Reliable Multicast for Time-Critical Systems

Mahesh Balakrishnan and Ken Birman
Cornell University, Ithaca, NY-14853
{mahesh, ken}@cs.cornell.edu

Abstract— **We are interested in communication support for time-critical reliable computing. Over a two-decade period, the distributed computing community has explored a number of reliable communication models. Yet time-critical applications in which rapid response matters more than absolute reliability have received comparatively little attention. We believe that this category of application is becoming common, and propose a new probabilistic reliability model for time-critical communication.**

I. INTRODUCTION

We suggest in this paper that reliable, time-critical communication is worthy of renewed attention. Advances in hardware capabilities, coupled with the growing popularity of cluster-style data centers, are creating new scenarios demanding reliable communication protocols that offer strong probabilistic guarantees of timely delivery.

To illustrate this point, we offer an analysis of the communication needs of the French Air Traffic Control System, with which we gained familiarity during 1995-2000, when we assisted developers charged with architecting and implementing subsystems that replicate data. Those subsystems mixed real-time needs with a requirement for consistency and availability, and were built using the Isis Toolkit [1][4]. The system is used throughout France and is a candidate for wide deployment in Europe.

At the time, the needs of an ATC system seemed atypical: few other systems of that period were implemented as structured application clusters running on large, distributed data centers with dedicated high-speed LAN hardware. However, the economics of computing have shifted. Recent dialog with companies operating very large data centers reveals that the time-critical communication needs of ATC systems are no longer at all unusual [10]. Indeed, perhaps surprisingly, even e-tailer data centers seem to be increasingly turning to a style of computing in which timely response is a primary goal [7]. In this paper, we argue that time-critical communication deserves a fresh look. Our Ricochet and Tempest projects are exploring this new model.

II. THE EVOLVING NEED FOR RELIABLE MULTICAST

A basic premise of our work is that the reliable communication needs of applications are heavily shaped by the hardware platforms on which they will be hosted. Until fairly recently, reliable computing was the provenance of server platforms, and servers addressing time-critical computing needs were typically hosted on dedicated high-performance processors, perhaps with specialized fault-tolerance features. Research on real-time responsiveness yielded new schedulers, operating systems, and application development styles.

Much of this work defined "real time computing" in ways focused on meeting deadlines and guaranteeing predictability subject to some worst-case fault scenario. For example, real-time multicast protocols such as Δ-t multicast [6] ensure ordered, nearly synchronous message delivery despite failures, at the cost of sending data redundantly. Delivery is delayed to ensure that even a worst-case failure pattern can be tolerated. The weakness of this definition of real-time and this style of "pessimistic" protocol is that in practical settings, the average-case overheads and delays required to overcome the worst imaginable failure pattern can be very high.

During the same period, a second style of reliable multicast was also explored, emphasizing fault-tolerance and consistency, but without real-time guarantees. This yielded models such as virtual synchrony (used in the Isis system), state machine replication (Paxos), and Byzantine replication (PRACTI replication), and platforms implementing these models (for details, see [5]).

III. THE FRENCH AIR TRAFFIC CONTROL SYSTEM

The experience of the designers of the PHIDIAS subsystem of the French Air Traffic Control System will illustrate the key insight behind our paper [8] [4]. The system was designed to support teams of three to five controllers using desktop computing systems to work as a group directing flights within some sector of an air space. An air traffic control center is supported by between fifty and two hundred of these console clusters. Centers operate autonomously, but are linked, sharing flight plans and other data.

PHIDIAS consists of a number of modular subsystems. Some interesting subsystems include:

1. *Radar image subsystem.* The radar devices used by the ATC systems multicast radar imagery at a fixed frequency (one image every few seconds). This "raw" data is typically shown as a background image on which other ATC data is superimposed.
2. *Weather alert subsystem.* This is one of a class of subsystems that send alerts when certain kinds of general conditions impacting portions of the overall airspace are detected (wind shear is an obvious

The authors are with the Department of Computer Science, Cornell University, Ithaca NY 14850; emails {mahesh,ken}@cs.cornell.edu. This work was supported by grants from NSF, DARPA, AFRL, AFOSR and Intel.

example). Such alerts trigger updates to relevant portions of the visible display.

3. *Track updates.* These subsystems process the raw radar imagery, identify tracks likely to be associated with aircraft, search the flight-plans database for known craft on the corresponding trajectory, etc. These result in updates to the screen and also are input to other subsystems that compute alerts, for example, if two planes are at risk of straying too close together.
4. *Updates to flight plans.* The ATC system maintains a central shared database in each region tracking flight plans and associated data for each flight known to the system. Some of this data is replicated onto the controller consoles and hence must be updated if changes occur. Updates include both critical data about flight planning, also more mundane information relating to the airline operating the flight, the gate at which it is scheduled to park, numbers of passengers expected to disembark on arrival, and so forth.
5. *Console to console state updates.* This is an important category of events specific to the French design. As noted, controllers work in small teams, and a goal of the system was to replicate the data associated with each team over the full set of consoles that team was using. The purpose is to ensure consistency and support fault-tolerance: if a machine crashes, the others can take over its responsibilities until the fault is repaired. Reliable multicast is used for these updates.
6. *System management, monitoring and control messages.* PHIDIAS has a system-wide monitoring architecture but does not use the multicast system in it (the Isis protocols don't scale to the necessary degree).
7. *Air control center to center updates.* Some types of controller actions or local events have important ramifications for other air traffic control centers that are likely to see the same flight, or that may be responsible for the flight before it reaches a given center; these updates need to be shared reliably using wide-area messaging.

When implementing their system, the French team basically split uses of multicast into categories:

1. Multicasts for which no special reliability protocol was needed. Without exception, these were cases in which the hardware itself had properties strong enough to provide desired application-level guarantees without additional work. Obviously, though, the counterpart of "doing nothing" to enhance reliability is that application software had built-in expectations. For example, each console knew that the radar system should send a refreshed image every few seconds and had a preprogrammed reaction to deal with missed data. Through analysis of possible scenarios, it was determined that loss of a single radar update or even two was safely tolerable, but that extended periods without updates posed risks. Accordingly, as time passed without radar updates, the system gradually reacted in an escalating manner: checking with other members of the cluster to see if other nodes were receiving data successfully, then displaying a low-level alarm to the user, and finally displaying a higher level alarm that offered the air traffic controller a choice between deliberately turning off the background radar images (e.g. if the radar itself was temporarily offline), or taking the console offline. In general, the philosophy was one of not overreacting, but still detecting problems, involving the controller when some threshold was reached, and having a "fail safe" backup option in all cases.
2. Multicasts for which virtual synchrony guarantees would be offered. These updates were guaranteed to reach all operational machines in the appropriate receiver group, and if a group member was detected as faulty, that console would be taken offline for repair and subsequent restart. So here we had a very strong reliability guarantee. As noted earlier, no group would ever have more than three to five members, even in a control center with hundreds of machines into total.

What we find interesting in this approach is that although quite a few of the "virtual synchrony" applications clearly have some form of timeliness requirement, that form of multicast primitive does not provide any sort of real-time guarantee at all. Something similar can be said of the "raw" multicast: this is as fast as they come (IP multicast implemented in hardware), but offers no guarantees of reliability. Our own recent studies of similar styles of communication make it clear that bursty packet loss can be a real problem for such uses of unreliable hardware multicast. The French system, in retrospect, mostly avoided this issue because the applications running on the LAN are carefully controlled and vetted – in their setting, some data loss is observed, but it isn't especially frequent. Had they tried to do the same thing on a modern clustered data center with a random mix of applications, raw multicast might not have been reliable enough for the intended use. Thus we have here an implicit reliability expectation, supported by a mixture of application-specific knowledge and hardware properties.

As noted, the second category of multicast offers a property seemingly remote from any real-time guarantee, namely virtual synchrony. This is an all-or-nothing ordered delivery property implemented through protocols that start with an IP multicast (or a set of one-to-one UDP transmissions), but then use background acknowledgement and retransmission mechanisms to identify and recover lost data in accordance with a very strong model – one that can be mapped to what the theory community calls *weak consensus*. The protocol just happened to be fast enough to satisfy the real-time needs of the system with overwhelmingly high probability.

This is fascinating, because several of the subsystems in question clearly had time-critical data delivery needs, combined with their reliability requirement. Viewed more closely, we find that there are two categories of applications that fall into this one category. One group needs a predictably high probability of successful event notification or message delivery. Virtual synchrony, in some sense, is a much stronger reliability and consistency model than these need, but because the Isis Toolkit was actually quite fast, satisfied the time-criticality goal while providing even more reliability than was strictly necessary. The second group of applications needed strong consistency for safety (for example, it is absolutely vital that at most one plane be routed into any one control sector at any instant in time, so routing decisions made by ground controllers have very strong reliable delivery obligations). Obviously, one also wants these to be delivered in a timely manner, but airplanes move fairly slowly in comparison to multicast messages, and the timeliness needs of such applications were often thousands of times slower than the point at which Isis would either successfully deliver its messages, or flag a communication problem as unrecoverable and force one of the offending consoles to restart.

For reasons of brevity, this paper cannot delve more deeply into the properties of the French system, although we recognize that our discussion may have left the reader interested in learning more. In addition to the paper cited here [4], the French government maintains a substantial amount of information about their system online at www.dgac.fr, and even more information is available through EURESCOM, the European Air Traffic Control standards organization.

IV. TIME-CRITICAL MULTICAST RELIABILITY

To summarize our observation, PHIDIAS offers several examples of subsystems that seem to have a natural need for timely data delivery. In building their system, the developers ultimately accepted a tradeoff. For some applications, they concluded that a combination of very rapid delivery with probabilistic physical "guarantees" obtained by application design, system management policies and analysis of the properties of hardware would be sufficient. For other purposes, the reliability need was felt to outweigh the need for time-criticality, and a strong reliability option was favored, taking advantage of the fact that at least in their setting, the communication loads were low enough and hence multicast data rates fast enough that messages would tend to be delivered quickly in any case. In effect, they were well aware of an implicit time-criticality need, but convinced themselves that even if the property wasn't formally guaranteed by the underlying technology, the system *as implemented* would achieve the required properties. This was later confirmed during their testing and validation effort.

Modern developments are now compelling a reexamination of these kinds of scenarios. The economics of scale have begun to favor data centers over other ways of obtaining scalable high performance, hence more and more developers are building services to run on commodity clusters or in data centers. These developers do expect reliability. Yet, because the actual purpose of scalability is often to ensure that an application will continue to give good time-critical responsiveness even when the number of clients using it becomes large, the even stronger need is for excellent time-critical message delivery.

The reason we need to formulate this as a new kind of reliability challenge is that the sort of reasoning used when building the French system is simply not appropriate in a modern data center hosting a more general mix of applications. Moreover, these systems may experience transient faults of a type never seen in the French ATC environment. To address the needs of these kinds of applications, a multicast primitive is required that treats those needs as first-class goals and meets them despite failures.

A. *Observed failure modes in clustered computers.*

One can see an example of the kind of faults that concern us in Figure 1, which is reprinted from a separate, forthcoming paper. Here we've experimented with the behavior of the UDP communication layer on a cluster when subjecting some of its nodes to bursts of incoming traffic of a type that might easily occur in a general mix of applications. What we see is that the nodes begin to lose packets (due to O/S resource exhaustion) and that the losses are bursty, with loss burst lengths ranging from a median value of 5 packets to as many as 75 packets for more intense spikes in incoming traffic.

Against this backdrop, we want to offer a reliable, time-critical multicast primitive. But what should time critical mean, or reliability mean, in this context? Again, the air traffic control application gives some insight.

Recall from the ATC discussion that many data sources basically send streams of updates from what can be loosely termed to be a "sensor". The ATC examples suggest that for some purposes, getting these updates delivered rapidly matters more than doing so with perfect reliability – loss of an event here and there would be acceptable as long as these losses occur with bounded probability that can be adjusted to match application needs.

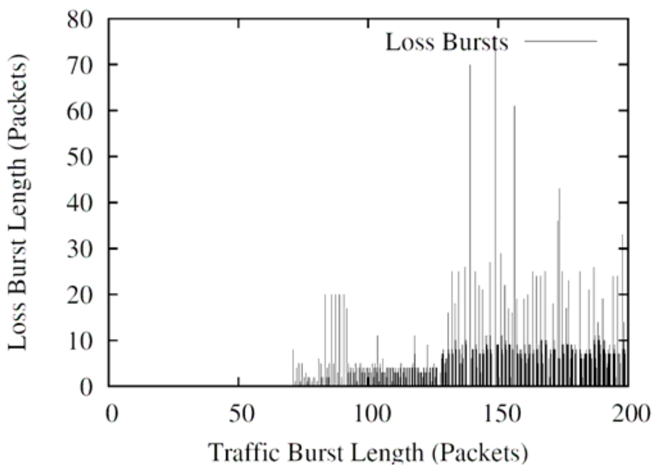


Figure 1: Loss Characterization for Cluster Nodes

How common is such a requirement? We raised this question with a number of industry developers at companies that include Amazon.com, Apache, Google, Raytheon, Tangosol ,Yahoo! and others. All are familiar with a wide range of applications, designed for scalability, in which some form of time-critical behavior is desired. We learned that a scalable services architecture first proposed by a team at Microsoft has gradually become popular among developers of solutions for such settings, and that quite a few of these kinds of applications need a time-critical reliable multicast.

B. *Farms, RACS and RAPS.*

In a widely cited paper, Jim Gray and others discussed terminology for a new kind of application that they were encountering in data centers and other scalable settings. They used the term "farm" to refer to these kinds of scalable systems as a whole, and suggest that a typical farm runs some number of scalable *services*. Each of these services, they found, is commonly partitioned for scalability: on the basis of a key, incoming requests are vectored to representatives using some partitioning function. They call these reliable arrays of partitioned services (RAPS) for this reason. Now, a RAPS would not be particularly reliable if a single failure could depopulate one of its partitions. Accordingly, they suggest that a RAPS should implement each of the partitions as a reliable array of *cloned* servers (a RACS). To clone the service state, one would make copies of any databases used to respond to incoming queries and multicast updates across the copies. Figure 2 illustrates the structure and access patterns of a single RACS.

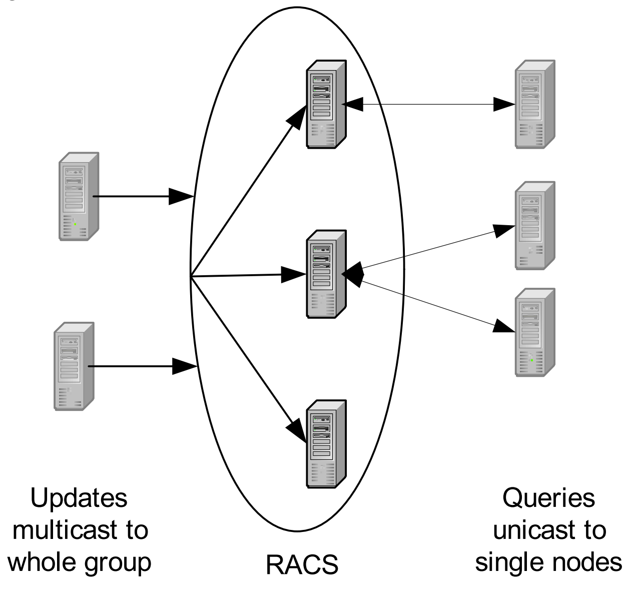


Figure 2: A service implemented by a single RACS

What sorts of services do developers implement in this manner? It turns out that they span a wide range, typified by relatively weak forms of state. Examples include:

- An inventory service that tracks the number of units of the products offered by an e-tailer available at its distribution sites or affiliates. Updates are deltas against the state: "add five" or "remove one", and hence are order-insensitive and commute with each other.
- Product popularity rankings: "customers who looked at X often looked at Y and Z too, and most purchase Z." Updates are typically computed by a background process and then distributed in bulk periodically, with the service itself handling a read-only query load against a database that only receives updates from the back-end.
- Product pricing data (similar to popularity)
- Data about popular products or responses to popular searches.

Notice that while the inventory service manages real data, the other examples are best understood as forms of caches, with back-end databases or other kinds of back-end data sources sending periodic updates, and the cache handling a read-mostly workload that originates with clients.

Interestingly, we learned that for these kinds of services, timely response is often more important than a completely accurate response. A timely response matters because the companies we spoke to are "graded" on the time it takes to compute an appropriate web page and return it to the end-user when a query is received. Sluggish responses mean lost sales. Moreover, when they invest to scale up, the new hardware is purchased in the expectation of extra responsiveness: twice the number of nodes, for example, to guarantee the same responsiveness for twice as many clients.

Complete accuracy for such services is an illusive goal. Developers explained to us that for large data centers, the raw incoming data is often noisy. An inventory system, for example, must be designed in the expectation that sometimes, bar codes don't read properly and an item is shipped without an appropriate debit transaction being issued, or an item is restocked without an "add items" transaction occurring. Breakage and theft may be issues. Compensating mechanisms are used to identify inconsistencies and correct them (audits, comparison with purchase and sale databases, etc). Thus a goal of perfect reliability might be overkill.

Finally, notice that for all of the examples given, updates don't need to be totally ordered. The reasons vary: for the inventory service, updates commute, hence the effect of applying them "out of order" won't be visible to end-users. Indeed, there isn't really a notion of correct state here: if a query occurs concurrent with an inventory change, the response can equally well be based on the inventory before or after the update, since these systems don't use locking, aren't transactional, and aren't trying to "guarantee" any form of serialization property. The actual purchase of that very last discounted digital camera will run through a different system, in any case, and at that stage a verification can be performed to make sure that there are still cameras left in stock.

For the cache services, on the other hand, while updates ordering does "matter", conflicting updates would never occur. The backend system that sends updates can easily be designed to never update the same item more often than once every T seconds, and so long as the multicast is reliable enough to get through within T seconds, out of order updates would never occur. Moreover, our respondents explained that like the inventory example, caches are understood to be potentially inaccurate and compensating mechanisms are built into the parts of their system that book actual purchases or other definitive transactions. And finally, the same point about the intrinsic noisiness of the query mechanism applies here: if a query for product popularity happens to be issued just as the popularity is updated, the client will probably be satisfied with *either* response; neither is "more correct".

C. *Time-critical reliable multicast*

Our review suggests that modern data centers need a communications primitive with properties that were also needed in the French air traffic system, but implemented using a protocol targeted to the characteristics of modern platforms.

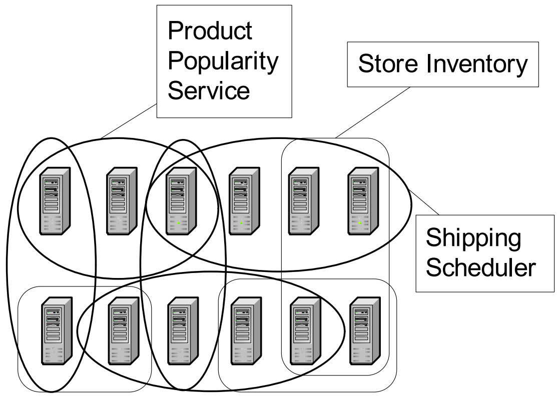


Figure 3: RACS-induced Multicast Groups

Needed is a multicast primitive that:

1. Delivers multicasts to a target *group* of processes. In the settings we've described, the numbers of groups could be large. Moreover, unlike the situation the French achieved through careful design, in general settings there might be many groups, their sizes may vary widely, and they may overlap, with a single node belonging to many groups. Figure 3 shows one reason for such communication patterns, where nodes host replicas of multiple services, causing RACS to overlap.
2. Delivers multicasts *quickly* even if faults occur. The basic goal should be to ensure that for a given level of protocol overhead, as high a percentage of multicasts as possible will be applied to each RACS as quickly as is practical, insuring that queries will reflect the most current data.
3. Tolerates failures that include packet loss, perhaps in bursts of as few as 1 or 2 packets but perhaps as many as 20 or 30, as well as node crashes.
4. Has probabilistic properties, perhaps even extending to delivery ordering and reliability.
5. Does not attempt to recover data if the timeliness of delivery would be seriously compromised.

Among these, we believe that all but the last requirement fall directly out of the scenarios we've analyzed. But what of the assertion that probabilistic *reliability* might suffice?

Our thinking is as follows. First, we do want the primitive to try and recover lost packets. But how hard should it try? After all, the presumption here is that the developer is using this primitive for time-critical updates and, in many cases (not all), doesn't even require reliability as long as losses are random. So clearly, we need a mechanism that can eliminate loss bursts – we've seen that this is a real issue. But on the other hand, a mechanism that tries so hard to recover data that it will do so even after a long effort might be counter-productive: for many purposes, the application might not even want data recovered beyond some sort of timeout, or beyond the next update from the data source (for example, the next radar image). Doing so would just overload the communication channel with no-longer-wanted bytes!

Thus we settle on the mixture of properties just enumerated. Clearly, we're no longer in the world of real-time multicast, virtual synchrony multicast, or best-effort hardware multicast. The need is for a new and different beast – indeed, a family of them, because even the requirements enumerated above could still correspond to quite a range of possible protocols (one can add ordering mechanisms, stronger fault-tolerance features, security features, additional consistency guarantees, etc).

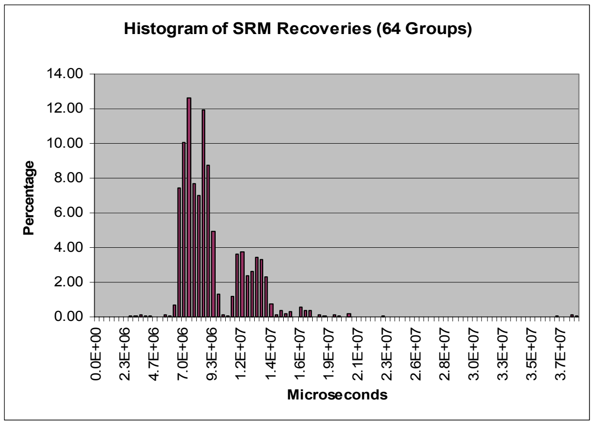
V. THE RICOCHET PROTOCOL

In work reported elsewhere, we describe two examples of protocols that match this new specification. One protocol, Slingshot [2], is aimed at settings with just a single group. The second, Ricochet [1], addresses scalability in the number and layout of groups, and is a closer fit to the full list of properties identified above. In the interest of not repeating work published elsewhere[1], we limit ourselves to a summary of the Ricochet protocol and describe why it fits the listed properties.

Ricochet works using *Lateral Error Correction*, a loss recovery mechanism in which multicast receivers create XOR repair packets from incoming data packets and send them to other randomly selected receivers. The key contribution of Ricochet is that it decouples packet recovery latency from the data rate in any single group or at any single sender. This is in contrast with existing scalable multicast protocols, which are invariably dependent on the rate of outgoing data at a single sender in a single group – for those existing protocols, the larger the number of senders to a group and the larger the number of groups splitting a receiver's bandwidth, the worse the latency of packet recovery.

By generating repairs at receivers and exchanging them across group barriers, Ricochet achieves performance that depends on the aggregate rate of incoming data at a receiver across all groups, and is insensitive to the number of groups in the system or their overlap patterns [property 1]. In practice, Ricochet recovers most lost packets within milliseconds using the proactive repair traffic. The fraction of lost packets recovered depends on the bandwidth overhead expended by the protocol, which is a tunable parameter [property 2]. Ricochet deals with bursty loss by staggering the creation of XORs over time and across groups, tolerating loss bursts of tens of packets. Further, it has no single point of failure and degrades gracefully in the face of node failure [property 3].

[1] We emphasize that the application analysis presented here did not appear in prior work and is the main original contribution of the present paper.



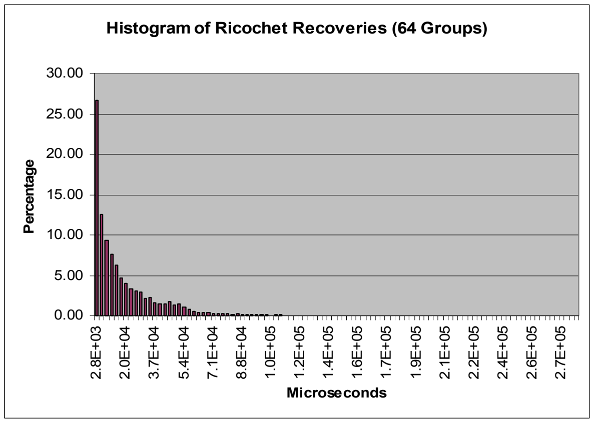


Figure 4: Ricochet versus SRM, 64 groups per node

For lost packets that cannot be recovered through the proactive repair traffic, Ricochet optionally initiates reactive recovery – by sending a retransmission request to the sender of the data – which leads to full data reliability within a few hundred milliseconds. Applications that do not require more than two or three nines of reliability can switch off reactive recovery, obtaining probabilistic reliability guarantees [property 4 and 5].

Figure 4 compares Ricochet with a well-known non-time-critical reliable multicast protocol, SRM [9]. This experiment was conducted on a 64-node cluster, with each node in 64 groups of size 10 each. The presence of many groups impacts SRM performance severely, pushing its recovery latency into seconds, while Ricochet recovers packets three orders of magnitude faster – details of this test can be found in [1].

Ricochet offers probabilistic reliability – we are building layers over it that provide stronger properties, such as probabilistic ordering. Readers interested in experimenting with Ricochet can download it from www.cs.cornell.edu.

VI. FUTURE WORK: THE TEMPEST SYSTEM

Our work with time-critical communication has only just started. Unlike Slingshot and Ricochet, which have been completed, Tempest is a future system, still being implemented as this paper was in preparation, with a completion target of mid to late 2006. The idea is to use Ricochet in a platform that offers turn-key scalability to developers working specifically in the Web Services architecture. Tempest takes a relatively standard Web Services application, built in Java, and linked with a library of our own design that offers a special Java "collection" class in which the application stores information that should be replicated within a RACS.

The user interacts with Tempest through a drag-and-drop GUI that elicits a variety of information about the application: the interfaces it offers to the outside world, which ones are queries and which ones are updates, how the partitioning key can be found in the arguments to these procedures, etc. With this information, Tempest automatically clones and partitions the service, deploys it onto a cluster running the Tempest runtime environment and tools, and than manages it dynamically to heal inconsistencies that can arise at runtime, for example after a crash or recovery, or in the event that Ricochet delivers a multicast to just some of the members of a RACS. Ricochet itself is just a transport protocol used by Tempest when an update is sent to a service under its control.

VII. CONCLUSIONS

We revisited old work on a real air traffic control system (in use in France since 1996), and noticed a previously unremarked collection of requirements associated with data replication under timing constraints. Dialog with developers of modern data centers revealed that similar needs have become common, but also that the solution used in the French system is not directly applicable in these new settings, primarily because they support a more general application mix. With this in mind, we proposed a new definition for a time-critical reliable multicast.

We then summarized work on implementing multicast protocols compatible with this new goal (over time, we think that a whole family of solutions could be developed), and pointed to an ongoing activity to integrate these kinds of protocols into an application development framework.

VIII. REFERENCES

[1] M. Balakrishnan, K. Birman, A. Phanishayee, S. Pleisch. Ricochet: Lateral Error Correction for Time-Critical Multicast. *In Submission*.

[2] M. Balakrishnan, S. Pleisch, K. Birman. Slingshot: Time-Critical Multicast for Clustered Applications. In *IEEE Network and Computing Applications (NCA).* July 2005

[3] K. P. Birman. Replication and fault-tolerance in the Isis system. In *Proceedings of the 10th ACM symposium on Operating Systems Principles,* pages 79--86. ACM Press, 1985.

[4] K. Birman, "A Review of Experiences with Reliable Multicast", *Software: Practice & Experience*, 29(9), pp. 741-774, July 1999.

[5] K.P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer; 1 edition (March 25, 2005)

[6] F Cristian , H Aghili , R Strong , D Dolev, Atomic broadcast: from simple message diffusion to Byzantine agreement, *Information and Computation*, v.118 n.1, p.158-179, April 1995

[7] B. Devlin, J. Gray, B. Laing, and G. Spix. Scalability terminology: Farms, clones, partitions, and packs: RACS and RAPS. Technical Report MS-TR-99-85, Microsoft Research, December 1999.

[8] Damien Figarol, Personal communications. 1995-1998.

[9] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. IEEE/ACM Transactions on Networking, 5(6): 784-803, 1997.

[10] Personal communications with developers at Amazon.com, Apache, Google, Raytheon, Tangosol ,Yahoo! and elsewhere. 2005.