Extensible Web Services Architecture for Notification in Large-Scale Systems

Krzysztof Ostrowski
Cornell University
krzys@cs.cornell.edu

Ken Birman
Cornell University
ken@cs.cornell.edu

Abstract

Existing web services notification and eventing standards are useful in many applications, but they have serious limitations precluding large-scale deployments: it is impossible to use IP multicast or for recipients to forward messages to others and scalable notification trees must be setup manually. We propose[1] *a design free of such limitations that could serve as a basis for extending or complementing these standards. The approach emerges from our prior work on QSM* [1]*, a new web services eventing platform that can scale to extremely large environments.*

1. Introduction

1.1. 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 Internet scales. The WS-Notification [2] and WS-Eventing [3] 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 issues of polling, and support proxy nodes that can be used to build scalable notification trees. Nonetheless, they embody restrictions:

- *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, formed by recipients, and without the dedicated infrastructure, is more appropriate.
- *Inability to use external multicast frameworks.* Both 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 could also be used to provide QoS guarantees or leverage the emergent technologies.
- *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).
- *Difficult to manage*. At Internet scales, it is hard to create and maintain a dissemination structure that would permit any node to serve as a publisher or subscriber, for this requires many parties to maintain common infrastructure, 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, etc.
- *Weak reliability*. Reliability in the existing schemes is limited to per-link guarantees, resulting from the use of TCP. In many situations, stronger guarantees are required, e.g. to support virtually synchronous, transactional or state-machine replication. Because receivers are assumed passive and cannot cache, forward messages or participate in multi-party protocols, even weak guarantees cannot be provided.

1.2. Our Contribution

In this document, we propose a principled approach to building large-scale systems for web services notification. We outline a design for an extensible notification scheme free of the limitations just described, which is the basis for Quicksilver [1], a new scalable and reliable publish-subscribe and notification platform under development at Cornell. Motivated by the end-to-end principle, we separate the implementation of loss recovery and strong reliability properties from the unreliable dissemination of messages. Accordingly, our design includes a *reliability framework* and a *dissemination framework*: two largely independent, yet complementary structures.

Both frameworks reflect the principles articulated below, and they share many elements. In particular, both employ hierarchical protocol stacks, an idea that is central to our architecture. These stacks permit the definition of an Internet-scale loss recovery scheme which can employ different recovery policies within different administrative domains. Likewise, they permit a construction of a global dissemination scheme that uses different mechanisms to distribute data within different administrative domains.

[1] Our effort is supported by AFRL/Cornell Information Assurance Institute.

IEEE International Conference on Web Services (ICWS'06)
0-7695-2669-1/06 $20.00 © 2006 IEEE

1.3. Design Principles

The limitations of the existing designs listed above and our experience 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 nodes 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 permit an external trusted entity or an agreement protocol to control it: determine the protocol it follows, install rules for message forwarding or filtering etc.
- *Hierarchical structure*. The principles listed above should apply to not just individual nodes, but also entire administrative domains such as LANs, data centers or corporate networks. This allows the definition and enforcement of Internet-scale forwarding policies, facilitating the cooperation among organizations in maintaining the global infrastructure. The way a message is delivered to subscribers across the Internet thus reflects policies defined at various levels.
- *Isolation and local autonomy*. A certain degree of local autonomy of the administrative domains must be preserved; such as how messages are forwarded internally, which nodes create which communication endpoints etc. In essence, the structure of a domain should be hidden from other domains it is peering with and from the higher layers. Likewise, details of its own subcomponents should as opaque as possible.
- *Channel negotiation*. Communication channel creation should permit a handshake. A recipient might be asked to e.g. join an IP multicast group, or subscribe to an external system. The recipient could then make a configuration decision on the basis of the information about the sender, e.g. a LAN asked to prepare a communication endpoint for receiving may choose a well-provisioned node to handle the anticipated load.
- *Managed channels*. Communication channels should be represented as *active contracts* in which receivers have a degree of control over the way the senders are sending. 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, might notify the source that messages should now be transmitted to two other nodes, instead.
- *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, e.g. by drag-and-drop, within a suitable GUI.

1.4. Basic Concepts

We employ the usual terminology, where notifications are associated with *topics* and produced by *publishers* and delivered to *subscribers*. We use the term "*group X*" to refer to the group of nodes subscribed to topic "*X*". More than one publisher may exist for a given topic. The prospective publishers and subscribers register with a *subscription manager*, which can be decentralized and independent of the publishers (see Figure 1, top). In our architecture, nodes reside in *administrative domains*. Nodes in the same domain are jointly managed. It is often convenient to define policies, such as for message forwarding or resource allocation, in a way that respects domain boundaries; either for administrative reasons, or because communication locally in a domain is cheaper than across domains, as it is often related to network topology. Publishers and subscribers may be scattered across organizations, which must cooperate in message delivery. This often presents a logistic challenge (see Figure 1, bottom).

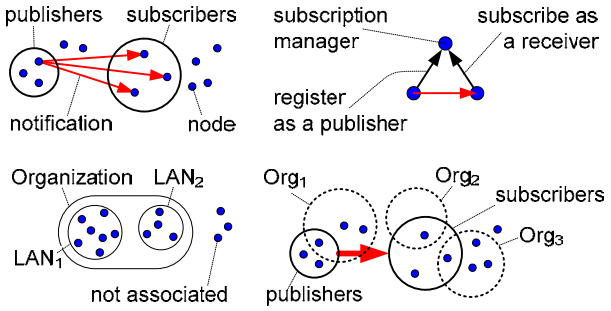

Figure 1. Publishers and subscribers register with the *subscription manager* (top). Nodes are scattered across *administrative domains* hierarchically divided into *sub-domains* (bottom).

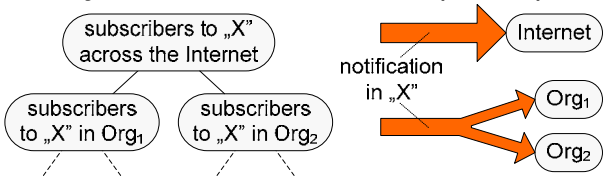

Figure 2. A hierarchical decomposition of the set of subscribers along the domain boundaries.

1.5. A Hierarchical View of the Network

A group X of subscribers for a given topic across the Internet can be divided into subsets Y_1, Y_2, ..., Y_N of subscribers in N top-level administrative domains (Figure 2). This can be continued recursively, leading to a hierarchical perspective on the set of all subscribers. By the principle of *isolation* and *local autonomy*, each administrative domain should manage the registration of its own publishers and subscribers internally, and decide how to distribute messages among them according to its local policy. Similar ideas were previously exploited in the context of content-based filtering [6], and in many scalable multicast algorithms, e.g. in RMTP [4]. This also reflects the principle of locality, implicit in many scalable protocols. Following this principle, groups of nodes, clustered based on proximity or interest, cooperate semi-autonomously in message routing and forwarding, loss recovery, managing membership and subscriptions, failure detection etc. Each such group is treated as a single cell within a global 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 overhead, e.g. buffer messages coming from different sources and locally disseminate such combined bundles etc. We make heavy use of this property in our QSM [1] system.

This principle of locality and the hierarchical view of the network outlined above form the basis for our design.

2. Design Overview

2.1. The Hierarchy of Scopes

Our design is constructed upon the following principal concepts: *management scope*, *channel*, *filter*, *forwarding policy*, *session*, *recovery algorithm*, and *recovery domain*.

A *management scope (*or simply a *scope)* represents a *set of jointly managed nodes*. It may include a single node, span over a group 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 the Internet. We do not assume a 1-to-1 correspondence between administrative domains and the scopes defined based on such domains, but it will often be the case, and we will refer to a *LAN scope* (or just a *LAN*) to mean the scope spanning over 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 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 in data centers. No such global infrastructure or administrative authority exists for the Internet, but organizations could provide servers to control the global scope in support of their own publishers or to manage the distribution of messages in topics of importance to them. Many independently managed global scopes could thus co-exist.

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

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 e.g. by a server managing a LAN and based on node subscription patterns, could also be considered scopes, all managed by the same server. Nodes would be members of clusters and clusters (not nodes) would be members of the LAN. As it will be explained below, each cluster, as a separate scope, could be locally and independently managed. In [1], such construction is a basis for our scalable multicast protocol.

A scope hierarchy is not a tree. There may be multiple global scopes, or many super-scopes for any given scope. However, a scope always decomposes into a tree of sub-scopes, down to the level of nodes. We refer to a *span of a scope X* as the set of all nodes at the bottom of a hierarchy of scopes rooted at X. For a given topic X, there always exists a single global scope responsible for it, i.e. such that all subscribers to X reside in the span of X. Publishing a message in a topic is thus equivalent to delivering it to all subscribers in the span of some global scope, which may be further decomposed into subscribers in the spans of sub-scopes (compare section 1.5 and Figure 2).

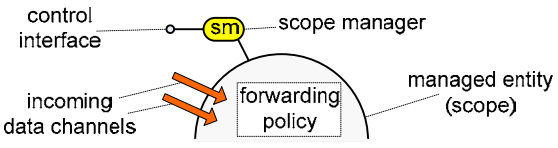

Figure 3. Accessed via a control interface and configured with a forwarding policy, a scope manager creates incoming data channels.

2.2. The Anatomy of a Scope

The infrastructure managing 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 on a set of nodes, perhaps outside of the scope it controls. It exposes a *control interface*, a web service hosted at a well-known address, to dispatch control requests directed to scopes it controls (Figure 3). SMs interact by calling each other's control web interfaces (see also [8]).

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 over 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. In an overlay network, a channel could lead to nodes that forward messages across the entire overlay.

A scope that spans over a set of nodes is governed by forwarding *policy* specifying how messages that originate within that scope or arrive through some communication channel are forwarded internally and to other scopes.

The reader will recognize in our construction the principles we formulated earlier. Scopes, whether individual nodes, LANs or overlays, are *externally controlled* using their control interfaces, may be *programmed* with policies that govern the way messages are distributed internally and forwarded to other scopes, and transmit messages via *managed* communication channels established through a dialogue between a pair of SMs.

2.3. 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 on. 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 sub-scope Y of X delivers messages internally is a decision made by Y autonomously. 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.

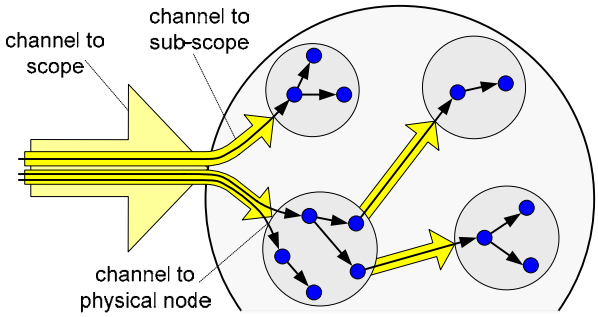

Figure 4. Channels created in support of forwarding policies defined at different levels.

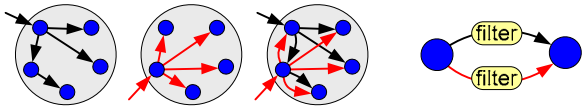

Figure 5. Forwarding graphs for different topics are superimposed. Two members may be linked by multiple channels, each with a different filter.

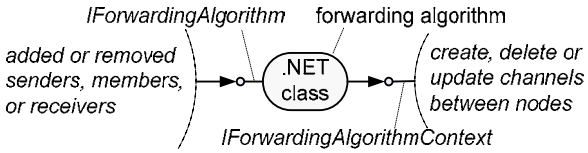

Figure 6. A forwarding policy as a code snippet.

Accordingly, a global policy may request organization X to 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. Finally, the policies of the respective LANs could delegate these tasks to individual nodes. When the policies defined at all the scopes involved are combined, the resulting *forwarding structure* completely determines the way messages are forwarded (see Figure 4).

In the examples above, the policies are simply graphs of connections: each message is always forwarded along every channel. In general, however, each channel may be optionally constrained by a *filter* that decides, on a per-message basis, whether to forward or not, and optionally tags the message with custom attributes. This allows us to express many popular techniques, e.g. using redundant paths, multiplexing between dissemination trees etc.

Every scope manager maintains internally a mapping from topics to policies: a graph of channels to create and filters on them. Such graphs of connections for different topics are superimposed (see Figure 5). Based on this, the SM asks the scope members to create channels and filters. When the structure is modified as a result of membership or subscription changes, the SM makes additional control requests to reflect this.

In our framework, a policy is defined as an algorithm that lives in an *abstract context*, with a fixed set of *events*

to respond to, standard set of *operations* and *attributes* to inspect. In a prototype we are developing, we implement a forwarding policy as a .NET class, stored in a DLL on an *algorithm library* server, that implements an abstract interface and interacts with an abstract *context* hiding the details of the environment (Figure 6). This allows our policies to be used within any scope.

2.4. Communication Channels

Consider a node X, a member of a scope Z that, based on a forwarding policy at Z, has been requested to establish a communication channel to scope Y to forward messages in topic T. Following the protocol, X asks the SM of Y for the specification of the channel to Y that should be used for messages in topic T. The SM of Y 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 Y may dictate that, in order to send to Y in topic T, X should establish channels to members A and B of Y, constrained with filters α and β. After X learns this from the SM of Y, it contacts SMs of A and B for details. Notice how the channel decomposes into sub-channels to A and B through a policy at a target scope Y.

This decomposition continues hierarchically, until the point when scope X is left with a tree containing filters in internal nodes and address/protocol pairs at the leaves (Figure 9). In order to send a message along the channel built in this way, X executes filters to determine which sub-channels to use, proceeding recursively, until it is left with a list of address/protocol pairs, then transmits the message. Filters will typically be simple, such as modulo-**n**; hence X could perform this procedure very efficiently.

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 sub-channels accompanied by filters. In the latter case, the filters jointly implement a multiplexing scheme that determines which sub-channels to use for sending, on a per-message basis (see Figure 7 and Figure 8).

Consider now the case when scope X, spanning over a set of nodes, has been requested to create a channel to scope Y. Through a dialogue with Y and its sub-scopes, X can get a detailed channel definition, but unlike in the example above, X now spans over a set of nodes, and as such, it cannot *execute* filters or *send* messages.

We propose two example generic techniques that solve this problem: *delegation* and *replication* (Figure 10). Both rely on the fact that if scope X receives messages in a topic T, then some of its members, Z, must receive them (for otherwise X would not be made part of a forwarding structure for topic T by X's super-scope). In case of *delegation*, X requests such a sub-scope Z to create the channel on behalf of X, essentially delegating the whole channel. The problem can be recursively delegated, down to the level where a single physical node is requested to create a channel. A more sophisticated use of delegation would be for X to delegate sub-channels. In such case, X would first contact Y to obtain the list of sub-channels and the corresponding filters, and for each of these sub-channels, delegate it to one of its sub-scopes. In any case, X delegates the responsibility for sending over a channel, in one way or another, to one or more of its sub-scopes.

In case of *replication*, scope X requests **n** of its sub-scopes to create the channel, but constrains each with a modulo-**n** filter based on a message sequence number (i.e. sub-scope **k** only forwards messages with numbers **m** such that **m** mod **n** equals **k**), effectively implementing a round-robin policy. Although all sub-scopes would create the same channel, the round-robin filtering policy ensures that every message is forwarded only by one of them.

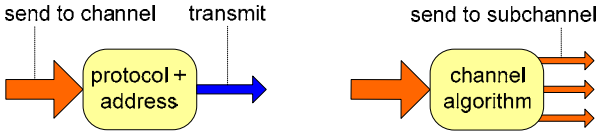

Figure 7. A channel may be an *address/protocol* pair (left), or it may consist of *sub-channels*, with an *algorithm* deciding what goes where (right).

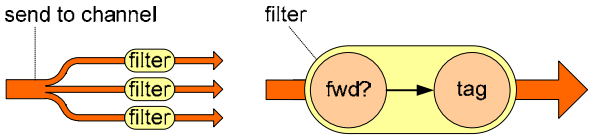

Figure 8. Channel algorithms are realized as sets of filters, one per subchannel, deciding whether to forward, and optionally adding custom tags.

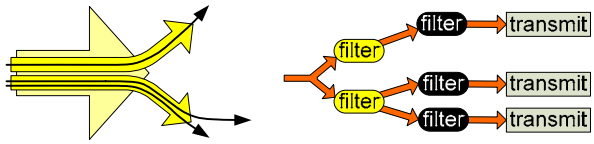

Figure 9. A channel split into sub-channels and a possible filter tree corresponding to it.

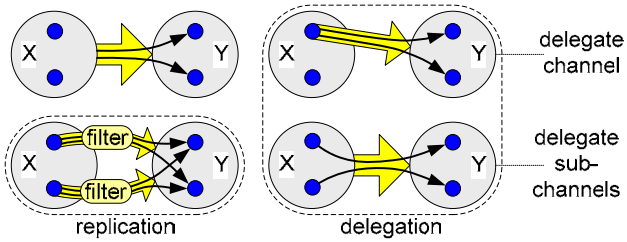

Figure 10. A distributed scope may delegate a channel or its sub-channels to members, or it may replicate them among members with filters that jointly implement a round-robin policy.

2.5. Reliability Scopes

The design of the reliability framework also relies on the concept of management scopes, referred to here as *reliability scopes* (in contrast to *dissemination scopes* in the dissemination framework). A reliability scope isolates and encapsulates the local aspects of loss recovery, hiding details from other scopes, just like a dissemination scope hides the local aspects of message delivery. Reliability scopes are also controlled by scope managers. Both kinds of scopes would typically overlap. For example, a single scope could be defined for an administrative domain such as a LAN, isolating local aspect of both dissemination and reliability. The scope could then be controlled by a single SM managing both dissemination and reliability.

The separation of dissemination from reliability makes it possible to combine an arbitrary unreliable notification mechanism, such as IP multicast or an overlay content delivery system, with a wide range of reliability protocols expressible in our reliability framework. This degree of reusability has not been possible with prior architectures.

2.6. Hierarchical Approach to Reliability

Our approach to reliability resembles our approach to dissemination. Just as channels are decomposed into sub-channels, in the reliability framework we decompose the task of repairing after message losses and providing other reliability goals. Recovering messages in a certain scope is modeled as recovering within sub-scopes, and then recovering "among" the sub-scopes (Figure 11). Just like recovery among single nodes, recovery among LANs may involve comparing their "state" (such as aggregated ACK or NAK information for the entire LANs) and forwarding lost messages. In section 2.9 we give examples of how recovery protocols may be defined and combined.

In our framework, different recovery schemes may be used in different scopes, to reflect differences in network topology, node or network capacity, the way subscribers are distributed (e.g. clustered vs. scattered around) etc.

Just like messages are disseminated through channels, 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 in **X**" that performs recovery-related tasks for a certain set of topics. The concept of a recovery domain is symmetric, dual to the notion of a channel. We present it via 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 may be multiple channels to a scope, e.g. for different sets of topics, multiple recovery domains, each for different topics, may exist within a single reliability scope.
- Just like channels may be composed of sub-channels, a recovery domain **D** defined at scope **X** may be composed of *sub-domains* $\mathbf{D_1}$, $\mathbf{D_2}$, …. $\mathbf{D_n}$ defined at sub-scopes of **X** (we will call them *members* of D). Each such sub-domain $\mathbf{D_i}$ handles recovery for a certain set of subscribers in the respective sub-scope, while **D** handles recovery "across" its sub-domains.
- Just like channels are composed of sub-channels via applying filters assigned by forwarding policies, a recovery domain **D** performs its recovery tasks using a *recovery algorithm*. Such an algorithm, assigned to **D**, specifies how to combine recovery mechanisms in the sub-domains of **D** into a mechanism for all of **D**. Recovery algorithms are defined in terms of how the sub-domains "interact" with each other. We will see how this is achieved in section 2.9.
- 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 for multiple topics. In both cases, reusing a single mechanism (a channel, a token ring etc.) may significantly improve performance due to the reduction in the total number of "control" messages. We evaluated this idea in [1].

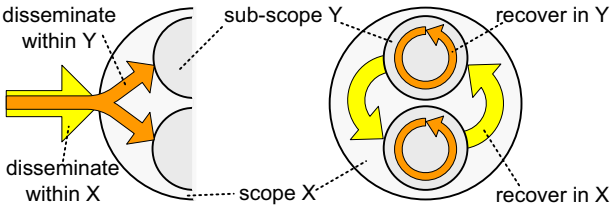

Figure 11. The similarity between hierarchical dissemination (left) and recovery (right).

Each individual node is a recovery domain on its own. In a distributed scope such as a LAN, on the other hand, two cases are possible. First, a single domain may cover the entire LAN. All internal nodes could form e.g. a token ring, exchange ACKs for messages in all topics, and use this to arrange for local repairs. Another possibility is that separate domains would be created for every individual topic. Subscribers to different topics would form separate structures, such as ring or trees, and run separate protocol instances in each, exchanging state and los messages.

As explained later, recovery domains in our system actually handle recovery for specific *sessions*, not just for specific topics. Sessions are introduced in section 2.7.

A recovery domain D of a data center could have as its members recovery domains created in LANs. Note that in this case, members of D would be sets of nodes. A recovery algorithm 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 shown in section 2.10, recovery algorithms are implemented through delegation, just like forwarding. A concept of a *recovery algorithm* is, to a certain extent, symmetric to the notion of a forwarding policy.

2.7. Sessions

Within our architecture, protocols that provide strong reliability guarantees express them in terms of *epochs*. An epoch corresponds to what in group communication literature is called a *membership view*. The lifetime of a topic is divided into a sequence of epochs. Whenever the set of subscribers to a topic changes as a result of a subscribe/unsubscribe request or a failure, the event initiates a new epoch. Subscribers are notified of the beginnings or endings of epochs. One then defines consistency in terms of which messages may be delivered to which subscribers and at what time, relative to epoch boundaries. The traditional term "membership view" reflects the fact that epochs begin and end with membership change events. The set of subscribers during a given epoch is fixed.

Although simple protocols, such as SRM or RMTP, do not rely on a consistent view of group membership, and their properties are not defined in terms of epochs, epochs are still a useful, if not a universal concept. In a dynamic system, configuration changes, especially those resulting from crashes, usually require reconfiguration or cleanup, e.g. to rebuild a distributed structure, release resources or cancel activity that is no longer necessary. Many simple protocols simply do not take this factor into account.

We introduce the idea of a *session*, a generalization of an epoch (membership view). A session is also an epoch in the prior sense, i.e. the lifetime of any given topic can always be divided into a sequence of sessions. Like before, any membership change marks the beginning of a new session and for a given session, membership is fixed. However, a new session may also be initiated even if membership is unchanged. The reliability properties of a group may vary to some extent in the subsequent sessions. An important example is an administrative change, where a new protocol is introduced, e.g. because it is more efficient or to fix a bug in the existing protocol. In Internet-scale systems such administrative changes must be performed online; session changes achieve this.

Session numbers are assigned globally for consistency. As explained before, for a given topic, a single "global" scope always exists such that all subscribers to that topic reside within the span of this scope. This is true for both dissemination and reliability frameworks. Usually, both global scopes overlap and are managed by a single SM. The top-level SM assigns and updates session numbers. Note that local topics (e.g. internal to an organization) could be managed by the local SM, much in a way local newsgroups are visible and managed locally.

Before discussing the mechanisms used to manage membership, we conclude the discussion of sessions by explaining how they 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 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 activity and *cleanup* resources pertaining to session **k**, completing the transition.

2.8. Constructing the Recovery Structure

Reliable protocols often rely on, or could benefit from, a consistent view of membership. It helps to determine which nodes have crashed or disconnected. In existing systems, this is achieved by a Global Membership Service (GMS) that monitors failures and membership changes, decides when to install new membership views for topics, and notifies all affected members of the new views. In our framework, the global SM for a given topic is responsible for announcing when sessions begin and end. However, if the global SM had to process all subscriptions, it would lead to a non-scalable design that violates the principle of isolation. To avoid this, for each topic **T** we distribute the information about membership of **T** across all SMs in the hierarchy of scopes for **T** (this hierarchy was defined in section 2.1). Each SM thus has only a partial membership view for each session. This scheme is outlined below.

In the reliability framework, if a scope **X** subscribes to a topic **T**, it specifies some local recovery domain **D** that should handle the recovery for topic **T** in **X**. The **X**'s super-scope **Y** processes this subscription request jointly with requests from other sub-scopes. It then creates its own recovery domains, with the newly subscribed and perhaps some existing sub-domains as members, and then issues its own subscription requests to its super-scope. This continues recursively up to the global scope.

The scheme used by the super-scope to create recovery domains must abide by three rules. First, the list of sub-

domains of a recovery domain is determined once at the time of creation, and fixed throughout its lifetime. This is necessary to ensure that a hierarchical structure employed for recovery in any given session does not change, which simplifies the overall design. Second, a recovery domain **D** at scope **X** is responsible for handling recovery for a specific set of topics, in specific sessions. If a change in membership in any of these topics occurs locally in **X**, a new recovery domain **D'** must be created, and when a new session is announced, it is installed in **D'**. This is because the existing recovery domain **D** no longer represents the current set of subscribers within **X**, hence a new distributed structure **D'** must be established. Finally, if a new session is announced for some topic **T**, but no membership changes occurred for **T** within scope **X** since the previous session, then an existing recovery domain should be re-used to handle recovery in the new session.

In a scope in which recovery for each topic is handled individually, we would maintain a separate sequence of recovery domains for each topic. A new domain would be created whenever the set of subscribers locally changes. In a scope in which recovery for all topic is performed jointly, such as e.g. in a cluster of nodes defined based on subscription patterns in which all nodes are subscribers to the same set of topics, there will be just a single sequence of recovery domains. We used the latter scheme in [1].

The above procedure effectively constructs a hierarchy of sub-domains, with the property that for each topic **T**, the recovery domains subscribed to **T** form a tree.

The global scope assigns new session numbers for all topics for which subscribe or unsubscribe requests have been received, and determines which of its local recovery domains should handle the new sessions. This represents a coarse-grained membership view, for each session only top-level recovery domains are specified, with no further details. The information about the new sessions is now sent down the tree of subscribers, and transformed along the way to filter out unnecessary details. The membership information a scope **X** receives for a session **S** is limited to one level "above" **X**, i.e. it includes **X**'s own recovery domain that got subscribed to **S** and the recovery domains of its *sibling* scopes (i.e. scopes that have the same super-scope). It is also coarse-grained, i.e. it does not provide any details at the level "below" **X** or its siblings.

2.9. Modeling Recovery Algorithms

The design of the reliability framework is based on an abstract model of a distributed protocol dealing with loss recovery and other reliability properties. When expressed within our framework, such protocols will be referred to as *recovery algorithms*. Recovery algorithms are the basic building blocks in constructing our hierarchical reliability protocols, much in a way channels and filters are the basic building blocks in our forwarding infrastructure.

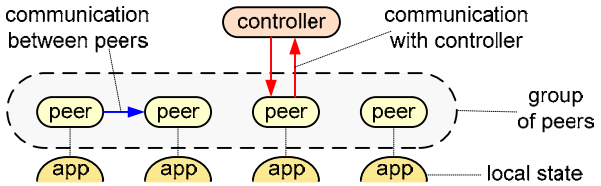

Figure 12. A group of *peers* in a reliable protocol.

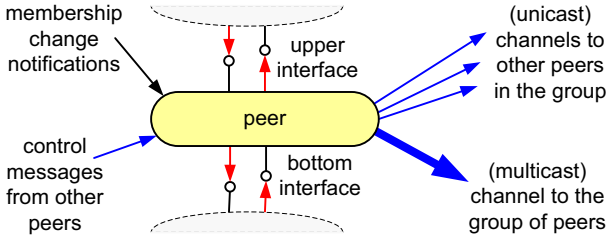

Figure 13. A peer modeled as a component living in abstract environment (events, interfaces etc.).

A protocol such as SRM, RMTP, or virtual synchrony is defined in terms of a group of cooperating *peers* that send control messages and forward lost packets to each other, and perhaps to a distinguished node, such as a sender or some node higher in a hierarchy, that we will refer to as a *controller* (Figure 12). 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, e.g. a sender, a higher-level protocol, or a layer in a hierarchical structure etc. Examples of such actions include requesting or performing a retransmission for all nodes, reporting which messages were successfully delivered to all nodes etc. Irrespectively of how exactly the interaction with a 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 it as an *upper interface*.

Each peer inspects and controls *local state*. Such state may include e.g. a list of messages received and perhaps copies of those that are cached (for loss recovery), the list and the order of messages delivered etc. Operations a peer may issue to change the local state could include e.g. retrieving/purging messages from a local cache, marking messages as "deliverable", handing a previously missed message to the application or assigning message sequence in a "totally ordered" group. We refer to such operations, used to view or control local state, as a *bottom interface*.

In protocols offering strong guarantees, peers typically know the membership of 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 can be modeled as running in an environment that provides the following: 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 a sender or a higher level in the hierarchy concerning the state of the whole group, (Figure 13). In some protocols, parts of the environment might be unavailable, e.g. in SRM peers might not know other peers. The bottom and upper interfaces 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 the RMTP protocol [4], the sender and the receivers for a given topic form a tree. Within this tree, each subset of nodes consisting of a parent and child nodes serves as 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, 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 the "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 14). 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 think of a node as running two "agents", each representing a different "half" of the node and serving as a peer in a separate peer group. Every group of peers, in this perspective, includes the "bottom agent" of a parent node and "upper agents" of 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 agents play different roles, as explained below.

The bottom agent of each node interacts via its *bottom interface* with the local state of the node. It also serves as a distinguished peer in the peer group composed of itself and the upper agents of 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, i.e. which messages were not recoverable in the group. This is communicated by the bottom agent, via its *upper interface*, to the upper agent.

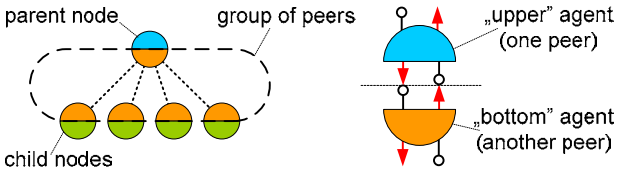

Figure 14. RMTP expressed in our model. A node hosts "agents" playing different roles.

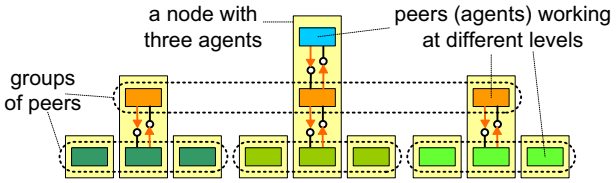

Figure 15. Another way to express RMTP. Each node hosts multiple "agents" that act as peers at different levels of the RMTP hierarchy.

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.

Such interactions, between a component that is a part of a "higher layer" and a component that resides in a "lower layer", 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, the upper agent of the root node communicates through its *upper interface* the state of the entire tree of receivers to the sender.

The second way to model RMTP captures the essence of our approach to combining protocols. It is similar to the first model, but instead of the "upper" and "bottom" agents, each node may host multiple agents, connected to each other, each working at a different level (Figure 15). In a LAN scope, all nodes host a "local agent" component (green), similar to the "bottom agents" above, that serves as a peer in the group of all LAN nodes. The *bottom interface* used by this agent interacts with the local state. These peers exchange ACKs and arrange for message forwarding, with one of them acting as a "parent" and all other as "children". On the node hosting the "parent", a "higher-level" agent is hosted (orange); we refer to it as a "LAN agent", for there is exactly one in each LAN, and it represents the entire LAN. It connects through its bottom interface to the local agent, which is a distinguished peer in a LAN peer group, to obtain information concerning the LAN it is controlling, e.g. ACKs. These LAN agents themselves form a "higher-level" peer group. One serves as a distinguished parent node, others as subordinates. The LAN agents are communicating with each other to arrange for forwarding messages, and they jointly calculate the ACK information for the entire scope, which in

this case could be e.g. a data center in which the LANs reside. The distinguished node that hosts the parent LAN agent also hosts a yet higher-level component, call it a "data center agent". This agent could communicate with the sender, or the construction might continue further in a similar fashion. Note how in this example the peer groups defined at various levels overlap with scope boundaries.

Note also that as long as their interfaces match, each peer group could run an entirely different algorithm. We believe this power 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.

The issue of how to select protocols at different levels in such a way that their interfaces would match is beyond the scope of this paper. In our forthcoming paper [7], we introduce a new mechanism that could help address this issue in a more systematic manner.

To keep the presentation simple, in the model and in the examples we discussed a peer group handles recovery in a single topic. In our full design, a group of peers can handle recovery in multiple sessions at once. Throughout the lifetime of the group, peers will be instructed to begin recovery for certain sessions, at some point later they will enter the flushing phase for specific sessions (while other sessions may still be active), and may finally be requested to cease any activity for specific sessions. Accordingly, a peer, via its bottom and upper interfaces, exchanges data and requests related to multiple sessions at once. One may think of a peer as having multiple pairs of bottom and upper interfaces, each pair for a different set of sessions. Also, peers hosted at a physical node will not necessarily form a vertical, linear stack, as in our examples, the same lower-level peer may interact with two or more peers at a level above it. We omit details for clarity. All techniques that we introduced here carry over to the full design.

2.10. Implementing Recovery Algorithms

In section 2.8 we have explained how a hierarchy of recovery domains is built, such that for each session, the domains "responsible" for it form a tree. In section 2.9 we gave an example of how an algorithm such as RMTP can be modeled in our framework as a network of agents that handle the recovery tasks at various levels. A distributed recovery domain **D** in our framework will correspond to a peer group. When **D** is created at some scope **X**, the latter selects an algorithm to run in **D**, e.g. a ring or a tree, and then every sub-domain $\mathbf{D_k}$ of **D** is requested to create an agent that acts as a "peer $\mathbf{D_k}$ in group **D**". Note how the membership algorithm provides membership view at one level "above", i.e. 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 $\mathbf{D_k}$ in a group **D** to learn the membership of its group.

When the SM of a scope **X** learns that an agent should be created for one of its recovery domains $\mathbf{D_k}$ in group **D**, two things may happen. If **X** manages a single node, the agent is created locally. Otherwise, **X** delegates the task to one of its sub-scopes. As a result, the agents that serve as "peers" at the various levels are delegated to individual nodes. We thus arrive at a structure just like on Figure 18, where each node has a stack of one or more agents, each operating at a different level, linked to one another. When the node hosting a "higher-level" agent crashes, the agent is delegated to another node. Since our framework would transparently recreate channels between agents, it looks 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). This requires that algorithms be defined in a way allowing 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.

3. Evaluation

The need for brevity precludes a detailed discussion of the performance of our architecture. The strength of this design lies in its extensibility, ability to accommodate a wide range of transport and recovery protocols, and in facilitating the cooperation among independent parties in creating a global publish-subscribe infrastructure. Such benefits are hard to quantify. However, in certain scenarios, our approach can also greatly improve scalability. In [8], we show how we used the model and principles presented here as the basis for the design of QSM [1], a new publish-subscribe platform offering a simple ACK-based reliability and extremely scalable in multiple dimensions. We are also in the process of creating a reference implementation of the infrastructure outlined here. Ultimately, this effort will lead to a set of specifications similar to [2].

5. References

[1] K. Ostrowski, K. Birman, and A. Phanishayee, "QuickSilver Scalable Multicast". In submission, 2006.

[2] http://ifr.sap.com/ws-notification/ws-notification.pdf

[3] http://ftpna2.bea.com/pub/downloads/WS-Eventing

[4] S. Paul, and K. Sabnani, "Reliable Multicast Transport Protocol". Journal of Selected Areas in Communications (1997).

[5] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang, "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing". IEEE/ACM TONS (1996).

[6] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman, "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems". ICDCS '99.

[7] K. Ostrowski, K. Birman, "Achieving Modularity and Scalability via Typed Communication Endpoints". Forthcoming.

[8] K. Ostrowski, K. Birman, "Extensible Web Services Architecture for Notification in Large-Scale Systems (Extended Version)". Cornell University Technical Report. Forthcoming.