of a node n are denoted as $\mathbb{N}(n)$, and nodes with only one neighbor are referred to as *outer* nodes.[1]

Operation	Description
APPEND(n_i, n_j)	Create a new node n_i and connect it to node n_j. Results in n_i being an outer node in the graph.
DETACH(n_i)	Remove an outer node n_i from the graph.
SHIFT(n_i, n_j, n_k)	Replace the link between n_i and n_j with one between n_i and n_k, where $\{n_i, n_k\} \in \mathbb{N}(n_j)$ and $n_i \neq n_k$.

TABLE I
PRIMITIVE OPERATIONS FOR PLANNED TRANSFORMATION

Operation	Description
MOVE(n_i, n_j, n_k)	Similar to the SHIFT primitive, but n_k may be any node.
INSERT(n_i, N)	Create a new node n_i with links to all nodes $n \in N$.
DELETE(n_i)	Remove node n_i from the graph.
SEQUESTER(n_i, n_j)	Remove all links from n_i except for the one to n_j, where $n_j \in \mathbb{N}(n_i)$. Results in n_i being an outer node in the graph.
PLACE(n_i, N)	Similar to the INSERT operation, but here n_i already exists in the graph.

TABLE II
COMPOSITE OPERATIONS FOR PLANNED TRANSFORMATION

The three proposed primitive operations are outlined in Table I. They can be used to add a new outer node to a graph (APPEND), remove an outer node from a graph (DETACH), or replace a connection to a node n with a connection to a neighbor of n (SHIFT).

Now, we prove with Theorem 1 that the above primitives are sufficient to perform any general transformation, given Claim 1, Lemma 1, and Corollary 1 [31]. Also, our proof uses the composite operations as outlined in Table II. The implementations of these operations are expressed in terms of other composite or atomic operations [31].

Claim 1. *The primitive and composite operations are safe.*

Lemma 1. *Given a sequence of nodes $n_1, \ldots, n_k$, there exists a sequence of* INSERT$(n_1, -), \ldots,$ INSERT$(n_k, -)$ *operations to generate any valid graph G consisting of nodes $n_1, \ldots, n_k$.*

Corollary 1. *Given a graph G, a graph G^A generated by an* APPEND *operation to G, and a graph G^I generated by an* INSERT *operation to G, there is a plan to transform G^A to G^I using only the* SHIFT *primitive (i.e., without adding or removing nodes).*

Theorem 1. *There is always a plan to transform any valid graph G to any other valid graph G'' using only the* APPEND, DETACH, *and* SHIFT *primitives.*

Proof: Let G' be a graph that consists of the same nodes as G''.

First we prove that there is a plan P to transform G to some G', and then prove there is a plan P' to transform G' to G''.

Plan P is relatively straightforward: for each node n in G but not in G'', DELETE(n); and then for each node n not in G but in G'', APPEND(n, n_k), where n_k is some node in both

[1] An outer node here is with respect to the broker overlay. In particular, a broker with clients connected to it is still an outer node if it has only one connection to another broker.

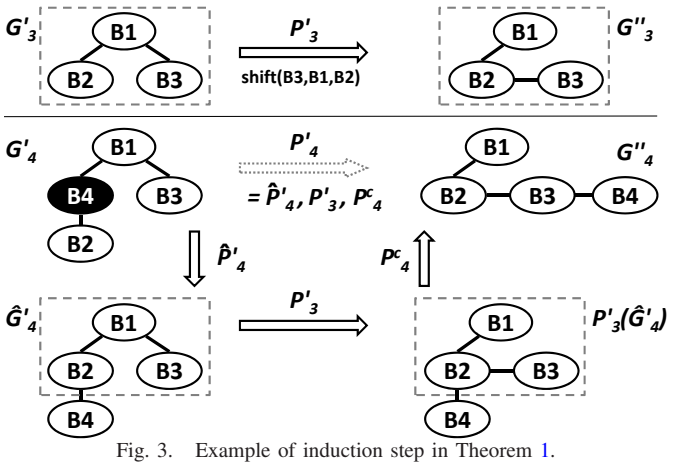


Fig. 3. Example of induction step in Theorem 1.

G and G''.

It remains now to be shown that there is a plan P' to transform G' to G'', and both graphs have identical nodes. We will show that plan P' exists by induction.

The induction will apply to a sequence of graphs $G'_1, \ldots, G'_m$, where G'_k is a k-node graph, and an INSERT$(n_{k+1}, -)$ operation applied to G'_k generates G'_{k+1}. Also, $G'_m = G'$ and hence G' consists of the nodes $n_1, \ldots, n_m$. By Lemma 1, this sequence of graphs exists. Similarly, graphs $G''_1, \ldots, G''_m$ exist, where $G''_m = G''$.

Let P'_k be a plan that transforms graph G'_k to graph G''_k. It is trivial to show that P'_k exists for $1 \leq k \leq 3$, that is, for graphs up to three nodes.

We will now show that if P'_k exists, then P'_{k+1} also exists. Specifically, we will construct a P'_{k+1} by moving node n_{k+1} "out of the way", applying plan P'_k on a subgraph, and then moving node n_{k+1} back to the correct position in the graph. The plan is as follows:

1) To begin, apply SEQUESTER$(n_{k+1}, -)$ to graph G'_{k+1} to generate a graph $\hat{G}'_{k+1}$ where node n_{k+1} is now an outer node in the graph. This operation will be referred to as plan $\hat{P}'_{k+1}$.
2) Now apply P'_k, the plan for the k-node graph, on $\hat{G}'_{k+1}$. This is safe because P'_k is reducible to a set of SHIFT operations which only affect the nodes in graph G'_k and the k-node subgraph of $\hat{G}'_{k+1}$ is identical to G'_k.
3) Finally, apply a plan P^c_{k+1} that transforms $\hat{G}'_{k+1}$ to G''_{k+1}. By Corollary 1, a plan P^c_{k+1} exists.

So, the plan P'_{k+1} consists of the sequence of plans $\hat{P}'_{k+1}$, P'_k, and P^c_{k+1}.

■

An example of the induction step in the above proof is shown in Fig. 3, where a plan P'_3 to transform a 3-node graph is used to construct a plan P'_4 to transform a 4-node graph.

In this section, we have proposed three primitive graph transformation operations: APPEND, DETACH, and SHIFT. We have also proven that these primitives can be used to perform any transformation of an acyclic, connected graph. Therefore, these primitives can be used to carry out a transformation of a content-based pub/sub overlay as dictated by existing optimal-overlay construction algorithms [2], [9], [15], [20].

Broker	Changes to routing tables
n_i	$\forall sub \in PRT$ where $sub.lasthop = n_j$ $sub.lasthop = n_k$ $\forall adv \in SRT$ where $adv.lasthop = n_j$ $adv.lasthop = n_k$
n_j	$\forall sub \in PRT$ where $sub.lasthop = n_i$ $sub.lasthop = n_k$ $\forall adv \in SRT$ where $adv.lasthop = n_i$ $adv.lasthop = n_k$
n_k	$\forall sub \in PRT$ where $sub.lasthop = n_j$ $sub.lasthop = n_i$ $\forall adv \in SRT$ where $adv.lasthop = n_j$ $adv.lasthop = n_i$

TABLE III
REQUIRED CHANGES TO BROKER ROUTING STATES FOR A SHIFT(n_i, n_j, n_k) OPERATION.

Note that we have only shown that a plan exists to perform any desired transformation, and have not provided an algorithm to generate a plan. While there are many plans possible for a given transformation, it is not even clear what the optimization criteria for a plan should be. A plan may attempt to minimize the number of primitive operations, isolate the modifications to a portion of the network, constrain the maximum degree of nodes in the intermediate steps, control paths lengths between certain nodes, or maintain some other invariant at each step. Constructing such paths is a rich avenue for future research, but is out of the scope of this paper.

B. Implementation

Having devised the abstract primitives, this section continues with a discussion on the implementation of these primitives in a class of content-based pub/sub overlays. In particular, we focus on advertisement-based acyclic pub/sub overlays that route messages by reverse path forwarding [6], [13].

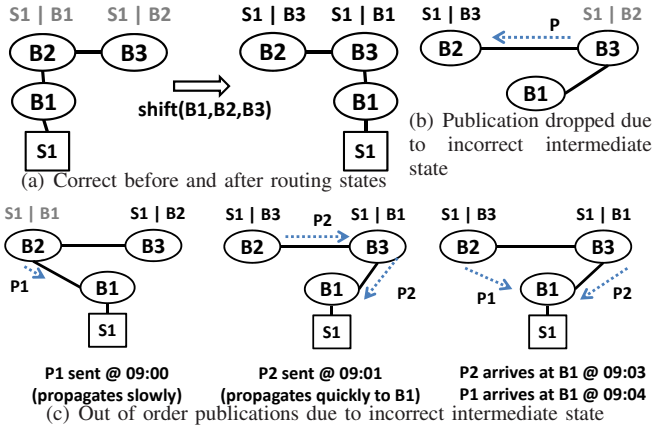
Briefly, the routing paths are constructed as follows. A publisher submits an advertisement that describes the collection of publications it may publish. These advertisements are flooded through the overlay with each broker recording the last hop of the message in its SRT (subscription routing table). A subscriber expresses its interest in publications with a subscription. Each broker forwards a subscription on the reverse path of any intersecting advertisements in the SRT, and also records the last hop of the subscription in its PRT (publication routing table). Finally, publications from publishers are forwarded along the reverse path of subscriptions until they are delivered to subscribers.

The APPEND and DETACH primitives constitute fundamental operations and are already supported by existing pub/sub systems [13]. Moreover, since they are concerned with outer brokers that do not have any pub/sub client connected to them, their implementations are not concerned with potential disruption. Therefore, the remainder of this section will concentrate only on implementing the SHIFT primitive.

As defined in Table I, the SHIFT operation replaces the connection between brokers n_i and n_j with one between brokers n_i and n_k. To accomplish this, the routing tables at n_i, n_j, and n_k must be modified as outlined in Table III.

For example, consider the transformation in Fig. 4(a) where a Broker B_1 with one subscription S_1 performs a SHIFT(B_1, B_2, B_3) operation. The figure shows the correct routing state at each broker both before and after the operation in terms of the last hops of the indexed subscriptions.

While Table III defines the correct final routing state after a SHIFT operation, the protocol that implements this operation



(a) Correct before and after routing states

(b) Publication dropped due to incorrect intermediate state

(c) Out of order publications due to incorrect intermediate state

Fig. 4. Examples of correct and inconsistent intermediate routing states due to a SHIFT(n_i, n_j, n_k) operation

must ensure that the consistency properties from Sec. II-A are preserved. The remainder of this section presents some examples of how these properties may be violated by an incorrect protocol, and then develops an implementation that does satisfy the consistency properties.

1) Inconsistent states during SHIFT *operation:* Since it is impossible to update the routing tables at all three brokers simultaneously in a distributed system, the routing configuration depicted in Fig. 4(b) represents a possible global state while the protocol is running. In this case, broker B_3 contains a stale last hop entry for subscription S_2, and a matching publication that arrives at broker B_3 is incorrectly routed to B_2 and is not delivered to the subscriber at B_1. In this example, a publication that is not delivered to all interested subscribers violates the strong complete consistency property.

Consider another possible sequence of intermediate states shown in Fig. 4(c). In this case, broker B_2 receives a publication P_1 just before it has updated its routing tables, and forwards P_1 to broker B_1 over a stale connection. After B_2's routing state has been updated, another publication P_2 from the same publisher arrives and is correctly forward to B_3 and then to B_1. Due to unpredictable delays in the network, it is possible that P_2 arrives at broker B_1 before P_1, violating the strong ordered consistency property.

2) Synchronous SHIFT *protocol:* One way to implement the SHIFT primitive that preserves the strong consistency properties is to first buffer all publications at brokers n_i, n_j, and n_k. Once the routing state has been updated, the buffered publication can be forwarded according to the new routing table entries. This is a three-phase protocol as sketched in Algorithm 1. Because the entire protocol is always confined to exactly three neighboring brokers in the overlay, the protocol can finish quickly and few publications will be delayed. Moreover, publications traversing over other brokers in the overlay are unaffected. We evaluate the performance of this protocol in Sec. V.

IV. On-demand adjustment

This section proposes primitive operations for on-demand adjustments of an overlay. These primitives can be used to handle unexpected service disruption due to congestion and failure. We develop protocols that support various combinations of the properties defined in Sec. II.

Algorithm 1: Synchronous SHIFT protocol

Input: n, the current broker.
1 **on receive** *SHIFTREQ*(n_i, n_j, n_k) **do**
2 start buffering non-control messages;
3 **if** $n = n_i$ **then** forward SHIFTREQ message to n_j;
4 **if** $n = n_j$ **then** forward SHIFTREQ message to n_k;
5 **if** $n = n_k$ **then** send SHIFTACK(n_i, n_j, n_k) message to n_k;
6 **on receive** *SHIFTACK*(n_i, n_j, n_k) **do**
7 modify routing state according to Table III;
8 **if** $n = n_k$ **then** forward SHIFTACK message to n_j;
9 **if** $n = n_j$ **then** forward SHIFTACK message to n_i;
10 **if** $n = n_i$ **then** send SHIFTFIN(n_i, n_j, n_k) message to n_i;
11 **on receive** *SHIFTFIN*(n_i, n_j, n_k) **do**
12 resume processing buffered messages;
13 **if** $n = n_i$ **then** forward SHIFTFIN message to n_j;
14 **if** $n = n_j$ **then** forward SHIFTFIN message to n_k;
15 **if** $n = n_k$ **then** // nothing else to do

Operation	Description
REPLICATE(n_i, n_j)	Replicate n_i into n_j to increase computing capacity.
CONSOLIDATE()	Remove under-utilized replicas.

TABLE IV
PRIMITIVE OPERATIONS FOR ON-DEMAND TRANSFORMATION

A. Primitive operations

As opposed to conventional approaches of over-provisioning an overlay where one must accept continuous synchronization overhead [4], [11], [17], [23], [27], [32], we opt for operations that are executed *on-demand*, as listed in Table IV. Similar to the primitives for planned overlay transformation, (de-)allocation of a replica requires only localized adjustments to an overlay. In particular, the on-demand primitive operations involve only the immediate neighbor brokers of the broker to be replicated (the *faulty broker*). Specifically, the routing entries of the faulty broker are populated on the newly deployed broker (the *replica*) by exchanging subscriptions and advertisements with the immediate neighbors.

As with planned overlay transformation, functional requirements can be violated. For example, messages that traverse one of the redundant paths created by the replicas can be delivered out-of-order, or messages delivered to a replica with only partially populated routing tables can be lost. The following sections introduce protocols to support different combinations of consistency and performance.

B. Synchronous replication protocol

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

Algorithm 2: 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 ∈ SRT where adv.lasthop = F do
6          adv.lasthop ← R;
7      foreach sub ∈ PRT where sub.lasthop = F do
8          sub.lasthop ← R;
9      A_n ← {adv|adv ∈ SRT ∧ adv.lasthop ≠ R};
10     forward every adv ∈ A_n to R;
11     send lastAdv message to R;
12 if receive replication_done message from R then
13     ResumeUserMessageHandling(OutputQueue_R);
```

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 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.

2) 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 2 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 are redirected 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.

The synchronous replication protocol achieves weak consistency by deferring sending publications until replication is complete so traditional pub/sub forwarding properties can be relied on. The weakness is the lengthy service disruption, as subscribers receive no publications until replication is complete. To address this, the next section develops an incremental replication protocol for time-sensitive pub/sub applications.

C. 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. 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.

1) Incremental publication delivery resumption: Publications destined to a broker undergoing replication are indexed, as illustrated in Fig. 5, in a wait-ready data structure 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. 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 replication is complete.

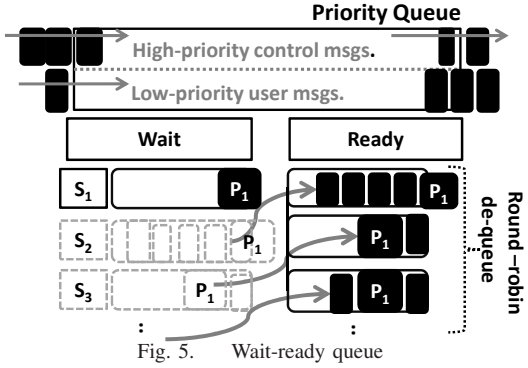


Fig. 5. Wait-ready queue

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.

Algorithm 3: Incremental pub. forwarding at replica R

```
   Input: N ← {n|n is a neighbor of faulty broker F}
1  /* Init. and handle advs as in synch. algo. */;
2  if receive subscription sub from any n ∈ N then
3      /* Match and forward sub as in synchronous algo
       then do the following. */;
4      ℙ ← matches[sub];
5      matches[sub] ← ∅;
6      foreach p ∈ ℙ do
7          if sub.lasthop ∉ sentTo[p] then
8              forward p to sub.lasthop;
9              sentTo[p] ← sentTo[p] ∪ sub.lasthop;
10 if receive publication pub from any n ∈ N then
11     𝕊 ← {sub ∈ PRT|sub matches pub ∧ sub.lasthop ≠ n};
12     foreach s ∈ 𝕊 do
13         forward pub to s.lasthop;
14         sentTo[pub] ← sentTo[pub] ∪ sub.lasthop;
15     𝕊′ ← extractMatchingSubs(pub);
16     foreach s ∈ 𝕊′ where s.lasthop ∉ sentTo[pub] do
17         matches[s] ← matches[s] ∪ pub;
```

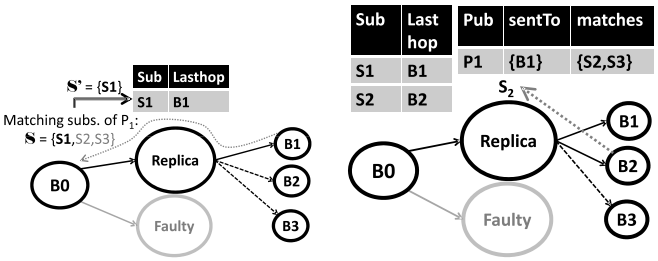
2) Incremental publication forwarding at replica: The second piece of the incremental replication protocol requires the replica to correctly send the incrementally forwarded publications to all interested subscribers. To ensure publications are correctly forwarded towards subscriptions that arrive late, the replica caches publications until the replication protocol is complete.

The incremental publication forwarding algorithm at the replica, shown in Algorithm 3, 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 3 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.

Fig. 6 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. 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 $\mathbb{S} = \{S_1, \ldots, S_n\}$ in this case. In Fig. 6(a), the replica first forwards P_1 to B_1, but then also notices that subscriptions



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

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

Fig. 6. Selective publication forwarding

$S_2, \ldots, S_n$ were included as matching subscriptions in P_1 but these subscriptions are not yet in the replica's routing table. As seen in Fig. 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 $S_2, \ldots, S_n$. 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 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 publications 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. Which factor dominates will depend on the workload.

D. Load-balancing and consolidation protocols

It is possible that the faulty broker is still functional, but simply operating at a rate that fails to fulfill its service level agreement. 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.

1) 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.

2) Message re-ordering at subscription-forwarding broker: To avoid out-of-order messages, 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 input queues except the last received messages in each queue are re-ordered and forwarded.

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. II-A. Furthermore, the message re-ordering algorithm can provide strong ordered delivery.[2]

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. IV-E.

3) Consolidation: Finally, consolidation of replicas is achieved by removing the message path to the replica to be deallocated. Specifically, the to-be-deallocated replica is removed from the list of $lasthops$ of every advertisement and subscription at the neighboring brokers. Incoming messages can continue to be routed to the remaining replicas without any disruption. We assume that consolidation of a broker does not overlap the replication operation. Also, consolidation is only applicable to replicas deployed for transient failures. Based on these assumptions, consolidation supports the strong consistency properties as the protocol strictly prohibits any messages going through the replica that is to be deallocated.

E. Discussion

Table V summarizes the properties satisfied by the on-demand protocols. The basic protocol does not employ any synchronization or incremental message forwarding, and thus is susceptible to message failures. It, however, ensures complete publication delivery for a subscription path established via the replica. This basic protocol serves as a base line approach to motivate the need for and evaluate the more advanced protocols.

Each of the four consistency properties in Table V may be required in different application scenarios. For example, for applications that emit a stream of location updates, such as online games, a weak complete delivery semantic may be acceptable as any information lost due to missed events will be obsoleted by newer events. The weak ordered semantic may be sufficient for a system that records the bills paid by customers or the completion of batch processing jobs. In these scenarios, the fact that a bill was paid or a job completed is more important than the order in which they occurred. As for the stronger properties, a strong complete delivery semantic is required by a system that monitors for radioactive alerts in a nuclear power generating station where, due to safety regulations, it is unacceptable to miss any measurement samples. Finally, strong ordered guarantees are essential in any electronic commerce or financial application where reliable transaction processing requires predictable ordering properties. There are clearly classes of applications for which each of the consistency properties are relevant, and each application can choose the protocols that support the required properties and achieve the desired performance.

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

TABLE V
PROTOCOL PROPERTIES (MS = MESSAGE SCHEDULING, RO = RE-ORDERING)

Note that a replica creates a localized cyclic subgraph. However, any ill effects, such as cyclic routing, are prevented by the selective forwarding of subscriptions and advertisements during replication [31]. Actually, the group of replicas can be viewed as a single virtual node with elastic capacity, which can be used in conjunction with the planned primitives. Detailed interaction of the replicas and the planned operations are the subject of future study.

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

V. EVALUATION

This section presents an experimental evaluation of our protocols in order to gain insights into the runtime disruptions that occur during the execution of the protocols. The protocols have been fully implemented and integrated into the PADRES[3] distributed content-based pub/sub system. As shown in Table VI, all experiments were performed on a cluster of IBM x3550 machines, with publications and subscriptions that are synthesized to reflect real-world subscription popularity [30].

Runtime environment	IBM x3550 cluster
Networking	1Gbps switched Ethernet connections
Node capacity	Two Intel Xeon 5120 dual-core processors 4GB of RAM.
Workload	Publication/subscription popularity follows Zipf distribution with degree 0.1 - 4.0; 100 - 1 000 publishers and 100 - 10 000 subscribers.

TABLE VI
EXPERIMENT SETUP SUMMARY

[2]The proofs for these claims are detailed in [31].

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

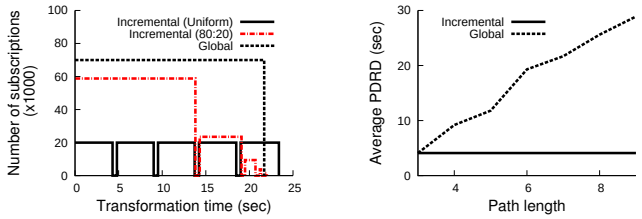
Subscriptions consist of $(attribute > value)$ predicates where $value$ follows a Zipf distribution over a default value range of $[1, 20]$. Similarly, publications are $(attribute, value)$ pairs where $value$ is Zipf distributed. Using the above message templates, and by simply varying the skewness 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 which plays a part in our protocols. However, this is an orthogonal concern to the design of the protocols, and would only obscure the relationship between the workload parameters and the resulting workload characteristics, rendering the experiments more difficult to control and analyze.

Recall that the key metrics to measure the non-functional disruptions are *publication delivery resumption delay* (PDRD), *last measured publication time* (LMPT), and *operation time (OT)*. PDRD is measured per subscription as the elapsed time since the subscriber neighbor broker gets the connection request from either a shifting broker or a replica broker until this neighbor broker receives the first publication matching the subscription through the new connection. The resumption delay indicates how quickly the protocols resume the service to subscribers during planned or on-demand transformations of an overlay. The LMPT is measured as the elapsed time since the subscriber neighbor broker gets the first publication matching the subscription via the new connection until it receives the last message pending at the advertisement-forwarding broker since the fault occurred. Particularly, we measured LMPT to reflect the ordering overhead imposed on on-demand replication protocols when handling transient failures. These experiments typically present the average delay across all subscriptions. OT is the duration from when a protocol is triggered to when it terminates. For the on-demand replication protocols, OT is measured as the time it takes to replicate the routing state of the faulty broker. Specifically, the replication time (RT) is the elapsed time since the replica broker sends advertisement forwarding requests to its neighbors until it receives all the subscriptions from its neighbors.

The key metric for assessing functional disruption is the order in which publications are delivered to subscribers. The order is quantified using a histogram which counts the frequency with which messages are displaced by various lengths [3]. 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 the maximum ordering degree, S^2, so as to allow comparisons across experiments with different sequence lengths. If a sequence is perfectly ordered, then the ordering degree equals to 1. The effect of ordering degree is application-dependent.

In the following subsections, we focus more on the replication protocols as they support various level of consistencies, and thus may exhibit more interesting runtime behavior. First, however, we evaluate the benefit of having primitives that are designed to be executed locally and incrementally.

a) Effect of incremental transformation: Our primitive operations are to be locally executed in order to minimize disruptions regardless of the scale of an overlay. In this experiment, we consider an overlay that includes a path of 7 brokers labeled B_1 to B_7 with an average of 10 000 subscriptions issued at each broker. We attempt to reconnect B_1 to B_7, once by directly rewiring the topology, and once incrementally by executing a sequence of five SHIFT operations (from Section III) waiting 0.5 second between each operation. Figure 7(a) shows that each incremental execution of the SHIFT primitive exhibits an average PDRD of approximately 4 seconds regardless of the distribution of subscriptions at the brokers along the path. The resumption delay for the direct wiring, however, took approximately 21.7 seconds as there are more routing states to update. In total the incremental plan took about 1.7 seconds longer than the direct wiring, but causes less disruption during the execution of the plan. In another experiment, when the subscription population is skewed (each successive broker along the path is assigned 80% of the remaining subscriptions in the workload), the PDRD for the SHIFT operator varied with the number of subscriptions at the broker. While the PDRD was sensitive to the number of updated routing entries in that experiment, this is not always the case. In another experiment shown in Fig. 7(b) where we varied the path length from 3 to 9, the average PDRD remained constant regardless of the path length of an overlay if our planned primitives were executed incrementally and locally. However, the average PDRD grew linearly with the path length when direct wiring was used.

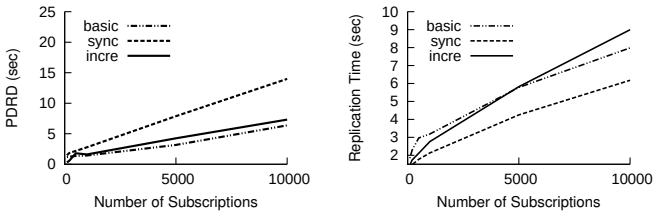


(a) PDRD during transformation. (b) Effect of path lengths on PDRD.

Fig. 7. Effect of incremental transformation in a planned rewiring of an pub/sub overlay.

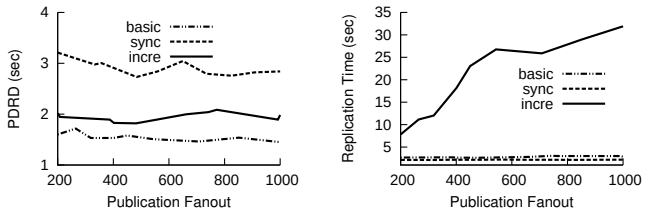
In the following sections, the replication protocols are evaluated under varying workloads in order to understand how the protocols scale with the subscription population, the publication fanout, the number of pending publications, the number of neighbors, and the congestion rate.

b) Subscription population: This experiment shows the effect of subscription populations on our replication protocols. Fig. 8(a) plots how the PDRD varies with the number of



(a) Resumption delay (b) Replication time

Fig. 8. Effect of subscription population.



(a) Resumption delay (b) Replication time

Fig. 9. Effect of publication fanout

subscriptions. The results show that the average PDRD 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 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 10 000 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 publication propagation as soon as an alternative subscription path is available through the replica. The basic replication protocol has a slightly better resumption delay than the incremental protocol, but it drops 2% of the publications, a result that is consistent across all the other experiments. We also observed 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. 8(b) 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 defers 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.

c) Publication fanout: In this experiment 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. 9. The number of publications and subscriptions is fixed at 500 and 1000, respectively.

The results in Fig. 9(a) indicate that while the PDRD 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 essentially random order.

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 are skewed towards popular publications and general subscriptions.

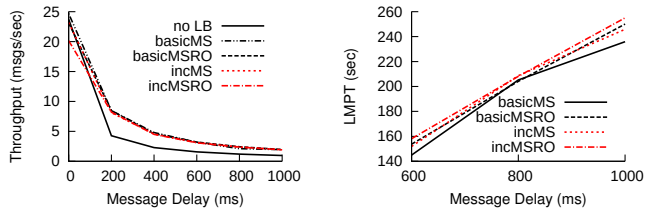
Other results, not shown here, indicate that, as expected, the synchronous protocol achieves perfect ordering regardless of the publication fanout. The incremental protocol, on the other hand, displays more interesting behavior: although the ordering degree (defined earlier in the evaluation setup) is 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.

We also observed that 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.

Moving on to the replication time, Fig. 9(b) 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.

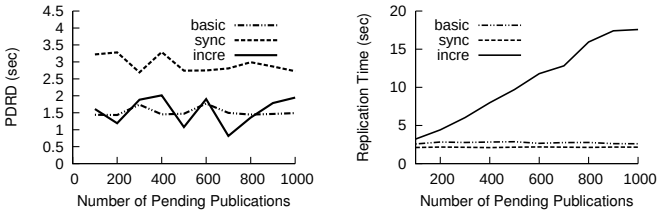
To summarize the results in Fig. 9, regardless of the popularity of publications (i.e., their fanout), the incremental protocol delivers publications sooner than the synchronous one at the expense of a more unordered publication stream. However, the publication ordering under the incremental protocol becomes more stable and approaches that of the synchronous protocol under workloads that exhibit high publication fanout.

d) Pending publications: This experiment studies 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. 9, in Fig. 10(a) the incremental protocol outperforms the synchronous one in terms of PDRD, and neither protocol is sensitive to the number of pending publications. Combined with the insensitivity to



(a) Throughput (b) Last Message Publication Time

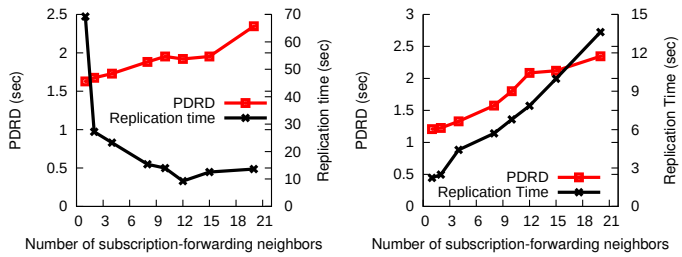
Fig. 12. Effect of congestion rate.



(a) Resumption delay (b) Replication time

Fig. 10. Effect of pending publications.

fanout seen in Fig. 9(a), it can be inferred that the delay results in Fig. 8(a) are due to the subscription population alone. In terms of publication ordering, the incremental protocol only achieves an ordering degree of about 0.8, but it is not sensitive to the number of pending publications. Comparing results, it is evident that 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. 10(b) 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.



(a) Constant workload (b) Proportional workload

Fig. 11. Neighboring Brokers

e) Neighboring brokers: This experiment focuses on the incremental publication forwarding at the replica (see Algorithm 3). 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. 11(a). The delay increases by only 44% despite a 20 fold increase in neighbors.

To further stress the protocol, in Fig. 11(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.

f) Congestion Rate: This experiment highlights the benefit of message scheduling in the case of handling transient failures. Load-balancing the publication stream to replicas caused significant message displacement. 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 are a cost-effective way to manage congestion.

VI. Related Work

This paper is put in context of existing techniques to pursue high-availability.

Optimal overlay design: Related work has focused on designing an optimal overlay to achieve high availability in terms of minimizing cost, latencies, path lengths, or node degrees, and maximizing network or processing capacities, given some client workload [2], [9], [15], [20]. These approaches, however, do not precisely consider the implications of modifying an overlay. Without considering the disruptiveness of the overlay transformation primitives as we have done in this paper, the construction of an optimal overlay in a real system is impractical.

Overlay reconfiguration: A more practical approach is to support dynamic reconfigurations in the overlay routing paths in order to bypass faulty brokers [10], [16], [24], including systems built on top of peer-to-peer networks [1], [7], [22]. However, these systems are engineered with proprietary routing and adaptation protocols and, unlike this paper, do not attempt to extract fundamental building block operations that can be used for arbitrary overlay transformation techniques in a wide variety of pub/sub overlays.

Overlay replication: There is some existing work on supporting redundancy in an overlay by replicating the broker state. 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 [4]. 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 [32]. 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.

Other approaches include dynamically load-balancing among a statically over-provisioned cluster of replica brokers [8] or reserving resources and constructing redundant paths with brokers with sufficient capacities to satisfy clients' quality-of-service requirements [5]. These replicas must be kept synchronized as is typical in distributed database replication [14], [29] or system-level replication [11], [23], [27], and there is no dynamic mechanism to add replicas. By contrast, our on-demand replication protocols are lightweight, and impose virtually no overhead to the existing system when replication is not taking place.

VII. Conclusion

In high availability applications, any modifications to a content-based pub/sub overlay should limit disruptions to the system, defined in this paper in terms of formalized consistency properties on the in-order delivery of all messages, and non-functional performance metrics.

To support such overlay modifications, five primitive operations were proposed that allow a content-based pub/sub broker overlay to be reconfigured for a variety of scenarios. Proofs affirm that the *planned* primitives (APPEND, DETACH, and SHIFT) are sufficient to carry out arbitrary transformations of an acyclic broker overlay, and the implementations of these primitives uphold the consistency guarantees, and minimize disruption by operating on at most three brokers in the overlay per atomic operation. Furthermore, two additional *on-demand* primitive operations (REPLICATE and CONSOLIDATE) allow a topology to be quickly adjusted around a misbehaving broker, whether due to a permanent failure, processing loads, or input or output queuing loads. Ten protocols implementing these primitives offer a range of trade-offs between the degree of consistency and performance required. For example, a publication re-delivery method based on a novel wait-ready queue and caching mechanism allows publications to be delivered more quickly while sacrificing in-order delivery guarantees.

Both sets of primitives require no overhead when the topology is not being modified, and their protocols are confined to a fixed set of brokers in the overlay making their effects on the overall topology more predictable. Evaluations of the fully implemented protocols running on a real system with various content distributions support their suitability to large-scale and time-sensitive distributed pub/sub applications.

References

[1] I. Aekaterinidis and P. Triantafillou. Pastrystrings: A comprehensive content-based publish/subscribe dht network. In *ICDCS*, 2006.

[2] R. Baldoni et al. Subscription-driven self-organization in content-based publish/subscribe. *ICAC*, 2004.

[3] T. Banka, A. A. Bare, and A. P. Jayasumana. Metrics for degree of reordering in packet sequences. In *LCN*, pages 333–342, 2002.

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

[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] A. K. Y. Cheung and H.-A. Jacobsen. Load balancing content-based publish/subscribe systems. *ACM Trans. Comput. Syst.*, 28:9:1–9:55, December 2010.

[9] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Constructing scalable overlays for pub-sub with many topics.

[10] G. Cugola et al. Minimizing the reconfiguration overhead in content-based publish-subscribe. In *SAC*, 2004.

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

[12] T. Fawcett et al. Activity monitoring: Noticing interesting changes in behavior. In *SIGKDD*, 1999.

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

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

[15] M. A. Jaeger, H. Parzyjegla, G. Mühl, and K. Herrmann. Self-organizing broker topologies for publish/subscribe systems. In *SAC*, 2007.

[16] Z. Jerzak, R. Fach, and C. Fetzer. Fail-aware publish/subscribe. In *NCA*, pages 113–125, 2007.

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

[18] I. Koenig. Event processing as a core capability of your content distribution fabric. In *Gartner Event Processing Summit*, 2007.

[19] G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Trans. Web*, 4(1):1–33, 2010.

[20] M. Migliavacca and G. Cugola. Adapting publish-subscribe routing to traffic demands. In *DEBS*, pages 91–96, 2007.

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

[22] V. Muthusamy. Infrastructureless data dissemination: A distributed hash table based publish/subscribe system. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 2010.

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

[24] G. P. Picco et al. Efficient content-based event dispatching in the presence of topological reconfiguration. In *ICDCS*, 2003.

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

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

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

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

[29] M. Wiesmann et al. Understanding replication in databases and distributed systems. *ICDCS*, 2000.

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

[31] Y. Yoon et al. On foundations for highly available content-based publish/subscribe overlays. Technical report, Univ. of Toronto, 2010.

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