A Hierarchical Proxy Architecture for Internet-scale Event Services

Haobo Yu, Deborah Estrin, Ramesh Govindan
Information Sciences Institute
University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292
{haoboy,estrin,govindan}@isi.edu

Abstract—The rapid growth of the Web has made it possible to build collaborative applications on an unprecedented scale. However, the request-reply interaction model of HTTP limits the range of applications that can be built. In this paper, we consider a complementary communication model—asynchronous event notification from servers to clients. Our focus in the paper is the design of an Internet-scale mechanism for event dissemination. Such a mechanism must scale to large numbers of participants and event types, as well as provide failure detection and handling. In this paper, we explore the design space of event dissemination architectures, and present a design of a hierarchical proxy architecture for event dissemination. Compared with previous approaches, our design reduces proxy states and provides failure detection and recovery mechanisms.

Keywords: event service, collaboration, web, proxy hierarchy

I. INTRODUCTION

The phenomenal growth of the Web during the past years has made it possible for people to collaborate on an unprecedented scale. However, designers of large-scale Web-based collaborative applications face one problem: HTTP [8] is a pure request-response model and does not allow servers to asynchronously notify clients of events at the server-side. Many distributed systems need such functionality, e.g., callbacks in distributed file systems [23], and gossip messages in lazy replication systems [15]. Some existing systems, such as mailing lists and ICQ [12], support large-scale notification using centralized proxies for relaying events from servers to clients. However, these notification mechanisms are specialized for their applications; it is desirable to design a general-purpose event dissemination infrastructure on which large-scale collaborative applications can be implemented.

To motivate the requirements of a general-purpose event notification architecture, let’s consider a hypothetical class of applications. Suppose every user (or program) in the world keeps a database of data items and a set of rules to modify the items. One example of a data item is an appointment in a calendar (“Meet Joe between 10 and 11 am on Tuesday”). An example of a rule associated with this appointment might be “if Joe cancels this meeting, try to set up a teleconference with Alice and Bob during the same slot”. This simple example brings out several interesting features. First, a change in one data item can lead to changes in other items. Thus, a canceled appointment in Joe’s calendar can affect Bob’s appointments. Second, a data item may be replicated at many users (e.g., an appointment for a group meeting) and modified by any one of them. Third, changes in data items can be effected by programmed interactions (rather than human-initiated interactions) defined by the associated rules. Finally, these programmed interactions may involve wide-area communication: Alice and Bob may be located in different parts of the world.

A global event notification service can be used to implement this class of applications. In our example, Joe’s cancellation of an appointment triggers an event sent to update our appointment, which in turn triggers other events. We call the origin of an event a source, and the destination receiver.¹

One key component of a global event notification service is a scalable event dissemination mechanism. Our example brings out some key requirements of such a mechanism. Every event may have many sources and receivers. Sources and receivers can both be mobile. Furthermore, both sources and receivers may be “thin”: our calendar appointments may be stored on hand-held devices, e.g., PDAs, with low-bandwidth wireless connections to a global network.

These requirements, and particularly the “thinness” of clients and servers, prompt us to consider a proxy-based architecture for event notification. In this architecture, each source delivers events to its nearest proxy. Re-

¹Here source and receiver are synonymous with publisher and subscriber [18], respectively.
receivers subscribe to events of interest with their neighboring proxies. The collection of proxies then conspire to deliver events from sources to receivers. The key challenge in designing this mechanism is that it must:

- Scale to large numbers of sources, receivers and event types.
- Provide a mechanism for detecting source, proxy and communication failure. Because events are pushed from sources to receivers, such a mechanism allows receivers to distinguish between the absence of events and any kind of failure.

In this paper, we first explore the design space of a proxy-based architecture and identify tradeoffs of different approaches (Section II). We then discuss related work in event notification (Section III). Next, we describe our design (Section IV), which uses a hierarchy of event forwarding proxies to deliver events from each source to each related receiver. We conclude the paper with a discussion of future work in Section V.

II. DESIGN SPACE

The key problem of event dissemination is rendezvous, i.e., where receivers “meet” sources and establish event delivery contracts. We identify and compare three classes of designs depending on the rendezvous location (Fig. 1): distributed, single-proxy and multi-proxy.

A. Distributed Architecture

In this architecture, a receiver sends subscriptions to all sources of events of interest. Later, whenever an event occurs, the source delivers the event directly to all subscribers. The entire procedure can be built on top of HTTP (enhanced with new methods, e.g., publish/notify [5]) so that it can be integrated into an existing browser.

This design raises two questions. First, every source has to process subscriptions and keep track of every subscriber. This scales poorly to a large number of subscribers. The problem becomes more serious when the source is mobile where both bandwidth and other resources are scarce [19]. It is arguable that multicast delivery will solve this problem. Using one multicast group per event type eliminates the subscriber list at the sender. However, how to scale multicast routing to large numbers of global groups, given a relatively small IP multicast address space, is still the subject of research [14], [20]. Second, because receivers do not explicitly poll for an event, they have to rely on a failure detection mechanism to know whether sources have failed or whether there are simply no events. The most often used failure detection mechanism is a heartbeat message [11]. In this situation, however, heartbeats between every source and each of its subscribers may impose heavy load on both the network and the receiver.

Furthermore, this architecture requires a directory service to enable receivers to locate all sources of an event. The Domain Name System [17] (DNS) does not provide a satisfactory solution for this. First, DNS supports only exact queries, but in an event service it may be desirable to perform inexact queries which contain wildcards or regular expressions [6]. Second, DNS names are not designed for general purpose structured data representation. Encoding event names using the DNS representation is likely to result in CGI-like cryptic names.

B. Single-proxy Architecture

In the single-proxy architecture, the proxy acts like a central directory, and maps all events to sources and receivers. Every receiver subscribes at the proxy, which records the subscription in its database. When a source generates an event, it sends the event to the proxy, which looks up the subscribers and forwards the event to them. With events going through the proxy, sources do not need to maintain their subscriber states.

Using clustering techniques [10], the proxy can be scalable enough to handle the amount of state and traffic, and

2It is possible to avoid a directory service using multicast to deliver every event to all receivers regardless of their subscription [18]. This approach performs well within a LAN environment, but clearly is unacceptable in the wide-area, because every event has to reach every receiver.
robust enough to provide uninterrupted service. However, some scalability and failure handling problems still exist. First, in order to provide failure detection between sources (and receivers) and the proxy, heartbeats should go from all sources to the proxy, and from the proxy to all receivers. This does not scale well to large numbers of sources or receivers. Moreover, when the sources are highly sporadic, i.e., it only occasionally sends an event, the overhead—measured by the ratio of heartbeat traffic to event traffic—will greatly increase and event delivery becomes less efficient. Second, even though the central proxy can be made extremely fault-tolerant, it is still susceptible to network partitions. In this case, a source and a receiver may not be able to communicate even if they can reach each other.

C. Multi-proxy Architecture

A natural solution to the above problems is to use multiple proxies. In this scheme, each receiver sends event subscriptions to its nearest proxy. Similarly, each source sends events to its nearest proxy. The proxies collectively implement an event dissemination mechanism that “routes”, at the application level, events towards the subscribers. Sources and receivers only send heartbeats to their local proxies, and proxies exchange heartbeats among themselves. This architecture is not susceptible to a single point of failure—a source may be able to reach some receivers even if some proxies fail or are disconnected from other proxies.

Unlike the distributed architecture, the multi-proxy architecture’s application-level routing scheme combines two functions: mapping event names to sources and receivers (the directory service) and delivering events from sources to receivers. This is a desirable feature. Sources need not explicitly maintain subscriber lists. Furthermore, failure detection can be done on a hop-by-hop basis and global heartbeats are avoided.

The design of the event dissemination and failure detection mechanisms depends on how the proxies are organized. One possible choice of proxy organization is the mesh—a connected graph. When proxies are organized thus, every proxy maintains a routing table in order to forward each event towards all interested receivers. Such a routing table can represent a scaling limitation. The Internet routes packets in a similar manner, but because IP addresses are approximately hierarchically assigned, routing table entries aggregate well [9]. Because events are named by attributes which may bear no relation to topological location, i.e., the same type of event may have topologically distant sources, the aggregatability of event forwarding tables can be poor. Organizing proxies into a mesh has two advantages, however. First, this organization is robust in that it can use an alternative route to deliver events when one route fails. Second, a mesh may achieve relatively small delivery latency if events are routed according to the shortest proxy-level path.

Another choice of organization is the hierarchy. In this scheme, each proxy has a single parent and may have one or more children. There is one designated root proxy. Unlike the mesh, only the root of the hierarchy is required to know where to route every existing event; other proxies can forward unrecognized events or subscriptions towards the root. In this sense, the hierarchy scales better than the mesh. A hierarchical organization may increase event delivery latency; we discuss this issue briefly in Section V. The hierarchy organization need not, however, sacrifice robustness; there exist several proposed solutions for protocols that automatically re-organize hierarchies in response to failures [22], [24].

III. RELATED WORK

There has been much previous work on Internet-scale event services. [5] proposes to implement event notification using HTTP. This work is applicable to proxy-based architectures, in that their proposed protocol can be used by receivers to register interest with proxies. However, this work does not address scalable event dissemination architectures. Carzaniga et al have proposed a proxy-based dissemination architecture for event service [4]. They discuss both a mesh and a hierarchy organizations, and have compared the two in terms of message traffic. For both organizations, however, their scheme installs routing state of every event in every proxy, which limits its scalability.

There has been much work in the area of event naming schemes [1], [3], [6], [13]. Our focus is on a complementary piece of the puzzle: how to achieve scalable dissemination given a fairly general event naming scheme. Because the choice of event naming schemes is still an area of active research, we have tried to make minimal assumptions about event naming in the design of our overall architecture. In this way, we hope to ensure that different event naming schemes impact the performance but not the correctness of the overall scheme. In this paper, we simply treat an event name as a list of attribute-value pairs.

Some event-based collaboration systems exist in the Internet today. Salamander [16] supports realtime interaction using event forwarding proxies. That paper does not discuss proxy organization in detail, thus it is difficult to evaluate its scalability. There are similar commercial products, e.g., SmartSockets [25] and WebLogic [27]. These systems emphasize API design for system integration, but do not emphasize proxy organization for event dissemination, which we believe is important to achieve

3 Similar ideas of using “application-level routing” are being explored in the context of web caching [29].
IP multicast routing protocols provide good examples of scalable dissemination architectures. Particularly, the bi-directional shared tree [2], [14] reduces multicast forwarding state by sharing a distribution tree among all sources of a multicast group. Our previous work, a web cache consistency architecture [28], adopted this shared tree idea and used a proxy hierarchy to deliver page invalidations to web caches. This can be seen as an event dissemination architecture customized for web caching. In this paper we extend this to a scalable dissemination architecture for general event services.

IV. A HIERARCHICAL PROXY ARCHITECTURE

A. Event

We define an event as a list of attribute-value pairs, and an event type as the list of attributes of the corresponding events. In order to express its interest in a particular event type, a receiver includes in an event specification the expected value ranges of the attributes of the event type. For example, a stock event type contains only two values: name and price. An event specification for “I only want to be notified when stock A drops below $10” can be in the form of \{Stock.Name=A, Stock.Price<10\}. Two or more event specifications may be merged to form a superset specification [4], e.g., \{Stock.Name=A, Stock.Price<10\} and \{Stock.Name=A, Stock.Price<15\} can be merged as \{Stock.Name=A, Stock.Price<15\}\footnote{More complicated expressions are possible with more powerful event naming schemes. Please refer to the discussion of event naming in Section III for more details.}.

B. Proxy Hierarchy

Our proxy hierarchy is glued together by multicast groups (Fig. 2). Each group is “owned” by one proxy, which is called the parent proxy; all other proxies in the group are called child proxies. Every source (and receiver) attaches to the hierarchy via a proxy, which is called its primary proxy. Every proxy joins the group owned by its parent, and its own group if it is not a leaf.

We do not address the issue of creation and maintenance of the hierarchy; protocols in previous work [22], [24] can be readily applied here.\footnote{The key idea behind these proposals is to use a single global multicast group in which participants periodically advertise their presence. With this information, they are able to form clusters based on proximity or other metrics. Because of the periodic advertisements, they are able to re-cluster when someone fails and its advertisements are no longer heard.}

All proxies in a group exchange periodic heartbeat messages. These heartbeats serve to indicate the liveness of each proxy to all other proxies within the group. These messages are delivered unreliable via IP multicast, which is used solely for its efficiency. Each proxy has a timer associated with each of its group members. Upon receiving a heartbeat from proxy \(P_2\), proxy \(P_1\) resets its timer for \(P_2\). If the timer expires, a failure recovery mechanism is triggered. In brief, this recovery mechanism (Section IV-F) invalidates \(P_1\)’s state for \(P_2\). Later, when \(P_1\) hears \(P_2\)’s heartbeat, it explicitly re-synchronizes its state with \(P_2\).

C. Basic Operations

Sources do not maintain subscriber lists. Instead, they simply send events to their primary proxies. Every proxy keeps an event routing table, whose entry is of the form \((E, IPL, OPL)\). This table completely describes the forwarding state at a proxy. It maps every event type \(E\) known to the proxy to a list of incoming proxies (IPL) and a list of outgoing proxies (OPL). The incoming proxies indicate “directions” towards which there exists at least one source of \(E\). The outgoing proxies indicate “directions” towards which there exists at least one receiver of \(E\). We will term the former source state, and the latter receiver state. Every outgoing proxy has an associated event specification. An event is forwarded to an outgoing proxy if and only if it matches the specification of the proxy, i.e., it satisfies all expressions in the specification.

The event routing table is established by two procedures: registration and subscription. Sources notify the proxy hierarchy of their existence using registrations. Intuitively, as registrations propagate within the proxy hierarchy, they set up source state at proxies. Similarly, receivers notify the proxy hierarchy of their interest in events using subscriptions. As subscriptions propagate within the proxy hierarchy, they establish receiver state at proxies.

The rules by which registrations and subscriptions propagate through the proxy hierarchy determine the amount of state at each proxy. At one extreme, each proxy can flood every subscription to every other proxy; this can result in significant receiver state at each proxy, but little or no source state. Conversely, if registrations are flooded everywhere, receiver states need to be established along the path from each receiver to each related proxy.
source. Our goal is to strike a balance between the amount of receiver states and source states, while avoiding flooding of events, registrations or subscriptions. To accomplish this, we leverage the restricted topology provided by a hierarchy (as opposed to the more general topology of the mesh) to propagate registrations “far enough” to avoid flooding subscriptions. The following subsections describe this idea in more detail.

C.1 Registrations

Because registration is meant to setup forwarding state for subscriptions, we first examine the requirements of subscription forwarding. Our goal is to forward subscriptions towards sources, thereby “pulling” down events towards receivers. In a proxy mesh, to avoid flooding a subscription, each proxy P must know which of its neighbors might lead to event sources. As we discussed before, it is unclear how well the mechanisms for setting up this state scale. In a proxy hierarchy, there are only two choices that a proxy P has for forwarding subscriptions: to its parent, or to some children. In order to make correct forwarding decision, P has to know which event types have sources in the directions of its parent and children. It is expensive to know all event types for P’s parent, because it virtually means to enumerate all existing event types in the hierarchy. However, due to the restricted topology in a proxy hierarchy, it is possible for P to keep source states only for the events that have sources in its subtree; if P encounters a subscription with an unknown source, it simply forwards it to its parent. With this mechanism, the amount of source state at any proxy P is proportional to the number of event types that have a source within the subtree of P. Next we use an example to illustrate the details of the registration procedure.

When a new source S1 of event type E starts, it sends to its primary proxy a registration message REG(E) which contains the definition of event type E (Fig. 3(a)). Reliable unicast (e.g., TCP) is used to propagate registration messages. After registration, the source starts to send periodic heartbeats to its primary proxy to indicate its liveness.

When a proxy P gets a registration from a source (or a neighbor proxy) S, P adds S in its incoming proxy list for E, then makes different forwarding decisions for three different scenarios:

- If P has no incoming proxy of E, it knows that E is a new event type from a source in P’s subtree, then forwards the registration to its parent (e.g., P2 in Fig. 3(a)). This ensures that every proxy knows about all event types in its subtree.

- If P has at least one incoming proxy, it does not propagate the registration to its parent. Furthermore, if P has exactly one incoming proxy, say H, H is not aware that P leads to another source. Therefore, when a new source registers, P should forward the registration to H. P should also send a registration towards the new source to inform it of the existing sources towards H. For example, in Fig. 3(b), when P1 receives the registration from the new source S2, it informs P2 of the new source S2, and inform P3 of the existing source S1. This ensures that a later subscription will be forwarded towards all sources in the hierarchy. When a new source S1 of event type E starts, it sends to its primary proxy a registration message REG(E) which contains the definition of event type E (Fig. 3(a)). Reliable unicast (e.g., TCP) is used to propagate registration messages. After registration, the source starts to send periodic heartbeats to its primary proxy to indicate its liveness.

- Finally, if P has more than one incoming proxy, it sends a registration towards the new source to inform the proxies along the path that P leads to other sources besides this newly registered source (e.g., P3 in Fig. 3(c)). Registrations are propagated reliably between proxies. An alternative approach would have been to periodically advertise registrations between proxies, with each

An alternative design is to embed a hierarchy in a proxy mesh, in a way identical to multicast routing. Doing so, however, does not invalidate our design. Because we do not prescribe how to construct a hierarchy, our design will also work even if the hierarchy is embedded in a mesh.

This does not violate our previous statement that a proxy only keeps state about events that have at least one source in the proxy’s subtree. H gets the registration because it has a source of E in its subtree; otherwise it will never hear anything about E.
advertisement *refreshing* the existing state at the proxy. These periodic advertisements can simultaneously indicate the liveness of proxies and sources. But, because this “soft-state” approach might incur significant traffic due to the potentially large number of events to be registered, we choose instead to use heartbeats for detecting proxy liveness, and to use a separate recovery protocol to re-establish state after a communication or proxy failure (Section IV-F). This is similar to the use of heartbeats in the Border Gateway Protocol in IP routing [21].

When a source wants to stop sending event type E, it de-registers E, so that no further subscriptions are forwarded towards it and no proxy state is unnecessarily consumed. The de-registration procedure is similar to registration. The source sends a de-registration to its primary proxy. Alternatively, if the source fails, the absence of its heartbeats causes the corresponding primary proxy to initiate the de-registration message. Upon receiving a de-registration from proxy S, proxy P removes S from its incoming proxy list of the event. If there is only one incoming proxy H left, P forwards the de-registration to H because P knows about no other sources. If no incoming proxy is left, P forwards the de-registration to its parent to inform it that no source exists towards P. Otherwise, the message stops at P.

**C.2 Subscriptions**

Requirements of event forwarding are slightly different from those of subscription forwarding. It is undesirable for proxies to forward every unknown event to the root. High frequency events may significantly increase processing load at the root. To avoid this, every proxy maintains receiver state for all event types that have at least one receiver or at least one source in its subtree. This allows unsubscribed events to be dropped before they reach the root, thus reducing bandwidth consumption and avoiding traffic concentration near the root. We discuss the detailed subscription protocol below.

To indicate its interest in an event type, a receiver sends a subscription message \( \text{SUB}(E_i) \) containing its event specification \( E_i \) to its primary proxy. As with registrations, reliable unicast is used to propagate all subscriptions. To indicate its liveness, the receiver then sends periodic heartbeats to its primary proxy.

When a proxy P receives a subscription for \( E_i \) from a receiver (or neighbor proxy) R, it makes forwarding decision based on its incoming proxy list for the event type corresponding to \( E_i \). If, at P, there already exists an entry E whose event type corresponds to \( E_i \), P adds R into the outgoing proxy list of E. Furthermore, if \( E_i \) matches any existing subscription associated with the outgoing proxy list for E, P does not propagate the subscription further. Otherwise, P tries to merge the subscription with existing ones, and sends the merged subscription to all incoming proxies (other than R).\(^8\)

If P has no entry whose event type E corresponds to \( E_i \), it creates a new routing table entry \( (E, \text{Parent}, \{P\}) \). Then, P forwards the subscription to its parent. In the worst case, the subscription will reach the root. If the root has no registered sources for the event type E, we allow receivers to specify whether the hierarchy should keep the subscription, or drop it. If the receiver wants to subscribe to an event regardless of whether there exists a source, it sets a PRE bit in its subscription and a lifetime \( L \), and the root will then keep this subscription for time \( L \). If there is no PRE bit or the subscription’s lifetime expires, the root will drop the subscription and send back an unsubscribe message (see below) to remove state in downstream proxies.

If a proxy receives a registration from a new source after it has got a subscription, it will send the subscription towards the new source. Take Fig. 4 as an example. When proxy P1 receives a registration from P4 and it already has one incoming proxy P3 for E, it knows that S2, a new source of E, has started in the direction of P4. Because when R subscribed there was no source in the direction of P4, P4 has not received any subscription, which prohibits event forwarding from S2 to R. In order to correctly forward events, P1 sends its subscription towards P4 to establish receiver state.

Cancelling a subscription can be done in the same way as canceling a registration. An unsupervised message UNSUB(E) is propagated in the hierarchy. It is handled in the same way as the de-registration, except that it deals with the outgoing proxy list instead of the incoming proxy list.

**C.3 Forwarding**

After the routing tables are setup, it is straightforward to forward events. The source simply sends all of its events to its primary proxy. Whenever a proxy \( P_i \) gets an event from \( P_j \), it forwards the event to those outgoing proxies whose subscriptions match the event (except

\(^8\)Please see Section IV-A for definitions of match and merge.
to $P_2$ if that is an outgoing proxy). If no routing table entry is found, or the entry contains no outgoing proxy, the event is dropped.

D. Sporadic Source

Using the basic operations, a source must register its event before it sends the event. It also keeps sending heartbeats after registration. When a source is highly sporadic, i.e., it only occasionally sends an event, the latency of registration and the overhead of heartbeat traffic may be unacceptable. In such cases, it is preferable to skip registration and directly send out an event. We support this by encapsulating an event in a registration message. An encapsulated registration is forwarded like an event, except that when there is no matching routing table entry, the message is forwarded to the parent instead of dropped. If the root gets an encapsulated registration for which there are no matching subscriptions, it drops the message. To make encapsulated registrations work, subscription state must exist at proxies when the registration arrives. Receivers may use the PRE bit to make sure that their subscriptions persist in proxies even if there is no currently known sources. Note that this approach is not scalable to large amount of sporadic sources, because every encapsulated registration must reach the source and receiver state must be kept in many proxies. It remains a future work to devise a scalable delivery mechanism for sporadic sources.

E. Temporal Forwarding

Using the basic operations, forwarding decision is solely based on the current event. In other words, every event is forwarded independently and does not change proxy state. Sometime it is more desirable to let events change the proxy state and affect the forwarding decision of future events of the same type. For example, if an event is used to invalidate a data item instead of updating it, it is only delivered once to every receiver. If a receiver refetch the updated item later, it can subscribe again to receive new invalidations. Because it introduces temporal relationship among forwarding decisions, we label this type of forwarding as temporal forwarding.

We provide a simple method to support temporal forwarding. Each receiver includes a maximum forwarding count in its subscription; this count is then kept in the outgoing proxy lists. Whenever a proxy forwards an event, it decreases the count of all matching entries in the outgoing proxy list. When the count drops to 0, the outgoing proxy is removed as if an unsubscription is received and no more events will be forwarded.

It is possible to provide more powerful mechanisms for temporal forwarding. For example, we can allow receivers to specify temporal conditions in the optional expression of their subscriptions. Proxies cache all events that have a subscription with temporal conditions and use the cached events to make forwarding decisions. On the extreme, an event may carry Java code segment and makes forwarding decision for itself, similar to the ideas in active network [7] and active naming [26]. This mechanism puts event-specific computation during the forwarding process, thus reduces the load of receivers and sources. It remains as future work to explore the potential of this mechanism.

F. Failure Handling and Recovery

When a proxy or a link between proxies fails for an extended period of time, the failure will be detected by neighbor proxies using heartbeats. In this section, we discuss how proxies handle detected failure and recover when the failure is healed. The mechanism discussed here does not address automatic repair of the hierarchy (Section II).

The failure of a proxy disrupts all event propagations that it is supposed to deliver to parent and child proxies. Suppose proxy $P_1$ detects a failure of a parent $P_2$. $P_1$ then invalidates entries in proxies downstream of it which might be affected by either sources or receivers reachable via $P_2$. Note that $P_1$ can determine the relevant sources and receivers by scanning its entire table. It then sends, to all its children, de-registrations and unsubscriptions for the sources and receivers respectively. When $P_1$ detects a failure of one of its children (say $P_3$), it sends similar de-registrations and unsubscriptions to its parent, and to other children.

After the failed proxy, say $P_2$, is heard from again, $P_1$ resynchronizes its state with $P_3$. To do this, $P_1$ sends a QUERY message which requests $P_3$ to re-send registrations and subscriptions corresponding to entries for which $P_1$ is either an incoming proxy or an outgoing proxy. In the event that connectivity between $P_1$ and $P_2$ had temporarily failed, $P_1$ will receive the replies. However, if $P_2$ had lost all state, it would, instead, send QUERY messages to its parent and all its children. These query messages would allow $P_2$ to reconstruct its state. QUERY messages should be sent periodically until the proxy gets a response. This is meant to deal with exceptions such as failure during recovery. QUERY messages can be sent via unreliable unicast, or piggybacked in heartbeats.

V. Conclusion and Future Work

Event service is an important building block for collaborative applications over the Internet. An event dissemination architecture is essential for an event service to be scalable. In this paper we compared the design choices of event dissemination architectures in terms of scalability and failure handling, and identified the limitations of
distributed and single-proxy event distribution. We then presented the design of a hierarchical event dissemination architecture, which extended the shared tree concept in IP multicast routing protocols. It reduces proxy state compared with previous approaches, and provides mechanisms to inform applications about failures and to recover from link failure or proxy failure.

We plan to continue this work in several directions. First, we will use simulations to quantitatively study the tradeoff between proxy mesh and proxy hierarchy in terms of event delivery latency and proxy forwarding state. The key issue here is to characterize the interaction between proxy topology, network topology and spatial locality of subscriptions. With this study, we will be able to better identify the class of applications for which this hierarchical architecture is suitable.

Second, we will explore the possibility of reducing delivery latency within the proxy hierarchy. Instead of using a proxy mesh which installs forwarding state of every event everywhere, we believe that it is possible to install more forwarding state in the hierarchy to provide “tree shortcuts” to reduce latency for a subset of events. This increases state at some proxies, but greatly reduces latency for the events that require it. We expect it to provide a middle ground between the hierarchy architecture and the distributed architecture.

REFERENCES