Transactional Mobility in Distributed Content-Based Publish/Subscribe Systems

Songlin Hu⋆, Vinod Muthusamy†, Guoli Li†, and Hans-Arno Jacobsen†
⋆Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
†University of Toronto, Toronto, Canada

Abstract

This paper formalizes transactional properties for publish/subscribe client mobility and develops protocols to realize them. Evaluations show that compared to traditional protocols, those developed in this paper, in addition to supporting transactional properties, are more stable with respect to message and processing overheads. Changes in factors such as the number of moving clients have little impact, making the protocols more scalable and simpler to administer due to predictable resource requirements.

1. Introduction

Many adaptive distributed applications require the stateful reprovisioning of software components. For example, applications on virtual grid infrastructures are redeployed based on application requirements and grid conditions [9], massive multiplayer games migrate game state among servers in response to changing workloads [2], distributed process execution systems dynamically schedule agents at various engines [18], adaptive stream processing engines reconfigure dataflow operators to optimize query execution [13], [20], load balancing algorithms move software modules between nodes [5], mobile agent frameworks migrate program code and state across nodes [16], [10], and in mobile applications such as location-based services, the nodes themselves are mobile and connect to nearby access points as they roam.

A distributed publish/subscribe system [4], [6], [14] provides a powerful communication infrastructure for components of a distributed application. The pub/sub model decouples application components (referred to as clients in the pub/sub model) with a simple yet powerful interface that supports complex interaction patterns. Enterprise applications may be spread across dozens of geographically distributed sites with thousands or millions of users or application components, a scale which distributed pub/sub networks are designed to support. For example, online game servers may be deployed at a handful of sites across the world with high concentrations of players, or an enterprise executing large business processes may maintain a data centre at each of its international branches.

Distributed reprovisioning of pub/sub applications requires the ability to move pub/sub clients in a well-behaved manner. For example, moving clients should neither miss messages nor receive duplicates, and should be able to send messages without interruption. To achieve this transparency, a protocol must provide guarantees at the pub/sub layer during routing table reconfiguration. However, there is little literature that develops well-defined transactional movement properties in a distributed content-based pub/sub network. Though some work on pub/sub mobility has been put forward [6], [17], they have used end-to-end protocols that relied on optimizations, such as covering, to achieve efficiency. We show in this paper that even with these optimizations, such protocols can be too costly in an environment characterized by frequent movement. We analyze the problems and shortcomings of traditional movement protocols, and propose an alternative protocol.

This paper makes three key contributions. (i) In Sec. 3 transaction properties are formalized for mobile clients in a distributed content-based pub/sub system. The properties are defined analogously to database ACID properties, and are separated into layers. Modularization of the properties allows protocols for the different layers to be developed independently and utilized in various combinations. (ii) In Sec. 4 efficient protocols to support the above transaction properties are developed. As well, a failure model is outlined and correctness proofs for the protocols are given under the stated failure conditions. (iii) Finally, in Sec. 5 the mobility protocols are implemented in a real system and their performance analyzed in detail with evaluations on a local testbed and a wide-area PlanetLab deployment.

2. Background and related work

Pub/Sub research has focused on developing efficient routing protocols [1], [4], fast matching algorithms [7], and features such as failure handling [12], load balancing [5], and client mobility [6], [17]. However, none address the problem of supporting transactional client movement guarantees in the pub/sub system. The work presented in this paper is therefore orthogonal to these approaches and presents an important and fundamental addition to the body of pub/sub knowledge. This paper also establishes the important

and surprising observation that the popular pub/sub covering optimizations may actually negatively affect performance.

Transaction models for pub/sub that define a transactional context for the *processing* of publications at subscribers have been proposed to simplify the development of event-based applications [15], [22]. The approaches are based on a central transaction coordinator, much like traditional transaction processing systems, and little experimental evidence to support the scalability of the approaches are presented. Our work addresses transactions in the context of mobile pub/sub clients aiming to preserve certain properties while clients move, which none of the prior approaches address.

Protocols to repair failures in pub/sub networks do not address routing table reconfigurations due to voluntary client movement [12], [19]. In the latter case broker state changes result from announced movement, not unexpected failures, and more efficient algorithms can be developed.

Mobility in pub/sub has mostly dealt with subscriber mobility, where the broker network stores and replays publications missed by a moving subscriber [6], [3], [17]. In prior work we define and evaluate end-to-end subscriber and publisher mobility protocols that rely on advertisement covering for efficiency [17]. A moving publisher issues an advertisement (unadvertisement) at the new (old) access point. By contrast, as evaluated in Sec. 5, the work presented in this paper performs reconfigurations in a more efficient hop-by-hop manner that is also less susceptible to interactions among clients than an end-to-end approach.

The work in this paper is implemented in the PADRES[1] distributed content-based pub/sub system which uses a language model where subscriptions are a conjunction of $(attribute, operator, value)$ predicates, and publications are a set of $(attribute, value)$ pairs [8], [14].

Each PADRES broker manages a Publication Routing Table and Subscription Routing Table. An advertisement from a publisher is inserted as an $\{adv, lasthop\}$ pair in the SRT and then forwarded to all neighbours. A subscription that intersects an advertisement is forwarded to the advertisement's last hop and inserted into the PRT as a $\{sub, lasthop\}$ pair. A publication matching a subscription in the PRT is forwarded to the last hop of the subscription hop-by-hop until it reaches the subscriber.

The *covering* optimization can reduce message propagation. If subscription s_1 covers subscription s_2 then the publications that match s_1 are a superset of those that match s_2, and so s_2 need not be forwarded. A similar optimization can be applied to covering advertisements.

3. Transaction properties

It is desirable for the movement of clients in a pub/sub system to be transparent to both the moving client and those it interacts with, such that an application consisting

of stationary clients behaves the same as one where the clients move. Among other things, this means that clients should not miss any notifications while moving, and their movement should not be visible to others.

This section defines strong properties for client mobility in a pub/sub system similar to the relational database *ACID* [11] properties of *A*tomicity, *C*onsistency, *I*solation, and *D*urability. We assume in-memory routing algorithms and hence will disregard the durability property. (Durability can be achieved by persisting state to stable storage.)

3.1. Movement operation

Consider Fig. 1 where Client A moves from broker B_1 to broker B_7 by issuing a *MOVE* command. The knowledge of where to move is application specific. For example, a virtual machine instance may wish to move to a less congested part of the network, or a stream processing operator may relocate to a machine with more memory.

The end result of a successful movement operation is that the client must sever its connection to broker B_1 and establish one with broker B_7 without
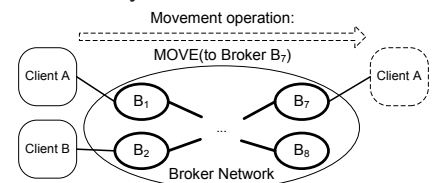

Fig. 1. Client movement

losing any messages in the process. The movement may fail for any number of reasons including the target broker rejecting the moving client (perhaps because the broker is overloaded, or the client is not authorized to make the connection), in which case the client should remain connected to broker B_1, again with no loss of messages. The guaranteed transactional properties required of the operation are presented in the remainder of this section.

While movement can be achieved purely by managing a client's connections, this paper considers the more general case where a client's execution location also moves. For example, a component managing a portion of a multiplayer game world may decide to migrate to a more optimal location in the network. While transferring the client computation and state, before the movement transaction has completed, there may be a copy of the client at both the source and target brokers (B_1 and B_7, respectively, in Fig. 1). This does not mean, however, that there are multiple functional instances of the client; the properties defined below ensure that only one copy of the client is "active" or visible to other clients in the system.

3.2. Pub/Sub system layers

A pub/sub system can be segmented into layers of well-defined functionality as depicted in Fig. 2. At a coarse grain, a client and broker interact with messages, including advertisements, subscriptions and publications from client to broker, and notifications from broker to client.

A client consists of application and pub/sub stub layers. The *application layer* encapsulates the application logic, be it an online game or a workflow management system. At this layer, transaction properties are domain specific but may rely on the properties of the lower layers. The *pub/sub stub layer* interfaces with a pub/

Fig. 2. P/S layers

sub broker. To support mobility, this layer must manage the phases of a moving client including queuing commands from the application, and retrieving notifications missed while moving. Since this paper does not consider the application layer, the pub/sub stub layer in the client is referred to interchangeably with the client itself.
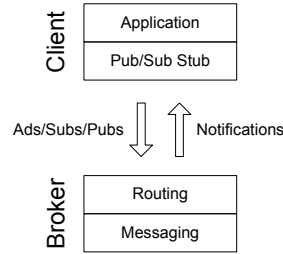
A pub/sub broker is divided into routing and messaging layers. The *routing layer* is concerned with how (un)advertisements, (un)subscriptions, and client movements influence the pub/sub routing tables, especially with routing state distributed across brokers. The *messaging layer* provides point-to-point communication between brokers. Messaging transactions, addressing concerns such as message ordering and synchronous versus asynchronous communication, are readily found in messaging products, such as MQSeries and JMS; they are not considered in this paper.

In Fig. 2 messages from the client affect the broker routing tables, but notifications from the broker are passed to the application layer without modifying the pub/sub stub state. However, since notifications are exposed to the application, it is necessary to define guaranteed properties for the notifications delivered to a client. The remainder of this section defines properties on the mobile *client state* (Sec. 3.3), the *notifications* delivered by the routing layer (Sec. 3.4), and the distributed *routing table state* (Sec. 3.5). Algorithms that satisfy the properties in each layer are presented, along with proofs, in Sec. 4.2, 4.3, and 4.4, respectively.

3.3. Client layer

We first focus on the correctness of a client movement protocol by way of properties on the state of clients. Each client must be in a unique state at all times. In standard pub/sub systems, a client may be in a *connected* or *disconnected* state. As we shall see in Sec. 4.2, in a system that supports mobility, more states are required.

Atomicity: We define two atomicity properties:

(a) After the transaction completes, a moving client must be either at its source or target broker, but not both.

(b) For a transaction consisting of a sequence of operations $\{o_1, ..., o_n\}$ that transition a client from state s_0 to $\{s_1, ..., s_n\}$, the final state of the client must be s_0 or s_n. That is, either all operations are completed, or none are.

Consistency: There must be at most one running instance of each client. This states that a movement should not result

in two instances of a client, with one instance at the source broker and another at the target.

Isolation: Suppose it is possible for an operation by a client to observe the state of another client. Given a transaction $T_x = \{o_1, ..., o_n\}$ that causes a client to change its state from s_0 to $\{s_1, ..., s_n\}$, an operation in another transaction T_y should only observe s_0 or s_n.

The properties above make a movement operation transparent in terms of the client's state. For example, if isolation is violated, then one of the intermediate states of a moving client may be observable, and the movement is no longer transparent. This point will become clearer in Sec. 4.2 where the state transitions of a moving client are described.

3.4. Notifications

We now define the properties required of the notifications delivered by the routing layer to the pub/sub clients. Let $P_i(t_a \to t_b)$ denote publications issued by client i in the time interval bounded by t_a and t_b, and $N_i(\cdot)$ refer to notifications received by client i. If necessary, notifications received by the copy of the client at the source or target broker are distinguished as $N_i^S(\cdot)$ or $N_i^T(\cdot)$, respectively.

Atomicity: Notifications—the delivery of publications to interested subscribers—are atomic. In the case of interested stationary clients, they are delivered exactly once; or in the case of interested moving clients, delivered exactly once to either the source or target client copy, but not both.

Consistency: Consider a client that initiates a movement operation at time t_0 in Fig. 3. The notifications $N^T(t_0 \to t_\infty)$ received by the client if the movement succeeds should be the same as those $N^S(t_0 \to t_\infty)$ it receives if the operation fails and it remains at the source. (Time t_∞ refers to the time when the client permanently disconnects from the pub/sub network, and hence $N^S(t_0 \to t_\infty)$ contains the set of notifications the client will ever receive starting from time t_0.) Notice that the location of a client may affect the order of notifications, which is why this property only requires that notifications are *eventually* delivered to the client.

Isolation: Consider a client C_i that initiates a movement operation at time t_0, and assume that the publications $P_i(t_0 \to t_\infty)$ issued by C_i if the movement is successful, are the same as the publications $P_i'(t_0 \to t_\infty)$ it issues if the movement fails. The notification received by every other client $C_j \neq C_i$ must be the same whether the movement completed or not: $N_j(t_0 \to t_\infty) = N_j'(t_0 \to t_\infty)$.

The notification properties above are designed to make a client's movement transparent, in terms of the notifications delivered both to itself and other clients in the network. For example, if atomicity is violated, a message may be processed twice by a client (once each by the copies of the client at the source and target brokers). The effects of this double processing of a message can be observed, making the movement visible, and thus may violate isolation.

3.5. Routing layer

Finally, we define the properties of the routing table state in a distributed content-based routing protocol.

Atomicity: For an operation o (such as advertise, subscribe, publish, or move) by a client, a pub/sub protocol defines routing table updates $U(o)$ that should occur. To be atomic, either all updates in $U(o)$ occur, or none.

Consistency: For every advertisement A that matches a subscription S issued by a client C, it must be that:

(i) At every broker B_i in the path from $publisher(A)$ to $C : S.lasthop \neq A.lasthop \land S.nexthop = A.lasthop$.

(ii) At every broker $B_i : B_i$ and $A.lasthop$ are neighbours in the path from B_i to $publisher(A)$.

These properties define the minimal set of routing table entries required to deliver notifications to all interested subscribers. Note that a routing table may have additional (perhaps stale) entries and still be considered consistent.

Isolation: Consider the advertisements A_i and subscriptions S_i issued by a moving client. Let RT_B be the routing table entries at broker B, and $RT_B(A_i)$ and $RT_B(S_i)$ be the subset of RT_B corresponding to A_i and S_i, respectively. To satisfy isolation, for every broker $B : [RT_B - RT_B(A_i, S_i)]_{beforemove} = [RT_B - RT_B(A_i, S_i)]_{aftermove}$. Informally, a movement should only update routing entries of the advertisements and subscriptions issued by the moving client; other clients' routing state should be unaffected.

The above properties are satisfied by well-known pub/sub routing protocols under non-failure conditions, and it is not difficult to adapt these protocols to tolerate failures in the case where faults are not permanent: brokers that crash are eventually restarted or replaced with a working broker, and link failures never permanently partition any set of brokers.

Under such failures, a pub/sub protocol can be made fault-tolerant by persisting the algorithmic and queue state of each broker, to recover from node and link failures, respectively. The algorithmic state—data managed by pub/sub protocols such as advertisements and subscriptions—is typically kept in memory, but can be persisted. The queue state includes unprocessed incoming messages at a broker and undelivered outgoing messages. The reliable delivery of these messages between brokers can be achieved using persistent queues. One implication of recovering broker failures locally without coordination among brokers, is that the recovery of a crashed broker may take arbitrarily long (perhaps requiring a manual restart), and so message delays may be unbounded, although eventual delivery is guaranteed.

The fault tolerance scheme above is straightforward and uses well-known technologies, and serves to demonstrate it is possible to implement a distributed pub/sub routing protocol that satisfies the atomicity, consistency, and isolation properties described above. Research on more sophisticated fault-tolerant algorithms is ongoing [12], [19] and the above properties can serve as a guide for such research. They are also used in this paper as a basis for the proofs of subsequent properties. In this way it is clear what properties are required of a robust pub/sub routing protocol in order to support the higher-level properties we address more fully below.

4. Client Movement Protocol

This section presents client movement protocols and proves they satisfy the transactional properties in Sec. 3. Unlike typical pub/sub mobility [6], where the client disconnects from one broker and reconnects to another, these protocols provide strong transactional movement guarantees and are more efficient, as we show in Sec. 5.

4.1. System model

We assume an acyclic overlay of pub/sub brokers that satisfy the properties in Sec. 3.5. Notably, node crashes or network faults in the pub/sub layer are masked by the routing protocols. A method to adapt existing pub/sub routing algorithms to be fault-tolerant was sketched in Sec. 3.5.

A *mobile container* associated with each broker encapsulates a *coordinator* (to execute the movement protocol) and the clients themselves. In this way, the middleware has full control over the deployment of clients. Also, we assume that the components within a container do no individually fail, so a crash failure of a coordinator implies a failure of the associated clients and vice versa.

Our movement algorithms are based on the three-phase commit (3PC) distributed transaction protocol [21][2]. As such, we can inherit the same failure model that this protocol assumes. In particular, crash failures of mobile clients and coordinators are allowed, and two network failure models are supported: (i) the network delivers messages within a bounded delay, in which case the non-blocking 3PC is used and movement transactions are guaranteed to complete within a bounded time; or (ii) message delays are unbounded in which case we use a blocking variant of the protocol and the movement algorithm may block.

4.2. Client layer

We present a movement protocol that satisfies the properties in Sec. 3.3. It consists of a conversation between the source broker a client is moving from and the target broker it is moving to as outlined in Fig. 3. While the protocol is handled by the source and target brokers, the brokers make use of a reconfiguration message (message (2) in Fig. 3) that is processed by all brokers along the path. The handling of this message is detailed in Sec. 4.4. The movement protocol modifies the state of the client and that of the source and target brokers as summarized in Fig. 4. The protocol proceeds as follows when a client moves from source broker B_i to target broker B_j:

---

[2]Unlike 2PC, 3PC supports non-blocking transactions, and nicely conforms to the message exchanges in our movement protocol in Sec. 4.2.

First, B_i sends message (1) to B_j with data about the moving client such as its ID, and its subscriptions and advertisements.

Fig. 3. Protocol overview

If B_j decides to accept the client, B_j initializes a transaction state for the new client, and then issues message (2) containing the client id and its subscriptions and advertisements. Message (2) executes routing table reconfiguration as described in Sec. 4.4. If B_j does not accept the client it sends a reject message (3) to B_i.

If B_i gets message (2), it stops the client, and sends message (4) to B_j, along with any queued publications for the client. Otherwise B_i receives message (3) and resumes the client.

B_j receives message (4), and dispatches it to the new client, which merges the notifications in the payload of this message with those in the queue at the target node. B_j also sends message (5) to B_i, upon receipt of which B_i finally cleans up any state associated with the client.

The client and coordinator at the source and target sites, all of which participate in the protocol, are modelled in Fig. 4. State transitions are labelled by the input transition trigger message and the generated output message. Messages between the application, mobile client, and coordinator are marked as indicated in the legend in Fig. 4.

The *global state* of a distributed protocol is defined by a vector of local states and outstanding messages in the network, and the global state transitions when a local state transition occurs. Transitions between global states create a reachable global state graph. For example, a possible global state is one where the source and target coordinators are in the `init` state with the `move` message (sent by the client to the coordinator) outstanding in the network. When this message is received, the global state transitions to one where the source coordinator moves to the `wait` state, the target coordinator remains in the `init` state, and the `negotiate` message (sent from the source to target coordinator) is in the network. Fig. 5 is the global reachable state graph for the local coordinator state graphs in Fig. 4. Note that the initials of the local coordinator state names are used to label the global states in Fig. 5.

The table embedded in Fig. 4 lists the possible concurrent client states for each coordinator state. For example, when both coordinators commit the transaction, the source client must be in the `clean` state, and the target client in the `started` state. From the table in Fig. 4, we see that for any global state in Fig. 5, two properties hold: (1) in a final global state, exactly one client is `started` and the other is `clean`; and (2) in any intermediate global state, at most one client is `started`.

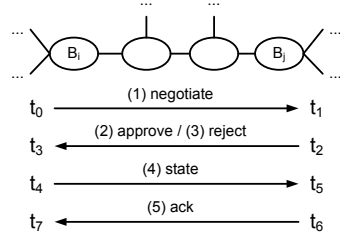Using these properties, we can prove the client state

properties in Sec. 3.3.

Atomicity Proof: (a) It follows from property (1) that a client only exists in the source or target broker after a movement. (b) Since we assume the client and coordinator experience the same faults (because they run on the same node), and the coordinator is guaranteed to abort or commit (barring an unrecoverable crash failure), the client will, according to Fig. 5, end up in its initial state (if the coordinator aborts), or the final state (if it commits). □

Consistency Proof: It follows from property (2) that there is at most one running instance of a client. □

Isolation Proof: A client's state can only be inferred by notifications received from it, in which case the client is in the running state. Since a client is never running until the movement has committed or aborted, it is not possible to observe an intermediate state of a moving client. □

4.3. Notifications

We now prove the notification properties from Sec. 3.4, assuming the use of a pub/sub layer that satisfies the routing state properties in Sec. 3.5.

Atomicity Proof: Correct routing state properties (namely consistency and atomicity) will ensure that the routing tables are configured so as to deliver notifications to stationary clients. For mobile clients, in any committed global final state in Fig. 5, the reconfiguration message would have been sent resulting in the appropriate updates to the routing state. Hence, a correct routing layer will deliver all messages to either the source or target client. □

Consistency Proof: Based on the protocol in Fig. 3, we wish to show that the set of notifications $N^S(t_0 \to t_\infty)$ received had the client not moved is equivalent to the notifications $N^T(t_0 \to t_\infty)$ if it does move. We first note that $N^T(t_0 \to t_\infty) = N^T(t_2 \to t_\infty) \cup N^S(t_0 \to t_3)$ since the target receives the latter messages from the source as part of the protocol. And, by definition, $N^S(t_0 \to t_\infty) = N^S(t_0 \to t_3) \cup N^S(t_3 \to t_\infty)$.

It suffices to show that $N^T(t_2 \to t_\infty)$ and $N^S(t_3 \to t_\infty)$ are equivalent. Note that only at t_3 have routing table entries for the target client been properly updated. Assuming routing states are setup correctly by the routing table layer, every publication $m \in N^T(t_2 \to t_\infty)$ also belongs to $N^S(t_3 \to t_\infty)$. Similarly, by virtue of an acyclic topology in which messages are processed in order, every publication $m \in N^S(t_3 \to t_\infty)$ also belongs to $N^T(t_2 \to t_\infty)$. Hence, $N^T(t_2 \to t_\infty)$ and $N^S(t_3 \to t_\infty)$ are equivalent. □

Isolation Proof: We assume a client C_i issues the same publications regardless of its location: $P_i(t_0 \to t_\infty) = P'_i(t_0 \to t_\infty)$. It remains to show each publication is issued exactly once. A client may only publish while in a running state. We have seen that a client is never in a running state at both the source and target, and that after the transaction completes, it will be in the started state at
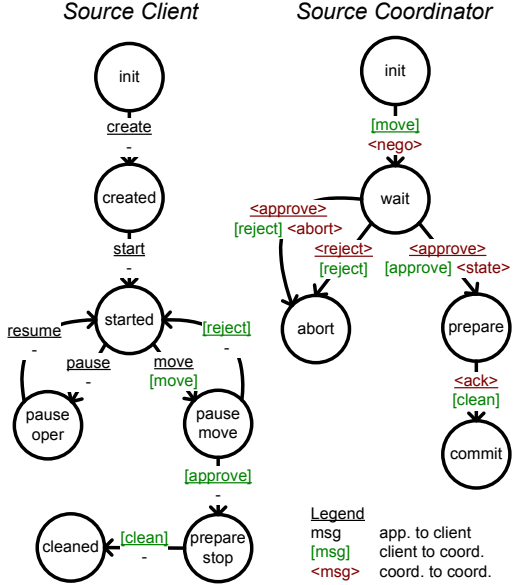
Fig. 4. Client and coordinator states at source and target sites
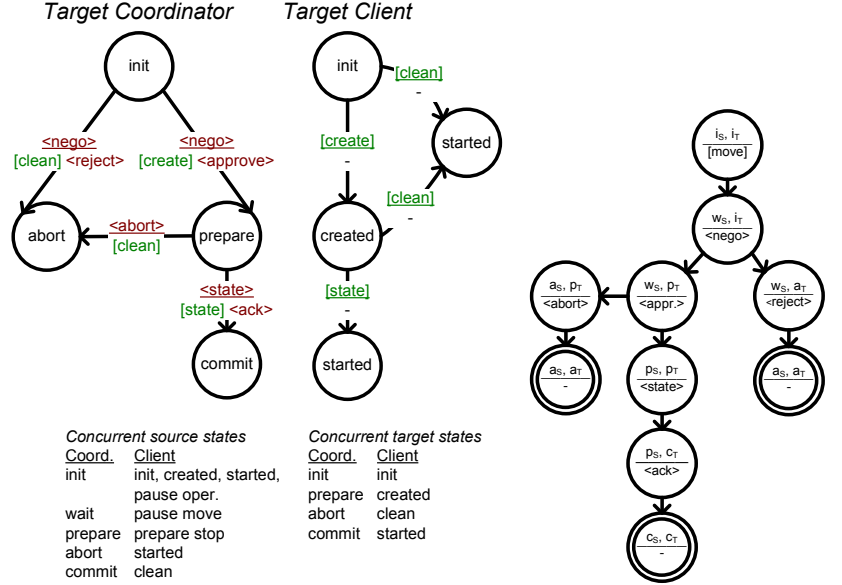

Fig. 5. Global states

the source or target, but not both. Therefore, a client cannot issue a publication from both the source and target, and since publications are buffered while in a non-started state, publications are not dropped, and hence each publication is issued exactly once. ☐

4.4. Routing layer

We now outline the routing layer protocol to achieve client movement according to the transactional properties defined in Sec. 3.5. The objective of the routing reconfiguration algorithm is to efficiently maintain valid routing configuration states during client movement.

In traditional client movement [6], [17], a client disconnects from its source broker after unadvertising and unsubscribing its history, and these messages propagate through the network. Then, the client connects to the target broker and reissues its advertisements and subscriptions, which again propagate. This is expensive, especially since (un)advertisements are flooded. The protocol can be improved by enabling advertisement and subscription covering, where (un)advertisements and (un)subscriptions are quenched by covering advertisements and subscriptions.

Contrary to traditional assumptions, with frequent mobility, enabling covering may actually degrade performance. Consider publisher pub_1 (pub_2) issuing advertisement adv_1 (adv_2) and let adv_2 cover adv_1. Suppose adv_1 is sent first, and flooded. Then, adv_2 is issued and also flooded. With advertisement covering, there are links where it is redundant to send both adv_1 and adv_2. For example, when a broker B' forwards adv_2 to broker B'', it will also issue an unadvertisement for the previously sent adv_1 since adv_2 covers adv_1. It turns out that adv_1 will have to be unadvertised over all links not on the path between pub_1

and pub_2. So, we have a situation where both adv_1 and adv_2 were flooded, *and* adv_1 was unadvertised throughout most of the network, which is more expensive than if covering is not enabled. Such situations are more likely when clients move frequently, and issue many advertisements and subscriptions.

We develop a routing reconfiguration protocol to achieve the best-case efficiencies of the traditional covering-based mobility algorithm, and not suffer from its pathological deficiencies. The algorithm reconfigures the routing table hop-by-hop along the path between source and target brokers.

Since the overlay is acyclic, there is only one route from a source broker B_i to a target broker B_j (assume $i < j$). This route, $RouteS2T$, is a sequence of brokers, $< B_i, B_{i+1}, \ldots, B_j >$, such that (B_m, B_{m+1}) is an edge in the network, where $i \leq m < j$. The predecessor and successor of B in $RouteS2T$ are denoted as $RouteS2T.pre(B)$ and $RouteS2T.suc(B)$, respectively.

Suppose a publisher at broker B_i moves to broker B_j ($i < j$). After movement, advertisement adv at the old publisher becomes adv' at the new publisher.

Claim 1: The routing configurations of adv and adv' are identical at all brokers except for brokers $B \in RouteS2T$.

Proof: Consider a broker $B \notin RouteS2T$. B must have zero or one neighbours $B_n \in RouteS2T$ (since the network is acyclic). In the former case, B must have a neighbour B'_n that is on the path to both B_i and B_j (since the topology is acyclic). In the latter case, the only path to B from either B_i or B_j is through B_n. Hence, in either case the routing configuration at B consists of an entry for adv or adv' from either B_n or B'_n. ☐

Claim 2: The routing configurations of adv and adv' at all brokers $B_l \in RouteS2T$ are different.

Proof: Consider a broker $B \in RouteS2T$ and B is

neither B_i nor B_j. Since B_i and B_j are at opposite ends of $RouteS2T$, the neighbour from which B receives adv (sent by B_i) must be different from which it receives adv' (sent by B_j), and hence the routing configuration of B is different in the two cases. Likewise, the routing configuration at B_i and B_j will be different. $\square$

This means that only the routing state at brokers along the route from target to source broker need to be modified. It is possible to simulate un-advertisement and re-advertisement by modifying the routing configuration $rc(adv)$ to be $rc(adv')$ hop-by-hop from B_i to B_j (source to target) or from B_j to B_i (target to source). Since the information about an advertisement at a broker influences the routing of matched subscriptions, both the records in the SRT and the records in the PRT must be modified.

To move a publisher that has issued an advertisement adv, the routing configuration at every broker $B_l \in RouteS2T$ is modified as follows. The records $(adv, RouteS2T.pre(B_l))$ in the SRT are modified to $(adv', RouteS2T.suc(B_l))$. For the records $(sub, lasthop)$ in the PRT where sub intersects adv, there are three cases to consider: (1) $sub.lasthop = B_x \notin RouteS2T$; (2) $sub.lasthop = RouteS2T.suc(B_l)$; and (3) $sub.lasthop = RouteS2T.pre(B_l)$

For the first case, since $adv.lasthop = adv'.lasthop$ in broker B_x, any subscription sub intersecting adv also intersects adv', and will be forwarded to B_l. If sub has not already been forwarded to $RouteS2T.suc(B_l)$, it also needs to be forwarded to $adv'.lasthop$. Note that at B_l, $adv'.lasthop = RouteS2T.suc(B_l)$.

For the second case, $sub.lasthop = adv'.lasthop$, which means that $sub \notin rc(adv')$. Unless sub intersects an advertisement besides adv, it is removed from the PRT.

For the third case, $sub.lasthop = adv.lasthop$, which means that $sub \notin rc(adv)$, so it must also match some other advertisement adv_1 where $adv_1.lasthop = RouteS2T.suc(B_l)$ or $adv_1.lasthop \notin RouteS2T$. If sub has not already been forwarded to $RouteS2T.suc(B_l)$, it needs to be forwarded there.

To guarantee atomicity of the movement transaction, it is necessary to construct a copy of $rc(adv')$, which is the revised version of $rc(adv)$, at each broker along $RouteS2T$. Because of Claim 1, the algorithm only needs to revise the routing configurations along $RouteS2T$. If the transaction commits, the old routing configuration $rc(adv)$ is deleted hop-by-hop, otherwise, $rc(adv')$ is deleted hop-by-hop.

5. Evaluation

The main conclusion of the evaluations is that the reconfiguration protocols exhibit more *stable* performance than the traditional covering-based movement protocol. The covering protocol's performance varies greatly and is more susceptible to pathological scenarios.

The protocols in this paper are implemented in the Java-based PADRES content-based pub/sub prototype. Experi-

ments performed on a cluster of 1.86 GHz machines with 4 GB of RAM that mimics an enterprise data centre environment and offers a controlled system for meaningful analysis. Evaluations are also conducted on heterogeneous nodes in the wide-area PlanetLab testbed, representing a geographically distributed system administered by one or more enterprises, and also serves to stress the protocols under the unpredictable and shared PlanetLab network.

By default the 14 broker overlay in Fig. 6 is used with each broker running on a separate machine. The network scale is appropriate for the motivating applications from Sec. 1; we
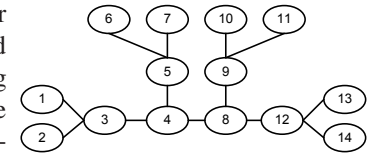
Fig. 6. Default topology

are not considering P2P applications with millions of nodes. A mobile coordinator is co-located with each broker allowing any broker to host mobile clients, and so we will not distinguish between brokers and mobile coordinators. Unless otherwise stated, clients connect to a random broker at startup, and then initiate their movement pattern, pausing for ten seconds at each broker between movements. Each subscriber is assigned a subscription randomly from the subscription workload.

Metrics include network traffic, movement duration and movement throughput. Network traffic, measured as the sum of messages transmitted over each overlay link, includes publications, (un)subscriptions, and (un)advertisements. Assuming roughly equal message sizes we approximate network traffic with message counts. Movement duration is the time to complete a client movement transaction, and movement throughput measures the number of movement transactions the system can process in a given time.

Each client issues a subscription chosen from the four subscription workloads in Fig. 7, where covering relationships are shown. For example, in Fig. 7(a), the root subscription covers all the others. Not shown is a Random workload in which subscriptions from all four workloads in Fig. 7 are selected uniformly. The details of individual subscriptions are omitted since it is primarily the covering relationships among them that affect the results.

Subscription Workload: We evaluate the sensitivity of the protocols to different subscription workloads. We first consider an experiment in which 400 clients, initially connected to Brokers 1 and 2, each repeatedly performs a movement between Brokers 1 and 13, and 2 and 14, waiting
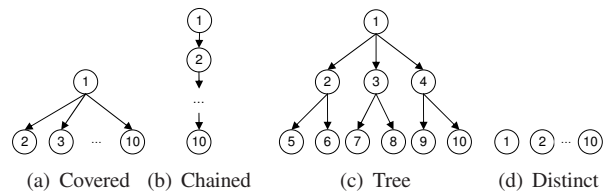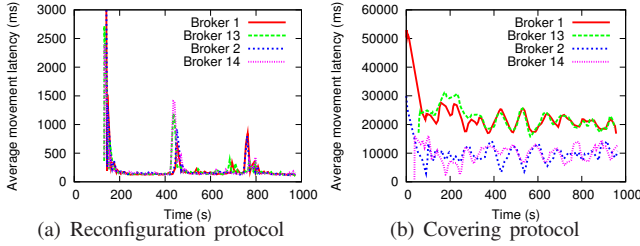
Fig. 7. Subscription covering relationships
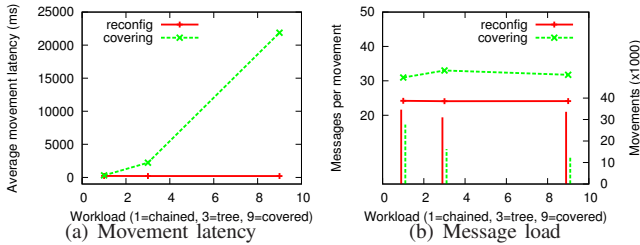
Fig. 8. Movement latency over time

Fig. 10. Number of clients

Fig. 9. Subscription workload

Fig. 11. Single client

for ten seconds at each broker before moving again.

The movement latency is shown in Fig. 8 for the reconfiguration and covering movement protocols. Each point in the plot represents the time to complete the protocol (vertical axis) for a movement that starts at a given time (horizontal axis). Notice that the reconfiguration protocol is more than an order of magnitude faster than the covering one. Also observe that movements at the beginning of the experiment take longer than those at the end due to the load imposed by joining clients. To avoid skewing steady state performance, we ignore this setup phase in subsequent results.

In Fig. 8(b), we see that clients moving between Brokers 1 and 13 are slower than those between Brokers 2 and 14. This is because odd valued subscriptions as numbered in Fig. 7 are initially assigned to Broker 1, and even ones to Broker 2. Since subscription 1 in the covered and tree workloads (Figs. 7(a) and 7(c)) cover more subscriptions than others, it is more likely to cause a pathological subscription propagation during movement as explained in Sec. 4.4. We confirm this with results that show there is almost no variance in movement latencies with the chained workload (where each subscription covers at most one other), and that the variance increases with the covered workload (where the root subscription covers the remaining nine).

Fig. 9(a) summarizes the latencies for different workloads, with the x-axis values corresponding to the number of covered subscriptions in the workload. The reconfiguration protocol exhibits little variation in latency, while the covering protocol performs worse—almost two orders of magnitude in the worst case—when more covering is present in the subscription workload.

Fig. 9(b) shows the number of messages normalized to the number of movements that occur during the experiment. Since clients keep moving during the experiment, a slow protocol will result in fewer movements, and so the per-
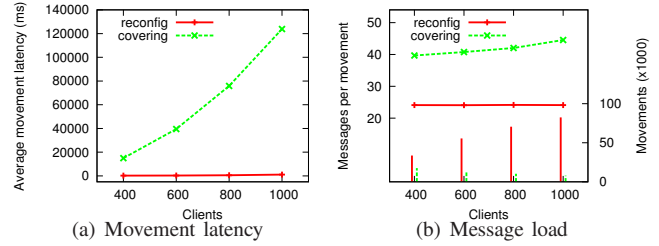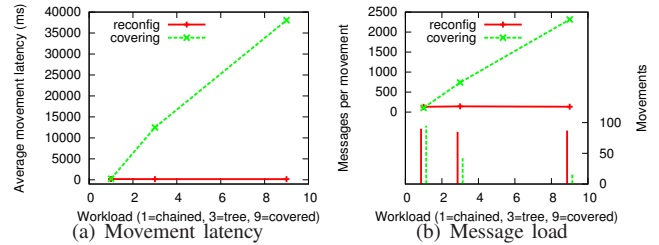
movement message counts are a more fair representation of a protocol's message overhead. The normalized message values are plotted as lines with values on the left vertical axis, and for completeness, the number of movements are plotted as impulses with values on the right vertical axis. First observe in Fig. 9(b) that the reconfiguration protocol maintains a stable message overhead regardless of workload, and is able to complete roughly the same number of client movements during the experimental duration. On the other hand, the covering protocol performs fewer movements with the tree and covered workloads, a direct consequence of each movement taking longer to complete as we saw in Fig. 9(a).

It seems odd that, compared to the tree workload, the covered workload imposes less per-movement message overhead as seen in Fig. 9(b) but results in a longer movement latency. This apparent discrepancy is due to an underlying bimodal behaviour of the covering algorithm with the covered workload. It is cheap for the covering protocol to move one of the non-root subscriptions in the covered workload (subscriptions 2 to 10 in Fig. 7(a)) because their propagation is quenched by the presence of the covering root subscription (subscription 1 in Fig. 7(a)). Moving the root subscription, however, is expensive, since it triggers the propagation of all the non-root subscriptions. The movement of the root subscriptions occurs seldom relative to the non-root ones which is why the message overhead of the covered workload is less than that of the tree one which contains fewer non-leaf subscriptions. However, when the root subscription does move, it causes a burst of messages that causes significant congestion and has a large impact on movement latencies.

We emphasize that unlike the covering protocol, the reconfiguration protocol's message load and latency results are *stable* with respect to subscription workloads.

Number of Clients: We evaluate scalability by varying the number of moving clients from 400 to 1000. Fig. 10(a)
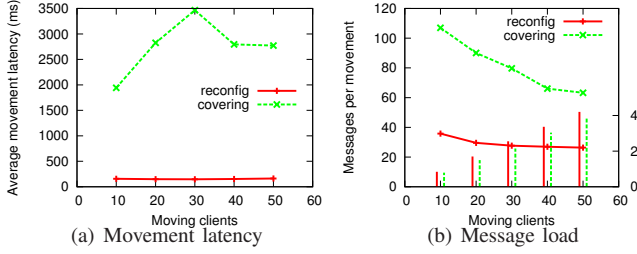
Fig. 12. Incremental movement

Fig. 13. Topology size

illustrates two important points: the reconfiguration algorithm performs much better than the covering algorithm, and the latter's performance degrades with more clients, while the former maintains stable performance. We see the same stable message overhead for the reconfiguration protocol in Fig. 10(b). There is an apparent paradox with the reconfiguration protocol achieving faster movement despite more total messages (which is the product of the normalized overhead and the number of movements). This occurs because it is able to isolate these messages within the path between source and target brokers, and because it does not suffer from bursty propagation of messages that congest the network. This also explains why a slight increase in the covering protocol's message overhead dramatically impacts the latency results in Fig. 10(a).

Single Client: This experiment isolates the effects of moving a single subscription in the covered workload with 400 clients. Only the root subscription (subscription 1 in Fig. 7(a)) is moved. Fig. 11 shows that the covering protocol has much worse movement latency and message load. Since only the root subscription moves here, we confirm the reason for this is precisely due to the pathological case for the covering protocol, where subscriptions (unsubscriptions) of the root subscription induce unsubscriptions (subscriptions) of the non-root subscriptions.

Incremental Movement: We evaluate the incremental effect of movement by keeping the number of clients constant at 400, and increasing the number of these that move. Fig. 12(a) again shows the superior and stable latencies of the reconfiguration protocol. The covering protocol exhibits an interesting behaviour. In the experiment each increment of ten moving subscriptions are successively chosen as follows: ten covering (i.e., root) subscriptions from the covered workload, ten covering from the tree workload, ten covering from the chained workload, ten covered (i.e., leaf) chosen randomly from the previous three workloads, and finally ten from the distinct workload. Notice that the first four sets of subscriptions have less and less covering, with the last two sets not covering any. For example, the first ten are chosen only from the covering workload (which has the most covering). And so, we expect the incremental effect of moving ten tree workload subscriptions to be greater than that of ten chained subscriptions. Indeed, we observe in
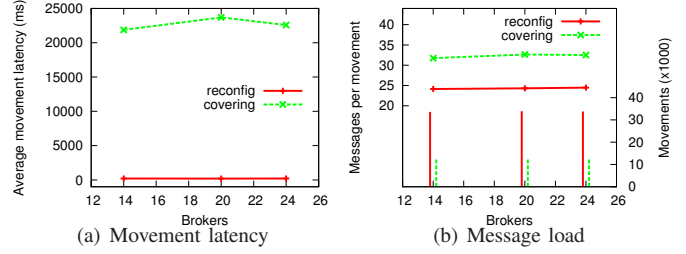
Fig. 12(a) that the slope of the covering protocol's latency between the first ten and twenty (tree workload) moving clients is slightly steeper than for the next ten (chained workload). The movement of subscriptions that do not cover any others, introduced when forty clients move, can be performed very quickly and contributes to a reduction in the average latency of the covering protocol. Likewise, the last ten subscriptions do not cover any, and they further reduce the average latency of the covering protocol. Fig. 12(b) also shows how the message overhead for the covering protocol decreases when moving subscriptions with less covering. The results in Fig. 12 nicely illustrate how the covering relationships of the subscription workload affect the performance of the covering protocol.

Topology Size: In this experiment we increase the number of brokers in the topology but keep the path length between source and target brokers constant by only moving clients between Brokers 1 and 12, and Brokers 2 and 14. The covered workload is used here to try to induce an exaggerated effect. Fig. 13 shows that increasing the topology size affects neither the latency nor message load drastically. This is expected since the reconfiguration protocol sends messages between the source and target brokers, and the covering protocol is primarily affected by congestion in the path between these two brokers. However, we wish to note that if there were clients moving in other parts of the network, they would be affected by the covering protocol but not the reconfiguration protocol.

Wide-area PlanetLab deployment: Evaluations on PlanetLab confirm the trends observed in the local testbed, but show longer movement latencies due to the more limited network and compute resources on PlanetLab. For example, similar to the experiment in Fig. 8, Figs. 14(a) and 14(b) now show the results of a 14 broker topology with 100 moving clients in the wide-area testbed. The reconfiguration protocol performs movements faster than the covering algorithm, but both take longer than in the local environment. Also the latencies vary more due to the unpredictable resource availability in the shared PlanetLab environment. As well, the trends in the local testbed in Fig. 9 occur in the wide-area deployment: Fig. 14(c) shows the covering protocol suffers with workloads with more covering, and Fig. 14(d) confirms the reconfiguration protocol imposes a smaller message over-
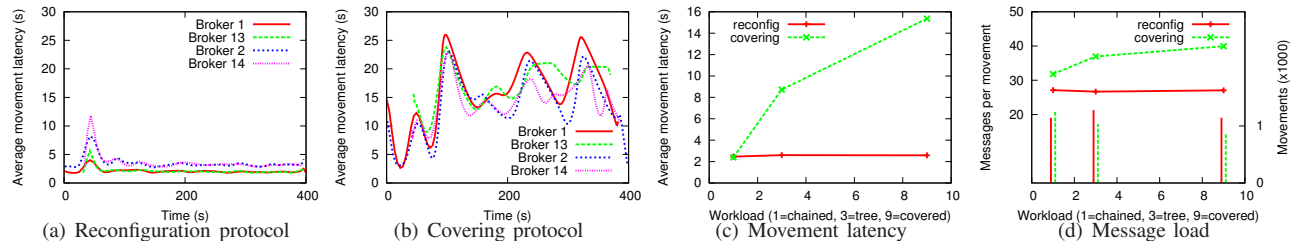
Fig. 14. Wide-area PlanetLab deployment

head than the covering protocol, and completes movement transactions at a faster rate. Other experiments on PlanetLab reinforce earlier conclusions such as the insensitivity of either protocol to the size of the network topology.

6. Conclusions

Distributed applications deployed on virtual grid infrastructures, stream processing engines, or distributed workflow execution systems require the dynamic reprovisioning of software components across nodes in the network. While the pub/sub model is well suited as a messaging layer for distributed applications, the movement of pub/sub clients between brokers is not guaranteed to be well-behaved.

This paper seeks to support the guaranteed movement of pub/sub clients according to well-defined properties. The transactional concerns of various layers are outlined, and atomicity, consistency, and isolation properties for three layers—client, notification, and routing—are specified. Furthermore, protocols to achieve these properties are described and correctness proofs are given. Unlike protocols where client mobility is handled by the end point brokers, a more efficient reconfiguration algorithm is developed in which brokers on the path from the source to target brokers participate in the client mobility protocol.

Evaluations in both a local data centre environment and a wide-area PlanetLab testbed indicate that the proposed reconfiguration mobility protocol outperforms the traditional protocol in terms of both the movement transaction time and network overhead, and confirm that the covering optimization is costly in a scenario with mobile clients. Moreover, the reconfiguration protocol exhibits much more stable behaviour: changes in the nature of the subscriptions, number of moving clients, and background pub/sub activity, such as unsubscriptions by non-mobile clients, hardly affect the performance of the reconfiguration protocol, whereas the traditional mobility protocol's performance varies greatly. This stability property not only illustrates the scalability of the protocol, but is also vital for administrators to plan the provisioning of a network's resources without having to be concerned with changing application workloads.

References

[1] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS*, 1999.

[2] P. B. Beskow, K.-H. Vik, P. Halvorsen, and C. Griwodz. Latency reduction by dynamic core selection and partial migration of game state. In *NetGames*, 2008.

[3] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE ToSE*, 2003.

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

[5] A. K. Y. Cheung and H.-A. Jacobsen. Dynamic load balancing in distributed content-based publish/subscribe. In *Middleware*, 2006.

[6] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE ToSE*, 2001.

[7] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of ACM SIGMOD*, 2001.

[8] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. The PADRES distributed publish/subscribe system. In *ICFI*, 2005.

[9] I. T. Foster. Globus toolkit: Software for service-oriented systems. In *NPC*, 2005.

[10] H. Gazit, I. Ben-Shaul, and O. Holder. Monitoring-based dynamic relocation of components in FarGo. In *ASA/MA 2000*, 2000.

[11] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, 1981.

[12] R. Kazemzadeh and H.-A. Jacobsen. Delta-fault-tolerant publish/subscribe systems. Technical report, CSRG-570, U. of Toronto, 2007.

[13] V. Kumar, Z. Cai, B. Cooper, G. Eisenhauer, K. Schwan, M. Mansour, B. Seshasayee, and P. Widener. Implementing diverse messaging models with self-managing properties using IFLOW. In *ICAC*, 2006.

[14] G. Li, V. Muthusamy, and H.-A. Jacobsen. Adaptive content-based routing in general overlay topologies. In *Middleware*, 2008.

[15] A. Michlmayr and P. Fenham. Integrating distributed object transactions with wide-area content-based publish/subscribe systems. In *DEBS*, 2005.

[16] A. L. Murphy and G. P. Picco. Reliable communication for highly mobile agents. In *ASAMA*, 1999.

[17] V. Muthusamy, M. Petrovic, and H.-A. Jacobsen. Effects of routing computations in content-based routing networks with mobile data sources. In *MOBICOM*, 2005.

[18] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *OOPSLA*, 2004.

[19] G. P. Picco, G. Cugola, and A. L. Murphy. Efficient content-based event dispatching in presence of toplogical reconfiguration. In *ICDCS*, 2003.

[20] V. Raman, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query processing. *ICDE*, 2003.

[21] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE ToSE*, 1983.

[22] L. Vargas, L. I. W. Pesonen, E. Gudes, and J. Bacon. Transactions in content-based publish/subscribe middleware. In *DEPSA*, June 2007.