Exploiting Gossip for Self-Management in Scalable Event Notification Systems

Ken Birman, Anne-Marie Kermarrec, Krzystof Ostrowski,
Marin Bertier, Danny Dolev, Robbert Van Renesse
Cornell University, Ithaca; INRIA/IRISA and IRISA/INSA, Rennes; Hebrew University, Jerusalem

Abstract

Challenges of scale have limited the development of event notification systems with strong properties, despite the urgent demand for consistency, reliability, security, and other guarantees in applications developed for sensitive tasks in large enterprises. These issues are the focus of Quicksilver, a new multicast platform targeted to large-scale deployments. An initial version of the system can support large numbers of overlapping multicast groups, high data rates and groups with large numbers of members. However, Quicksilver still requires manual help when discovering the system configuration and can't easily enforce certain types of application monitoring and integrity constraints. In this paper, we propose to extend Quicksilver by introducing gossip mechanisms, yielding a self-managed event notification platform. The two technologies are presented through a single interface and appear to end users as live distributed objects*, side-by-side with other kinds of typed components.*

1. Introduction

As we look to the next generation of distributed computing platforms, it is hard not to feel concern at the accelerating deployment of systems that will play sensitive roles, and yet will be built using fragile technologies. For example, an electronic health records system must achieve high levels of availability and consistency, be largely self-configuring, and maintain privacy and security. A typical deployment scenario involves decentralized systems linked over networks, integrating subsystems running at hospitals, other care providers, laboratories, insurance companies, pharmacies, etc. Electronic monitoring devices and other sensors running both in the hospital and at home will contribute time-sensitive data, and some therapeutic and drug delivery devices will be remotely controlled.

To reduce cost and leverage standardization, a system of this sort would probably be constructed using COTS platform technologies, such as web services. Doing so also brings productivity benefits, in the form of development tools and runtime support, and makes it easy to integrate pre-supplied functions with new application-specific ones. However, today's solutions lack the sorts of strong properties needed for sensitive uses. Our objective is to extend these platforms by adding robust tools that bridge gaps while complying with standards.

The nerve center of a modern service-oriented architecture is its event notification subsystem. Event notification services can distribute sensor readings and other kinds of updates to widely distributed system components, and can be used to replicate information where an application or a record is available at multiple locations. By decoupling publishers from subscribers, these services make it easy to upgrade an application over time and to integrate components that run on dissimilar platforms or were implemented using very different technologies. On the other hand, traditional event notification platforms lack the strong guarantees needed for medical decision making and other critical roles.

If we can create a new kind of scalable, robust event notification architecture that fits seamlessly into modern development platforms such as Windows .net or J2EE, and yet has strong properties that reduce to rigorously specified protocols that the end user can count upon and reason about, we can help application developers create robust applications for sensitive uses.

In this paper, we focus on scalability, robustness and self-management, deferring issues of security and privacy for the future. For scalable event notification with strong reliability guarantees, we've developed Quicksilver: a high-performance multicast technology that can implement a variety of reliability models, including consensus-based ones [1][2]. Traditionally, systems implementing reliable multicast have scaled poorly, but as reported below, this problem can be overcome. Moreover, although we don't tackle the question here, we believe that Quicksilver can be secured using digital certification certificates, by authenticating access to information resources, and encrypting all network traffic using per-event-channel keys that can be refreshed whenever the set of subscribers changes.

The existing version of Quicksilver is weaker with respect to self-configuration and self-management; both critical requirements for the sorts of applications we hope to support. In our target environments, the pace of reconfiguration could be very rapid: if a patient falls ill, providers might (in effect) hand the family a box full of equipment to be deployed throughout the home. Needs change as the patient's care plan evolves. Patients are moved from unit to unit. Thus one must imagine a highly dynamic, rather unpredictable environment in which the

[1] Contact: ken@cs.cornell.edu; The Cornell research group was supported by grants from AFRL/IFSE, AFOSR, NSF and the Intel Corporation.

sets of components, their configurations, and their communication patterns change constantly. Against this backdrop, we seek an event notification infrastructure that can configure itself, that can adapt as conditions evolve, and that can be leveraged to support self-configuring applications.

Fault-tolerance poses closely related problems. Today, Quicksilver offers fault-tolerance through models such as virtual synchrony, where applications are structured into groups and, if desired, will be notified when membership changes. But not all integrity constraints map easily to group membership tracking. For example, the decoupling of publisher from subscriber is advantageous from a development perspective, but sometimes correct function requires that there be an active subscriber associated with certain topics. One such case involves logging accesses to patient records for offline audits. If this functionality is implemented using event notification, it important that the logging service be running when audit events are published. Yet even if built upon a substrate such as Quicksilver, today's event notification APIs lack mechanisms to express such constraints, and hence can't trigger exceptions when they are violated.

To address self-* needs, both within Quicksilver and in applications built using it, we propose to use technology emerging from work on *gossip* protocols. Gossip encompasses a large class of protocols that exploit randomness to achieve surprising robustness under a wide range of operating conditions. They can be made self-configuring, adapt rapidly after disruption, and support a diversity of useful end-user functionality.

The integration of gossip with multicast in a single setting poses non-trivial systems-engineering challenges. Here, we propose such a unification. Although our new system is still under development, it will offer a seamless infrastructure in which Quicksilver runs side-by-side with gossip-based mechanisms to provide a self-managed scalable event notification capability. The system will expose these gossip mechanisms so that applications can exploit them directly in the same paradigm used to expose Quicksilver's multicast functionality. Here we sketch out the architecture and discuss some research challenges it poses; several appear to be of broader relevance.

The paper is structured as follows. First, we spend a moment discussing the strengths and limitations of gossip technologies. The goal is not to be exhaustive, but rather to identify styles of gossip that are both highly effective and well matched to our self-management objectives. Next, we review Cornell's new platform, Quicksilver, touching both on its scalability and its unusual embedding into the Windows .net framework. The latter topic emerges as a source of leverage in what we are now proposing to do. Finally, we explore the options for integrating the two, arriving at an architecture that (we believe) is interesting in several respects. First, it sets gossip side by side with scalable event notification. Next, the system offers an elegant embedding into Windows so that developers can benefit from that system's powerful component integration functionality and development tools (a Linux version is also under design). And finally, it suggests a path for future evolution of service-oriented architectures and standards. The paper concludes by discussing open research questions.

2. Gossip protocols

A *gossip protocol* is one with the following properties:

1. The core of the protocol involves periodic, pairwise, inter-process interactions.
2. The information exchanged during these interactions is of (small) bounded size.
3. When node *a* interacts with node *b*, the state of *a* evolves in a way that reflects the state of *b* (and vice versa). For example, if *a* pings *b* merely to measure RTT, this is not a gossip interaction.
4. Reliable communication is not assumed.
5. The frequency of the interactions is relatively low when compared to typical message latencies.
6. There is some form of randomness in peer selection.

There are three prevailing styles of gossip protocol.

1. *Dissemination (rumor-mongering) protocols*. These use gossip to spread information; they basically work by flooding nodes in the network, but in a manner that produces bounded worst-case loads:
   a. An *event dissemination* protocol runs in response to events and can be understood as using gossip to carry out multicasts, although the events don't actually trigger the gossip (since gossip runs periodically).
   b. A *background data dissemination* protocol gossips continuously to track the evolution of state at participating nodes.
2. *Anti-entropy protocols* repair replicated data by comparing replicas and reconciling differences.
3. *Aggregation protocols* compute a network-wide aggregate by sampling information at the nodes in the network and combining the values to arrive at a system-wide value – the number of nodes in the system, the sum or average of some value, etc

Our definitions are rather broad; indeed, many protocols that predate the earliest use of the term "gossip" fall within our definition. In particular, notice that a gossip substrate can "mimic" a standard routed network. That is, nodes could "gossip" about traditional point-to-point messages, in effect tunneling normal traffic through a gossip layer. Bandwidth permitting, this implies that a gossip system can potentially support any classic protocol or distributed service. Nonetheless, when we talk of gossip, we rarely intend such a broadly inclusive interpretation. More typically we have in mind protocols that run in a regular, periodic, relatively lazy, symmetric and decentralized manner; the high degree of symmetry among nodes is particularly characteristic. To illustrate this point, consider that one could run a 2-phase commit protocol over a gossip substrate, piggybacking the messages on gossip traffic. In our view, doing so would be at odds with the spirit of the definition: there's nothing wrong with such a protocol, but it isn't gossip!

2.1 The Limitations of Gossip

The stylized manner in which we normally use gossip introduces significant limitations. First, consider the implications of the small, bounded message sizes and the relatively slow periodic message exchanges. These combine to limit the information carrying capacity of a gossip algorithm. For example, if gossip is used to disseminate information (often, in a form of flooding), the system-wide capacity for new events will be limited simply because the aggregate "bandwidth" available is bounded. The problem is that gossip protocols keep the nodes in a network busy while information spreads – typically, a process that requires O(log(n)) time. It follows that the "rate" at which events can be introduced will be proportional to 1/log(n).

The relatively slow spread of gossip can also be an obstacle. While it is common to claim that users need only tune the gossip rate to match their goals, requirement 5 complicates the picture. Gossip rates approaching the network RTT are out of the question.

Finally, gossip can be fragile in the face of malicious behavior (components that malfunction, for example by running the protocol incorrectly, disseminating incorrect data, and so forth). Recent work on BAR Gossip [21] tries to overcome some of the issues by using verifiable pseudo-random peer selection to avoid selfish and malicious behaviors. But this is just a first step.

2.2 Strengths of Gossip

Although gossip has limitations, these protocols do have substantial power. Among the most cited strengths are these:

• *Convergent consistency.* Properly designed gossip protocols, when not overwhelmed by a higher rate of incoming "events" than the information-carrying bandwidth of the underlying channels, should have a logarithmic mixing time – any new event will, with high probability, affect all nodes that need to learn about it within time logarithmic in the system size.

• *Emergent structure.* Earlier, we contrasted a classic deterministic protocol for building a spanning tree by leader-initiated flooding with a decentralized way of building such a tree using gossip. In the gossip style, the tree "emerges" from randomized pairwise interactions between peers. The term emergent structure is intended to evoke the image of a data structure that emerges with probability 1.0 in this manner. The structure may then continue to evolve over time as further gossip occurs.

• *Simplicity.* Most (but not all) gossip protocols are extremely simple and highly symmetric, with all participants running the same code.

• *Bounded load on participants.* Many classic (non-gossip) distributed protocols are criticized because they can generate high surge loads that overload individual components. Gossip is normally used in ways that produce strictly bounded worst-case loads on each component, eliminating the risk of disruptive load surges. In some situations, where network capacity is also a concern, peer-selection is further biased to control load imposed on network links.

• *Topology independence.* If running on a sufficiently connected networking substrate, and with sufficient bandwidth, a gossip protocol will often operate correctly on a great variety of underlying topologies.

• *Ease of local information discovery.* Many gossip protocols are used for purposes of discovery, for example to find a nearby resource (these are usually protocols in which gossip occurs between neighbors, not between arbitrarily distant peers). Unlike local flooding, which scales poorly, gossip would typically find local information less quickly but with bounded costs: perhaps, a constant or a delay logarithmic in the system size.

• *Robustness to transient network disruptions.* As time elapses, there are exponentially many routes by which information can flow from its source to its destinations. However, not all uses of gossip are robust in all ways. For example, unless data is self-verifying, dissemination protocols are often vulnerable to data corruption. Anti-entropy protocols may similarly be at risk if a replica becomes corrupted. And aggregation protocols are vulnerable not just to the introduction of faulty information, but also to computational errors that result in a faulty computation of the aggregate.

2.3 Appropriate roles for gossip

The foregoing discussion suggests a number of natural roles for gossip in large-scale event notification systems.

The earliest uses of gossip were to disseminate information in large-scale systems [22]. Scalability and robustness were cited as the primary benefits in these uses: the load on each node grows in a logarithmic manner as the system scales and information can be reliability disseminated in the presence of a high proportion of node failures [20]. Such properties rely on the fact that each node samples network state randomly. This pseudo-randomness can nonetheless be controlled or "shaped". For example, Lpbcast [19] and Cyclon [15] are protocols in which each peer periodically selects another peer with which it gossips; they differ in the details of target selection, and in the way they merge information gathered through the gossip exchange with their own.

Generalizing these ideas, gossip may be used to create unstructured overlay networks, achieving properties close to those of random graphs [12]. Having used gossip to create such a graph, gossip protocols can also run over them, for example to create an overlay optimized with respect to an application-specific metric. For example, T-man builds overlays that use application-supplied quality functions to bias neighbor selection [10]. In [14], the gossip itself is biased; users with shared interests are structured into peer groups for file sharing, substantially improving response times in a search application.

Similarly, GosSkip [17] and Sub-2-Sub [13] build content-based publish-subscribe systems in which the overlay topology matches the subscription pattern. In GosSkip, subscriptions are organized into a skiplist structure so that events will be routed to interested subscribers in a logarithmic number of hops. In Sub-2-Sub, several gossip-protocols are layered to efficiently support range subscriptions. The lowest layer uses random peer sampling to ensure connectivity and robustness, a second layer creates clusters of "close" subscriptions, and the third layer structures overlapping subscriptions to ensure an exact and exhaustive dissemination of events.

This flexibility comes at a price. Gossip-based publish-subscribe overlays are often slow: the technology is wonderful for matching publishers with subscribers, but says little about getting events delivered rapidly, robustly, and with strong reliability properties. Indeed, we like to think of these kinds of applications as having two disjoint aspects: a gossip infrastructure that, in these cases, builds an overlay; and then a distinct dissemination structure that uses the overlay to reliably distribute events.

This way of thinking leads back to our current goals. We hope to systematically ask how gossip can be valuable in event-notification systems such as Quicksilver and in the applications that run over it. A number of options seem to be worth exploring. For example, as just seen, a gossip-constructed overlay network could be useful for efficient dissemination. In this case, Quicksilver itself would provide the "quality metrics" used to optimize the overlay, and the associated cost functions would reflect the mechanisms Quicksilver uses for dissemination and for recovery of lost packets.

More broadly, we hope to use gossip to materialize a form of distributed "picture" of the application network, which would become an input to an auto-configuration application that would generate configuration files. These would advise the end-user application (in addition to the Quicksilver event notification infrastructure) of the topology on which it should operate and the appropriate parameter settings to use. Later, as conditions evolve, the same approach could be used to reconfigure the running system so as to repair damage caused by a failure, or to integrate new components with the existing infrastructure.

Another possible role for gossip would be to track overall loads, loss rates and other status in the system. We have experience with a gossip-based system used for this purpose. Astrolabe is a distributed monitoring and data mining system that uses gossip to construct a virtual hierarchical database that can be queried much like a normal database [5]. The database is extremely useful for self-optimization and problem diagnosis. Because Astrolabe is fully replicated it has no single point of failure or load-related hot-spots, and the underlying gossip protocol remains robust even under stress that can shut down most other system functionality. In our new system, we believe aggregation mechanisms can play even more roles, including parameter setting and dynamic adaptation [11]. Aggregation can even be used for resource allocation, for example by using gossip to sort peers according to an application-specific metric [16].

Finally, we will use gossip to support background diffusion of system information that won't be needed immediately, but could be of high value "later". A tool permitting discovery of available information sources would be one possible use for such a mechanism. Other possibilities include mechanisms for tracking contact nodes or other services, finding information stored elsewhere in the network, etc. By using gossip to disseminate the underlying information, we can be certain that data will get through even if the system configuration changes (or is disrupted), and hence will be available when and where needed.

To exploit these kinds of gossip mechanisms, we need to tackle some significant software engineering issues that prior work has largely overlooked. To make gossip useful as a tool, one needs appropriate embeddings of these abstractions into the runtime environment. For these purposes, we propose to extend a feature of Cornell's Quicksilver platform, discussed below.

3. Quicksilver

Cornell's Quicksilver project [3][4] offers a scalable event notification infrastructure that can support strong properties on a per-topic basis. An application can subscribe to large numbers of communication channels, with the properties of each channel matched to the data it carries. Krzysztof Ostrowski is the lead architect and developer for Quicksilver, in collaboration with Ken Birman, Danny Dolev and Robbert van Renesse. We start by reviewing prior work on Quicksilver, and then suggest some of the extensions our new effort will explore.

A key objective for Quicksilver is scalability in multiple dimensions: numbers of applications using the platform, numbers of event channels to which each application subscribes, data rates, tolerance of disruption, etc. Our underlying premise is that inadequate scalability has limited the uptake of group-multicast in general, and has prevented its widespread use in support of event notification. This sometimes manifests itself through throughput that degrades gracefully as the system is deployed into a larger setting, but more dramatic consequences are also observed. For example, many large-scale event notification platforms become unstable in large deployments, oscillating from very low throughput to overwhelmingly high data rates in which traffic generated by the platform can actually shut down the communications bus by swamping it with data, retransmissions, nack and ack messages and other forms of overhead – a so called broadcast storm effect. In designing Quicksilver, our goal was to demonstrate stability in this problematic domain.

This is not the right setting for a detailed discussion of the Quicksilver architecture. Instead, we summarize some key ideas very briefly:

- *Separation of concerns.* Quicksilver treats event dissemination separately from recovery of lost packets, flow control, and implementation of stronger consistency ("properties").
- *Regions of overlap.* A single node will often subscribe to many event channels. If each channel is treated as a separate multicast group, one encounters obvious problems of scale. Accordingly, Quicksilver maps from overlapping channels down to *regions*, defined to be sets of nodes with similar subscriptions. Dissemination is on a per-region basis; recovery is done in an aggregated manner over regions, etc.
- *Scalable recovery.* Quicksilver uses a novel hierarchy of token rings to achieve scalable detection of lost packets and, when possible, to recover data between peers in a region, offloading work from the sender.
- *Per-channel reliability properties.* The reliability properties of each channel can be matched to its role.
- *Managed runtime environment.* Quicksilver runs in managed settings, allowing it to leverage strong type checking, memory management, etc.

Details of the architecture and protocols appear in [3][4].

Quicksilver has been running since June 2006. For the moment, all our users are building datacenters – WAN scenarios are a goal once the new gossip-based mechanisms are available, but the current system doesn't run in WAN settings. In our datacenter experiments, we've set up groups with up to 200 nodes (larger runs are planned), than subjected them to extremely high throughputs and injected various forms of stress.

Up to the present, we have seen only minimal throughput degradation and no signs of instability or throughput fluctuations even in the largest configurations. In contrast, such problems are easy to provoke in most existing technologies for multicast in the same settings, even with much smaller groups of just 50 to 75 members [2]. Quicksilver can saturate a 100Mbit ethernet interconnect with just 20-40% CPU loads on the inexpensive PC's making up our test cluster; experiments with our prior systems peaked at about a tenth these data rates and generated much heavier loads. Perhaps most important, processes are able to access large numbers of groups. For this reason, when used to support event notification, Quicksilver can maintain steady performance even when each process joins as many as 8000 separate event channels [3][4]. Obviously, this capsule summary oversimplifies in some important ways (in particular, not all configurations of processes and event streams are supported), but they do give a sense of what the system should be able to achieve.

Of primary relevance here is the manner in which Quicksilver embeds event notification channels into Windows. Traditionally, event notification platforms have been treated as a free-standing technology that lives separately from the operating system. Quicksilver can be used this way too, through a conventional publish-subscribe infrastructure that generalizes the web services eventing standards (in [6] we discuss our reasons for extending these standards rather than working entirely within ws-notification or ws-eventing).

But Quicksilver also offers a second, deeper embedding into Windows in which event notification channels can be accessed either as a new kind of distributed *live object* visible in the file system side-by-side with other named objects. These objects are best understood as distributed abstract data types. A program accesses such an object much as it would access a file in Windows: given appropriate permissions, it can open the object, read the current state, and will receive events as the state is subsequently updated. This, however, is an illusion: the "object" is really an event channel, and the state is a checkpoint produced by some existing subscriber when a new program subscribes. State persistence is available, but optional.

We've emphasized the similarity between the way that a system such as Windows understands file "types" as an association between the data in some object and the programs that implement operations on that kind of object, and the way that Quicksilver associates a type with each event notification channel. For Quicksilver, the type corresponds to an object class, but also is associated with a definition of the properties the channel should implement. The effect is to confer a distributed semantics on the group of objects as a whole. The approach is flexible enough to support weak properties such as best-effort notification, stronger consensus-based properties such as the virtual synchrony model, or even very strong models such as transactional 1-copy serializability. Quicksilver implements a domain-specific programming language within which the properties associated with each event channel can be specified. The system basically compiles these property definitions into pseudo-code which it can execute to achieve the desired behavior.

4. A unified platform

For our purposes, the key point of leverage involves the embedding of Quicksilver's live objects (event channels) into Windows. Consider the integration of abstract data types such as Excel spreadsheets or Word documents into the Windows file system. Windows uses the filename extension to understand the "type" of the object, allowing it to interpret operations on the object as method invocations on an appropriate application program. Web services standards are used in conjunction with these componentization mechanisms: active components such as the Excel application register their interfaces using the Web Services framework built into .net, at which point the Windows platform can function as a component integration environment using Web services standards and protocols to perform tasks such as method invocation. Of course, this component-to-component type system is somewhat primitive, but one could imagine taking the idea much further; indeed, there are projects underway at Microsoft to do just that. It isn't unreasonable to imagine that future versions of Windows will incorporate a full-fledged distributed type system at the component level.

As suggested above, Quicksilver extends Windows to support abstract data types with "live" content, and allows a variety of event stream providers to support the live aspects of the abstraction. A Quicksilver event notification channel has a name that can be visible in the file system name space, and a type, corresponding to the properties associated with the event channel. When an application binds itself to an event channel, Windows passes the binding event to Quicksilver, and we can perform type compatibility checking, or can even perform some kinds of dynamic type coercion (for example by introducing an encryption/decryption layer in order to integrate a component that doesn't support encryption with an event channel that requires stronger forms of security). The same mechanisms also work from the Windows shell: if a user right-clicks on a Quicksilver event channel, the shell extensions framework passes us the request. Quicksilver can then identify applications that can connect to this kind of channel, and can even generate dynamically created virtual folders, for example displaying thumbnail-size images from a video streaming application.

Quicksilver is thus on a path towards the same kind of tight integration with Quicksilver event streams as is seen with other Windows communications options such as DCOM. The approach enables developers to leverage existing Windows application development and debugging tools while benefiting from co-existence in a managed framework. If Windows evolves in the manner currently anticipated, type checking will become possible even across component boundaries. Because Quicksilver uses the CLR memory management layer, no copying occurs when a large object is multicast. Of course, such a positioning of the technology also brings challenges of its own (for example, to maximize performance in a managed environment requires protocol designs quite different from those one uses in a Linux/C multicast implementation [3]) but the problems are solvable and we believe the result is well worth the effort. We should comment that although Windows is our initial target, everything we are doing should port (using Mono) to Linux and would then be accessible from J2EE or even Corba applications.

This, then, is the core contribution of the present paper: a vision of how one might unify these three worlds: objects in a platform such as Windows on the one hand, and both gossip and of scalable event notification on the other, all in a single framework. A first step towards this vision requires that the Quicksilver multicast framework be separated from the mechanisms that embed Quicksilver objects into Windows; Ostrowski is already developing this capability as part of version 2.0 of the system. As is the case in Quicksilver today, the basic abstraction will be that of a distributed object having a "state" and an associated event stream. However, rather than assuming that the live content is transported by Quicksilver's reliable multicast protocols, there will be at least two possible communication infrastructures – the other being gossip-based. Down the road one might imagine additional options, such as an IP-TV streaming layer, or one focused on real-time communication.

Thus, referring back to the examples of gossip-based mechanisms mentioned in Section 3, one could build a gossip-based topology and configuration discovery service that, in effect, produces an annotated picture of the state of the system. An end-user could access that picture by clicking on an associated file name; doing so would launch some sort of browser capable of visualizing this kind of information and might let the user explore the network, for example to pin down a bottleneck that is impacting performance. Application programs could use the picture to configure themselves. And Quicksilver's event notification infrastructure could use that picture to construct overlays for disseminating events that use IP multicast when possible, but tunnel data through overlay trees where IP multicast is not feasible (and these same overlay networks would also be available to application designers, through some form of abstract data type). The remarkable robustness of the gossip protocols ensures that even when all else is disrupted, applications can still

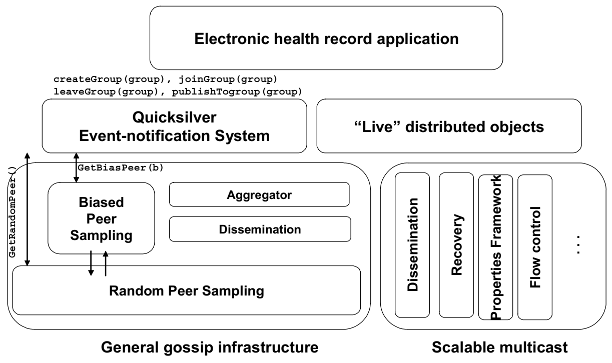
Electronic health record application
createGroup(group), joinGroup(group)
leaveGroup(group), publishTogroup(group)
Quicksilver Event-notification System
"Live" distributed objects
GetBiasPeer(b)
GetRandomPeer()
Biased Peer Sampling
Aggregator
Dissemination
Random Peer Sampling
Dissemination
Recovery
Properties Framework
Flow control
...
General gossip infrastructure
Scalable multicast

Figure 2: Overall System Architecture

monitor the system to set parameters, configure themselves, and adapt when conditions change.

But we believe we can do more than to simply import gossip functionality into Quicksilver. Gossip systems of the types we reviewed share substantial commonalities across their various presentations. For example, many gossip mechanisms require random peer selection, either within the full system (a kind of *anycast*) or within a set of neighbors of a node (a local variant on *anycast*). The thinking is that this and other low-level primitives can be standardized within the gossip subsystem, and then reused across gossip-based objects. Doing so poses interesting research challenges: if a single object employs *anycast*, one can implement a "greedy" solution. But suppose that on some single node there are tens or even hundreds of gossip-based objects, all using *anycast.* Could we aggregate, so that a single message can carry information on behalf of multiple objects?

One can pose similar questions at a higher level. Many gossip algorithms are highly stylized: the nature of a gossip exchange is rather similar across most gossip-based mechanisms, even if the details of what "state" is exchanged and how it is "merged" differ. This immediately suggests that one might design an abstract gossip state-machine that could be instantiated in multiple objects, parameterized with appropriate state marshalling and merge functions.

The resulting architecture is summarized in Figures 1 and 2. Figure 1 illustrates the overall system architecture, with the gossip infrastructure hosted side-by-side with the scalable multicast infrastructure and accessed either through a generalized publish-subscribe interface, or in the form of live distributed objects. As noted earlier, internal details for Quicksilver can be found in [3] and will not be repeated here. Figure 2 gives some additional detail for the gossip infrastructure.

5. Electronic health record example

We conclude the discussion by revisiting our electronic health record example, assuming now that the gossip mechanisms and the Quicksilver-based event notification solution are available side-by-side.

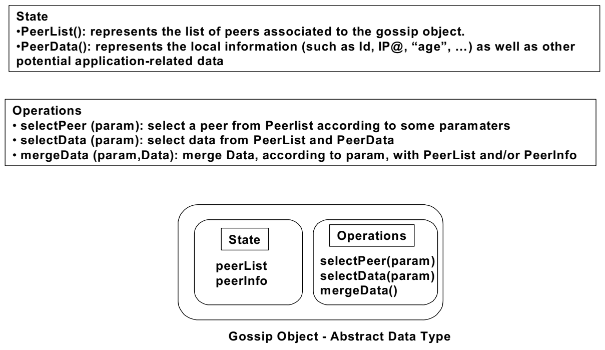
State
•PeerList(): represents the list of peers associated to the gossip object.
•PeerData(): represents the local information (such as Id, IP@, "age", ...) as well as other potential application-related data
Operations
• selectPeer (param): select a peer from Peerlist according to some paramaters
• selectData (param): select data from PeerList and PeerData
• mergeData (param,Data): merge Data, according to param, with PeerList and/or PeerInfo
State
peerList
peerInfo
Operations
selectPeer(param)
selectData(param)
mergeData()
Gossip Object - Abstract Data Type

Figure 1: Generalized Implementation of a Gossip Object

Let's start with roles for the gossip mechanisms. For the time being, we've decided to focus on uses in which the gossip components will be simple enough so that we can verify correctness, able to "sanity check" data collected from the environment, and unlikely to come under attack; these assumptions mitigate the security concerns mentioned earlier. For example, with gossip it isn't difficult to build a system that can track locations of system components: servers, client platforms, sensors, other devices. When a change occurs, the updated configuration should become visible with delay proportional to the log of the size of the system – in the scenarios we have in mind case, probably within 10 or 15 rounds of gossip. This capability could be the basis for a highly robust plug-and-play technology, whereby the health-care system would adapt in tens of seconds as conditions evolve. Although such a system might collect incorrect information about a platform that has some form of scrambled configuration state, the "damage" would be limited to the annotation of that component on the map, and the gossip objects can be designed to sense and reject implausible inputs.

Gossip could also be used to monitor system invariants (such as: "there should always be at least one instance of the auditing service"). Here, Quicksilver's notion of membership offers very rapid event detection and reaction, but if enough damage occurs while the system is running to seriously disrupt event notification, the gossip layer could guide a timely discovery of the problem and dynamic repair or adjustment of the parameters. The remarkable robustness of gossip mechanisms gives us reason for confidence that they will be able to continue to operate reliably even when other infrastructure components are severely degraded by a disruptive event.

Gossip can also be used to help system components connect themselves in appropriate ways. For example, a component might keep track of the locations of the various servers so that in the event of a fault that prevents connection to one server, the clients using it can seamlessly roll over to others offering backup functionality. When the first server recovers, the clients can shift back. Gossip mechanisms can be used to monitor system health, assisting managers in diagnosing and repairing problems that arise because of software bugs or other disruptive events. If a firewall or server comes under attack (or just becomes overloaded), gossip based tracking mechanisms can help client systems discover the problem, identify fall-back options, and gracefully adapt.

Gossip also offers an antidote to certain kinds of fragility. For example, suppose that we want to track the physical location of patients in our hospital complex. In the most obvious standard implementation of an electronic health record system, one would probably place some sort of active component on the patient's gown or bed; it would continuously track its own location (somehow) and report that data to the central database. With gossip, new and potentially more robust options arise. Now, client systems can gossip with one-another about patient "sightings". With many observers and many paths by which information can spread, we obtain a patient location-tracking database at low cost, and with guarantees of extremely robust behavior even in the event of a disruptive condition, such as a malfunctioning application that generates extremely high network loads and loss rates. (Recall from our discussion of Astrolabe that a gossip management infrastructure might help in this case too, by assisting the system administrator in localizing the problem).

What about high-speed event notification and streaming? Our system could exploit this functionality in a great many ways. If we assume that health care records are, in effect, replicated throughout the system as a whole, when an update occurs, it will be important to consistently update all copies. Here we see a form of event notification that requires relatively strong reliability and delivery semantics – corresponding to a consensus-based model such as virtual synchrony or state machine replication, both available within Quicksilver as group "types". Event notification can support a publish-subscribe relationship between the database servers in the hospital and client systems operated in private practices and other satellite locations. Bedside or nursing station display systems may need to be refreshed. Similarly, if the update is relevant to a patient's prescriptions, the event might be pushed out to participating pharmacies. One can also imagine high-throughput event channels. For example, television cameras and other sensors monitoring infants in a neo-natal unit could stream images to the nursing station; pediatricians would be able to subscribe as necessary to keep an eye on their patients: a robust, scalable IP-TV architecture

The Quicksilver properties mechanisms would be beneficial here, by permitting the system to match the properties of each type of event channel, or live object, to the requirements associated with that category of object. In fact we doubt that there would be a huge number of cases, but there are clearly subsystems that would value real-time data delivery over other guarantees, subsystems that need the sorts of consistency afforded by virtual synchrony or state machine replication, and subsystems that need transactional "ACID" properties. These can all be supported, side-by-side, on a per-event-channel basis.

These example illustrates a point worth reiterating: by using the publish-subscribe paradigm, the publishing side of the enterprise can be designed independently from the data consuming side; both can be incrementally extended over time as new applications are added, and will automatically accommodate varying runtime configurations. In effect, we are able to separate the information representation standards used within the system (including the hierarchy of topics) from the data sources and the data consumers. The communications infrastructure provides the needed guarantees, and when a new component is introduced, existing event-generating applications don't need to be modified. Because Quicksilver has a strong notion of types associated with event channels and live objects, we can do far more type checking than is traditionally feasible in publish-subscribe settings. For example, we can potentially ensure that the properties of a channel match the expectations of the application that binds itself to that channel. Moreover, to the extent that we need instant detection and reaction to a failure, because Quicksilver extends the publish-subscribe eventing model to also offer (optional) information about subscription changes when processes join and leave a channel, all sorts of rapid fault-tolerance mechanisms can be implemented.

We've avoided discussion of privacy and security issues, despite their central importance in electronic health care systems. This is in part because Quicksilver currently lacks a comprehensive security architecture, although we do have some ideas for how we might build one. Our thinking is to focus on capabilities enabled by the secure replication of security keys using the algorithms of Reiter [8][9] or Rodeh [7]; these offer ways to refresh keys when the set of nodes in the replication group (the event channel) changes because of a failure or a join. However, prior research has never explored scalability implications of these kinds of secure key replication schemes, and we believe the topic will require a substantial research effort to fully resolve. Use of security keys in gossip settings represents an additional intriguing option for study.

7. Research topics

Our vision raises a number of questions:

1. Given a proposed large-scale application, what is the most effective development methodology for mapping it down to application-specific functionality, as opposed to platform-supplied functionality? How should the developer make decisions concerning the aspects that are best matched to gossip communication, those best matched to event notification, and those that require hand-coded logic? Given that both gossip and event notification systems can support "guaranteed" properties, how should the developer decide which properties are needed by a given application, and how best to achieve them? Is there are large-scale methodology for specification of overall properties of a complex system that might lend itself to a formal verification process analogous to the ones used to reason about and ultimately prove correctness for non-distributed systems? Can the properties mechanisms used in Quicksilver today be extended to include gossip protocols?
2. If a single computer system supports multiple "live" data objects, high performance often requires that protocols be designed to amortize costs. Much of the innovation in Quicksilver is at this level: the system looks for ways to disseminate data, recover from packet loss and control data rates that are aggregated across potentially huge numbers of objects. When we introduce new classes of objects supported by gossip, the gossip infrastructure will need to address similar questions.
3. We alluded to the need to secure the platform, and to the risk that gossip mechanisms might be incapacitated by certain kinds of malicious behaviors. Our architecture poses significant opportunities for research on security, ranging from questions of precisely how one might secure a gossip protocol to broader issues of scalability that arise if an application subscribes to a large number of secured objects. How should one secure a high-speed event channel? What issues arise as one scales a security abstraction in a setting where each separate event channel or live object might have its own security requirements?
4. The creation of appropriate abstractions for the gossip infrastructure is an important challenge. At the lowest level, one imagines mechanisms for random peer selection, state exchange and merge, aggregation, etc. Ideally, these should be highly standardized. Yet some gossip protocols bias peer selection, implement "tricky" state exchange/merge mechanisms, or perform aggregation in unusual ways. Needed is a platform that can function well as a black box, and yet that can also expose functionality as needed.
5. We need to better understand the correct set of gossip mechanisms needed for purposes of self-management and self-configuration in Quicksilver. The modern internet is complex, and while it is easy to evoke a vision of an autonomic infrastructure that can support plug-and-play behavior in almost arbitrary settings, implementing that vision is quite a different matter.
6. Applications running on the event notification infrastructure will also need self-management and self-configuration functionality. Quicksilver's needs are somewhat peculiar to its role; will the same autonomic mechanisms that work for Quicksilver be adequate for other purposes, or are other kinds of gossip tools needed?
7. Obtaining high performance in large-scale settings that involve managed frameworks (C# in .net, in our case) is surprisingly hard [3]. It is likely that we will need to overcome similar challenges as we implement a gossip-based infrastructure and then tune it to cooperate cleanly with Quicksilver.
8. We commented that one key to scalability in Quicksilver is the mapping of event channels down to regions of approximate overlap – sets of nodes with similar subscription sets. A basic assumption underlying the system is that this can actually be done and that large systems will exhibit high degrees of overlap, or at least that they can be designed to have this property. But how can overlap regions be discovered in the first place? We are thinking that gossip mechanisms could be very useful in discovering applications and their "potential" subscription sets, enabling an offline analysis (perhaps with a human designer in the loop) to identify regions of overlap and configure Quicksilver. In contrast, the alternative of trying to discover regions at runtime by analysis of subscription patterns as programs come and go raises a number of thorny problems and may not be the best approach.

9. Conclusions

Scalable event notification systems capable of offering strong properties may be the key to enabling a new generation of trustworthy distributed applications, but only if they can be integrated naturally into the most powerful development environments and made autonomic: self-monitoring, self-configuring, and self-managing. For these latter purposes, we propose to build a new kind of distributed abstraction that embeds into Windows much like a typed object, but can be supported either by Quicksilver's scalable event architecture or by gossip-based protocols. A system realizing this vision is now under joint development at IRISA/INRIA in Rennes and at Cornell University.

7. References

[1] Reliable Distributed Systems Technologies, Web Services, and Applications. Birman, Kenneth P.

2005, XXXVI, 668 p. 145 illus., Hardcover ISBN: 0-387-21509-3

[2] A Review of Experiences with Reliable Multicast. K. P. Birman Software Practice and Experience Vol. 29, No. 9, pp, 741-774, July 1999

[3] Implementing Scalable Publish-Subscribe in a Managed Environment. Krzysztof Ostrowski, Ken Birman. In Submission (November, 2006).

[4] QuickSilver Scalable Multicast. Krzysztof Ostrowski, Ken Birman, and Amar Phanishayee. Cornell University Technical Report TR2006-2063 (April, 2006).

[5] Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. Robbert van Renesse, Kenneth Birman and Werner Vogels. ACM Transactions on Computer Systems, May 2003, Vol.21, No. 2, pp 164-206

[6] Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. Krzysztof Ostrowski, Ken Birman, and Danny Dolev. Submitted to International Journal of Web Services Research.

[7] The Architecture and Performance of the Security Protocols in the Ensemble Group Communication System. Ohad Rodeh, Ken Birman, Danny Dolev. Journal of ACM Transactions on Information Systems and Security (TISSEC). Vol. 4, No 3, pp 289-319, Aug 2001

[8] A Security Architecture for Fault-Tolerant Systems. Michael K. Reiter, Kenneth P. Birman, Robbert van Renesse. ACM Trans. Comput. Syst. 12(4): 340-371 (1994)

[9] How to Securely Replicate Services. Michael K. Reiter, Kenneth P. Birman. ACM Trans. Program. Lang. Syst. 16(3): 986-1009 (1994)

[10] T-Man: Gossip-based overlay topology management. Mark Jelasity and Ozalp Babaoglu. In ESOA 2005, Revised Selected Papers, vol 3910 of LNCS, 1-15.

[11] Gossip-based aggregation in large dynamic networks. Mark Jelasity, Alberto Montresor and Ozalp Babaoglu. ACM Transactions on Computer Systems, 23(3): 219-252, August 2005.

[12] The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. Mark Jelasity, Rashid Guerraoui, Anne-Marie Kermarrec, Marteen van Steen. Middleware 2004, volume 3231 of LNCS, 79-98, Springer-Verlag, 2004.

[13] Sub-2-Sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks. Spryros Voulgaris, Etienne Riviere, Anne-Marie Kermarrec and Maarten van Steen. Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS), Santa-Barbara, CA, February 2006.

[14] Epidemic-style Management of Semantic Overlays for Content-based Searching. Spyros Voulgaris and Maarten van Steen, Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par), Lisbon, Portugal, August 2005.

[15] CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. Spyros Voulgaris, Daniela Gavidia, Maarten van Steen. Journal of Network and Systems Management, vol. 13(2):197-217.

[16] Ordered Slicing of Very Large-Scale Overlay Networks. Mark Jelasity and Anne-Marie Kermarrec. In The Sixth IEEE Conference on Peer to Peer Computing (P2P), Cambridge, UK, 2006.

[17] GosSkip, an Efficient, Fault-Tolerant and Self Organizing Overlay Using Gossip-based Construction and Skip-Lists Principles. Rachid Guerraoui, Sidath Handurukande, Kevin Huguenin, Anne-Marie Kermarrec, Fabrice Le Fessant and Etienne Riviere. In The Sixth IEEE Conference on Peer to Peer Computing (P2P), Cambridge, UK, September, 2006.

[18] From Epidemics to Distributed Computing**.** Patrick Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. IEEE Computer, 37(5):60-67, May 2004.

[19] Lightweight Probabilistic Broadcast. Patrick Eugster, Sidath Handurukande, Rachid Guerraoui, Anne-Marie Kermarrec, and Petr Kouznetsov. ACM Transaction on Computer Systems, 21(4), November 2003.

[20] Probabilistic Reliable Dissemination in Large-Scale Systems**.** Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. IEEE Transactions on Parallel and Distributed Systems*,* 14(3), March 2003.

[21] BAR Gossip**.** Harry Li, Allen Clement, Edmund Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, Mike Dahlin, Proceedings of the 2006 USENIX Operating Systems Design and Implementation (OSDI), Nov 2006 .

[22] Epidemic algorithms for replicated database maintenance. Alan Demers, Dan Greene, Carl Houser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, Doug Terry. ACM SIGOPS Operating Systems Review Volume 22 , Issue 1 (Jan., 1988), 8 - 32