# Scalable Multicast Platforms for a New Generation of Robust Distributed Applications

Ken Birman, Mahesh Balakrishnan, Danny Dolev, Tudor Marian, Krzysztof Ostrowski, Amar Phanishayee

## ABSTRACT

As distributed systems scale up and are deployed into increasingly sensitive settings, demand is rising for a new generation of communications middleware in support of application-level critical-computing uses. Ricochet, Tempest and QuickSilver are multicast-based systems developed to respond to this need. Ricochet and QuickSilver are multicast platforms; both are exceptionally scalable and support fault-tolerance properties that match closely with the needs of high-availability applications. *Ricochet* was designed to support time-critical applications replicated for scalability on data centers and clusters. These are typically coded in Java and run under Linux. *Tempest* is layered over Ricochet and automates most tasks of programming services for data centers. In contrast, *QuickSilver* focuses on high throughput and is targeted towards very large deployments of desktop computing systems, in support of publish-subscribe, event notification or media dissemination applications. In this paper we offer an overview of the systems and some of the new systems embeddings that, we believe, make them far easier to use than was the case in prior multicast platforms.

## I. INTRODUCTION

Distributed computing systems are confronting a wide range of challenges associated with limits of the prevailing service oriented architectures and platforms. By *service oriented architectures*, or SOAs, we refer to Web Services, CORBA and J2EE: the most popular standards for interconnecting client computing systems with servers over the Internet. These invite the developer to interconnect systems that previously have been relatively incompatible. They standardize such aspects as connection establishment, synchronous and asynchronous request invocation, and representation of application data in messages. They support transactions, and a reliability model based on message queuing middleware. SOAs make it so easy to integrate applications that developers with almost no specialized distributed computing skills can now create sophisticated distributed systems. Industry leaders such as Bill Gates and Larry Ellison speak of a services "revolution", and moves are underway to create new kinds of electronic health record systems, military command and control systems, government services systems, banking systems and a host of other similar applications that will use these standards.

Yet as we deploy new generations of sensitive computing applications on SOAs, we'll face tough technical challenges. Some of these systems will require high availability, and hence need a way to replicate data and application state. Distributed security and management applications need ways to sense events occurring at many locations system-wide and to report those events to groups of actuators where appropriate actions may be triggered. Some applications need to stream data at high rates to groups of recipients, a pattern seen in collaboration systems, computer gaming, embedded control systems, IPTV and other media delivery systems. Some applications must guarantee rapid responsiveness, even as they scale up to accommodate potentially high loads. Adaptive mechanisms are required to handle surges in demand. Self-monitoring/self-repair mechanisms are needed to automate problem diagnosis and repair after a crash or other disruption occurs. Distributed security architectures need ways to replicate keys or security policies at groups of endpoints where those keys will be used to encrypt or decrypt traffic, and to update those kinds of information as conditions change.

Unfortunately, existing development platforms offer limited options for solving the kinds of distributed communications problems just mentioned. Our belief is that this is primarily a result of a communications issue: the service architecture standards have embraced point-to-point TCP connections to an overwhelming degree. By placing such a strong emphasis on the client-server data pathway, SOAs have been inattentive to server-to-{set-of-clients} and server replication data patterns that arise in the cases just enumerated.

Thus, for large classes of applications, a modern developer faces a tough choice. One option is to work within the existing architectural standards and tools, but this implies that the system will be limited to the kinds of scalability options available in three-tier database architectures. A second option is to simply abandon such goals as fault-tolerance, scalability, distributed coherent caching for high performance, or certain forms of distributed security. And the third option is to build

[1] Contact information for first author: ken@cs.cornell.edu; Dept. of Computer Science, Cornell University. Dolev was on sabbatical from the Hebrew University in Jerusalem, and Phanishayee is now a graduate student at CMU. This work was support in part by grants from AFRL, AFOSR, DARPA, NSF and Intel Corporation.

some form of non-standard, home-brew solution that departs from the most widely used standard tools and platforms, and hence will make the application harder to build, harder to deploy and operate, and ultimately more costly.

At Cornell, a new research project is underway that seeks to break through these barriers by creating a fourth option. Our goal is to create new communication solutions that:

- Can be elegantly embedded into existing service-oriented platforms, in a way that will make it as easy to exploit these tools as it is to build client-server systems today.
- Scale extremely well. One reason for the problems we've cited is that prior generations of reliable communications platforms scaled poorly (most critically, in the number of groups to which an application can belong). These systems need to scale in multiple dimensions: groups, processes, network size, data rates and message sizes.
- Permit reliability and quality of service options to be tailored to match application requirements.
- Perform extremely well. We seek to equal or exceed the performance developers could achieve with specialized hand-crafted solutions.

This paper is limited in length, making it impossible to discuss these issues while also providing technical details for the multicast platforms reported below, Ricochet and QuickSilver, and for Tempest, the development tool that runs on Ricochet. Accordingly, in this paper we limit ourselves to a broad overview that touches on the major innovations of the effort but omits specifics such as details of protocol designs, engineering decisions, integration with the host platforms (Windows and Linux), and evaluation. However, the interested reader will find this sort of detail for QuickSilver in [4, 3, 10, 11, 12], for Ricochet in [6, 7, 2] and for Tempest in [1]. These systems are available for no-fee download from Cornell's web server [9], with source provided on request.

## II. TARGET ENVIRONMENT

Early in our effort, we realized that even though multicast applications arise in many settings, the type of system appropriate for addressing server replication issues in a data center or cluster differs from the type of system needed in corporate LAN settings or the wide-area Internet.

### A. Ricochet

Modern data centers consist of large numbers of relatively homogeneous nodes interconnected with high-speed, low latency switched networks to host applications that are cloned. Doing so allows work to be load-balanced either as requests arrive directly from web clients, or after they are vectored through a front-end that parallelizes them by issuing concurrent requests to one or more services. The need for scalable multicast arises in the context of replicating the state of the cloned services. Most data centers are moving to web services standards, and Linux and Java are the most common platforms used to build new applications in such settings.

We targeted Ricochet to this environment, and started by asking ourselves what reliability, scalability and quality of service requirements arise in such settings:

- *Reliability.* We undertook a series of experimental studies aimed at quantifying the types of failures that actually occur in large data centers. From this we determined that packet loss is extremely uncommon in data center networks, but surprisingly easy to provoke within the relatively slower commodity operating systems that run on the inexpensive computing nodes popular in such settings. Moreover, when an overloaded operating system drops packets, it often drops several in a row; histograms of packet loss revealed that short bursty episodes of loss were common (these loss bursts rarely exceeded 20 packets). Node and application instability (crashes, freeze-up or temporarily sluggish operation) was also common. Knowing that these problems actually arise in practice, a reliable multicast communication layer should be designed to overcome them.

  *Scalability.* When services are developed using modern tools, it is common for what the end-user sees as a single service to be implemented as a set of communicating components. These are often grouped on a single node to exploit shared-memory or other high speed communication options. Thus, when a service is cloned to scale it out, it is common to see sets of components that must be cloned, such that one instance of each will run on each of a set of nodes. Moreover, caches are widely used to offload read-only work from heavily loaded servers such as databases or file systems, and these co-reside with applications that issue reads. Accordingly, if multicasts are used to update the caches and to update the cloned server states, nodes within a data center will often "belong" to large numbers of heavily overlapping communication groups. Publish-subscribe gives rise to a similar pattern, if one thinks of each topic as mapping to an underlying communication group.

  Accordingly, data centers that use multicast (or publish-subscribe) need support for communication in *large numbers of overlapping groups.* As the data center scales up, the aggregate load on the multicast system will rise; this makes it important that the implementation be fully decentralized. This is important because most off-the-shelf multicast products vector multicast communication through some form of central server. In the settings we care about, that server would become overloaded and emerge as a bottleneck. However, groups will rarely become very large, because it is rare to clone a service onto more than a few tens of nodes – in dialog with companies such as Amazon, Google, eBay and others, we've heard of a few examples of very large groups, but many examples that require smaller-scale replication.
- *Time-criticality and other quality of service properties.* A major end-user objective in porting a service to a data center is to achieve rapid responsiveness for a much larger workload than would be possible with a single instance of the service, even on a high-speed computing node. For

this reason, rapid response is a key metric in the eyes of users – even users who don't think of themselves as needing a "real time" solution. The insight is probably obvious: one buys extra nodes to maintain rapid response despite higher load. This time-criticality property is not supported in any existing system or product. Many applications also require ordered delivery. Only some applications require absolute reliability guarantees or "hard" real-time delivery; surprisingly often, a high quality solution that offers extremely good speed and scalability, but that might be unable to recover some small percentage of lost packets, is quite acceptable. Instead, one would prefer an end-to-end consistency repair mechanism that can detect and repair inconsistent clones; such a mechanism would do double-duty, fixing application level problems caused by bugs, but also kicking in if a message is dropped infrequently.

These goals and insights motivated our design of Ricochet and Tempest. Ricochet implements a new time-critical multicast protocol that offers tunable but high reliability, can recover from bursty packet loss, scales extremely well in the number of groups to which each node belongs, and is particularly notable for guaranteeing rapid message delivery with very predictable latency distributions. Tempest, built over Ricochet, automates the steps in cloning a web services application and provides end-to-end consistency checking and repair mechanisms that can recover the application if it suffers some form of transient fault.

At the core of Ricochet is a new time-critical repair mechanism that employs *lateral error correction* [6]. This is a scheme whereby a receiver periodically computes an error correction packet by XOR-ing together packets it receives across some subset of groups it belongs to and then sending that packet to other receivers that belong in that same set of groups (Figures 1, 2). IP Multicast is used to transmit messages unreliably, and the error correction packets enable Ricochet to recover lost messages very quickly. Receivers can reconstruct any single missing packet from the error correction packets they receive, and detect multi-packet losses. Using an appropriate "stride" (meaning that the messages XOR-ed together in any given LEC packet are sufficiently separate from one-another in the receive sequence), Ricochet also overcomes bursty loss.

Most lost packets are recovered by Ricochet within a few milliseconds through the proactive lateral error correction traffic. For the small percentage that can't be recovered proactively, Ricochet uses reactive negative acknowledgments to request retransmissions of the packet from the sender, achieving any desired level of reliability on the multicast data.

Ricochet achieves extreme scalability in the number of groups in the system, allowing nodes to join as many as a thousand groups without impairing loss recovery latencies. Whereas existing reliable multicast protocols discover packet loss with latency inversely proportional to the data rate at a single sender

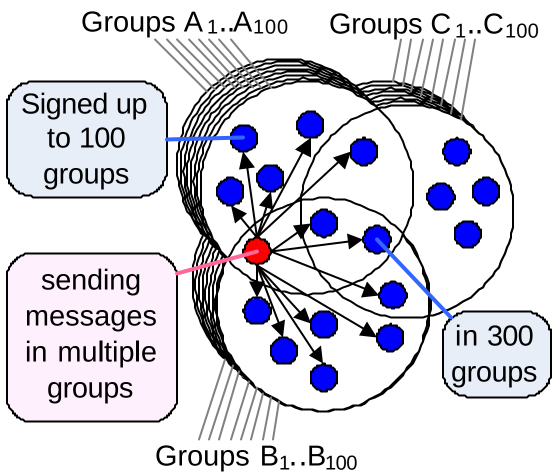

**Figure 1: A collection of overlapping groups, with a sender transmitting multicasts in two groups.**

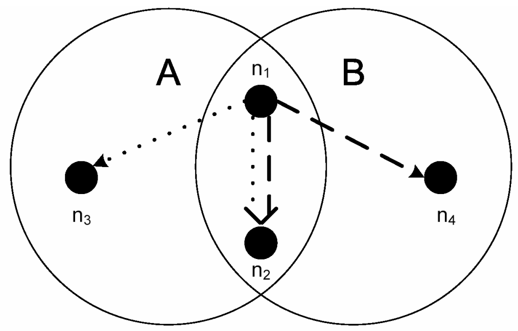

**Figure 2: Ricochet introduces "lateral error correction", whereby error correction data is proactively multicast by other receivers in overlap regions. Lost data is often recovered before the sender can detect the loss.**

in a single group, Ricochet's discovery latency is dependent on the total incoming data rate at a receiver, across all groups and all senders.

Ricochet does not provide ordered message delivery on its own – this is done by a separate ordering layer called PLATO [2]. Traditional ordering protocols deliver messages to the application only after conservatively establishing the delivery order. In contrast, Plato belongs to a class of protocols which *optimistically* deliver messages to the application and rely on *rollbacks* to undo incorrectly ordered message deliveries.

PLATO optimistically delivers only those messages for which it believes there to be little risk of out-of-order delivery. To assess this risk, it leverages several observations. Keep in mind that IP Multicast messages are delivered almost simultaneously at all receivers in a data center network. Thus, nodes are at risk of receiving messages in different order only when two messages from different senders are *sent at almost the same time*, or secondly, *if one receiver loses a burst of messages in its kernel buffer and another doesn't,* causing a gap in the delivery sequence. PLATO predicts the occurrence of these two events by buffering incoming messages long enough to observe the inter-arrival delays of consecutive messages. *A small delay between consecutive messages predicts heightened likelihood of out-of-order arrival.*

To see this, notice that in the first case just mentioned, if two messages are multicast by two senders at around the same time, they will arrive at each receiver with a small inter-arrival delay. In the second case, if a receiver experiences a buffer overflow in its kernel, it will subsequently attempt to catch up by rapidly dequeueing packets into application space, resulting in a sequence of arrivals with very low delays in between.

Hence, if two messages are separated by a sufficiently long gap, PLATO delivers the first without waiting for additional ordering information. On the other hand, if messages arrive in quick succession, PLATO withholds them from the application until it establishes the correct order. This is done by designating one group member as a *sequencer* node that periodically multicasts the canonical ordering for the group.

In practice, the PLATO scheme delivers a large fraction of messages with very little delay (2 or 3 milliseconds), a smaller fraction with the delay of a conservative ordering protocol (tens of milliseconds), and a miniscule fraction (<1%) with a delay of a hundred milliseconds or more, consisting of optimistically delivered messages which were later found to be in incorrect order and rolled back. Together, PLATO and Ricochet exploit the natural properties of the underlying hardware to provide very fast communication in the average case. The probabilistic techniques employed by the protocols scale very well in multiple dimensions.

### *B. Tempest*

It is important to embed solutions into appropriate platforms in natural ways. Ricochet is integrated with the Apache Axis2 web services platform, but many users prefer higher-level tools. Accordingly, we are building a new system called Tempest [1], which automatically transforms web services applications into cloned groups of servers (Figures 3, 4). These use Ricochet for update propagation, but also have a Tempest-provided mechanism for detecting crashes and restarting failed services, checkpointing, cold startup, and even detecting and repairing any data inconsistency that might arise at runtime. The repair mechanism can compensate in the unlikely event that Ricochet is unable to recover a lost packet, but can also detect and repair many application-level problems that leaves one clone inconsistent with the others.

Tempest accomplishes of this by intervening at two places. First, when a web service is invoked, Tempest substitutes its own invocation protocol for the one used by standard client-side web services platforms; this protocol captures the invocation, adds a unique identifier and some other information, and then sends it via Ricochet. The required stub can be inserted automatically in many cases. Second, Tempest intervenes within the service itself. The application developer is asked to manage any "distributed state" by storing it in a special collection classes inherited from the Tempest framework. The collection classes mimic the standard Java collections while the stored collection items mimic Java Beans, hence both should look familiar to many developers. Knowing where the state for the service is stored, Tempest can access it to make checkpoints, compare states between components, etc. The whole mechanism is almost completely transparent to the developer.
All aspects have been designed to ensure that a cloned time-

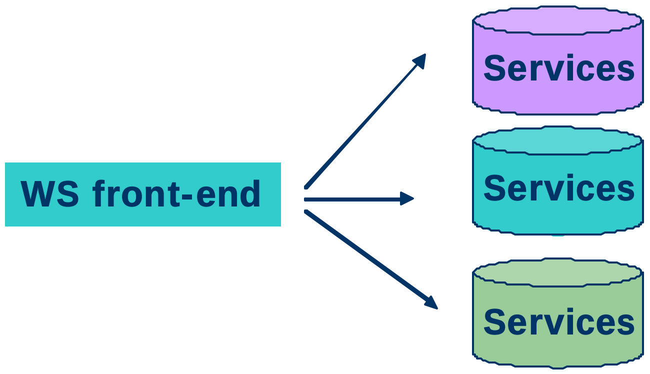

**Figure 3: A web service composed of a front-end and three back-end services.**

critical application will have excellent timing properties. Tempest thus enables componentized data center application development, where a programmer with few specialized distributed systems skills is able to build a high-performance, scalable, fault-tolerant, self-managed and self-repairing solution that scales out and achieves high performance even when subjected to heavy loads.

### *C. QuickSilver*

In contrast to Ricochet, QuickSilver focuses on platform support for new kinds of client applications in which group communication is used for event and media stream delivery, or for the other kinds of purposes outlined at the outset of this paper. Unlike the nodes in a data center, client systems are overwhelmingly desktop systems based on PC standards. These applications gain enormous flexibility from the component integration features of Windows and its .NET framework. Accordingly, QuickSilver targets this platform mixture as its primary runtime environment, although we do plan a port that will run in Linux settings under Mono. Our initial focus is on large enterprise LANs such as corporations might operate, but over time we do hope to tackle Internet WAN scenarios. In what follows, we discuss QuickSilver from the "ground up", starting with the network layer, then

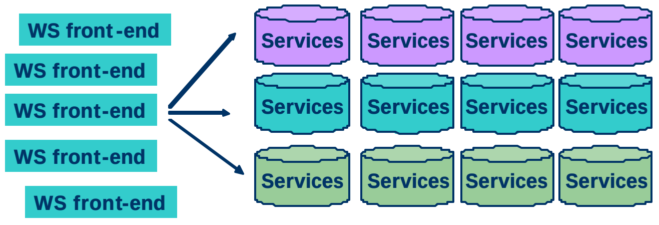

**Figure 4: Tempest transforms the application into a high-availability service that runs on a data center.**

discussing a new and flexible framework for endowing groups with reliability properties, and finally explaining how these are exposed to the end user through a new form of typed communication channel.

***Network layer.*** Up to now, we have discussed data centers. Enterprise LAN systems pose different goals and architectural considerations. For example, data center applications benefit from Ricochet's rapid message delivery in part because Tempest cleans up any inconsistencies that arise between clones. But in a LAN, the receivers in a group won't be clones and we can't fall back on Tempest to deal with inconsistencies. This means that the communication system will often need to support stronger reliability models, such as virtual synchrony, consensus, group authentication/security, or one-copy serializability (these models are discussed in [5]).

In data centers application placement is tightly controlled, hence groups will generally be small and will overlap in regular ways. QuickSilver lives in a world of LAN groups, which may be large and will often overlap in irregular ways. Ricochet is optimized for time-critical latency, but in a LAN, the more important metric is throughput. On the other hand scalability in the number of groups will be critical: the application programmer probably thinks of groups in terms of publish-subscribe topics, hence a single node could easily belong to an enormous number of groups, perhaps with slightly different reliability properties for each.

QuickSilver's network-layer protocols treat *data dissemination, basic reliability and rate control* separately, and provide strong properties such as the ones just mentioned at a higher layer. The *dissemination framework* is optimized to deliver messages using as few network operations as practical – ideally, a single IP multicast, and indeed, if possible, a single multicast into which multiple application-level messages have been packed. *Reliability* is implemented on a regional basis (similar to Ricochet), but uses a rotating token scheme, with a further subdivision of each region into a set of "partitions" to handle the case of a very large region: each partition has a token of its own, and a secondary aggregating token runs among leaders selected on a per-partition basis; a group membership service tracks node status and handles overall configuration. The token ring is also used by a novel *rate control mechanism* that replaces traditional ACK/NACK flow

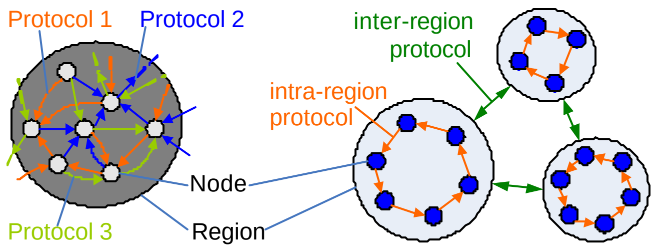

**Figure 5: QuickSilver amortizes costs over many groups. Lacking such an approach, chaos can occur, as is seen on the left, where each group builds its own recovery infrastructure (here, a tree-like overlay).**

control. Packet loss is detected using data tracked by the tokens, in a manner that aggregates across all the groups that map to any given region, and recovery is peer-to-peer, between nodes in the same region, with the sender involved only if an entire region is lacking a message (for more details, see [12]).

***Event Processing.*** Sustaining high performance in large-scale configurations involves a difficult balancing act. The network architecture described above is part of the story. But the actual handling of events such as timers, incoming messages, recovery tokens and control data was also critical. QuickSilver uses a single-threaded event-driven architecture to eliminate exposure to uncontrolled scheduler decisions. This freed us to design a scheduling policy to handle situations where multiple events occur simultaneously.

We mentioned earlier that inexpensive computing platforms can easily be overwhelmed by high rates of incoming messages, triggering bursty packet loss in the kernel. Recall that Ricochet uses proactive repair packets to overcome such losses: this incurs higher overhead, but allows Ricochet to repair packet loss before most platforms would even detect a problem. QuickSilver adopts a different approach: the system is designed to minimize such occurrences using rate controllers that estimate maximum sending rate and try to match it, lowering the overall control traffic and system overhead, but also pulls packets from the kernel as rapidly as possible, reducing the risk that the socket queue might overflow.

Because QuickSilver cannot completely avoid transient disruptive events such as garbage collection and scheduling, it is not unusual for this event handler to confront long bursts of queued packets. The question arises of which ones to process first. It might seem as if a FIFO strategy would be best, but in fact we found that such policies result in poor performance.

Experimentally, we discovered that versions of our QuickSilver protocols were prone to convoy phenomena, whereby throughput oscillates under heavy load. Perhaps this should not be a surprise: researchers who developed earlier systems have often suggested that such phenomena represent fundamental barriers to scalability with high data rates. As it turned out, however, we were able to control the problem, at least under most conditions.

The approach we adopted is best understood by thinking in terms of traffic on a high-speed throughway. Such a road can achieve high capacity if cars are widely spaced but moving rapidly. Closely spaced cars, however, are prone to virtual traffic jams: one vehicle slows; the one behind it brakes reactively, and a wave of congestion sweeps backwards up the highway, ultimately creating a strange form of stop and go traffic that may persist for long periods of time.

In QuickSilver, the "throughway" is the communication platform itself, and the vehicles are the messages. We want these to pass through our platform as smoothly as possible without lingering. On the other hand, some forms of backlogs are unavoidable. As just noted, if a receiver is temporarily busy, messages pile up in the kernel; when QuickSilver resumes execution, it reads all the queued data out of the kernel before processing any of it to minimize the risk of kernel buffer overloads. This tactic reduces the frequency of packet loss, but doesn't eliminate it; indeed such packet loss that *does* occur will probably happen precisely at this moment. Moreover, if a message *was* lost, QuickSilver must obtain the missing data before it can deliver subsequent packets.

Small bursts of packets won't pose much of a problem, but longer convoys of delayed multicasts can have an impact that snowballs through the system, causing nodes to act upon stale state, and leading to redundant recovery traffic that further aggravates the problem. These phenomena cause fluctuating throughputs: they corresponded to periods of low throughput when memory consumption soars, followed by periods of very high and bursty throughput, mostly wasted on largely redundant loss recovery. Our challenge is to minimize the probability of this disruptive scenario.

Event prioritization is the key to doing so. In QuickSilver, the highest priority task is to maintain a good degree of state synchrony between nodes. If node A has a good idea of the state of node B, A is unlikely to perform costly but inappropriate actions such as sending repair packets to B when B has long-since repaired missing data. On the other hand, when data really does get dropped, we need to prioritize repair packets over "new" multicasts, since those new messages must wait for the gaps to be filled, in any case.

Accordingly, QuickSilver defines a three-level prioritization scheme. Arriving messages are pulled rapidly from the kernel, and then classified. The tokens QuickSilver uses to track node status within each region are dispatched first. Next, repair packets are dispatched and last, new incoming multicasts. When combined first with rate control mechanisms that limit the sending rate so that network interface cards and other hardware components won't get overloaded and second with mechanisms for delaying the creation of messages until just before their transmission ensuring that all transmitted packets reflect the most recent state of the sender, this rather simple "innovation" eliminated performance fluctuations. Details on this and other engineering insights gained while building QuickSilver are discussed in [4]; many should be applicable in other systems.

*Typed group endpoints.* Just as the integration of Ricochet with web services is important to making it easy for developers to use, we believe that the integration of QuickSilver with the powerful component integration framework supported by Windows and .NET could make the technology accessible to a very large user community that needs multicast and replicated data. Accordingly, QuickSilver was developed to run as a managed application under the .NET framework. This brings immediate advantages: in Windows, end-user applications and systems have access to very transparent component integration tools, and benefit from high performance and strong type-checking if all the components run within the managed framework. But doing so also poses challenges. Windows achieves much of the power of its integration framework by leveraging type information. Yet the common language runtime environment in Windows lacks any notion of a typed communication channel expressive enough to capture notions such as groups with strong reliability properties (like virtual synchrony), strong security properties, or other behaviors.

To bridge this gap we implemented a new kind of typed communication channel that integrates seamlessly into Windows. Elements of the solution include a component integration technology, somewhat similar to web services, but designed for multi-point channels and capable of expressing semantically-rich contracts, and a form of "shell extension" that serves as the user interface. The idea is similar to the mechanism used by Windows to bind file extensions to programs supporting the corresponding file type. Basically, each group has an associated type, which describes the functionality and features offered by the group as well as requirements placed on the clients. Similarly, applications expose typed interfaces, which are verified against group types by the runtime. This type system is accompanied by compiler tools for generating and importing type information associated with remote objects, offering the developer signature-based application development and debugging support of the sort familiar to any visual studio user, as well as static and dynamic type checking.

We are also able to support a novel form of user-mediated component integration, which works in a manner familiar to users of Windows applications such as "Winzip," "News" or the "MyPhotos" subsystem. Basically, we are able to portray available group communication topics as a namespace that can be browsed by the user. Right-clicking on a topic – a communication group – brings up a list of applications that can bind to this topic. The latter is determined by comparing the type signatures of known types of applications with the type signature of the topic. For example, a user could bind the camera on a PC to a group through an application that periodically snaps a photo and multicasts it. Various data consuming applications would be coded separately and registered with the runtime – for example, an application that reads a photo from a group channel and then displays it as a new desktop background image. The user would assemble the desired configuration by a few mouse-clicks – in our example, creating a live desktop background that updates each time a new image is multicast in the group. Similarly, an incoming video feed could be bound to a web page, a stock "ticker" to an automated trading application, etc.

*Extensible properties framework.* Supporting these application-visible typed group endpoints is a subsystem we call the "properties framework" ([3]). This framework supports a simple programming language that allows us to express strong reliability properties (and, ultimately, security or other properties) at a high level, similar in spirit to Lamport's Temporal Logic of Actions (TLA [8]). The idea is that each different group can have its own associated properties, defined through rules in the properties language. Thus, virtual synchrony can live side by side with weaker properties (such as the basic best-effort reliability property of QuickSilver's underlying scalable multicast), or stronger but more costly ones such as state-machine consensus or even transactional one-copy serializability, which requires synchronization with persistent storage.

The framework automatically generates and deploys a hierarchical protocol for each communication group based on a set of generic mechanisms. The latter include traditional "features" such as failure detection, tracking membership or state transfer, along with a mechanism for aggregating and disseminating control variables, some of which may have characteristics such as monotonicity or atomicity, and either be tied to local actions, or produced according to a set of rules. Because different protocols use shared mechanisms, overhead across protocols can be amortized as in [4]. By allowing properties to be defined on a per-group basis, one can imagine applications in which different subsystems use different properties, yet benefit from a single shared framework.

To give the reader some intuition, in Table 1 and Table 2 we list example properties and rules for a simple topic with a last-copy recall semantics, i.e. such that messages are recovered in a peer-to-peer manner, and eventually cleaned. Each property has a name. A single instance of a property with such name and for a given topic may exist at every level of a hierarchy, ranging from nodes, through groups of nodes, up to the entire topic. "Received" in the context of a node will thus represent the set of messages received by that node, and in the context of a group of nodes, messages received at any of them.

Default behaviors include aggregation (Agg) and propagation (Msg). If a property is aggregated, its value at a given scope is periodically aggregated over values in the sub-scopes of that scope. This way, the value of "Received" in a LAN could be aggregated over all nodes in it. The values of this property are sets of message identifiers. Parameter "$\cup$" specifies that aggregation is done by merging such sets. Parameter "+" specifies that the new values of a property should be merged with the existing ones. For dissemination, "*" specifies that values are propagated to each of the sub-scopes, and "select" that they are split among the sub-scopes.

An important type of constraint is monotonicity ("Mono"). If a property is monotonic, new values have to be aggregated over values at least as fresh as those used in previous aggregations. Additional constraints are placed also on members joining a topic. In our example, applying this behavior to the "Stable" property guarantees that it is non-decreasing, which prevents message cleanup before it is delivered to all nodes. Bindings determine how values of a property are bound to the actions of the application. In our example, most properties are bound to "standard" actions *recv*, *fwd* and *clean*, representing the receipt, forwarding and cleanup of a message, respectively.

Finally, rules define how values of properties can be generated from existing values of other properties. Rules are periodically executed either at each scope ("*"), or only at nodes ("local") or for the entire topic ("global"). Target property is assigned a value obtained by evaluating the given expression. Properties in rules can be unqualified, representing the local instance of a property at the scope at which a rule is executed, or qualified, with "x" or "y", keywords representing two halves into which the scope can be subdivided. In the latter case, the rule applies to every possible sub-division of the scope.

| **Name** | **Value Type** | **Behaviors** | **Bindings** |
|---|---|---|---|
| Received | IdSet | Agg($\cup$) | Get [recv] |
| Cleaned | IdSet | Agg($\cap$) | Get [clean] |
| Cached | IdSet | | |
| Stable | IdSet | Agg($\cap$), Mono | |
| Fwd | Addr×IdSet | +Msg(select) | Add [fwd] |
| NoFwd | Addr×IdSet | +Msg(*) | Del [fwd] |
| Clean | IdSet | +Msg(*), +Agg($\cup$) | Set [clean] |

**Table 1. The properties for a very simple topic with peer-to-peer message forwarding and cleanup.**

| **Scope** | **Property** | **Expression** |
|---|---|---|
| * | Cached | Received \ Cleaned |
| * | Fwd(x) | <y, Cached(x) \ Received(y)> |
| * | NoFwd(x) | <y, Received(y)> |
| local | Stable | Received |
| global | Clean | Stable |

**Table 2. The rules to accompany the properties in Table 1.**

At the start of this section, we commented on our scalability goals. QuickSilver scalable multicast achieves scalability for reliable multicast delivery, but this doesn't imply that the properties framework will also scale well. Accordingly, an important near-term goal for us is to develop scalable implementations for the important classes of properties (such as the ones listed above). We are focusing on virtual synchrony, state machine replication and transactions, and will report our experience in a future paper.

*Status.* Ricochet is available today, as is QuickSilver scalable multicast; the Tempest system and the full QuickSilver platform should both be completed sometime in late 2006. Once QuickSilver's properties framework and typed endpoint mechanisms are working our plan is to integrate the platform with presentation layers such as the web services eventing API (it may be necessary to extend this standard, however, since as explained in [10], it is written in a way that precludes true multicast communication), toolkits for multiuser game development, collaboration tools, and so forth. Beyond the end of 2006, our focus will shift to security: we hope to show that prior work on group security can be made more scalable using the mechanisms available within the properties framework.

## III. PERFORMANCE

Our effort has emphasized performance, in the belief that ease and speed of system development lures users, but that raw speed and scalability are central to holding them over time. However the most relevant metrics for performance depend very much on the system and setting.

For our work on Ricochet and Tempest, the central goal is to support a new kind of scalable time-critical system that will be hosted on commodity data centers. Accordingly, the metrics against which we measured our own accomplishments revolve around these characteristics. In [6], we show that when compared with other multicast substrates having similar reliability properties, Ricochet can achieve as much as a three orders of magnitude reduction in multicast delivery latency while guaranteeing a very high (albeit probabilistic) level of delivery reliability. Moreover, the distribution of delivery latencies is smooth, offering the designer an intuitively appealing model against which to work.

For Tempest, the question is not so much one of delay until Ricochet delivers a message, but rather the perceived "quality of service" for the services replicated using the methodology. Although our work has not yet been completed, we have already found that even with very simple gossip-based consistency protocols, Tempest can augment the basic Ricochet guarantees into higher level service guarantees [1]. These provide external clients of the service with extremely high levels of perceived reliability and extremely good timeliness properties, even when nodes fail or are restarted while the system is in use.

Performance in QuickSilver focuses not so much on delay as on raw throughput and stability under stress. In [4] we report on experiments that stressed a 110 node QuickSilver configuration with multicasts in up to 8000 active groups, and showed it to tolerate well the common types of perturbances, including bursty loss, node failures, churn, and several others. Performance of the system with the full properties framework in use has not yet been investigated, and represents our next major target.

The work on QuickSilver reliable multicast showed that even with small numbers of senders, we can sustain throughputs close to 100Mbits/second, the speed of the cluster interconnect we used. We were able to send roughly 9,000 1k byte packets per second (and keep in mind that in many multicast systems, a single message can contain multiple updates). Memory consumption rises with the number of groups to which a node belongs, but without becoming enormous, and memory costs are sublinear in the number of groups. Moreover, although our experimental configuration only had 110 nodes, the scalability of the system seems to be extremely good, with no evidence of insipient performance problems. We hypothesize that similar performance could be achieved even with many hundreds of nodes – and we hope to experiment with such configurations as a serious user community emerges.

## IV. CONCLUSIONS

New styles of distributed computing are emerging, and with them the need for new and more powerful communications options has arisen. The lack of solutions is inhibiting the development of reliable, secure, self-managed applications, and yet the displacement of critical applications to distributed settings demands that we build such applications. We believe that the Ricochet, Tempest and QuickSilver platforms shed light on the real nature of the problem, and offer a possible path to solutions that could be broadly useful even for developers who lack any sort of special training in the theory and development of reliable distributed systems and protocols.

All aspects of our work are available under public licenses, and we welcome potential collaborators who might be in a position to deploy QuickSilver or Ricochet in demanding settings. Download instructions can be found at [9].

## IV. ACKNOWLEDGEMENTS

We are grateful to Greg Morrisett, Robbert van Renesse, Einar Vollset and Maya Haridasan, who offered a great many comments and suggestions over the course of the project.

## V. REFERENCES

1. **A Scalable Services Architecture**. Tudor Marian, Ken Birman, and Robbert van Renesse. To appear in Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS 2006). Leeds, UK. October 2006.
2. **PLATO:Predictive Latency-Aware Total Ordering.** Mahesh Balakrishnan, Ken Birman, and Amar Phanishayee. To Appear in Proceedings of the SRDS 2006: 25th IEEE Symposium on Reliable Distributed Systems, Leeds, UK. October 2006.
3. **Properties Framework and Typed Endpoints for Scalable Group Communication.** Krzysztof Ostrowski, Ken Birman, Danny Dolev. In Submission (July, 2006).
4. **QuickSilver Scalable Multicast.** Krzysztof Ostrowski, Ken Birman, Amar Phanishayee. Cornell University Technical Report (April, 2006).
5. **Reliable Distributed Systems Technologies, Web Services, and Applications**. Birman, Kenneth P. 2005, XXXVI, 668 p. 145 illus., Hardcover ISBN: 0-387-21509-3
6. **Ricochet: Low-Latency Multicast for Scalable Time-Critical Services**. Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, and Stefan Pleisch. Cornell University Technical Report.
7. **Slingshot: Time-Critical Multicast for Clustered Applications**. Mahesh Balakrishnan, Stefan Pleisch, Ken Birman. IEEE Network Computing and Applications 2005 (NCA 05). Boston, MA
8. ***Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*****.** Leslie Lamport. Addison-Wesley, 2003.
9. The QuickSilver project web site at Cornell, for downloads**:** **http://www.cs.cornell.edu/projects/QuickSilver/**
10. **Extensible Web Services Architecture for Notification in Large-Scale Systems.** Krzysztof Ostrowski, Ken Birman. To appear in International Conference on Web Services (IEEE ICWS 2006).
11. **Scalable Group Communication System, for Scalable Trust.** Krzysztof Ostrowski, Ken Birman. To appear in The First Workshop on Scalable Trusted Computing (ACM STC 2006).
12. **The Power of Indirection: Achieving Multicast Scalability by Mapping Groups to Regional Underlays.** Krzysztof Ostrowski, Ken Birman, Amar Phanishayee. Cornell University Technical Report (November, 2005).