Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification

Krzysztof Ostrowski, Cornell University, USA

Ken Birman, Cornell University, USA

Danny Dolev, The Hebrew University of Jerusalem, Israel

ABSTRACT

Existing Web service notification and eventing standards are useful in many applications, but they have serious limitations that make them ill-suited for large-scale deployments, or as a middleware or a component-integration technology in today's data centers. For example, it is not possible to use IP multicast, or for recipients to forward messages to others, scalable notification trees must be setup manually, and no end-to-end security, reliability, or QoS guarantees can be provided. We propose an architecture that is free of such limitations and that may serve as a basis for extending or complementing the existing standards. The approach emerges from our work on QuickSilver, a new, extremely modular and extensible platform for high-performance, scalable, reliable eventing.

Keywords: *architecture; eventing; extensible; multicast; notification; publish-subscribe; reliable; scalable*

INTRODUCTION

Motivation

Notification is a valuable, widely used primitive for designing distributed systems. The growing popularity of RSS feeds and similar technologies shows that this is also true at the Internet scales. The WS-Notification (Graham et al., 2004) and WS-Eventing (Box et al., 2004) standards have been offered as a basis for interoperation of heterogeneous systems deployed across the Internet. Unlike RSS, they are subscription-based and, hence, free of the scalability problems of polling, and they support proxy nodes that could be used to build scalable notification trees. Nonetheless, they embody restrictions that make them unsuitable as a middleware technology in large-scale systems:

- **No forwarding among recipients:** Many content distribution schemes build overlays within which content recipients participate in message delivery. In current Web services notification standards, however, recipients are *passive* (limited to data reception). For example, given the tremendous success of BitTorrent for multicast file transfer, one could imagine a future event notification system that uses a BitTorrent-like protocol for data transfer. But BitTorrent depends on direct peer-to-peer interactions by recipients.
- **Not self-organizing:** While both standards permit the construction of notification trees, such trees must be manually configured and require the use of dedicated infrastructure nodes ("proxies"). Automated setup of dissemination trees by means of a protocol running directly between the recipients is often preferable, but the standards preclude this possibility.
- **Weak reliability:** Reliability in the existing schemes is limited to per-link guarantees resulting from the use of TCP. In many applications, end-to-end guarantees are required, and often of strong flavor, for example, to support virtually synchronous, transactional, or state-machine replication. Because receivers are assumed passive and cannot cache, forward messages, or participate in multiparty protocols, even weak guarantees of these sorts cannot be provided.
- **Difficult to manage:** It is hard to create and maintain an Internet-scale dissemination structure that would permit any node to serve as a publisher or as a subscriber, for this requires many parties to maintain a common infrastructure and agree on standards, topology, and other factors. Any such large-scale infrastructure should respect local autonomy, whereby the owner of a portion of a network can set up policies for local routing, availability of IP multicast, and so forth.
- **Inability to use external multicast frameworks:** The standards leave it entirely to the recipients to prepare their communication endpoints for message delivery. This makes it impossible for a group of recipients to dynamically agree upon a shared IP multicast address, or to construct an overlay multicast within a segment of the network. Yet such techniques are central to achieving high performance and scalability, and can also be used to provide QoS guarantees or to leverage emergent technologies.

In this article, we propose a principled approach to Web service notification in large-scale systems, free of the limitations listed above, which is modular and highly extensible. The design presented here is a basis for Quicksilver (Ostrowski & Birman, 2006c; Ostrowski, Birman & Dolev, 2006), a novel, reliable, and extremely scalable platform for publish-subscribe eventing and notification, under development at Cornell. While this architecture is inspired by our prior work on QuickSilver, it is designed to be generic, and it is compatible, in general, with a wide range of existing protocols.

Model

We employ the usual terminology, where events are associated with *topics*, produced

Figure 1. Publishers and subscribers register for a topic with the subscription manager

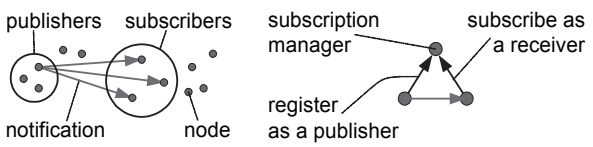
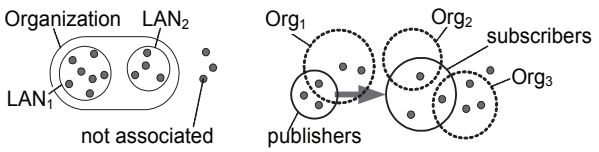


Figure 2. Nodes can be scattered across administrative domains hierarchically divided into subdomains



by *publishers,* and delivered to *subscribers*. We use the term "*group X*" to refer to the set of nodes subscribed to topic "*X*." More than one node may publish to a given topic. The prospective publishers and subscribers register with a *subscription manager*. This entity can be independent of the publishers (Figure 1). The manager may be replicated to tolerate failures, or hierarchical, to scale. A single manager may track the publishers and subscribers for many topics, and many independent managers may coexist. Nodes may reside in many *administrative domains* (LANs, data centers, etc.). Nodes in the same domain may be *jointly managed*. It is often convenient to define policies, such as for message forwarding or resource allocation, in a way that respects domain boundaries, for example, for administrative reasons, or because communication within a domain is cheaper than across domains, as it is often related to network topology. Publishers and subscribers might be scattered across organizations. These need to cooperate in message dissemination, which often presents a logistic challenge (Figure 2).

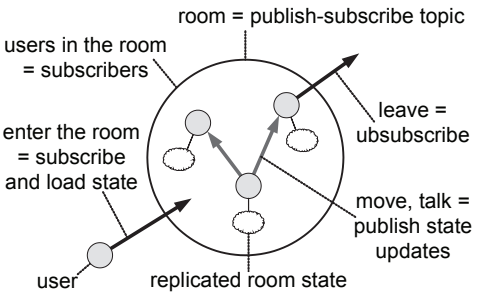
Example

To facilitate discussion, the article uses a running example. Obviously, the architecture is not limited to any particular application; the architecture is intended to be flexible enough to serve as a general, multipurpose middleware component-integration technology, and to be used in settings such as large data centers, trading systems, military infrastructure, and so forth. However, the example helps us illustrate the architecture with specific scenarios that highlight the role of specific features.

The example involves a possible vision for the future of the Internet. Even today, the Web is moving towards very dynamic and interactive content. Massively multiplayer online gaming and virtual realities, such as the World of Warcraft and Second Life, are becoming increasingly popular. However, current techniques are insufficient for massive-scale deployments. As of this writing, the latter platform is reported to host online 17,000 of a total of 2 million of its users, on a server farm of nearly 2,600 dual-core Opteron machines. As the number of online users simultaneously browsing through such virtual realities will grow to tens and hundreds of millions, as users start to expect a smoother and more realistic experience, and they start transmitting high-resolution audio, video, and animation streams, it will become very difficult to host entire virtual worlds in a centralized manner. Centralized systems are invariably costly and suffer from bottlenecks and high latencies. We believe that the most cost-effective (and perhaps ultimately the only feasible) way to implement such scenarios is for the virtual worlds to be decentralized, and for the users to interact directly, without any expensive servers in the middle.

Apart from the scalability aspect mentioned above, we also believe that the Internet community is unlikely to accept a situation where all content is controlled by a handful of providers. Instead, we envision that just as today users create Web pages, they would eventually want to be able to create their own virtual rooms or landscapes, where participants could interact with one-another much as they do in online multiplayer games. A successful technology base of this sort could eventually transform

Figure 3. A virtual room in a virtual world, modeled as a "publish-subscribe" topic



the Web into a multi-verse, a federation of millions of interconnected virtual places or entire worlds, created and hosted by the Internet community collaboratively, with a mixture of infrastructure services, services hosted on data centers, and services introduced and hosted by individual users.

A high-performance, scalable, and reliable variant of the *publish-subscribe* paradigm could be the enabling technology for such applications. To see this, think of each location in a virtual world as a separate publish-subscribe topic, and of the users that currently reside inside the location as subscribers (Figure 3). The state of the room, for example, its interior, user positions, actions in progress, all objects inside the room, and so forth, is replicated among all the subscribers. The state is loaded by the users upon entering the room, and can be updated in a consistent manner by multicasting any updates (such as users speaking, moving, handing objects to each other, etc.) reliably to the set of all subscribers. The state can be retained while no visitors are in the room by a room *guardian*, a special entity that can either stay in the room at all times, or slip in only if the last "regular" user leaves, depending on how the room is setup. Unlike in the centralized approaches, here users interact directly with each other, with no server in between. Although this solution does require infrastructure components, for example, to track subscriptions, inform users of each other's existence, control the "guard," and so forth, none of the required infrastructure sits on the "critical path" and acts as a proxy or intermediary. If one user speaks or projects a video clip to others, the data can be transmitted directly to participants, without the involvement of any such infrastructure. Infrastructure that controls a virtual location could thus be hosted even on a home machine of the user who created it. The network of users' home machines, hosting their virtual rooms connected by virtual corridors, can thus form a background *backbone* structure that controls the virtual reality in a way similar to how DNS serves as a backbone of the Internet.

To realize the vision outlined above, we need a publish-subscribe platform that can provide very high performance, scalability in multiple dimensions, such as the size of the system, the number of publish-subscribe topics, or data rates, and so forth, end-to-end reliability guarantees, and a way to integrate with modern development platforms. In work reported elsewhere, we have created a system with these attributes. Initial results from experiments on the system (Quicksilver Scalable Multicast, or QSM) suggest that these goals can be achieved (Ostrowski & Birman, 2006b, 2006c). However, for truly massive adoption, we need publish-subscribe interoperability standards that allow a large number of independent users, residing in different administrative domains scattered across the Internet, to collaboratively form a single infrastructure for reliable publish-subscribe notification. This was our original reason to explore the architectural proposal that is the subject of this article.

Let's now revisit the problems listed in the section entitled "Motivation" in the context of our example. Consider a user sitting in a cafeteria with his laptop that enters a virtual room and starts interacting with friends in our three-dimensional virtual reality. The user would act as one of several publishers and subscribers. Because such interactions would be ad-hoc, the technique that the users would use to disseminate data between them should be *self-organizing*, that is, it should not rely on proxies, or other dedicated infrastructure.

A simple way to meet this requirement could be for each publisher to send updates directly to all subscribers over TCP, but this is not acceptable for several reasons. First, to ensure that all users see a consistent history of events, we'd need *reliability guarantees*, such as a global ordering of all updates published by different users (so that all of them see events occurring in the same order, or that two users don't pick up the same object), atomicity (so that if one user can see something happening, then so do all the others) and so forth. While solutions to such problems are well known from the literature, they need more than a plain point-to-point TCP-based dissemination scheme. Thus, the users would need to run a suite of special reliability protocols.

Second, the wireless link of the user in the cafeteria, or his laptop, may not be fast enough to simultaneously send updates to ten other people who may be in the virtual room. If other users are connected, for example, directly to the campus network, over a wire, it may be desirable to arrange it so that the wireless user publishes updates to a user on a campus LAN, which is then responsible for *forwarding* it to the others. If the campus LAN is configured to enable IP multicast, it would be desirable to be able to exploit such *external mechanisms*. The users should thus be able to quickly form small overlays that can efficiently utilize whatever resources are available.

Finally, note that users will often reside in different administrative domains, for example, in a cafeteria wireless network, in different campus networks, on a cable LAN, and so forth. The administrators of these domains may need to impose acceptable use policies that specify how dissemination should be performed internally in the domain they own. For example, one domain might disallow IP multicast, while another might permit IP multicast provided that various rules are respected. Policies could govern sharing of connections, what data rates are acceptable, what multicast protocol to use, and so forth.

Different domains will often have distinct administrative policies. And yet, users residing in those domains would expect a smooth operation, as if the entire Internet formed a single, fully-connected administrative domain. Such *management* issues are a logistic headache, and require better *interoperability* standards. An update published by the user in the cafeteria should be disseminated in every campus network, or among wireless users, locally according to the local policies setup by the domain administrators, but all these domains need to cooperate with each other, ideally without reliance on costly *proxies*. Existing eventing standards have overlooked such issues, but they form the core of our proposal.

Design Principles

The limitations of the existing architectures, listed in the section on "Motivation," and our experience in designing scalable multicast systems, led us to the following design principles:

- **Programmable nodes:** Senders and recipients should not be limited to sending or receiving. They should be able to perform certain basic operations on data streams, such as forwarding or annotating data with information to be used by other peers, in support of local *forwarding policies*. The latter must be expressive enough to support protocols used in today's content delivery networks, such as overlay trees, rings, mesh structures, gossip, link multiplexing, or delivery along redundant paths.
- **External control:** Forwarding policies used by subscribers must be selected and

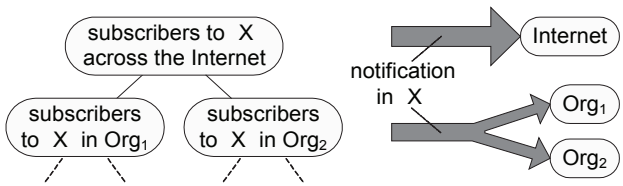
updated in a consistent manner. A node cannot predict a-priori what policy to use, or which other nodes to peer with; it must thus permit an external trusted entity or an agreement protocol to control it: determine the protocol it follows, install rules for message forwarding or filtering, and so forth.

- **Channel negotiation:** The creation of communication channels should permit a handshake. A recipient might be requested to, for example, join an IP multicast address or subscribe to an external system. The recipient could also make configuration decisions on the basis of the information about the sender. For example, a LAN domain asked to create a communication endpoint for receiving could select a well-provisioned node as its entry point to handle the anticipated load.
- **Managed channels:** Communication channels should be modeled as *contracts* in which receivers have a degree of control over the way the senders are transmitting. In self-organizing systems, reconfiguration triggered by churn is common and communication channels often need to be reopened or updated to adapt to the changing topology, traffic patterns, or capacities. For example, a channel that previously requested that a given source transmits messages to one node may notify the source that messages should now be transmitted to some two other nodes.
- **Hierarchical structure:** The principles listed above should apply to not just individual nodes, but also to entire administrative domains, such as LANs, data centers, or corporate networks. This allows the definition and enforcement of Internet-scale forwarding policies, facilitating cooperation among organizations in maintaining the global infrastructure. The way messages are delivered to subscribers across the Internet thus reflects policies defined at various levels (for example, policies "internal" to data centers, and a global policy "across" all data centers).
- **Isolation and local autonomy:** A degree of a local autonomy of the individual administrative domains, such as how messages are forwarded internally, which nodes are used to receive incoming traffic or relay data to other domains, and so forth, should be preserved. In essence, the internal structure of an administrative domain should be hidden from other domains it is peering with and from the higher layers. Likewise, the details of the subcomponents of a domain should be as opaque as possible.
- **Reusability:** It should be possible to specify a policy for message forwarding or loss recovery in a standard way and post it into an online library of such policies as a contribution to the community. Administrators willing to deploy a given policy within their administrative domain should be able to do so in a simple way, for example, by drag-and-drop, within a suitable GUI.
- **Separation of concerns:** Motivated by the end-to-end principle, we separate implementation of loss recovery and other reliability properties from the unreliable dissemination of messages, as well as from message ordering, security, and subscription management. Accordingly, our design includes *reliability*, *dissemination*, *ordering*, *security*, and *management* frameworks, five independent, yet complementary structures. This decoupling gives our system an elegant structure and a degree of modularity and flexibility unseen in existing architectures.

Hierarchical View of the Network

A group X of subscribers for a given topic across the entire Internet can be divided into subsets Y_1, Y_2, ..., Y_N of subscribers in N top-level administrative domains (Figure 4). This may continue recursively, leading to a hierarchical perspective on the group X. Hierarchies of this sort have been previously exploited in scalable multicast protocols, for example, in RMTP (Paul, Sabnani, Lin, & Bhattacharyya, 1997), or in the context of content-based filtering

Figure 4. A hierarchical decomposition of the set of subscribers along the domain boundaries



(Banavar, Chandra, Mukherjee, Nagarajarao, Strom, & Sturman, 1999). The underlying principle, implicit in many scalable protocols, is to exploit locality. Following this principle, sets of nodes, clustered based on proximity or interest, cooperate semi-autonomously in message routing and forwarding, loss recovery, managing membership and subscriptions, failure detection, and so forth. Each such set is treated as a single cell within a larger infrastructure. A protocol running at a global level connects all cells into a single structure. Scalability arises as in the *divide-and-conquer* principle. Additionally, the cells can locally share workload and amortize dissemination or control overheads, for example, buffer messages from different sources and locally disseminate such combined bundles, and so forth.

In the architecture described here we go one step further. Following our principle of *isolation* and *local autonomy*, each administrative domain should manage the registration of its own publishers and subscribers internally, and it should be able to decide how to distribute messages among them or how to perform loss recovery according to its local policy. Unlike in most hierarchical systems, where hierarchy and protocol are inseparable, and hence the "subprotocols" used at all levels of the hierarchy are identical, in our architecture we decouple the creation of the hierarchy from the specific "subprotocols" used at different levels of the hierarchy and we allow the "subprotocols" to differ. Thus for example, our architecture permits the creation of a single, global dissemination scheme for a topic that uses different mechanisms to distribute data in different organizations or data centers. Likewise, it permits the creation of a single Internet-scale loss recovery scheme that employs different recovery policies within different administrative domains. Previously, this has only been possible with proxies, which can be costly, and which introduce latency and bottleneck. In this article, we propose a way to do this efficiently, and in a very generic, flexible manner. This novel hierarchical protocol "composition" approach, motivated by the principles of locality and local autonomy, is central to our architecture.

ARCHITECTURE

The Hierarchy of Scopes

Our architecture is organized around the following key concepts: management scope, forwarding policy, channel, filter, session, recovery protocol, recovery domain, and agent.

A *management scope* (or simply *scope*) represents a set of jointly managed nodes. It may include a single node, span over a set of nodes residing within a certain administrative domain, or include nodes clustered based on other criteria, such as common interest. In the extreme, a scope may span across the entire Internet. We do not assume a 1-to-1 correspondence between administrative domains and the scopes defined based on such domains, but that will often be the case. A LAN scope (or just a LAN) will refer to a scope spanning all nodes residing within a LAN. The reader might find it easier to understand our design with such examples in mind.

A scope is not just *any* group of nodes; the assumption that they are *jointly managed* is essential. The existence of a scope is dependent upon the existence of an infrastructure that

maintains its membership and administers it. For a scope that corresponds to a LAN, this could be a server managing all local nodes. In a domain that spans several data centers in an organization, it could be a management infrastructure, with a server in the company headquarters indirectly managing the network via subordinate servers residing in every data center. No such global infrastructure or administrative authority exists for the Internet, but organizations could provide servers to control the Internet scope in support of their own publishers or to manage the distribution of messages in topics of importance to them. Many global scopes, independently managed, could thus co-exist.

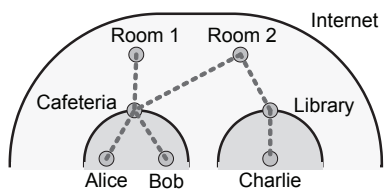
Like administrative domains, scopes form a hierarchy, defined by the relation of *membership*: one scope may declare itself to be a *member* (or a "subscope") of another. If X declares itself to be a member of Y, it means X is either physically or logically a part (subset) of Y. Typically, a scope defined for a subdomain X of an administrative domain Y will be a member of the scope defined for Y. For example, a node may be a member of a LAN. The LAN may be a member of a data center, which in turn may be a member of a corporate network. A node may also be a member of a scope of an overlay network. For a data center, two scopes may be defined, for example, *monitoring* and *control* scopes, both covering the entire data center, with some LANs being a part of one scope, or the other, or both. The corporate scope may be a member of several Internet-wide scopes, and so forth.

The generality in these definitions allows us to model various special cases, such as clustering of nodes based on interest or other factors. Such clusters, formed, for example, by a server managing a LAN and based on node subscription patterns, could also be treated as (virtual) scopes, all managed by the same server. Nodes would thus be members of clusters, and clusters (not nodes) would be members of the LAN. As we shall explain below, every such cluster, as a separate scope, could be locally and independently managed. For example, suppose that we are building an event notification system that needs to disseminate events reliably, and is implemented by an unreliable multicast mechanism coupled to a reliability layer that recovers lost packets. In the proposed architecture, different clusters could run different multicast or loss recovery protocols; this technique is used in the QSM platform (Ostrowski & Birman, 2006b, 2006c). Thus, if one cluster happens to reside within a LAN that permits the use of IP multicast, it could use that technology, while a different cluster on a network that prohibits IP multicast, or consisting of a large number of nodes across the Internet, could instead form an end-to-end multicast overlay.

The scope hierarchy need not necessarily be a tree (Figure 5). There may be many global scopes, or many superscopes for any given scope. However, a scope decomposes into a tree of subscopes, down to the level of nodes. The *span of a scope X* is the set of all nodes at the bottom of the hierarchy of scopes rooted at X. For a given topic X, there always exists a single global scope responsible for it, that is, such that all subscribers to X reside in the span of X. Publishing a message to a topic is thus always equivalent to delivering it to all subscribers in the span of some global scope, which may be further recursively decomposed into the sets of subscribers in the spans of its subscopes.

Suppose that Alice and Bob are sitting with their laptops in a cafeteria, while Charlie is in a library. Both the cafeteria's wireless network and the local network in the library are separately managed administrative domains, and they define their own *management scopes*. Alice's and Bob's laptops are also *scopes*, both of which

Figure 5. An example hierarchy of management scopes in a game



become members of the cafeteria's scope. Now suppose that Alice opens her virtual realities browser and enters a virtual place "Room1" in the virtual reality, while Bob and Charlie enter "Room2." Each of the virtual rooms defines a global, Internet-wide scope that could be thought of as "the scope of all users in this room, wherever they are." If the networking support for Alice and Bob is still provided by the cafeteria and library wireless networks respectively, when the students enter the rooms, the cafeteria's and library's scopes become *members* of these global scopes (Figure 5).

The Anatomy of a Scope

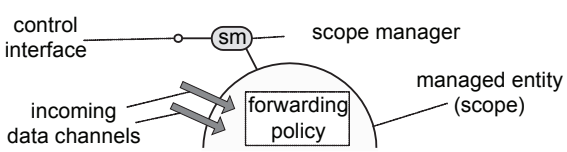
The infrastructure component administering a scope is referred to as a *scope manager* (SM). A single SM may control multiple scopes. It may be hosted on a single node, or distributed over a set of nodes, and it may reside outside of the scope it controls. It exposes a *control interface*, a Web service hosted at a well-known address, to dispatch control requests (e.g., "subscribe") directed to the scopes it controls (Figure 6).

A scope maintains communication *channels* for use by other scopes. A *channel* is a mechanism through which a message can be delivered to all those nodes in the span of this scope that subscribed to any of a certain set of topics. In a scope spanning a single node, a channel may be just an address/protocol pair; creating it would mean arranging for a local process to open a socket. In a distributed scope, a channel could be an IP multicast address; creating it would require all local nodes to listen at this address. It could also be a list of addresses, if messages are to be delivered to all of them, or if addresses are to be used in a random or a round-robin fashion. In an overlay network, for example, a channel could lead to a small set of nodes that forward messages across the entire overlay. In general, a scope that spans a set of nodes can thus be governed by a *forwarding policy* that determines how messages originating within the scope, or arriving through some communication channel, are disseminated internally, within the local scope.

Continuing our example, the cafeteria and the library would host the managers of their scopes on dedicated servers, and each of the student laptops would run a local service that serves as a scope manager for the local machine. The library's SM may be on a campus network with IP multicast enabled. When the cafeteria's SM requests a channel from the library's SM, the latter might, for example, dedicate some machine as the entry point for all messages coming from the cafeteria, instruct it to retransmit these messages to the IP multicast address, and instruct all other laptops to join the IP multicast address. Similarly, the cafeteria's SM might setup a local forwarding tree. In most settings, a scope manager would live on a dedicated server. It is also conceivable to offload, in certain scenarios, parts of the SM's functionality to nodes currently in the scope (e.g., to Alice's and Bob's laptops), but a single "point of contact" for the scope would still need to exist.

The control interfaces "exposed" by scopes (interfaces exposed by their scope managers) are "accessed" by other scopes (i.e., by their SMs). When interacting, scopes can play one of a few standard roles, corresponding to the three principal interaction patterns: *member-*

Figure 6. A scope is controlled by a scope manager, which exposes a standardized control interface, and may create a number of incoming data channels, to serve as "entry points" for the scope



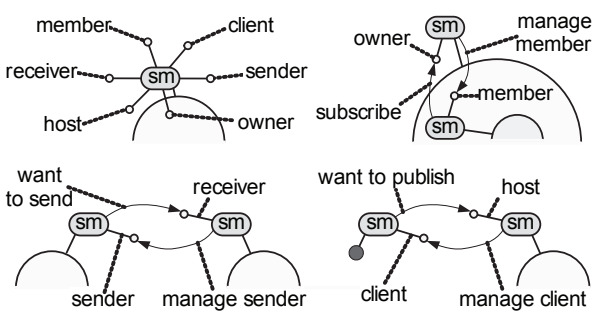
owner, *sender-receiver*, and *client-host* (see Figure 7). A *member* registers with an *owner* to establish a relation of *membership*, that is, to "declare" itself as a subscope of the owner. After such a relationship has been established, the member may then register with the owner as a publisher or subscriber in one or more topics. Depending on the topics for which the member has registered and its role in these topics, it may be requested by the owner to perform relevant actions, for example, by forwarding messages or participating in the construction of an overlay structure. At the same time, the owner is responsible for tracking the health of its members, and in particular for detecting failures and performing local reconfiguration.

The *client-host* relationship is similar to *member-owner*, in that a *client* can register with a *host* as a publisher or as a subscriber. However, unlike the *member-owner* relationship, which involves a mutual commitment, the *client-host* relationship is more casual, in the sense that the host cannot rely on the client to perform any tasks for the system, or even to notify the host when it leaves, and similarly, the client cannot rely on the host to provide it with the same quality of service and reliability properties as the regular members. Thus for example, while a member can form a part of a forwarding tree, a client will not; the latter will also typically get weaker reliability guarantees, because the protocols run by the scope members will not be prevented from making progress when transient clients are unreachable. The same applies to publishers. A long-term publisher that forms a part of a corporate infrastructure will register as a member, but handheld devices roaming on a wireless network will usually register as clients.

The *sender-receiver* relationship is similar to a publisher registering as a client or member in that the *sender* registers with the *receiver* to send data. However, whereas a *member* registers with its *owner* to publish to the set of all subscribers in a topic, including nodes inside as well as nodes outside of the *owner*, a *sender* will register with a *receiver* to establish a communication channel between the two to disseminate messages only within the scope of the receiver. When a publisher registers as a member with an owner scope that is not the global scope for the given topic, the owner may itself be forced to subscribe with its superscope. The sender-receiver relationship is horizontal; no cascading subscriptions take place. On the other hand, while a member scope never exchanges data with the owner (instead, it is "told" by the owner to form part of a dissemination structure that the owner controls), the sender-receiver relationship serves exactly this purpose; the two parties involved in the latter will negotiate protocols, addresses, transmission rates, and so forth.

The reader should recognize in our construction the design principles we articulated earlier. Scopes, whether individual nodes, LANs

Figure 7. When interacting, scopes follow one of a few standardized relationship patterns, in which they can play one of a few standard roles



or overlays, are *externally controlled* using the control interfaces exposed by SMs, may be *programmed* with policies that govern the way messages are distributed internally, forwarded to other scopes, and so forth, and transmit messages via *managed* communication channels, established through a dialogue between a pair of SMs, and dynamically reconfigured.

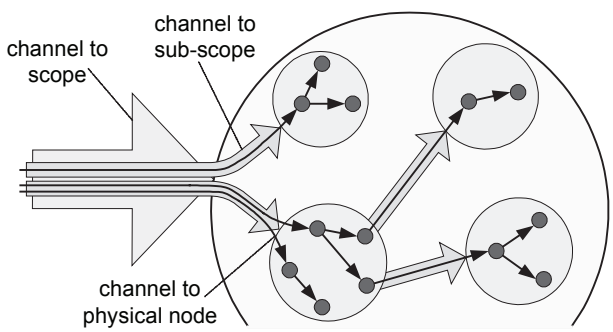
Hierarchical Composition of Policies

Following our design principles, we propose to solve the issue of a large-scale global cooperation in message delivery between independently managed administrative domains by introducing a hierarchical structure in which forwarding policies defined at various levels are merged into a single dissemination scheme. Each scope is configured with a policy dictating, on a per-topic (and perhaps a per-sender) basis, how messages are forwarded among its members. For example, a policy governing a global scope might determine how messages in topic T, originating in a corporate network X, are forwarded between the various organizations. A policy of a scope of the given organization's network might determine how to forward messages among its data centers, and so forth. A policy defined for a particular scope X is always defined at the granularity of X's members (not individual nodes). The way a given subscope Y of X delivers messages internally is a decision made autonomously by Y. Similarly, X's policy may specify that Y should forward messages to Z, but it is up to Y's policy to determine how to perform this task.

Accordingly, a global policy may request that organization X forward messages in topic T to organizations Y and Z. A policy governing X may then determine that to distribute messages in X, they must be sent to LAN_1, which will forward them to LAN_2. The same policy might also specify which LANs within X should forward to Y and Z, and finally, the policies of these LANs will delegate these forwarding tasks to individual nodes they own. When all the policies defined at all the involved scopes are combined together, they yield a global *forwarding structure* that completely determines the way messages are disseminated (Figure 8). In the examples given, the forwarding policies are simply graphs of connections: each message is always forwarded along every channel. In general, however, a channel could be constrained with a *filter* that decides, on a per-message basis, whether to forward or not, and may optionally tag the message with custom attributes (more details are in the section "Communication Channels"). This allows us to express many popular techniques, for example, using redundant paths, multiplexing between dissemination trees, and so forth.

Every scope manager maintains a mapping from topics to their forwarding policies. A forwarding policy is defined as an object that

Figure 8. Channels created in support of forwarding policies defined at different levels



lives in an *abstract context*, and that exposes a fixed set of *events* to which members must be prepared to react, and the *operations* and *attributes*. A scope manager might be thought of as a *container* for objects representing *policies*. The interfaces that the container and the policies expose to each other and interact with are standardized, and may include, for example, an event informing the policy that a new member has been added to the set of members locally subscribed to the topic, or an operation exposed by the container that allows the policy to request a member to establish a channel to another member and instantiate a given type of filter (Figure 9). A scope manger thus provides a *runtime environment* in which policies can be hosted. This allows policies to be defined in a standard way, independent not only of the platform, but also of the type of the administrative domain. For example, one can imagine a policy that uses some sort of a novel mesh-like forwarding structure with a sophisticated adaptive flow and rate control algorithm. Our architecture would allow that policy to be deployed, without any modifications, in the context of a LAN, data center, corporate network, or a global scope. In effect, we've separated the policy from the details of how it should be implemented in a particular domain. Policies may be implemented in any language, expose and consume Web service APIs, stored online in *protocol libraries*, downloaded as needed, and executed in a secure manner.

Graphs of connections for different topics, generated by their respective policies, are superimposed (Figure 10). The SM of the scope maintains an aggregate structure of channels and filters, and issues requests to the SMs of its members to create channels, instantiate filters, and so forth. If multiple policies request channels between the same pair of nodes, the SM will not create multiple channels, but rather a single channel, for multiple topics. To avoid the situation where every member talks to every other member, the SM may use a single forwarding policy for multiple topics, to ensure that channels created for different topics overlap.

Communication Channels

Consider a node X, which is a member of a scope Y that, based on a forwarding policy at Y, has been requested to create a communication channel to scope Z to forward messages in topic T. Following the protocol, X asks the SM of Z for the specification of the channel to Z that should be used for messages in topic T. The SM of Z might respond with an address/protocol pair that X should use to send over this channel. Alternatively, a forwarding policy defined for T at scope Z may dictate that, in order to send to Z in topic T, scope X should establish channels to members A and B

Figure 9. A forwarding policy as a code snippet

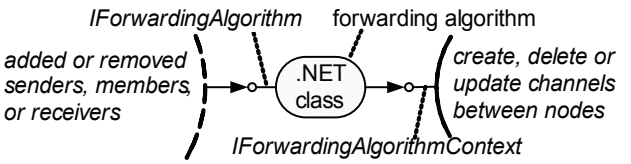
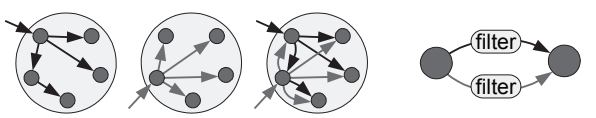


Figure 10. Forwarding graphs for different topics are superimposed. Two members may be linked by multiple channels, each with a different filter



of Z, constrained with filters α and β. After X learns this from the SM of Z, it contacts SMs of A and B for further details. Notice that the channel to Z decomposes into subchannels to A and B through a policy at a target scope Z. This procedure will continue hierarchically, until the point when X is left with a tree with filters in the internal nodes and address and protocol pairs at the leaves (Figure 11). Now, to send a message along a channel constructed in this way, X executes filters, starting from the root, to determine recursively which subchannels to use, proceeding until it is left with just a list of address/protocol pairs, and then transmits the message. Filters will usually be simple, such as modulo-**n**; hence X can perform this procedure very efficiently. Indeed, with network cards becoming increasingly powerful and easily programmable (Weinsberg, Dolev, Anker, & Wyckoff, 2006), such functionality might even be offloaded to hardware.

Accordingly, to support the hierarchical composition of policies described in the preceding section, we define a channel as one of the following: an address/protocol pair, a reference to an external multicast mechanism, or a set of subchannels accompanied by filters. In the latter case, the filters jointly implement a multiplexing scheme that determines which subchannels to use for sending, on a per-message basis (Figure 12, Figure 13).

Consider now the situation where scope X, spanning a set of nodes, has been requested to create a channel to scope Y. Through a dialogue with Y and its subscopes, X can obtain a detailed channel definition, but unlike in prior examples, X now spans a *set of nodes*, and as such, it cannot *execute* filters or *send* messages. To address this issue, we now propose two simple, generic techniques: *delegation* and *replication* (Figure 14). Both of them rely on the fact that if X receives messages in a topic T, then some of its members, Z, must also receive

Figure 11. A channel split into subchannels and a possible filter tree corresponding to it

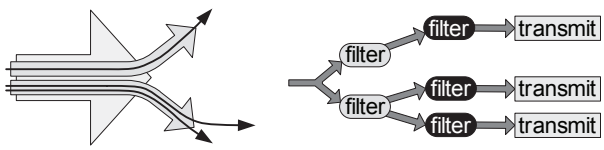


Figure 12. A channel may be an address/protocol pair (left), or it may consist of subchannels, with an algorithm deciding what goes where (right)

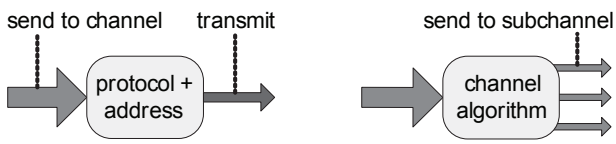


Figure 13. A distributed scope may delegate a channel or some of its subchannels to its members, or it may replicate the channel among members with filters that jointly implement a round-robin policy, and so forth

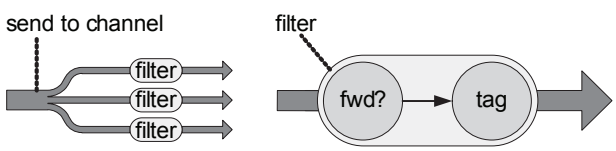
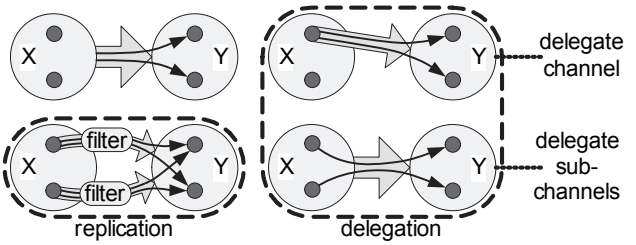


Figure 14. Channel algorithms are realized as sets of filters, one per subchannel, deciding whether to forward, and optionally adding custom tags.



them (for otherwise X would not declare itself as a subscriber and it would not be made part of the forwarding structure for topic T by X's superscope). In case of *delegation*, X requests such a subscope Z to create the channel on behalf of X, essentially "delegating" the entire channel to its member. The problem can be recursively delegated, down to the level where a single physical node is requested to create the channel, and to forward messages it receives along the channel. A more sophisticated use of delegation would be for X to delegate subchannels. In such case, X would first contact Y to obtain the list of subchannels and the corresponding filters, and for each of these subchannels, delegate it to its subscopes. In any case, X delegates the responsibility for forwarding over a channel to its subscopes.

Our approach is flexible enough to support even more sophisticated policies. An example of one such policy is a *replication* strategy, outlined here. In this scheme, scope X could request that **n** of its subscopes create the needed channel, constraining each with a modulo-**n** filter based on a message sequence number. Hence, while each of the subscopes would create the same channel on behalf of its parent scope (hence the name "replication"), subscope **k** would only forward messages with numbers **m** such that **m** mod **n** equals **k**. By doing this, X effectively implements a round-robin policy of the sort proposed in protocols such as MIT's SplitStream. Although all subscopes would create the same channel, the round-robin filtering policy would ensure that every message is forwarded only by one of them. This technique could be useful, for example, in the cases where the volume of data in a topic is so high that delegation to a single subscope is simply not feasible.

Our point is not that this particular way of decomposing functionality should be required of all protocols, but rather that for an architecture to be powerful enough and flexible enough to be used with today's cutting-edge protocols and to support state-of-the-art technologies, it needs to be flexible enough to accommodate even these kinds of "fancy" behaviors. Our proposed architecture can do so. The existing standards proposals, in contrast, are incredibly constraining. Each only accommodates a single rather narrowly conceived style of event notification system.

Constructing the Dissemination Structure

A detailed discussion of how forwarding policies can be defined and translated to filter networks is beyond the scope of this article. We describe here just one simple, yet fairly expressive scheme.

Suppose that the forwarding policies in all scopes define forwarding trees on a per-topic basis and possibly also depending on the location at which the message locally originated. By saying that a message *locally originated from* a member X of scope Y, we mean that either the message was created by X (if X is itself a node) or a member of X, or that the message was created outside of Y, but X is (or contains) the first node in all of Y to which the message

Figure 15. An example hierarchy of scopes with cascading subscriptions

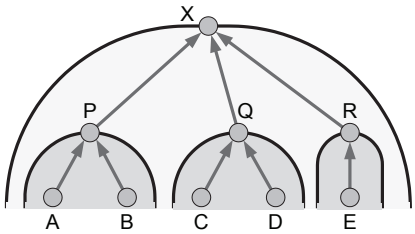
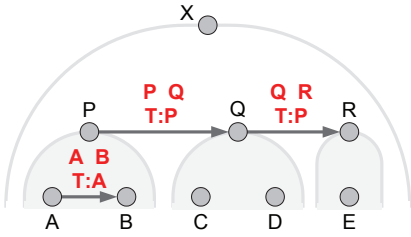


Figure 16. Channels created by the policies based on subscriptions



was forwarded. The technique described here implicitly assumes that messages do not arrive along redundant paths, that is, that the forwarding policy at each level is a tree, not a mesh. This scheme may be extended to cover the more general case with redundant paths and randomized policies, but we omit it here, for it would unnecessarily complicate our example. A comprehensive treatment of dissemination policies is beyond the scope of this article.

A scope manager thus maintains a graph, in which the scope members are linked by edges, labeled with constraints such as β:X, μ:Y, …, meaning that a message is forwarded along the edge if it was sent in topic β and locally originates at X, or if it was sent in topic μ and locally originates at Y, and so on. We shall now describe, using a simple example, how a structure of channels is established based on scope policies, and how filters are instantiated. We shall then explain how messages are routed.

Consider the structure of scopes depicted on Figure 15. Here A, B, C, D, and E are student laptops. P, Q, and R are three departments on a campus, with independent networks, hence they are separate scopes. X represents the entire campus. All students subscribe to topic T. Topic T is local to the campus, and X serves as its root. The scopes first register with each other. Then, the laptops send requests to subscribe for topics to P, Q, and R. Laptop A requests the publisher role, all others request to be subscribers. None of P, Q, or R are roots for the topic, hence they themselves subscribe for topic T with X, in a cascading manner. Now, all scopes involved create objects that represent local forwarding policies for topic T, and feed these objects with the "new member" events. The policy at P for messages in T originating at A creates a channel from A to B. Similarly, the policy at X for messages in T originating at P creates channels P to Q and Q to R (Figure 16). Each channel has a label of the form "**X-Y, T:Z**", meaning that it is a channel from X to Y, for messages in T originating at Z. Note that no channels have been created in scope Q. Until now, Q is not aware of any message sources because neither C nor D is a publisher, and because no other scope has so far requested a channel to Q, hence there is no need to forward anything. Channels are now delegated to individual nodes, as described in the section "Communication Channels." P delegates its channel to B, and Q delegates to D (Figure 17, delegated channels are in blue).

Figure 17. B and D contact Q and R to create channels. Q and R select C and E as entry points. Q now has a local message source and creates its own local channel, from C do D

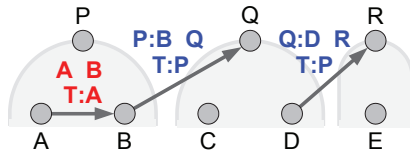
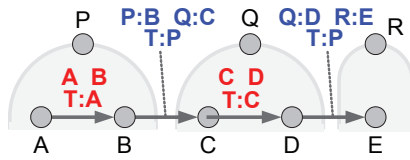


Figure 18. Channels are delegated

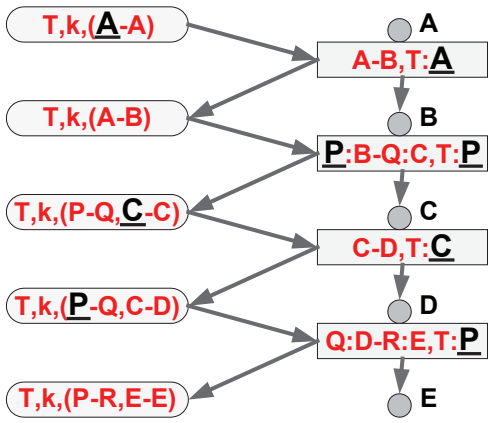


Channel labels are extended to reflect the fact that they have been delegated, for example, **P-Q** becomes **P:B-Q**, which means that P has delegated its source endpoint to its member B, and so forth. At this point, B and D contact the destinations Q and R and request the channels be created. Q and R select C and E as their *entry points* for these channels. Now Q also has a local source of messages (node C, the entry point), so now it also creates an instance of a forwarding policy, which determines that messages in T, locally originating at C, are forwarded to D (Figure 18). The entire forwarding structure is complete.

A message in transit is a tuple of the form (**T**, **k**, **r**), where **T** is the topic, **k** is the identifier that may include the source name, one or more sequence numbers, and so forth, and **r** is a *routing record*. The routing record is an object of the form $(\mathbf{X_K}–\mathbf{Y_K})$, $(\mathbf{X_{(K-1)}}–\mathbf{Y_{(K-1)}})$, …, $(\mathbf{X_1}–\mathbf{Y_1})$, in which every pair of elements $(\mathbf{X_i}, \mathbf{Y_i})$ represents the state of dissemination in one scope; $\mathbf{X_i}$ is the member of the scope that the message locally originated from and $\mathbf{Y_i}$ is the member of the scope that the message is moving inside of or that it is entering. Pair $(\mathbf{X_1}, \mathbf{Y_1})$ represents individual nodes, and for each **i**, scopes $\mathbf{X_i}$ and $\mathbf{Y_i}$ are a level below $\mathbf{X_{i+1}}$ and $\mathbf{Y_{i+1}}$, respectively, and $\mathbf{Y_i}$ is always a member of $\mathbf{Y_{i+1}}$. This list of entries does not need to "extend" all the way up to the root. If entries at a certain level are missing, they will be filled up when the message jumps across scopes, as it is explained below.

When a message arrives at a node, the node iterates over all of its outgoing channels, and matches channel filters against the routing record. Message **T**, **k**, $((\mathbf{X_K}–\mathbf{Y_K}), (\mathbf{X_{(K-1)}}–\mathbf{Y_{(K-1)}}), …, (\mathbf{X_1}–\mathbf{Y_1}))$ matches channel $(\mathbf{P_L}:\mathbf{P_{L-1}}:…:\mathbf{P_1} – \mathbf{Q_L}:\mathbf{Q_{L-1}}:…:\mathbf{Q_1}, \mathbf{T:R})$ when $(\mathbf{K} \geq \mathbf{L} \wedge \mathbf{X_L} = \mathbf{R} \wedge \mathbf{K} < \mathbf{L} \wedge \mathbf{P_L} = \mathbf{R})$ holds. This condition has two parts. If $\mathbf{K} \geq \mathbf{L}$, then in the scope in which the channel endpoints $\mathbf{P_L}$, $\mathbf{Q_L}$ and **R** are members, the message originated at $\mathbf{X_L}$ and is currently at $\mathbf{Y_L}$. According to our rules, the message should be forwarded iff $\mathbf{X_L} = \mathbf{R}$ (and $\mathbf{Y_L} = \mathbf{P_L}$, but this is always true). If $\mathbf{K} < \mathbf{L}$, then the routing record does not carry any state for the scope at which $\mathbf{P_L}$, $\mathbf{Q_L}$ and **R** are members. This means the message must have originated in this scope (the recovery record is filled up when the message is forwarded, as explained below), hence the condition $\mathbf{P_L} = \mathbf{R}$. Note that there might be several channels originating at the node; the message is forwarded across each one it matches. Now, when the message

Figure 19. The flow of messages (rounded rectangles, left), and the channels (square rectangles, right) in the scenario of Figure 17. Elements compared against each other are shown as black, bold, and underlined



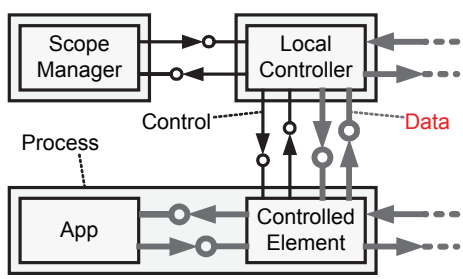
is forwarded, its routing record is modified as follows. First, the record is extended. If $\mathbf{K} < \mathbf{L}$, then for each **i** such that $\mathbf{i} > \mathbf{K}$ and $\mathbf{i} \leq \mathbf{L}$, we set $\mathbf{X_i} = \mathbf{P_i}$ and $\mathbf{Y_i} = \mathbf{P_i}$. Then, the details internal to the scope that the channel is leaving are replaced with the details internal to the scope the channel is entering, so for each **i** such that $\mathbf{i} < \mathbf{L}$, we set $\mathbf{X_i} = \mathbf{Q_i}$ and $\mathbf{Y_i} = \mathbf{Q_i}$. Finally, the current scope at the level at which the channel is defined is updated, we set $\mathbf{Y_L} = \mathbf{Q_L}$. The original value of $\mathbf{X_L}$, once set, is not changed until the message is forwarded out of the scope of which $\mathbf{X_L}$ is the member. Entries for $\mathbf{i} > \mathbf{L}$ also remain unchanged.

The flow of messages and matching in the example of Figure 18 is shown on Figure 19. When created on the publisher **A**, the message initially has a routing record **A**–**A**, since we know that it originated locally at **A** and that it is still at **A**. No entries are included at this point for routing in the scopes above **A**. There is no need for doing so since no routing has been done so far at those higher levels, so there is no state to maintain. Now the routing record is compared against channel **A-B, T:A**. The local origin **A** does match the channel's constraint **A**, so the message is forwarded along **A-B**, with a new record **A-B** to reflect the fact that it is now entering **B**. While matching this record to channel **P:B**–**Q:C** at **B**, we find that the record lacks routing information for the scope at which this channel is defined. This means that the message must have originated at the channel's source endpoint (which we know is **P** from the channel's description). We find that the channel's constraint **P** is the same as the channel's source endpoint **P**, so we again forward the message. While doing so, we extend its routing record with a new entry **P-Q** to record the fact that we made progress at this higher level. We also replace information local to P (the entry **A-B**) with new information local to **Q** (the new entry **C-C**). Forwarding at **C** and **D** works similarly to how it worked on **A** and **B**.

The Local Architecture of a Dissemination Scope

So far we focused in our description on peer-to-peer aspects of the dissemination framework, but we have not described how applications use this infrastructure, and how leaf scopes are internally organized. Of a number of possible architectures of the leaf scopes, the most general one involves the following components: *scope manager*, *local controller*, and *controlled element* (Figure 20). The *controlled element* is a very thin layer that resides in the application process. Its purpose is to serve as a hookup to the application process that allows for controlling the way the application communicates over the network. As such, it serves two principal purposes: (a) passing to the local controller requests to subscribe/unsubscribe, and (b) opening send and receive sockets in the process per request from the controller. The controlled element does not forward messages, nor does it include any other peer-to-peer functionality, and it is not a part of the management network. This simplicity allows it to be easily incorporated into legacy applications. It can be implemented in any way, for as long as it exposes a *control endpoint* (Figure 20, thin, black), a standardized Web service interface, and as long as it can

Figure 20. The architecture of a leaf scope in the most general scenario, with a local scope manager, a thin controlled element linked to the application, and with a local controller to handle peer-to-peer aspects, for example, forwarding

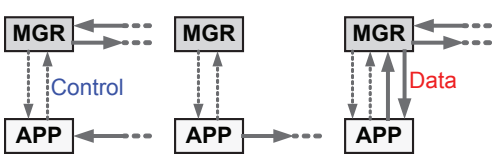


create *communication endpoints* (thick, red) to receive or connect to remote endpoints to send. The local controller implements all the peer-to-peer functionality such as forwarding, and so forth, and hosts such elements as channels or filters. It may also create communication endpoints, and may send data to or receive it from the controlled element. The controller is not a part of the management network, and it does not interact with any scope mangers besides the local one. These interactions are the job of the local SM, which, on the other hand, never sends or receives any messages itself.

In general, the three components can live in separate processes, or even physically on different machines, and may communicate over local sockets or over the network. However, in the typical scenario, the scope manager and local controllers are located in a single process ("manager"). The manager runs as a system service and may control multiple applications, either local or running on subordinate devices managed by the local node, through the control elements embedded in these applications. Within this scenario, we can distinguish three basic subscenarios, or three patterns of usage (Figure 21), depending on whether applications participate in sending or receiving directly, or only through the manager.

In the first scenario, the applications only receive data, directly from the network (Figure 21, left). When the manager is requested to create an incoming channel to the scope, it may either arrange for all applications to open the same socket to receive messages directly from the network (if the applications are all hosted one the same machine), or it may make them all subscribe to the same IP multicast address (if they are running on multiple subordinate devices), or it may have them create multiple endpoints, in which case the definition of the local channel endpoint will include a set of addresses rather than just one. The manager does not sit on the critical path; hence we avoid bottleneck and latency. If the local scope is required to forward data to other scopes, the manager also creates an endpoint (opens the same socket, or extends the receive channel endpoint definition to include its own address), to receive the data that needs to be forwarded.

Figure 21. Three example architectures with the scope manager and the local controller merged into a single local system "daemon"



In our example, the library's server might act as a leaf scope, and students' laptops might act as subordinate devices that do not host their own local scope managers and do not forward messages. Applications on the laptops would have the embedded controlled elements that communicate with the local controller on the library server via a standard Web interface. When students subscribe to a local topic, such as an online lecture transmitted by another department, the server chooses an IP multicast address and has all the laptops subscribe to it. The data arriving on the local network is received by all devices without any intermediary. If the library needs to forward messages, the server also subscribes to the IP multicast address and creates all the required channels and filters.

In the second scenario, the applications act as publishers (Figure 21, center). There is no need to forward data, hence the manager does not create any send or receive channels. In order to support this scenario, the controlled element must allow transmitting data to multiple IP addresses; embed various headers provided by the local controller, and so forth. There can be different "classes" of controlled elements, depending on what functionality they provide; this scenario might be feasible only for some such classes. This scenario avoids proxies, thus it could be useful, for example, in streaming systems.

In the third scenario, the applications communicate only with the local controller, which acts as a proxy (Figure 21, right). Unlike in the first two scenarios, this introduces

a bottleneck, but since the controlled elements do not need to support any external protocols or to be able to embed or strip any external headers, this scenario is always feasible. This scenario could also be used to interoperate with other eventing standards, such as WS-Eventing or WS-Notification. Here, the manager could act as the publisher or a proxy from the point of view of the application, and provide or consume messages in the format defined by all those other standards, while using our general infrastructure "under the hood." And similarly, for high-performance applications that reside on the server, an efficient implementation is possible using shared memory as a communication channel between the manager and the applications, and permitting applications to deserialize the received data directly from the manager's receive buffers, without requiring extra copy or marshaling across domains.

Sessions

We now shift focus to consider architectural implications of reliability protocols. Protocols that provide stronger reliability guarantees traditionally express them in terms of what we shall call *epochs*, corresponding to what in group communication systems are called *membership views*. In group communication systems, the lifetime of a topic (also referred to as a *group*) is divided into a sequence of *views*. Whenever the set of topic subscribers changes after a "subscribe" or an "unsubscribe" request, or a failure, a new view is created. In group communication systems, the corresponding event initiates a new epoch. Subscribers are notified of the beginnings or endings of epochs, and of the membership of the topic for each epoch. One then defines consistency in terms of which messages can be delivered to which subscribers and at what time relative to epoch boundaries. The set of subscribers during a given epoch is always fixed.

Whereas group communication views are often defined using fairly elaborate models (such as virtual synchrony, a model used in some of our past research, or consensus, the model used in Lamport's Paxos protocol suite), the architectural standard proposed here needs to be flexible enough to cover a range of reliability protocols, include many that have very weak notions of views. For example, simple protocols, such as SRM or RMTP, do not provide any guarantees of consistent membership views for topics.

In developing our architectural proposal, we found that even for protocols such as these two, in which properties are not defined in terms of epochs, epochs can still be a very useful, if not a universal, necessary concept. In a dynamic system, configuration changes, especially those resulting from crashes, usually require reconfiguration or cleanup, for example, to rebuild distributed structures, release resources, or cancel activities that are no longer necessary. Most simple protocols lack the notion of an epoch because they do not take such factors into account and do not support reconfiguration. Others do address some of these kinds of issues, but without treating them in a systematic manner. By reformulating such mechanisms in terms of epochs, we can standardize a whole family of behaviors, making it easier to talk about different protocols using common language, to compare protocols, and to design applications that can potentially run over any of a number of protocols, with the actual binding made on the basis of runtime information, policy, or other considerations.

Our design includes two epoch-like concepts: *sessions*, which are global to the entire topic, across the Internet, are shared by different frameworks (reliability, ordering, etc.), and which we discuss in this section, and *local views*, which are local to scopes, and which are discussed in the section on "Building the Hierarchy of Recovery Domains".

A *session* is a generalization of an epoch. In our system, sessions are used primarily as means of reconfiguring the topic to alter its security or reliability properties, or for other administrative changes that must be performed online, while the system is running.

The lifetime of any given topic is always divided into a sequence of sessions. However, session changes may be unrelated to member-

ship changes of the topic. Since introducing a new session involves a global reconfiguration, as described further, it is infeasible for an Internet-scale system to introduce a new session, and disseminate membership views, every time a node joins or leaves. Instead, such events are handled locally, in the scopes in which they occur, and without introducing a new, global, Internet-wide epoch.

Session numbers are assigned globally, for consistency. As explained before, for a given topic, a single global scope ("root") always exists such that all subscribers to that topic reside within its span. Although, as we shall explain, dissemination, reliability, and other frameworks may use different scope hierarchies, the root scope is always the same and all the dissemination, reliability, and other aspects of it are normally managed by a single scope manager. This top-level scope manager maintains the topic's metadata; it is also responsible for assigning and updating session numbers. Note that local topics, for example, internal to an organization, may be rooted locally, for example, in the headquarters, and managed by a local SM, much in a way local newsgroups are managed locally. Accordingly, for such topics, sessions are managed by a local server (internal to the organization).

To conclude, we explain how sessions impact the behavior of publishers and subscribers. After registering, a publisher waits for the SM to notify it of the session number to use for a particular topic. A publisher is also notified of changes to the session number for topics it registered with. All published messages are tagged with the most recent session number, so that whenever a new session is started for a topic, within a short period of time no further messages will be sent in the previous session. Old sessions eventually quiesce as receivers deliver messages and the system completes flushing, cleanup, and other reliability mechanisms used by the particular protocol. Similarly, after subscribing to a topic, a node does not process messages tagged as committed to session **k** until it is explicitly notified that it should receive messages in that session. Later, after session **k+1** starts, all subscribers are notified that session **k** is entering a *flushing* phase (this term originates in *virtual synchrony* protocols, but similar mechanisms are common in many reliable protocols; a protocol lacking a flush mechanism simply ignores such notifications). Eventually, subscribers report that they have completed flushing and a global decision is made to cease any activity and *cleanup* all resources pertaining to session **k**, thus completing the transition.

Incorporating Reliability, Ordering, and Security

As mentioned earlier, we rooted our design in the principle of *separation of concerns*, and we implement tasks such as reliability, ordering, security, or scope management independently from dissemination. In the section entitled "The Local Architecture of a Dissemination Scope," we explained how the management and the dissemination infrastructures interact in our system. The remaining frameworks, reliability, security, and ordering, are decomposed in a similar manner, and they also include three base components: (a) the *controlled element* that lives in the application processes and implements only base functionality, related to sending or receiving from the applications, but none of the peer-to-peer or management aspects, (b) the *local controller* that may live outside of the application process, and where all the peer-to-peer aspects are implemented, and (c) the *scope manager* that implements the interactions with other scope managers, but that is not involved in any activities related to the data flows, such as forwarding, calculating recovery state, managing encryption keys, assigning message order, and so forth.

In general, each of the *dissemination*, *reliability*, *security,* and *ordering* frameworks has a separate hierarchy of scopes and a separate network of scope managers. For example, reliability scopes isolate and encapsulate the local aspects related to reliability, such as loss recovery, and so forth, and hide their internal details from other scopes, just like dissemination scopes manage local dissemination and

hide the local aspects of message delivery. In some cases, the different scopes would overlap. This will be normally the case, for example, with the four "flavors" of scopes (ordering, security, reliability, and dissemination) local to a node. In such cases, a single local service would act as a scope manager for all the scopes of all four flavors. The same would typically be the case for the servers that control administrative domains, such as a departmental LAN, a wireless network in a cafeteria, a data center, a corporate network, and so forth. Scopes of all flavors would again overlap, and they would be managed by a single server.

Irrespective of whether components of the four different frameworks overlap, or are physically hosted on the same machine or in the same process, the frameworks always logically converge in the application (Figure 22). The complete local architecture includes a *multiplexer* (MUX), which serves as the entry point for messages from the application, and assigns messages to sessions, and a separate protocol stack for each session (rows on Figure 22). Elements of the per-session stacks are subcomponents owned by the four "controlled elements" (columns on Figure 22): *security* (SEC), *dissemination* (DISS), *reliability* (REL), and *ordering* (ORD), each of which exposes the standard Web interface required for interaction with its corresponding local controller. Now, when the application sends a message, it is first assigned to a session by the multiplexer, and assigned a local sequence number within the session. It is then passed to the appropriate per-session protocol stack, simultaneously to the subcomponents that handle security and ordering. Each of these two subcomponents processes the message independently and concurrently. The security component may encrypt and sign it, if necessary, and then pass it further to the dissemination component for transmission, and independently, to the reliability component to place it in a local cache for the purpose of retransmission, forwarding, and so forth, and to update the local structures to record the fact that the message was created. At the same time, the ordering component, also working in parallel with the dissemination and reliability components, records the presence of the message in its own structures, which are used later by the ordering infrastructure to generate *ordering requests*, to be submitted to the orderer (for details, see the section entitled "Ordering").

On the receive path, the process would look similar (Figure 23). Messages may arrive either through the dissemination framework, in the normal case, or via the reliability framework if they were initially lost, and have been later recovered. Messages that arrive from the dissemination framework are routed via the reliability subcomponent so that they are registered, can be cached, or so that delivery can be suppressed. When ordering arrives from the ordering framework, messages can be decrypted, placed in a buffer (BUF), and delivered in the appropriate order to the application.

The exact manner in which the subtasks performed by the four subcomponents are syn-

Figure 22. Internal architecture of the application process

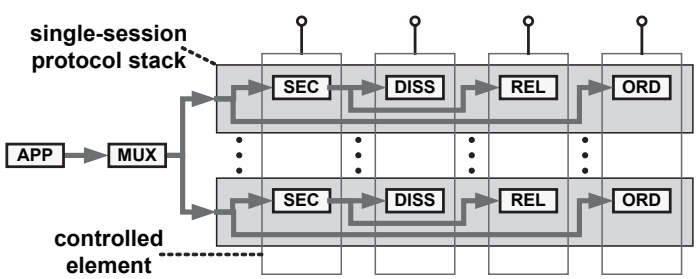
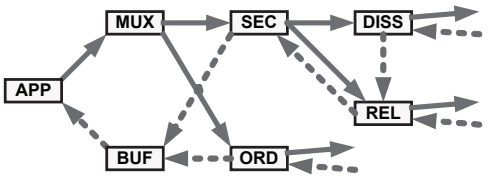


Figure 23. Processing messages on the send path (red and solid lines) and on the receive path (blue and dotted lines)



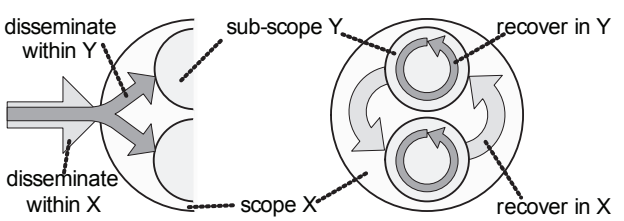
chronized may vary. On the send path, messages may not be transmitted until the entire protocol stack for a given session can be assembled, that is, if the dissemination framework learned of a new session, but the reliability framework has not, the transmission might be postponed until the information about the new session propagates across the reliability framework as well, to avoid problems stemming from such misalignments. On the receive path, the decryption of the message might be postponed until the ordering is known, to avoid maintaining two copies of the message in memory, one for the purpose of loss recovery (encrypted) and one for delivering to the application (decrypted), and so forth.

Hierarchical Approach to Reliability

Our approach to reliability resembles our hierarchical approach to dissemination. Just as channels are decomposed into subchannels, in the reliability framework we decompose the task of loss repair and providing other reliability goals. Recovering messages in a given scope is modeled as recovering within each of its subscopes, concurrently and independently, then recovering across all the subscopes (Figure 24). For example, suppose that scope **X** has members $\mathbf{Y_1}$, $\mathbf{Y_2}$, …, $\mathbf{Y_K}$. A simple reliability property **P**(**X**) requiring that "if some node **x** in the span of scope **X** receives a message **m**, then for as long as either **x** or some other node keeps a copy of it, every other node **y** in the span of **X** will also eventually get **m,**" can be decomposed as follows. First, we ensure $\mathbf{P}(\mathbf{Y_i})$ for every $\mathbf{Y_i}$, that is, we ensure that in each subscope $\mathbf{Y_i}$ of scope **X**, if one node has the message, then so eventually do the others. The protocols that lead to this goal can run in all these subscopes independently and concurrently. Then, we run a protocol across subscopes $\mathbf{Y_1}$, $\mathbf{Y_2}$, …, $\mathbf{Y_K}$, to ensure that if any of them has in its span a node **x** that received message **m**, then each of the other $\mathbf{Y_i}$ also eventually has a node that received **m**. When these tasks, that is, recovery in each $\mathbf{Y_i}$ plus the extra recovery across all $\mathbf{Y_i}$, are all performed for sufficiently long, **P**(**X**) is eventually established.

Coming back to our example, assume that students with their laptops sit in university departments, each of which is a scope. Suppose that some, but not all of the students received a message with a homework problem set from their professor sitting in a cafeteria. We would like to ensure that the problem set gets reliably delivered to all students. In our architecture, this would be achieved by a combination of protocols: a protocol running in each department would ensure that internally, for every

Figure 24. The similarities between a hierarchical dissemination (left) and hierarchical recovery (right)



pair **x**, **y** of students, if **x** got the message then so eventually does **y**, and likewise a protocol running across the departments ensures that for each pair of departments **x**, **y**, if some students in **x** got the message, so eventually do some students in **y**. In the end, this yields the desired outcome.

Just like recovery among individual nodes, recovery among LANs might also involve comparing their "state" (such as aggregated ACK/NAK information for the entire LAN) or forwarding lost messages between them. We give an example of this in the section on "Recovery Agents." As mentioned earlier, in our architecture, different recovery schemes may be used in different scopes, to reflect differences in the network topologies, node or communication link capacities, the availability of IP multicast and other local infrastructure, the way subscribers are distributed (e.g., clustered or scattered) and so forth.

For example, in one department, the machines of the students subscribed to topic **T** could form a spanning tree. The property we mentioned above could be guaranteed by making neighbors in the tree compare their state, and upon discovering that one of them has a message **m** that the other is missing, forwarding **m** between the two of them. The same approach may also be used across the departments, that is, departments would form a tree, the departments "neighboring" on that tree could compare what their students got, and perhaps arrange for messages to be forwarded between them. For the latter to be possible, the departments need a way to calculate "what their students got," which is an example of an aggregated, "department-wide" state. Finally, some departments could use a different approach. For example, a department enamored of gossip protocols might require that student machines randomly gossip about messages they got; a department that has had bad experiences with IP multicast and with gossip might favor a reliability protocol that runs on a token ring instead of a tree, and a department with a site-license for a protocol such as SRM (which runs on IP multicast) might favors its use, where the option is available. In each department, a different protocol could be used locally. As long as each protocol produces the desired outcome (satisfies the reliability property inside of the department), and as long as the department has a way to calculate aggregate "department-wide" state needed for inter-department recovery, these very different policies can be simultaneously accommodated.

Just as messages are disseminated through channels, forming what might be termed *dissemination domains*, reliability is achieved via *recovery domains*. A recovery domain **D** in scope **X** may be thought of as a "distributed recovery protocol running among some nodes within **X** that performs recovery-related tasks for a certain set of topics."

For example, when some of the students sitting in a library subscribe to topic **T**, the library might create a "local recovery domain for topic **T.**" This domain could be "realized," for example, as a spanning tree connecting the laptops of the subscribed students and running a recovery protocol between them. The library could internally create many domains, for example, many such trees of student's laptops.

The concept of a recovery domain is dual to the notion of a channel; here we present the analogy:

- Just like a channel is created to disseminate messages for some topics $\mathbf{T}_1$, $\mathbf{T}_2$, …. $\mathbf{T}_k$ in scope **X**, a recovery domain is created to handle loss recovery and other reliability tasks, again for a specific set of topics, and in a specific scope. Just like there could exist multiple channels to a scope, for example, for different sets of topics, there could also exist multiple recovery domains within a single reliability scope, each ensuring reliability for different sets of topics.
- Just as channels may be composed of subchannels, a recovery domain **D** defined at a scope **X** may be composed of *subdomains* $\mathbf{D}_1$, $\mathbf{D}_2$, …. $\mathbf{D}_n$ defined at subscopes of **X** (we will call them the *members* of **D**). Each such subdomain $\mathbf{D}_i$ handles recovery for a set of subscribers in the respective

subscope, while **D** handles recovery across the subdomains. The hierarchy of recovery domains reflects the hierarchy of scopes that have created them, just as channels are decomposed in ways that reflect the hierarchy of scopes that have exposed those channels.

- Just as channels are composed of subchannels via applying filters assigned by forwarding policies, a recovery domain **D** performs its recovery tasks using a *recovery protocol*. Such a protocol, assigned to **D**, specifies how to combine recovery mechanisms in the subdomains of **D** into a mechanism for all of **D**. Recovery protocols are defined in terms of how the subdomains "interact" with each other. We explain how this is done in more detail in the section entitled "Hierarchical Approach to Reliability."
- Just like a single channel may be used to disseminate messages in multiple topics, a recovery domain may run a single protocol to perform recovery simultaneously for a set of topics. In both cases, reusing a single mechanism (a channel, a token ring, a tree, etc.) may significantly improve performance due to the reduction in the total number of control messages and other such optimizations. Indeed, we implemented and evaluated this idea in QSM (Ostrowski & Birman, 2006b, 2006c).

Each individual node is a recovery domain on its own. On the other hand, in a distributed scope such as a LAN, the library in our example, many cases are possible. In one extreme, a single domain may cover the entire LAN. All internal nodes could thus form a token ring, or gossip randomly to exchange ACKs for messages in all topics simultaneously, and use this to arrange for local repairs. In the other extreme, separate domains could be created for every individual topic; subscribers to the different topics could thus form separate structures, such as separate rings and trees, and run separate protocol instances in each of them, exchanging state and the lost messages. In our system, recovery domains actually handle recovery for specific *sessions*, not just specific topics. Each of the recovery domains created internally by a scope performs recovery for some set of sessions, and these sets are such that for each session in which this scope has subscribers, there is a recovery domain in this scope that performs recovery for this session.

A recovery domain **D** of a data center could have as its members recovery domains created by the LANs in that data center (by the SMs of these LANs). Note that in this case, members of **D** would themselves be distributed domains, that is, sets of nodes. A recovery protocol running in **D** would specify how all these different sets of nodes should exchange state and forward lost messages to one another. Note the similarity to a forwarding policy in a data center, which would also specify how messages are forwarded among sets of nodes. As explained in the section on "Recovery Agents" and the section on "Implementing Recovery Domains with Agents," recovery protocols are implemented through delegation, just like forwarding. A concept of a *recovery protocol* is, to some extent, dual, symmetric to the notion of a *forwarding policy*.

Building the Hierarchy of Recovery Domains

Before we show how the hierarchical recovery scheme can be implemented, we need to explain how domains created at different scopes are related to each other. As explained in the preceding section, domains are organized by the relation of membership: domains in superscopes can be thought of as containing domains in subscopes as members. Just as was the case for scopes, a given domain can have many parents, and there may be multiple global domains, but for a given topic, all domains involved in recovery for that topic always form a tree. Domains know their *members* (subdomains) and *owners* (superdomains), and through a mechanism described below, also their *peers* (other domains that have the same parent). This knowledge of *membership* allows the scopes that create those

domains to establish distributed structures in an efficient way.

A consistent view of membership is the basis for many reliable protocols, and could benefit many others that don't assume it. Knowing the members of a topic helps to determine which nodes have crashed or disconnected. In existing group communication systems, this is usually achieved by a Global Membership Service (GMS) that monitors failures and membership changes for all nodes, decides when to "install" new membership views for topics, and notifies the affected members of these new views, including the lists of topic members. Nodes then use those membership views to determine, for example, what other nodes should be their neighbors in a tree, who should act as a leader, and so forth.

In our framework, the manager of the root scope for a given topic is responsible for creating the top-level recovery domain, announcing when sessions for that topic begin or end, and so forth. However, if the root SM, which in case of an Internet-wide scope would "manage" the entire Internet, had to process all subscriptions, and respond to every failure across the Internet, it would lead to a non-scalable design: beyond a certain point the system would be constantly in the state of reconfiguration, trying to change membership or install new sessions, and hence unable to make useful progress. It would also violate the principle of isolation: the higher-level scopes would process information that should be local, for example, a corporate network would have to know which nodes in data centers are subscribers, whereas according to our architectural principles, the administrator of a corporate network, and the policies defined at this level, should treat the entire data centers as black boxes.

To avoid the problem just mentioned, rather than collecting all information about membership in a topic **T** and processing it centrally, we distribute this information across all scope managers in the hierarchy of scopes for topic **T** (recall this hierarchy defined in the section entitled "The Hierarchy of Scopes"). Each SM thus has only a partial membership view for each topic and session. This scheme is outlined below.

In the reliability framework, if a scope **X** subscribes to a topic **T**, it first selects or creates a local recovery domain **D** that will handle the recovery for topic **T** locally in **X**, and then sends a request to one of its parent scopes, some **Y**, asking to subscribe this specific domain, to topic **T**. At this point, it is not significant which of its parent scopes **X** directs the request to. **X** may be manually setup by an administrator with the list of parent scopes, and to send requests in all topics that have names matching a certain pattern to a given parent, or it could use an automated, or a semi-automated scheme to discover the parent scopes that it should subscribe with. Exactly how such a discovery scheme can be most efficiently constructed is beyond the scope of this article, but we do hope to explore the issue in a future work.

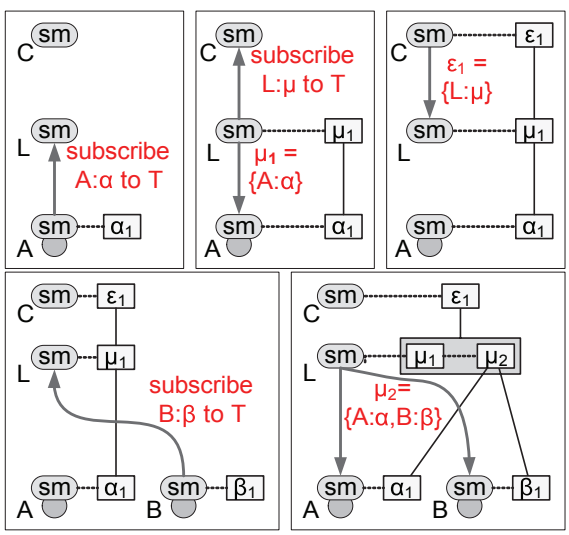
The superscope **Y** processes the **X**'s subscription request jointly with requests from other of its subscopes, for example, batching them together for efficiency. It then either joins **X** and other subscopes to an existing recovery domain or creates a new one, some **D'.** When joining an existing recovery domain, **Y** follows a special protocol, some details of which are given in the section entitled "Reconfiguration". In any case, scope **Y** informs all scopes of the new membership of domain **D'**. So for each recovery domain **D"** that is a member of **D',** the subscope of **Y** that owns this domain will be notified by **Y** that domain **D'** changed membership, and will be given the list of all subdomains of **D'** together with the names of the scopes that own those subdomains. Finally, if domain **D'** has just been created, and scope **Y** is not the root scope for the topic, **Y** itself sends a request to one of its parent scopes, asking to subscribe domain **D'** to the topic. Topic subscriptions thus travel all the way to the root scope for the topic, in a cascading manner, creating a tree of recovery domains in a bottom-up fashion.

In our example, when a student **A** sitting in a library **L** enters a virtual room **T**, **A**'s laptop creates a local recovery domain $\mathbf{D}_{\mathbf{A}}$, and sends a request to the library server. Suppose that there

are other students in the library that are already subscribed to **T**, so the library server already has a recovery domain $\mathbf{D}_L$ that performs recovery in topic **T**. In this case all that is left to do for the library server is to update the membership of $\mathbf{D}_L$, and to inform student **A**'s laptop, as well as the laptops of the other students subscribed to **T**, of the new membership of $\mathbf{D}_L$, so that the laptops can update the distributed recovery structure, and build a new distributed structure. For example, if laptops used a spanning tree, or a token ring, to compare the sets of messages they have received and exchange messages between neighbors, the spanning tree or the ring may be updated to include the new laptop. On the other hand, suppose that student **A** is the first in the library to subscribe to **T**. In this case, the library server creates a new recovery domain $\mathbf{D}_L$, with $\mathbf{D}_A$ as its only member, and sends its own request, in a cascading manner, to a campus server **C**, asking to subscribe $\mathbf{D}_L$ to topic **T**, and so on.

The above procedure effectively constructs a hierarchy of subdomains, with the property that for each topic **T**, the recovery domains subscribed to **T** form a tree. At the same time, a membership hierarchy is built in a distributed manner. Specifically, for each domain $\boldsymbol{\mu}$ in some scope **S**, **S** will maintain a list of the form $\{ \mathbf{X}_1{:}\boldsymbol{\beta}_1, \mathbf{X}_2{:}\boldsymbol{\beta}_2, \ldots, \mathbf{X}_K{:}\boldsymbol{\beta}_K \}$, in which $\boldsymbol{\beta}_1, \boldsymbol{\beta}_2, \ldots, \boldsymbol{\beta}_K$ are the *members*, that is, subdomains of domain $\boldsymbol{\mu}$, and $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_K$ are the names of the scopes that own those subdomains (i.e., that created them). In the process of establishing this structure, each of the scopes $\mathbf{X}_i$ receives the list, along with any future updates to it. This information is not "pushed" all the way down to the leaf nodes. Instead, every scope maintains the membership of the domains it created (so, for example, scope **S** maintains the list mentioned above), plus a copy of the membership of all superdomains of the domains it created (so, for example, each $\mathbf{X}_i$ has a copy of the list above), but not the membership of any domains created below it (so, for example, **S** would not track the membership of any of the domains $\boldsymbol{\beta}_1, \boldsymbol{\beta}_2, \ldots, \boldsymbol{\beta}_K$), more than one level above it (so, for example, while scope **S** would know what are the peers of its own domain $\boldsymbol{\mu}$, scopes $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_K$ would not know those

Figure 25. Node A subscribes to topic T with the library L. Library L subscribes with campus C. Membership information and view numbers are passed one level down (never up) the hierarchy.



peers, because this information is, logically, two levels "above"), or any internal details of its peers (so, for example, while all of $\mathbf{X_1}$, $\mathbf{X_2}$, …, $\mathbf{X_K}$ would know the membership of $\boldsymbol{\mu}$, that is, the entire list { $\mathbf{X_1}$:$\boldsymbol{\beta}_1$, $\mathbf{X_2}$:$\boldsymbol{\beta}_2$, …, $\mathbf{X_K}$:$\boldsymbol{\beta}_K$ }, they would not know the membership of any of the domains $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, …, $\boldsymbol{\beta}_K$ besides their own; they only know the names of their peer domains).

Figure 25 shows an example of the structure the system would construct in a scenario similar to the one above. Laptop **A** creates a new domain $\boldsymbol{\alpha}$, which may have a version number, like any other domain, say $\boldsymbol{\alpha}_1$. Then, **A** directs a request subscribe **A**:$\boldsymbol{\alpha}$ to **T** to the library **L**. Note that the version number of $\boldsymbol{\alpha}$ was not included in the request to **L**. This is a detail internal to **A** that **L** does not need to know about. Now, the library creates a new domain $\boldsymbol{\mu}$, and gives it a version number, say $\boldsymbol{\mu}_1$, and directs subscribe **L**:$\boldsymbol{\mu}$ to **T** to the campus **C** (omitting the version number). Concurrently, **L** notifies **A** that **A:**$\boldsymbol{\alpha}$ is now a member of $\boldsymbol{\mu}_1$. This means that a domain $\boldsymbol{\mu}$ has been created at **L**, and version (view) number **1** of $\boldsymbol{\mu}$ has just a single member **A:**$\boldsymbol{\alpha}$. Similarly, **C** creates a new domain $\boldsymbol{\varepsilon}$ with initial version $\boldsymbol{\varepsilon}_1$ that includes a single member **L:**$\boldsymbol{\mu}$ and notifies **L**. Later, another laptop **B** in the library **L** also joins topic **T**. This time, no request is sent to campus **C**. The library handles the request internally. A new version (view) $\boldsymbol{\mu}_2$ of domain $\boldsymbol{\mu}$ is created with two members **A:**$\boldsymbol{\alpha}$ and **B:**$\boldsymbol{\beta}$, and both **A** and **B** are notified of this new view. **A** and **B** undergo a special protocol to "transition" from recovery domain $\boldsymbol{\mu}_1$ to recovery domain $\boldsymbol{\mu}_2$ in a reliable manner, and the protocol running for $\boldsymbol{\mu}_1$ eventually quiesces. The protocols that run at higher levels are unaffected. Domain $\boldsymbol{\varepsilon}_1$ still has only a single member **L:**$\boldsymbol{\mu}$, and the view change that occurred internally in domain $\boldsymbol{\mu}$ is transparent to **C**, and to the protocols that run at this level, and handled internally in the library **L**, between nodes **A** and **B**.

By keeping the information about the hierarchy of domains distributed, and by limiting the way in which this information is propagated to only one level below, we remain faithful to the principles of isolation and local autonomy laid out earlier. At the same time, this enables significant scalability and performance benefits. Because parent domains are oblivious to the membership of their subdomains, and reconfiguration can often be handled internally, as in the example above, churn and failures in lower layers of the hierarchy do not translate to churn and failures in higher layers. A failure of a node or a mobile user with a laptop joining or leaving the system does not need to cause the Internet-wide structure of recovery domains, potentially spanning across tens of thousands of nodes, to fluctuate and reconfigure.

As stated earlier, a single recovery domain may perform recovery for multiple topics (sessions), simultaneously. Additionally, recall that the domain hierarchy, with multiple topics, may not be a tree. We now present an example of how and why this could be the case. Suppose that nodes in a certain scope **L** are clustered based on their interest. Nodes **A** and **B** would be in the same cluster if **A** and **B** subscribed to the same topics. Clusters are thus defined by sets of topic names, for example, cluster $\mathbf{R_{XY}}$ would include nodes that have subscribed to topics **X** and **Y**, and that have not subscribed to any other topics besides these two. The set of nodes subscribed to each topic would thus include nodes in a certain set of clusters. We might even think of as topics "including" clusters, and the clusters including the individual nodes (Figure 26). Accordingly, a scope manager in **L** might create a separate recovery domain for each cluster, and then a separate recovery domain for each topic. If node **A** subscribes to topic **X**, and nodes **B** and **C** subscribe to topic **Y**, then **L** could create a recovery domain $\mathbf{R_X}$ for cluster $\mathbf{R_X}$ and a recovery domain $\mathbf{R_{XY}}$ for cluster $\mathbf{R_{XY}}$. Domain $\mathbf{R_X}$ would have a single member **A:**$\boldsymbol{\alpha}$, while $\mathbf{R_{XY}}$ would have two members, **B:**$\boldsymbol{\beta}$ and **C:**$\boldsymbol{\varphi}$. Scope **L** would also create a local domain **L:X** for topic **X** and **L:Y** for topic **Y**. Domain **L:X** would have members **L:**$\mathbf{R_X}$ and **L:**$\mathbf{R_{XY}}$ while domain **L:Y** would have a single member **L:**$\mathbf{R_{XY}}$. Domains **L:X** and **L:Y** defined at scope **L** could themselves be members of some higher-level domains, defined at a higher-level scope **C**, and so on. Now, the protocol running in

domain $\mathbf{R_{XY}}$ at scope **L**, for example, would perform recovery simultaneously for topics **X** and **Y**. As said earlier, the protocol running in $\mathbf{R_{XY}}$ would also be used to calculate aggregate information about domain $\mathbf{R_{XY}}$, to be used in the higher-level protocols. In our example, the information collected by the protocol running in $\mathbf{L{:}R_{XY}}$ would be used by two such protocols, a protocol running in domain **L:X** and a protocol running in **L:Y**.

While the structure just described may seem complex, the ability to perform recovery in multiple topics simultaneously is important in systems like our virtual worlds, where the number of topics (virtual rooms) may be very large. QSM, mentioned previously, uses the architecture just presented, and is able to scale to thousands of publish-subscribe topics.

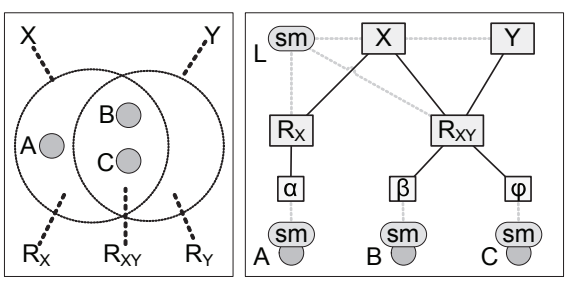
To complete the discussion of recovery hierarchy, we now turn to sessions. As explained earlier, in our architecture recovery is always performed in the context of individual sessions, not topics, because whenever a session changes, so can the reliability properties of the topic. The creation of the domain hierarchy, outlined above, is mostly independent of the creation of sessions. The only case when these two processes are synchronized arises when the last member, across the entire Internet, leaves the topic, or when the first member rejoins the topic after a period when no members existed, for in such cases, it is impossible to handle the event via a local reconfiguration between members (such as transferring the state from some existing member to the newly joining member, or "flushing" any changes from the departing member to some of the existing members). Such an event will force an existing session to be flushed or a new session to be created.

In any case, sessions for a topic **T** are created by the scope that serves as the root for **T**. The root maintains topic metadata and the information about sessions in persistent storage. It assigns new session number whenever a new session is created, and then *installs* the new session and *flushes* the existing session in the top-level recovery domain that it created to perform recovery in topic **T**. More on how the *installing* and *flushing* of a session are realized will be explained in the section "Implementing Recovery Domains with Agents." Now, concurrently with installing of a new session and flushing of the old session in the top-level recovery domain, the scope manager passes the session change event further, to the subdomains that are members of this global recovery domain, by communicating with the scope managers that created those subdomains. This notification travels down the hierarchy of recovery domains in a cascading manner, until the session change events are disseminated across the entire structure.

Recovery Agents

The reader will have noticed by now that the structure of recovery domains we've just described "exists" only virtually, inside the scope managers. These recovery domains are

Figure 26. A hierarchy of recovery domains in a system that clusters nodes based on interest



"implemented" by physical protocols running directly between physical nodes, the publishers and subscribers, in a manner similar to how we implemented channels between scopes, by *delegating* the tasks that the recovery domains are responsible for to physical nodes. Just as channels between scopes are "implemented" by physical connections between nodes that can be constrained with filter chains and that are "installed" in the physical nodes by their superscopes, in the reliability framework recovery domains are "implemented" by *agents*. Similarly to filters, these agents are also small, "downloadable" components, which are installed on physical nodes by their superscopes. Before going into details of how precisely this is done, however, we first explain how existing recovery protocols can be modeled in a hierarchical manner that is compatible with our architecture.

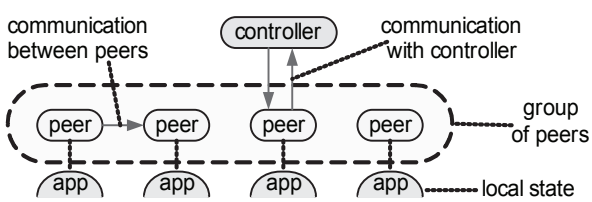
Modeling Recovery Protocols

The *reliability framework* is based on an abstract model of a scalable distributed protocol dealing with loss recovery and other reliability properties. In this model, a protocol such as SRM, RMTP, virtual synchrony, or atomic commit, is defined in terms of a group of cooperating *peers* that exchange control messages and can forward lost packets to each other, and that may perhaps interact with a distinguished node, such as a sender or some node higher in a hierarchy, which we will refer to as a *controller* (Figure 27). The controller does not have to be a separate node; this function could be served by one of the peers. The distinction between the peers and the controller may be purely functional. The point is that the group of peers, as a whole, may be asked to perform a certain action, or calculate a value, for some higher-level entity, such as a sender, a higher-level protocol, or a layer in a hierarchical structure. Examples of such actions include retransmitting or requesting a retransmission for all peers, reporting which messages were successfully delivered to all peers, which messages have been missed by all peers, and so forth. Irrespectively of how exactly the interaction with the controller is realized, it is present in this form or another in almost every protocol run by a set of receivers. We shall refer to the possible interactions between the peers and the controller as the *upper interface*. Notice that some reliability protocols aren't traditionally thought of as hierarchical; we would view them as supporting only a one-level hierarchy. The benefit of doing so is that those protocols can then be treated side by side with protocols such as SRM and RMTP, in which hierarchy plays a central role.

Each peer inspects and controls its *local state*. Such state could include, for example, a list of messages received, and perhaps copies of those that are cached (for loss recovery), the list and the order of messages delivered, and so forth. Operations that a peer may issue to change the local state could include retrieving or purging messages from cache, marking messages as deliverable, delivering some of the previously missed message to the application, and so forth. We refer to such operations, used to view or control the local state of a peer, as a *bottom interface*.

In protocols offering strong guarantees, peers are typically given the membership of

Figure 27. A group of peers in a reliable protocol



their group, received as a part of the initialization process, and subsequently updated via *membership change* events. Peers send control messages to each other to share state or to request actions, such as forwarding messages. Sometimes, as in SRM, a multicast channel to the entire peer group exists.

To summarize, in most reliable protocols, a peer could be modeled as a component that runs in a simple environment that provides the following interface: a *membership view* of its peer group, *channels* to all other peers, and sometimes to the entire group, a *bottom interface* to inspect or control local state, and an *upper interface*, to interact with the sender or the higher levels in the hierarchy concerning the aggregate state of the peer group (Figure 28). In some protocols, certain parts of this interface might be unavailable, for example, in SRM peers might not know other peers. The bottom and upper interfaces also would vary.

This model is flexible enough to capture the key ideas and features of a wide class of protocols, including virtual synchrony. However, because in our framework protocols must be reusable in different scopes, they may need to be expressed in a slightly different way, as explained below.

In RMTP, the sender and the receivers for a topic form a tree. Within this tree, every subset of nodes consisting of a parent and its child nodes represents a separate local recovery group. The child nodes in every such group send their local ACK/NAK information to the parent node, which arranges for a local recovery within the recovery group. The parent itself is either a child node in another recovery group, or it is a sender, at the root of the tree. Packet losses in this scheme are recovered on a hop-by-hop basis, either top-down or bottom-up, one level at a time. This scheme distributes the burden of processing the individual ACKs/NAKs, and of retransmissions, which is normally the responsibility of the sender. This improves scalability and prevents ACK implosion.

There are two ways to express RMTP in our model. One approach is to view each recovery group consisting of a parent node and its child nodes as a separate group of peers (Figure 29). Since internal nodes in the RMTP tree simultaneously play two roles, a "parent" node in one recovery group and a "child" node in another, we could think of each node as running two "agents," each representing a different "half" of the node, and serving as a peer in a separate peer group. In this perspective it would be not the nodes, but their "halves" that would represent peers. Every group of peers, in this perspective, would include the "bottom agent" of the parent node, and the "upper agents" of its child nodes. When a node sends messages to its child nodes as a result of receiving a message from its parent, of vice versa, we may think of those two "agents" as interacting with each other through a certain interface that one of them views as upper, and the other as bottom. These two types of agents play different roles in the protocol, as explained below.

The bottom agent of each node interacts via its *bottom interface* with the local state of

Figure 28. A peer modeled as a component living in abstract environment (events, interfaces, and so forth)

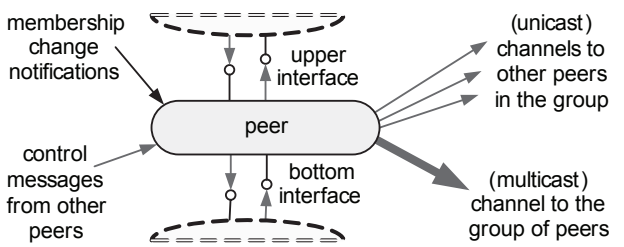
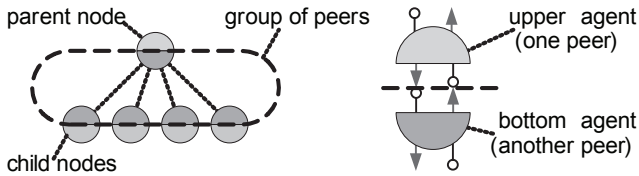


Figure 29. RMTP expressed in our model. A node hosts "agents" playing different roles

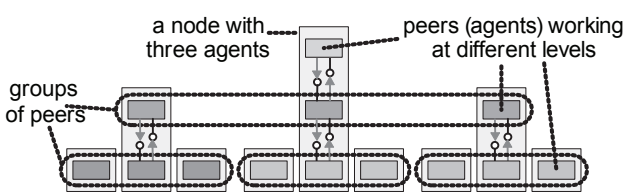


the node. It also serves as a distinguished peer in the peer group, composed of itself and the upper agents of the child nodes. A protocol running in this peer group is used to exchange ACKs between child nodes and the parent node and arrange for message forwarding between peers, but also to calculate collective ACKs for the peer group, that is, which messages were not recoverable in the group. This is communicated by the bottom agent, via its *upper interface*, to the upper agent. The upper agent of every node interacts via its *bottom interface* with the bottom agent. What the upper agent considers as its "local state" is not the local state of the node. Instead, it is the state of the entire recovery group, including the parent and child nodes, that is collected for the upper agent by the bottom agent though the protocol that the bottom agent runs with the upper agents in child nodes. Such interactions, between a component that is logically a part of a "higher layer" ("upper agent") with components that reside in a "lower layer" ("bottom agent"), both components co-located on the same physical node, and connected via their upper and bottom interfaces, are the key element in our architecture.

At the top of this hierarchy is the sender, the root of the tree. The bottom agent of the sender node collects for the upper agent the state of the top-level recovery group, which subsumes the state of the entire tree, and passes it to the upper agent through its upper interface. The upper agent of the sender can thus be thought of as "controlling" through its *bottom interface* the entire receiver tree.

The second way to model RMTP, which builds on the concepts we just introduced, captures the very essence of our approach to combining protocols. It is similar to the first model, but instead of the "upper" and "bottom" agents, each node can now host multiple agents, again connected to each other through their "bottom" and "upper" interfaces. Each of these agents works at a different level. We may think of every node as hosting a "stack" of interconnected agents (Figure 30). In this structure, the sender would not be the root of the hierarchy any more. Rather, it would be treated in the very same way as any of the receivers. The same structure could feature multiple senders, and a single recovery protocol would run for all of them simultaneously.

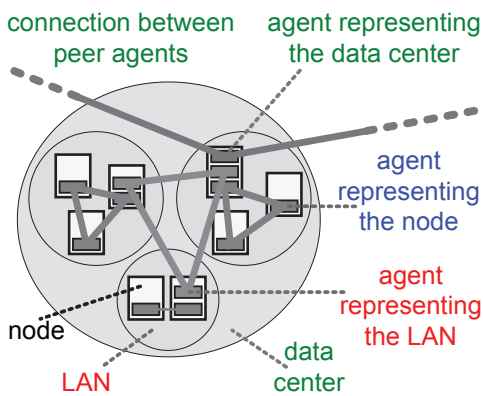
Figure 30. Another way to express RMTP. Each node hosts multiple "agents" that act as peers at different levels of the RMTP hierarchy



Focusing for a moment on a concrete example, assume that nodes reside in three LANs, which are part of a single data center (Figure 30). Each of these administrative domains is a scope. Each node hosts, in its "agent stack," a "node agent" (bottom level, green, Figure 30). The "local state" of the node agent, accessed by the node agent through its bottom interface, is the state of the node, such as the messages that the node received or missed, and so forth. Now, in each LAN, the node agents of all nodes in that LAN form a peer group and communicate with each other to compare their local state, or to arrange for forwarding messages between them. One of these node agents in each LAN serves as a leader (or "parent"), and the others serve as subordinates (or "children"). The leader collects the aggregate ACK/NAK information about the LAN from the entire peer group. The node that hosts the leader also runs anther, higher-level component that we shall call a "LAN agent" (middle level, orange, Figure 30). The LAN agent accesses, through its bottom interface, the aggregated state of the LAN that the "leader" node agent, co-located with it on the same "leader" node, calculated. The LAN agent can therefore be thought of as controlling, through its bottom interface, the entire LAN, just like a node agent was controlling the local node. Now, all the LAN agents in the data center again form a peer group, compare their state (which are aggregate states of their LANs), arrange for forwarding (between the LANs), and calculate aggregate ACK/NAK information about the data center. Finally, one of the nodes that host the LAN agents hosts an even higher-level component, a "data center agent" (top level, blue, Figure 30). The aggregate state of the data center, collected by the peer group of LAN agents, is communicated through the upper interface of the "leader" LAN agent to the data center agent; the latter can now be thought as controlling, through its bottom interface, the entire data center. In a larger system, the hierarchy could be deeper, and the scheme could continue recursively.

Note the symmetry between the different categories of agents. In essence, for every entity, be it a single node, a LAN, or a data center, there exists exactly one agent that collects, via lower-level agents, the state of the entity it represents, and that acts on behalf of this entity in a protocol that runs in its peer group. The agent that represents a distributed scope is always hosted together with one of the agents that represent subscopes. By now, the reader should appreciate that this structure corresponds to the hierarchy of recovery domains we intro-

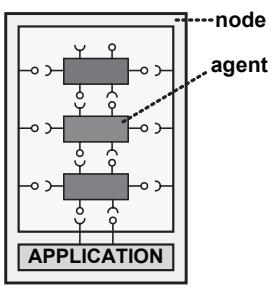
Figure 31. A node as a "container" for agents



duced in the section "Building the Hierarchy of Recovery Domains." In our design, every recovery domain is represented, as a part of some higher-level domain, by an agent that collects the state of the domain, which it represents, and acts on behalf of it in a protocol that runs among the agents representing other recovery domains that have the same parent (Figure 31). In order to be able to do their job, these agents are updated whenever a relevant event occurs. For example, they receive a membership change notification, with the list of their peer agents, when a new recovery domain is created. To this end, each agent maintains a bi-directional channel from the scope that created the recovery domain represented by this agent, down to the node that hosts the agent, along what we call an agent "delegation chain."

Note also that as long as the interfaces used by agents to communicate with one-another are standardized, each group of agents could run an entirely different protocol, because the only way the different peer groups are connected with each other is through the bottom and upper interfaces of their agents. For example, in smaller peer groups with small interconnect latency agents could use a token ring protocol, whereas in very large groups over a wide area network, with frequent joins and leaves and frequent configuration changes agents might use randomized gossip protocol. This flexibility could be extremely useful in settings where local administrators control policies governing, for example, use of IP multicast, and hence where different groups may need to adhere to different rules. It also allows for local optimizations. Protocols used in different parts of the network could be adjusted so as to match the local network topology, node capacities, throughput or latency, the present of firewalls, security policies, and so forth. Indeed, we believe that the best approach to building high-performance systems on the Internet scale is not through a uniform approach that forces the use of the same protocol in every part of the network, but by the sorts of modularity our architecture enables, because it can leverage the creativity and specific domain expertise of a very large class of users, who can tune their local protocols to match their very specific needs.

Figure 32. A hierarchy of recovery domains and agents implementing them



The flexibility enabled by our architecture also brings a new perspective on a node in a publish-subscribe system. A node subscribing to the same topics in different portions of the Internet, joining an already established infrastructure (existing recovery domains and agents implementing them running among existing subscribers) may be forced to follow a different protocol, potentially not known in advance. Indeed, if we exploit the full power of modern runtime platforms, a node might be asked to use a protocol that must first be downloaded and installed, in plug-and-play fashion. Because in our architecture, elements "installed" in nodes by the forwarding framework (filters and channels) and by the recovery framework (agents) are very simple, and communicate with the node via a small, standardized API (recall the abstract model of a peer in Figure 28, which can also be interpreted as the abstract model of an agent in the recovery framework), these elements can be viewed as downloadable software components.

Thus, a filter or a recovery agent can be a piece of code, written in any popular language, such as Java or one of the family of .NET languages, that exposes (to the node hosting it, or to agents above and below in the agent stack) and consumes a standardized interface, for example, described in WSDL. Such components could be stored in online repositories, and downloaded

as needed, much as a Windows XP user who tries to open a file in the new Vista XPS format will be prompted to download and install an XPS driver, or a visitor to a Web page that uses some special Active-X control will be given an opportunity to download and install that control. In this perspective, the subscribers and publishers that join a publish-subscribe infrastructure, rather than being applications compiled and linked with a specific library that implements a specific protocol, and thus very tightly coupled with the specific publish-subscribe engine, can now be thought of as "empty containers" that provide a standard set of hookups to host different sorts of agents. Nodes using a publish-subscribe system are thus runtime platforms, programmable "devices," elements of a large, flexible, programmable, dynamically reconfigurable runtime environment, offering the sort of flexibility and expressive power unseen in prior architectures.

We believe that in light of the huge success of extensible, component-oriented programming environments, standards for distributed eventing must incorporate the analogous forms of flexibility. To do otherwise is to resist the commercial, off-the-shelf (COTS) trends, and history teaches that COTS solutions almost always dominate in the end. It is curious to realize that although Web services standards were formulated by some of the same companies that are leaders in this componentized style of programming, they arrived at standards proposals that turn out to be both rigid and limited in this respect.

One part of our architecture, for which API standardization options may not be obvious, includes the upper and bottom interfaces. As the reader may have realized, the exact form of these interfaces would depend on the protocol. For example, while a simple protocol implementing the "last copy recall" semantics of the sort we used in some of our examples require agents to be able to exchange a simple ACK/NAK information, more complex protocols may need to determine if messages have been persisted to stable storage, to be able to temporarily suppress the delivery of messages to the application, control purging messages from cache, decide on whether to commit a message (or an operation represented by it) or abort it, and so forth. The "state" of recovery domain and the set of actions that can be "requested" from a recovery domain may vary significantly.

As it turns out, however, defining the upper and bottom interfaces in a standard way is possible for a wide range of protocols. In our technical report (Ostrowski et al., 2006), we have outlined elements of a novel architecture being developed by the three authors of this article, called the "QuickSilver Properties Framework," and based on the very architecture presented here, that achieves precisely this form of standardization. Moreover, the properties framework allows a large class of protocols, including such protocols as virtually synchronous multicast and multicast with transaction semantics, and so forth, to be implemented in a declarative manner, using a special, domain-specific rule-based language, without requiring that the developer worry about performance and scalability aspects.

The key idea behind this approach is based on the observation that the state of most distributed protocols can be accurately described by a set of "properties." Properties are essentially variables that can be associated with various distributed entities (the reader might think of these entities as recovery domains, which they would indeed be, were the system described there to be implemented within the architecture described here). An example of a property is **Received**(**x**), parameterized by an entity name **x**. The value of this variable would be the set of identifiers of all messages that have been received by at least one node that is still in **x**. Other examples, values of all of which would again be sets of message identifiers, include **Cached**(**x**) – the messages cached at some nodes in **x**, **Cleaned**(**x**) – the messages received, but no longer cached in **x**, **Stable**(**x**) – messages received by all nodes in **x**, and so on.

In Ostrowski et al. (2006), we argue that the logic of most protocols could be modeled as a set of rules that determine how the values of such properties are created and propagated.

For example, some properties are aggregated. For a distributed entity **x**, **Received**(**x**) can be defined as the set sum of **Received**(**y**) for all **y** that are members of **x**. If this rule is applied recursively to a distributed entity, it will yield the set of messages that are received by any node in the span of that entity, as requested. Similarly, **Stable**(**x**) can be defined as the set intersection of **Stable**(**y**) for all **y** that are members of **x**. A rule that implements message cleanup could be modeled as **CanClean**(**root**) ← **Stable**(**root**), where root is the top-level entity (the top-level recovery domain), and **CanClean**(**x**) is a property, the value of which is disseminated rather than aggregated, that is, passed in a top-down fashion, from the root down to the individual nodes. As it turns out, such rules can be implemented on top of the architecture outlined in this article, using a special version of an agent that supports a few simple mechanisms, such as different flavors of property aggregation or dissemination, and the ability to produce a property based on a value of a certain expression, either periodically, or in response to events, such as receiving a message, and so forth.

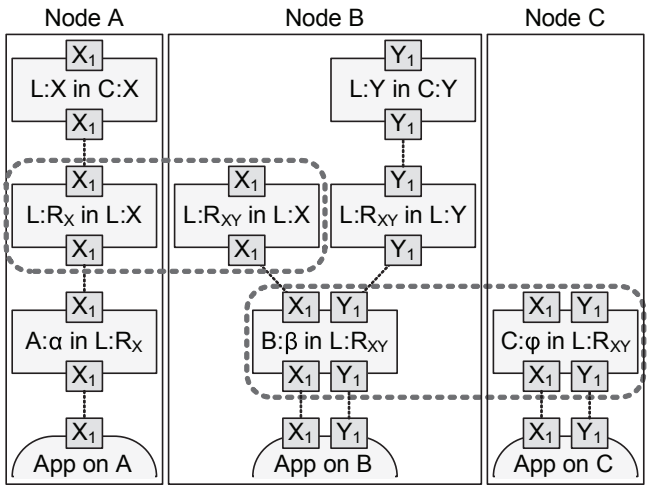
We believe that even without using the properties framework, a *set of properties*, expressed in a standard manner, including their "types," is a good candidate for the upper or bottom interface. Agents could still be implemented in an imperative manner, explicitly use the messaging API and the membership notifications from the scopes controlling them, but using the "properties" API to interact with other agents in the agent stack. The details of how exactly such interface could be defined is, however, beyond the scope of this article. Moreover, other similarly expressive schemes may exist. Indeed, it is conceivable that agents co-located on the stack could be able to "negotiate" the manner in which they interact, and download appropriate "converter" components if necessary to ensure that their upper and bottom interfaces match against each other.

To conclude this section, we now turn to recovery in many topics at once. Throughout this section, the discussion focused on a single topic, or a single session, but as mentioned before, the recovery domains created by the reliability framework, and hence the sets of agents that are instantiated to "implement" those recovery domains, may be requested to perform recovery in multiple sessions at once, for reasons of scalability. As mentioned earlier, after recovery domains are established, and agents instantiated, the root scope may issue requests to install or flush a session, passed along the hierarchy of domains, in a top-down fashion. These notifications are a part of the standard agent API. Agents respond to the notifications by introducing or eliminating information related to a particular session in the state they maintain or control messages they exchange.

For example, agents in a peer group could use a token ring protocol and tokens circulating around that ring could carry a separate recovery record for each session. After a new session would get installed, the token would start to include the recovery record for that session. When a flushing request would arrive for a session, the session would eventually quiesce, and the recovery record related to that session would be eliminated from the tokens, and from the state kept by the agents. The exact manner in which introducing a new session and flushing are expressed would depend on how the agent implements it. The available agent implementation might not support parallel recovery in many sessions at once; in this case, the scope manager could simply create a separate recovery domain for each topic, or even for each session, so that separate agents are used for each.

If an agent performs recovery for many sessions simultaneously, its "upper" and "bottom" interfaces would be essentially arrays of interfaces, one for each session. Likewise, agent stacks might no longer be vertical. The stacks for a scenario of Figure 26 are shown on Figure 33. Here, agents on node B form a tree. Two parts of the upper interface of one agent that correspond to two different sessions that the agent is performing recovery for, are connected with two bottom interfaces of two independent higher-level agents. At this point, the structure depicted in this example may seem

Figure 33. Agents stacks that are not simply vertical that may be created on nodes in the scenario from Figure 26. Agents are shown as gray boxes, with the parts of their upper or bottom interfaces corresponding to particular sessions as small yellow boxes. On node B, the bottom-level agent connects the two parts of its upper interface to two different higher-level agents, one to each of them. The peer groups are circled with thick dotted red lines



confusing. In the section entitled "Implementing Recovery Domains with Agents," we come back to this scenario, and we explain how such structures are built.

Implementing Recovery Domains with Agents

In the section entitled "Building the Hierarchy of Recovery Domains," we've explained how a hierarchy of recovery domains is built, such that for each session, there is a tree of domains performing recovery for that session. In the section on "Recovery Agents," we indicated that the recovery domains are "implemented" with agents, and in the section entitled "Modeling Recovery Protocols," we explained how recovery protocols can be expressed in a hierarchical manner, by a hierarchy of agents that represent recovery domains. We now explain how agents are created.

A distributed recovery domain **D** in our framework (i.e., a domain different than a node, not a leaf in the domain hierarchy) will correspond to a peer group. When **D** is created at some scope **X**, the latter selects a protocol to run in **D**, and then every subdomain $\mathbf{Y_k:D_k}$ of **D** is requested to create an agent that acts as a "peer $\mathbf{Y_k:D_k}$ within peer group **X:D**". We will refer to an agent defined in this manner as "$\mathbf{Y_k:D_k}$ in **X:D.**" Note how the membership algorithm provides membership view at one level "above," that is, the scope that owns a particular domain would learn about domains in all the sibling scopes. This is precisely what is required for each peer $\mathbf{Y_k:D_k}$ in a peer group **X:D** to learn the membership of its group. Hence, the scopes $\mathbf{Y_k}$ that own the different domains $\mathbf{D_k}$ will learn of the existence of domain **X:D**, each of them will realize that they need to create an agent "$\mathbf{Y_k:D_k}$ in **X:D,**" and each of them will receive from **X** all the membership change events it needs to keep its agent with an up to date list of its peers.

For example, on Figure 26, domains **B:β** and **C:φ** are shown as members of $\mathbf{L:R_{XY}}$, so according to our rules, agents "**B:β** in $\mathbf{L:R_{XY}}$" and "**C:φ** in $\mathbf{L:R_{XY}}$" should be created to implement $\mathbf{L:R_{XY}}$. Indeed, the reader will find those agents on Figure 33, in the protocol stacks on nodes **B** and **C**.

When the manager of a scope **Y** discovers that an agent should be created for one of its recovery domains $\mathbf{D}_k$ that is a member of some **X:D**, two things may happen. If **X** manages a single node, the agent is created locally. Otherwise, **Y** delegates the task to one of its subscopes. As a result, the agents that serve as peers at the various levels of the hierarchy are eventually delegated to individual nodes, their definitions downloaded from an online repository if needed, placed on the agent stack and connected to other agents or to the applications. We thus arrive at a structure just like in Figure 30, Figure 31, Figure 32, and Figure 33, where every node has a stack of agents, linked to one another, with each of them operating at a different level.

While agents are delegated, the records of it are kept by the scopes that recursively delegated the agent, thus forming a *delegation chain*. This chain serves as a means of communication between the agent and the scope that originally requested it to be created. The scope and the agent can thus send messages to one another. This is the way membership changes or requests to install or flush sessions can be delivered to agents.

When the node hosting a delegated agent crashes, the node to which that agent is delegated changes. That is, some other node is assigned the role of running this agent, and will instantiate a new version of it to take over the failed agents responsibilities. Here, we again rely on the delegation chain. When the manager of the superscope of the crashed node (e.g., a LAN scope manager) detects the crash, it can determine that an agent was delegated to the crashed node, and it can request the agent to be redelegated elsewhere. Since our framework would transparently recreate channels between agents, it would look to other peers agents as if the agent lost its cached state (not permanently, for it can still query its bottom interface and talk to its peers). On the one part, this frees the agent developer from worrying about fault-tolerance. On the other part, this requires that agent protocols be defined in a way that allows peers to crash and "resume" with some of their state "erased." Based on our experience, for a wide class of protocols this is not hard to achieve.

Reconfiguration

Many large systems struggle with costs triggered when nodes join and leave. As a configuration scales up, the frequency of join and leave events increases, resulting in a phenomenon researchers refer to as "churn." A goal in our architecture was to support protocols that handle such events completely close to where they occur, but without precluding global reactions to a failure or join if the semantics of the protocol demand it. Accordingly, the architecture is designed so that management decisions and the responsibility for handling events can be isolated in the scope where they occurred. For example, in the section on "Building the Hierarchy of Recovery Domains," we saw a case in which membership changes resulting from failures or nodes joining or leaving were isolated in this manner. The broad principle is to enable solutions where the global infrastructure is able to ignore these kinds of events, leaving the local infrastructure to handle them, without precluding protocols in which certain events do trigger a global reconfiguration.

An example will illustrate some of the tradeoffs that arise. Consider a group of agents implementing some recovery domain **D** that has determined that a certain message **m** is locally cached, and reported it as such to a higher-level protocol. But now suppose that a node crashed and that it happens to have been the (only) one on which **m** was cached. To some extent, we can hide the consequences of the crash: **D** can reconfigure its peer group to drop the dead node and reconstruct associated data structures. Yet **m** is no longer cached in **D** and this may have consequences outside of **D**: so long as **D** cached **m**, higher level scopes could assume that **m** would eventually be delivered reliably in **D**; clearly, this is no longer the case. Thinking back to the properties framework, the example illustrates the risk that a property such as **Cached**(**x**), defined earlier, might not grow

monotonically: here, **Cached(x)** has lost an item as a consequence of the crash

This is not the setting for an extended discussion of the ways that protocols handle failures. Instead, we limit ourselves to the observations already made: a typical protocol will want to conceal some aspects of failure handling and reconfiguration, by handling them locally. Other aspects (here, the fact that **m** is no longer available in scope **D**) may have global consequences and hence some failure events need to be visible in some ways outside the scope. Our architecture offers the developer precisely this flexibility: events that he wishes to hide are hidden, and aspects of events that he wishes to propagate to higher-level scopes can do so.

Joining presents a different set of challenges. In some protocols, there is little notion of state and a node can join without much fuss. But there are many protocols in which a joining node must be brought up to date and the associated synchronization is a traditional source of complexity. In our own experience, protocols implementing reconfiguration (especially joins) can be greatly facilitated if members of the recovery domains can be assigned certain "roles". In particular, we found it useful to distinguish between "regular" members, which are already "up to date" and are part of an ongoing run of a protocol, and "light" members, which have just been added, but are still being brought up to date.

When a new member joins a domain, its status is initially "light" (unless this is the first membership view ever created for this domain). The job of the "light" members, and their corresponding agents, is to get up-to-date with the rest of their peer group. At some point, presumably when the light members are more or less synchronized with the active ones, the "regular" agents may agree to briefly suspend the protocol and request some of these "light" peers to be promoted to the "regular" status.

The ability to mark members as "light" or "regular" is a fairly powerful tool. It provides agents with the ability to implement certain forms of agreement, or consensus protocols that would otherwise be hard to support. In particular, this feature turns out to be sufficient to allow our architecture to support virtually synchronous, consensus-like, or transactional semantics.

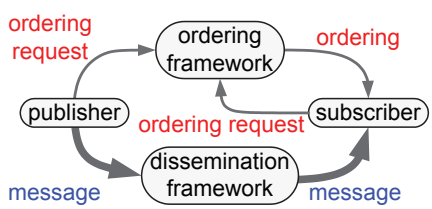
Ordering

As mentioned in the section on "Incorporating Reliability, Ordering, and Security," ordering is implemented independently of dissemination. When a publisher submits a message to the dissemination framework, it simultaneously submits an *ordering request* into the ordering framework. An ordering request is a small object that lists the sender identifier, message topic and a sequence number. In response to the ordering requests, the ordering framework produces *orderings*. Orderings are small objects assigning indexes to batches of messages, perhaps coming from different publishers. These orderings are then delivered to the interested subscribers (Figure 34).

Because the ordering framework is designed to process requests from different publishers together, it is possible to, for example, totally order messages in each topic, or even totally order messages across different topics.

Messages transmitted by publishers may be marked as *ordered*, as a hint for subscribers that these messages should not be delivered until

Figure 34. For all messages for which ordering across multiple senders is required, publishers create ordering requests and submit these in batches to the ordering framework. The latter produces orderings in response to the ordering requests, and delivers these orderings to the subscribers



an ordering for these messages arrives through the ordering framework. The subscribers that do not care about ordering will simply not subscribe to the ordering framework, and will ignore such markings.

The ordering framework consists of two components: a component that collects ordering requests from multiple sources and produces orderings in response to these requests, and a component that delivers these orderings to subscribers. The first of these components is essentially an aggregation mechanism, and the second is similar to the dissemination framework.

Like every other element of our architecture, ordering is performed in a hierarchical manner that respects isolation and local autonomy of administrative domains. Thus, the ordering framework also relies on the concept of *management scopes*, in this case the *ordering scopes*, which may or may not overlap with the other flavors of scopes. Each scope can define its own policies that govern the way ordering is performed: how ordering requests are collected, which member of the scope should serve as the *orderer* (collect these requests and produce orderings), and how these orderings should be disseminated to subscribers. The resulting architecture is a product of policies defined at various levels, just as it was the case for dissemination and reliability.

To prevent the article from becoming excessively long, we shall omit the details of the ordering framework. The design shares common elements and ideas with the dissemination and reliability frameworks, and will be covered comprehensively in the first author's Ph.D. thesis.

Other Possible Frameworks

Up until now, we've focused on the dissemination, recovery, and ordering frameworks within our overall architecture. However, the same structure can also support additional frameworks, which exist as logical "siblings" to the ones already presented. These include:

- Flow and rate control. Architectural mechanisms in support of flow and congestion control handling, negotiating rates, managing leases on bandwidth, and so forth.
- Security. Architectural support for integrating security (managing keys, granting or revoking access, managing certificates, etc.) into scalable eventing systems.
- Auditing. Self-verification, detection of inconsistencies (e.g., partitions, invalid routing, and so forth).
- Failure detection and health monitoring. Scalable detection of faulty or underperforming machines.

CONCLUSION

We have argued that new and more flexible event notification standards are going to be needed if the Web services community is to gain the benefits of architectural standardization while also exploiting the full power of component architectures and integration platforms. The proposal presented here draws heavily from our experience building Quicksilver, a new and extremely scalable eventing infrastructure that scales in multiple dimensions, integrates seamlessly with modern component-style platforms, and has the flexibility to support a wide range of reliability models including best-effort, virtual synchrony, consensus, and transactional ones. In contrast, our experience trying to express Quicksilver within the existing Web services options was discouraging; we found them to be narrowly conceived and incapable of offering needed flexibility and scalability.

ACKNOWLEDGMENT

Our research was supported by AFRL/Cornell Information Assurance Institute. We acknowledge Mahesh Balakrishnan, Lars Brenna, Yejin Choi, Maya Haridasan, Tudor Marian, Ingrid Jansch-Porto, Robert van Renesse, Yee Jiun Song, Einar Vollset, and Hakim Weatherspoon for their feedback.

REFERENCES

Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R., & Sturman, D. (1999, May 31-June 4). An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS) (IEEE Computer Society),* Washington, D.C. (p. 262).

Box, D., Cabrera, L.F., Critchley, C., Curbera, D., Ferguson, D., Geller, A., et al. (2004). Web services eventing (WS-eventing). Retrieved August 30, 2007, from http://www.ibm.com/developerworks/Webservices/library/specification/ws-eventing/

Floyd, S., Jacobson, V., Liu, C., McCanne, S., & Zhang, L. (1996). A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking, 5*(6), 784-803.

Graham, S., Niblett, P., Chappell, D., Lewis, A., Nagaratnam, N., Parikh, J., et al. (2004). Web services brokered notification (WS-brokered-notification). Retrieved August 30, 2007, from http://www.ibm.com/developerworks/library/specification/ws-notification/

Keidar, I., & Dolev, D. (1995, May 22-25). Increasing the resilience of atomic commit, at no additional cost. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '95),* San Jose, CA (pp. 245-254). New York, NY: ACM Press.

Keidar, I., Sussman, J., Marzullo, K., & Dolev, D. (2002). Moshe: A group membership service for WANs. *ACM Transcations on Computer Systems*, *20*(3), 191-238.

Ostrowski, K., & Birman, K.P. (2006a, September 18-22). Extensible Web services architecture for notification in large-scale systems. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06)* (Vol. 00, pp. 383-392). Washington, D.C.: IEEE Computer Society.

Ostrowski, K., & Birman, K.P. (2006b, November 3). Scalable group communication system for scalable trust. In *Proceedings of the First ACM Workshop on Scalable Trusted Computing (STC '06)* Alexandria, VA (pp. 3-6). New York, NY: ACM Press.

Ostrowski, K., & Birman, K.P. (2006c). *Scalable publish-subscribe in a managed framework* (Tech. Rep.). Cornell University.

Ostrowski, K., Birman K.P., & Dolev, D. (2006). *Properties framework and typed endpoints for scalable group communication* (Tech. Rep). Cornell University.

Paul, S., Sabnani, K.K., Lin, J. C.-H., & Bhattacharyya, S. (1997). Reliable multicast transport protocol. *IEEE Journal of Selected Areas in Communications, 15*(3), 407-421.

Weinsberg, Y., Dolev, D., Anker, T., & Wyckoff, P. (2006, November). Hydra: A novel framework for making high-performance computing offload capable. In *Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN 2006)*. Tampa, FL.

Krzysztof Ostrowski is a PhD student in the Department of Computer Science at Cornell University. He is currently building QuickSilver, a new type of a development platform with an extremely fast and scalable group communication engine at the core, offering a range of strong reliability properties and a smooth integration with Windows, Visual Studio, and the web service technologies. This new platform is aimed at enabling a new style of programming, characterized by casual use of publish-subscribe topics to represent distributed, interactive content. Before joining Cornell, Ostrowski spent four years in the industry.

Ken Birman is a professor of computer science at Cornell University. He currently heads the QuickSilver project, which is developing the world's fastest and most scalable publish-subscribe system and a new, highly automated, platform aimed at making it dramatically easier to build scalable clustered applications. Previously he worked on fault-tolerance, security, and reliable multicast. In 1987 he founded a company,

Isis Distributed Systems, which developed robust software solutions for stock exchanges, air traffic control, and factory automation. For example, Isis currently operates the New York and Swiss Stock Exchanges, the French air traffic control system, and the US Navy AEGIS warship. The technology permits these and other systems to automatically adapt themselves when failures or other disruptions occur, and to replicate critical services so that availability can be maintained even while some system components are down. In contrast to his past work, Birman's recent work has focused on issues of scale, self-management and self-repair mechanisms for complex distributed systems, such as large data centers and wide-area publish-subscribe. The very large scale of these kinds of applications poses completely new challenges. For example, while protocols for data replication on a small scale are closely tied to dahabase concepts such as two-phase commit, these large scale applications are best viewed as probabilistic systems, and the most appropriate technologies are similar to techniques seen in peer-to-peer file sharing applications. Birman is the author of several books. His most recent textbook, Reliable Distributed Computing: Technologies, Web Services, and Applications*, was published by Springer-Verlag in May of 2005. Previously he wrote two other books and more than 200 journal and conference papers, including one that appeared in Scientific American in May, 1996. Dr. Birman was also editor in chief of* ACM Transactions on Computer Systems *from 1993-1998 and is a fellow of the ACM.*

Danny Dolev received his BSc degree in mathematics and physics from the Hebrew University, Jerusalem in 1971. His MSc thesis in applied mathematics was completed in 1973, at the Weizmann Institute of Science, Israel. His PhD thesis was on Synchronization of Parallel Processors *(1979). He was a post-doctoral fellow at Stanford University, 1979-1981, and IBM Research Fellow 1981-1982. He joined the Hebrew University in 1982. From 1987 to 1993 he held a joint appointment as a professor at the Hebrew University and as a research staff member at the IBM Almaden Research Center. He is currently a professor at the Hebrew University of Jerusalem. His research interests are all aspects of distributed computing, fault tolerance, security and networking—theory and practice.*