A Review of Experiences With Reliable Multicast

Kenneth P. Birman[1]
Dept. of Computer Science
Cornell University

1 Abstract

By understanding how real users have employed reliable multicast in real distributed systems, we can develop insight concerning the degree to which this technology has matched expectations. This paper reviews a number of applications with that goal in mind. Our findings point to tradeoffs between the form of reliability used by a system and its scalability and performance. We also find that to reach a broad user community (and a commercially interesting market) the technology must be better integrated with component and object-oriented systems architectures. Looking closely at these architectures, however, we identify some assumptions about failure handling which make reliable multicast difficult to exploit. Indeed, the major failures of reliable multicast are associated wit failures. The broader opportunity appears to involve relatively visible embeddings of these tools int h attempts to position it within object oriented systems in ways that focus on transparent recovery from server o object-oriented architectures enabling knowledgeable users to make tradeoffs. Fault-tolerance through transparent server replication may be better viewed as an unachievable holy grail.

2 Introduction

Multicast is an important communications tool for high reliability distributed computing systems, yet the technology continues to seek a permanent home in major vendor platforms. Originally proposed in the context of theoretical work on distributed consensus, reliable multicast found its way into practical settings with the development of multicast-based communication systems and products for real-world applications [CZ85, BJ87, BvR94, Birman97]. Some of these have been quite successful – yet the technology has never gained the degree of mainstream support seen for transactional reliability tools, remote method invocation, or protocols such as TCP/IP.

This begs several questions. Where multicast has been a success, one can ask whether the technology was actually central to the success, or simply found an accidental match with the application. Where multicast failed, was the mismatch fundamental, or an accident of the setting in which multicast was used and the presumptions of the users? Would reliable multicast be a runaway success if it were positioned differently?

Underlying this paper is the belief that reliable multicast is here to stay. But if reliable multicast tools do become standard, what should "reliable" mean? What would be the best way to present the tools, so that they promote a high degree of application robustness without forcing design decisions on developers? How can reliable multicast best be integrated with the most popular software development methodologies? The present paper explores these questions by studying real applications.

With respect to reliability options, the basic issue revolves around tradeoffs between properties and costs. Our review will suggest that applications have used multicast technology in ways that differ fundamentally: reliability properties don't fall on some simple spectrum, and the problem is not merely one of chosing the optimal design point. On the contrary, it seems that any general reliable multicast technology needs to

[1] This work was supported by DARPA/ONR under contract N0014-96-1-10014. Contact information: Ken Birman; Dept. of Computer Science; Cornell University; Ithaca, New York 14853. ken@cs.cornell.edu. Web site http://www.cs.cornell.edu/Info/Projects/Ensemble. This paper has been significantly revised and extended from an invited essay that appeared in the IEEE TCOS Newsletter in Spring 1998 (volume 10, No. 1).

leave the user with flexibility to select among a variety of reliability guarantees, some of which are mutually incompatible. Fortunately, architectures offering this sort of flexibility are well known.

From a software engineering perspective, reliable multicast is often viewed as a "middleware" service. Such layers provide basic facilities by which processes (active programs) can find, join and leave process groups, a means for representing data in messages and multicasting to such groups, and a means for initializing a new member at the time it joins. These are typically integrated with a failure-detection mechanism whereby the system monitors the status of group members and automatically triggers a membership change in the event of a failure. At the other extreme, they sometimes offer a much more elaborate set of interfaces. One can look at these projects relative to the tools they used to understand how they used them and what was central to success.

Reliable multicast can also be hidden behind some other abstraction. For example, reliable multicast is found at the core of message-bus and object-oriented event stream architectures, and has been used to support reliable 1-n file transfer for critical data. The technology is also fundamental to cluster management solutions. Thus even in a world where reliable multicast becomes standard, one could package it so that the average user would not be aware of its use. Again, we can look at applications to understand when and where such embeddings make sense.

This paper starts with a review of several well known, major projects that used reliable multicast. In each case we summarize the structure of the application, outline the uses made of reliable multicast, and describe the reliability properties that were important. We then draw conclusions about the questions just posed.

3 Major Projects

The purpose of this section is to relate some true stories. Several stem from the experiences of users who worked with the Isis Toolkit [BJ87, BvR94] (these are relatively detailed because the author has direct experience with this group of users, and because several are documented through publications). To the degree that experiences with other technologies have been documented, we review them too, but not to the point of speculating about unreported work done by others.

Rather than simply including every experience story available to us, the material that follows is quite selective. We start by looking at financial systems that use reliable multicast: stock markets and trading floor systems (of the type seen in brokerage settings). The latter is a major commercial computing market, and these systems span an important and rather characteristic set of application concerns and styles. However, their primary objective involves data distribution, not fault-tolerance. We then shift attention to some demanding fault-tolerance problems arising in the context of air traffic control, an application that has been tackled with considerable success in France, but less so in the United States. The section closes by looking briefly at cluster management systems and a few other specialized classes of systems built using reliable multicast.

3.1 Stock Exchanges

At first glance, stock exchange “floor” support systems represent an obvious match with reliable multicast (Figure 1), but successes in this area have been slow to emerge. Multicast solutions must displace video-style solutions in which a central server generates some number of channels which are displayed on overhead television systems along the lines of an in-house cable TV. It is only recently, in response to pressure from traders who demand greater flexibility and customization demanding more computing power, that the need for multicast has become strong.

It seems likely that future trends will greatly accelerate this movement. One is the interest in transitioning from exchanges with physical floors to virtual electronic exchanges, typified by the Swiss Exchange, which we describe below. Most exchanges view the challenge of rewiring their trading floors as a first step into an all-electronic world, in which physical trading floors will eventually be outmoded – as, indeed, has happened in Zurich. At the same time, the exchanges see financial advantages to integrating trading

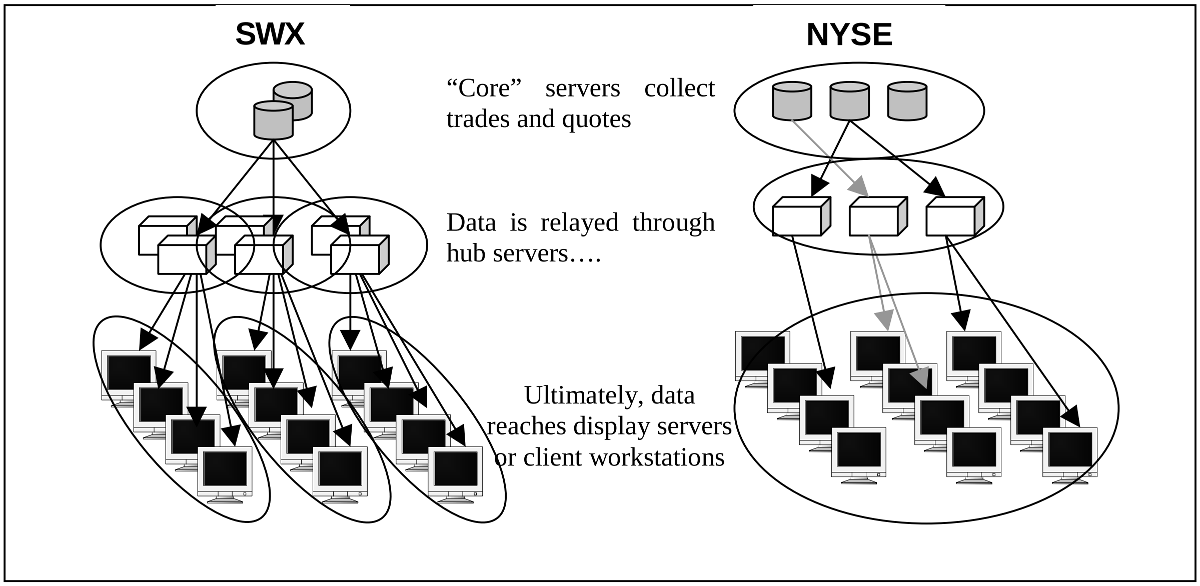

Figure 1: In the Swiss Exchange (SWX, left), the core servers are configured using a primary-backup scheme, as are the hub servers. All updates to the exchange state are fully replicated by reliably multicasting each event to the full set of client systems. In the New York Stock Exchange (NYSE, right), the core servers back one-another up, but each handles a share of the load, and similarly for hub servers. In the NYSE, a given data item is forwarded only to those display systems which subscribe to the item, normally a very small subset of the total number of displays. A consequence is that the SWX is best understood as a tree of process groups, while the NYSE is best understood as a set of three large process group (in fact, both systems are somewhat more complex, but we omit details for brevity). In the SWX, reliable multicast is used to deliver quotes directly the leaf nodes. In the NYSE, quotes flow over a point-to-point forwarding architecture, and reliable multicast is employed mostly to track the topology of the tree, reconfigure after failure, and to track interests of the leaf nodes (which change over time).

systems with other aspects of the market, such as trade clearing. In the long term, these factors will surely transform stock markets into sophisticated networks. The maturing of reliable multicast technology is a critical gating factor in enabling such a path.

3.1.1 The Swiss Exchange

The Swiss Exchange [PS97] is an ambitious all-electronic stock exchange: the physical trading floor has been supplanted by a collection of workstations running exchange-developed software situated in member bank and brokerage trading rooms. The old physical trading floor now houses the development and management teams which run the electronic system. The benefits to exchange members are several: Rather than calling in orders to traders on the floor, order processing occurs in the same room where institutional trading strategy is developed. Trading support tools can thus be exploited without revealing proprietary methods, and the floor representative of the bank can participate directly in strategy discussions without being observed by competitors.

The task of the software implementing the Swiss Exchange is to distribute to each participating trader every trade and every quoted price in the identical order, fault-tolerantly, throughout the full set of perhaps 1000 active traders at 60 member banks and institutions. The latency of the architecture must be low (at worst, a few seconds), and data rates peak at approximately 45 events per second, each represented by a message in the 512-1024 byte range. The exchange guarantees fairness to its customers by giving each exactly the same information, in exactly the same order, and at nearly the same time. The basic architecture of the exchange system is hierarchical: a server at the core sees all trades and all quotes; the latter dominate by about 20:1. It broadcasts these to hubs located at each of the participating institutions; they, in turn, relay the information to the workstations of the traders. Reliable multicast enters this picture in two ways. Its primary role is for transmission of the quotes. A less demanding role is in replication of the servers used and triggering fail-over when a server crashes. The Isis Toolkit is used for both purposes.

The Swiss Exchange is clearly a major success (in the words of its developers the system is “the most sophisticated” existing exchange and the “first fully computerized stock exchange system integrating trading, clearing, settlement and member back offices”). They cite reliable multicast as fundamental to this success and indicate that it greatly simplified their development task.

Nonetheless, the developers found that Isis limits the performance and scalability of their architecture. In particular, they comment that when running Isis reliable multicast over IP multicast, a gradual performance degradation is observed as a function of scale, and that this limits each hub server to approximately 100 clients. They observe that even a single “slow” client can cause performance degradation visible throughout the entire set of clients handled by the corresponding hub, forcing the use of an extremely sensitive failure detection threshold. These phenomena can be traced to the positive and negative acknowledgement mechanisms used in Isis and to its flow control protocol. More fundamentally, however, they are due to its use of a reliability model called “virtual synchrony”, about which we will have more to say shortly. In effect, to enforce the reliability model, Isis sometimes chokes back the sender when a data recipient appears temporarily unresponsive and data buffering limits are reached. This is in contrast to other technologies which might drop messages under such conditions, even if they overcome other less severe forms of failure or message loss.

We should ask whether these problems are fundamental: Isis is an old technology. With respect to raw performance and scaling, Isis has long since been surpassed by other systems supporting the same virtual synchrony model, but with more efficient, lower overhead protocols. For example, Kaashoek’s Ameoba multicast protocol [Kaa94] and the ones implemented by the Horus and Ensemble systems (both successors to Isis) offer higher performance and cut overhead from perhaps 160 bytes per message in Isis to as few as 8 or 16 bytes [vR96]. A new generation of flow-control protocols employ gossip-based mechanisms [Guo98] to give steady data delivery with low overhead. In [Birman97] there is a list of about a dozen virtual synchrony implementations, many of which would substantially outperform Isis.

Unfortunately, the concern about slow clients is not fully addressed by citing faster protocols. The problem of providing stable throughput in scalable settings under load has been a recent research interest of the author. It turns out that while many protocols are described as scalable, few have been studied carefully with respect to stability of throughput when failures are injected while the system is heavily loaded. Recent work suggests that most reliable multicast protocols are weak in this respect. The Horus and Ensemble virtual synchrony protocols are much faster than the ones in Isis, yet there are conditions under which they exhibit precisely the problem cited by the Swiss Exchange developers [BH98]. Similarly, the widely cited Scalable Reliable Multicast protocol (SRM [FJ95]) performs well in a network with low loss rates, yet performs poorly if the network as a whole exhibits a significant degree of message loss or delay [Liu97, Lucus98]. In effect, these protocols have excellent best-case performance and scalability, but degrade when placed under stress.

If we accept a somewhat different reliability goal, however, stable throughput can certainly be achieved. The Bimodal Multicast [BH98] offers a strong reliability guarantee, and includes a guarantee of steady throughput even if a bounded rate of transient failures occurs, including as many as 25% of “slow” clients, or a similar percentage of randomly dropped messages. The properties of this protocol are not identical to the reliability properties of Isis, but seem strong enough to implement the client-side goals of the Exchange as currently designed.

Figure 2, drawn from [BH98], illustrates these points. Here we see a graph constructed by injecting multicast messages into a group of 16 processes at a steady rate of 100 messages/second. Either 1 or 5 of the receiving processes are “perturbed” by causing it to fall asleep for 100ms at a time with the probability shown on the x-axis. Thus, 0.3 corresponds to a process which tends to sleep for about 300ms out of each second. The rate of message delivery to a healthy process elsewhere in the network is now measured. We would hope that the rate remains close to 150 msgs/second, and this is indeed the case for the bimodal multicast (top). When using a virtual synchrony protocol, in contrast, the unperturbed receivers experience

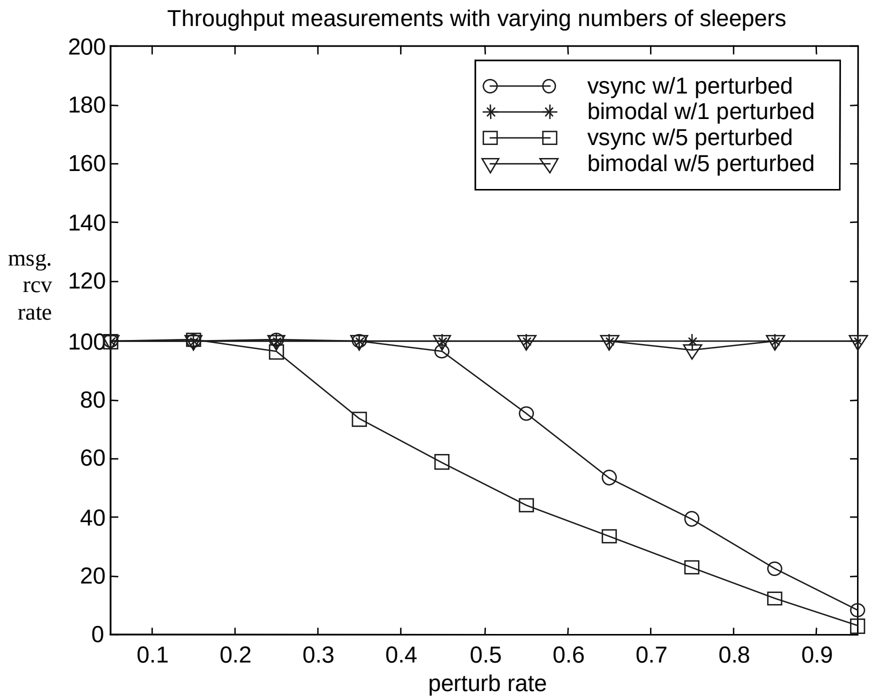

Figure 2: Comparing stability of multicast protocols when a receiver is slow.

a degraded data rate (bottom). With virtual synchrony, no messages are actually lost, but they accumulate in the sender (which is forced to choke back) if data is presented at a rate exceeding the capacity of the system. In [BH98] we show that as the size of a system is increased, sensitivity to perturbation of this sort grows – in effect, the system becomes more fragile as a function of size. This particular graph doesn't show the situation for the perturbed receivers. With both protocols the perturbed processes see reduced data rates: in the case of the bimodal multicast, messages to them can be lost, while the virtually synchronous protocol chokes back the sender. (This graph was prepared on an SP2 configured to behave like a local area network.)

The very steady data rates seen with the bimodal multicast suggest that, with modern versions of reliable multicast, there exists a plausible technology route for projects such as the Swiss Exchange, permitting scaling beyond the current limits of the system, supporting substantially higher performance and stable throughput, and sustaining the reliability goals (albeit in somewhat different ways)[2]. Faster hardware will no doubt help, too.

To summarize:

- Reliable multicast works can be used successfully even in extremely demanding, large-scale settings.
- Roles for the multicast protocols range from data distribution to replication of critical servers and databases.
- Virtually synchronous properties simplify design, but have an Achilles Heel: such protocols suffer from throughput stability problems when failures occur in heavily loaded, large-scale configurations.
- Emerging probabilistic reliability models represent an evolutionary path overcoming this concern.

[2] Obviously, this remark reflects the author's views; the group that developed the Swiss Exchange did not have access to this class of reliable multicast protocols, which emerged subsequent to the completion of their project.

3.1.2 The New York Stock Exchange Overhead Display Architecture

The New York Stock Exchange (NYSE) also solves a multi-point data distribution problem, using a special-purpose Isis-based product called the Isis Reliable Message Distribution Service [Glade98]. However, the goals of the exchange are quite different from those of the Swiss Exchange (Figure 1). Whereas the SWX seeks to fully replicate the entire exchange and to deal with all aspects of trading and trade clearing, the NYSE use of multicast arises only in the context of overhead displays. In particular, the architecture operates to report pricing data to customizable overhead display screens on the trading floor, feeds associated with various wire services, and monitoring systems and archival databases used by the Securities Exchange Commission (SEC) for enforcement. Although the NYSE has a much greater volume than the SWX, one could argue that this data distribution problem is actually simpler because the infrastructure is under NYSE control, and the machines are dedicated ones running homogeneous hardware and software. In contrast the SWX endpoints are machines of varying capabilities running in the client bank infrastructures, and hence exposed to the vagaries of an unknown external environment.

While message sizes are comparable for the NYSE and SWX, it is hard to compare data rates because the data patterns delivered to different overhead displays differ widely: Each display potentially subscribes to its own mixture of data items. NYSE data volumes for an individual display system peak at about 100 messages per second. However, an average display system receives at most a few messages per second under normal conditions. At the root nodes, data rates in the range of 50-100 messages per second are common, with bursts reaching about 250 messages per second.

The Isis Message Distribution Service (MDS) employs a tree-structured data distribution scheme. The leaves of this tree are the overhead consoles. The tree itself is formed by forwarding information about which categories of data are required up a routing tree, establishing a routing path for each possible category. MDS routes data fault-tolerantly (even if nodes in the tree fail, every data item reaches every subscribing leaf node), provides causally consistent event ordering, automates the handling of a replicated network, and aggregates data items heading "in the same direction." The system employs IP-multicast where a node has large fanout.

As one might imagine, management of a tree structure such as this is difficult under dynamic conditions, for example when routing nodes fail or recover. The MDS system leaves the detection and reporting of such events to an Isis-based process group membership algorithm. The idea is that Isis process groups track membership of the MDS machines involved in managing the system and report failures or recoveries to active systems in a consistent manner, permitting them to reconfigure rapidly and consistently. Multicast is also used to replicate information about the data interests of nodes; this must be reliable to ensure that nodes receive the data they require.

In this architecture, data transport is mostly point-to-point: Isis group multicast arises primarily at the very last stage of these trees, as they fan out to the actual display computers. Thus, whereas the SWX system reliably multicasts every quote or transaction report to every machine in its network, the NYSE protocol has a much less regular structure focused on permitting the owners of each display to configure the data items it will receive. One consequence is that the slow client problem seen in the SWX has not been observed in the NYSE: although a machine may well perform poorly, the impact would not typically be system-wide (any backlog that arises is on the link from its server to that machine), and the consequences are similarly local (normally, the machine in question is identified as faulty and dropped from the system). The NYSE thus derives important benefits from an architecture in which different data patterns are needed by each of its displays.

The overall story of the NYSE is very positive. For example, during the summer of 1997, the NYSE system functioned flawlessly during a billion-share trading day[3], and the technology has ridden through a number of failures of various kinds since it was first placed into active use in 1993. A final stage of system

[3] Many of these were block trades in which thousands of shares change hands in one transaction, so the actual volume of communication would have been millions, not billions, of messages. On the other hand, pricing is quoted very frequently compared to the frequency with which trades are completed, so that the communication volume handled by the system may be several orders of magnitude larger than the volume of finalized transactions.

rollout was completed shortly before the time of this writing (late in 1998), reaching many hundreds of display systems. Although the stock exchange experienced a major outage just as this rollout was underway (a one-hour trading failure in November 1998), press releases subsequently noted that a hardware problem caused the failure. Indeed, had the software architecture been fully deployed, the failure would not have even been noticed by traders.

To summarize:

- Reliable multicast can simplify management of a complex message-routing infrastructure even if reliable multicast isn't the primary data transport protocol for stock quotes.

- The hierarchical tree-structured architecture of the New York Stock Exchange system scales well and is well matched to the patterns of data delivery which arise in the exchange.

3.1.3 Reliability in Stock Exchange Systems

An attribute shared by the Swiss Exchange and the NYSE is their insistence upon strong message reliability. Both offer "causal" and "total" ordering for message delivery (discussion of these and other ordering properties can be found in [Birman97]). The mixture of properties ensures that all observers see identical sequences of events in the order that they happened. One reason is to avoid any perception of unfairness. There are also applications that use such properties, including the SEC, which monitors patterns of trading for evidence of illegal price fixing, insider trading, or other inappropriate activities. For purposes such as these (and any analysis that tracks trade-by-trade market trends), knowledge of the actual sequence of events which occurred can be important.

This requirement for strong properties outweighs (at least at this time, for these projects) the negatives: the potential system-wide performance impact of a single slow client cited by the Swiss Exchange developers, the potential for a system-wide crash triggered by a bug in the group communication system itself, and so forth. One can respond to the former class of concerns by deploying new protocols immune to the scaling and throughput problems raised by the Swiss, and can overcome the latter using formal methods and extreme testing regimes, an approach being applied successfully to the Ensemble system. Nonetheless, such considerations must be viewed as a cost associated with the decision to opt for strong consistency.

There have been many other attempts to create all-electronic stock exchange systems, but none has been successful on the scale of these projects. The Swiss project, for example, is not just the most ambitious exchange deployed to date, but also the first success after several previous attempts to build electronic exchanges in Switzerland were abandoned. We can take this as empirical evidence that reliable multicast can solve these problems (albeit with limitations) and that other technologies are poorly suited to the need.

Yet the worldwide demand for stock exchange trading systems is limited. If we grant that reliable multicast is the sort of esoteric technology needed for these very ambitious, financially-critical systems, it might still be less relevant and important for solving "normal" reliability problems. A great deal of money is spent developing systems like the two just discussed, but the market is not of the nature that has traditionally attracted the major software vendors. Perhaps systems like the Swiss Exchange will eventually scale to such a large size that they will attract the interest of vendors. But in the meantime, we should look outward towards the brokers and bankers who use trading data in their day to day work. A large bank may have thousands of such individuals worldwide, and the aggregate number of desktop trading support systems is probably in the millions, augmented by tens of millions of non-professional traders who simply enjoy tracking the performance of their individual portfolios. Does reliable multicast have a role to play out where the grass is greener?

3.2 Brokerage Applications and other uses of Message Bus Technologies

The question just posed distinguishes two kinds of systems: professional trading support systems, and non-professional environments where a browser has been configured to display a stock ticker. Multicast plays an insignificant role in supporting personalized ticker applications, because each user has different interests. Most common is a private TCP connection from the browser to a server operated by an Internet service provider. One can raise questions about how this scales, but to do so is beyond our present interests, so we instead turn to the professional trading systems.

The dominant architectural paradigm for such systems involves publish-subscribe message bus architectures, an important application of reliable multicast. Before plunging into details, we should spend a moment to look at the corresponding form of reliability.

The consistency requirements of in-house trading systems are fairly weak. In these systems, most traders work independently, so there is little need to worry about reporting identical data to different individuals at the same time. On the contrary, trading systems reflect the premise that pricing data is at least slightly stale by the time it is received. A broker works by instructing agents on the floor to carry out trades within some window of acceptable pricing and volumes over a period of time. This contrasts with the use of stock quotes on stock exchange trading floors, where the entire floor bases its moment by moment actions on near-simultaneous observations of identical reported information, and where instantaneous reactions are a key to trading success.

Knowing that data might be a little inaccurate and that actions will lag the trading decisions, most brokerage applications can tolerate a lower degree of reliability and ordering than is required in situations where the data is considered to be current and accurate. A possible exception is program trading, where computers direct the purchase or sale of baskets of stocks, but here the numbers of machines involved is fairly small. Thus, in the large scale settings where reliable multicast might be most useful, the main reliability issue is to keep the system running and to overcome serious instances of message loss (for example, a brief failure that causes all messages to be dropped for a period of time). A randomly dropped quote at some random trader will probably pass unnoticed, but fortunes could be lost if the delivery of data is seriously disrupted. Developers of technology for these settings often call this a “best effort” reliability requirement: the system should rapidly restart itself, but a short outage is acceptable.

The most common architecture[4] for providing reliable multicast to traders is through publish/subscribe products running on redundant network hardware (and often, on computers which provide fault-tolerant hardware). The application subscribes to “subjects” of interest, which are text strings or patterns representing various financial instruments or financial news sources. Each quote is published with its corresponding subjects. The application subscribes by registering handlers for the subjects of interest to it, and the communication system works to filter messages so that each application receives only desired messages, and so that the communication hardware is used efficiently. Such systems are evocative of the old USENET News system, but operate at a program-to-program level instead of between humans working with textual messages. The connection to multicast was first observed by researchers developing the V system [CZ85], who suggested that each process group could carry the quotes associated with a particular subject. Subsequent work on the approach retained this flavor [BJ87, OP93, iBus98].

The successful products in this area are often fault-tolerant, but not in the same sense as a stock exchange. They typically can recover if a component fails and will switch from a malfunctioning stock feed to a working one automatically. However, the guarantees offered to the end-user are weaker than the ones cited in the previous section. Typical trading systems simply work to ensure that message loss will be infrequent, that the delivery of information to different platforms seems consistent, and that recovery from hardware failures will be prompt. At the upper end of the reliability spectrum, systems like iBus [iBus98] or the older Isis Message Distribution System [Glade98] can be configured to offer the very strong “virtual synchrony” properties associated with the Isis process group technology mentioned earlier [BJ87, Birman98]. State transfer is provided by a message-replay utility that logs messages and provides back copies from which a joining program can initialize itself. However, the strongest reliability properties would typically be enabled selectively. For example, banks might place a high reliability premium on multicasts reporting

[4] A different style is the pull-oriented database approach, in which the DBMS is treated as a bulletin board. Clients pull data via queries from this database. For applications doing some form of online monitoring, the push scheme is more appropriate, for those doing exploration (data mining), the pull scheme is more appropriate. Pull solutions make no use of reliable multicast, and hence represent a tangent to the topic of interest here.

large transactions that impact an overall risk-management posture, while opting for steady throughput but relatively weak reliability and ordering guarantees for mundane messages reporting bid and offered pricing from a stock market. It is not uncommon to see transactional architectures used for these more demanding aspects, with reliable multicast employed only for less stringent purposes.

Publish-subscribe architectures have also been applied in workflow settings and process control systems, for example of the sort used in VLSI fabrication lines. The requirements associated with such uses tend to be more oriented towards real-time properties, whereas fault-tolerance and ordering requirements may be reduced. On the other hand, certain classes of events demand strong fault-tolerance guarantees. A message associated with an alarm on a factory floor may be of such importance that its loss in the system would have catastrophic consequences. These settings emphasize timely delivery of data; and could be supported by protocols such as [CASD85] and [BH98].

To summarize:

- Trading floors have strong similarities with stock exchanges, but exhibit weaker reliability requirements emphasizing quick restart but not absolute consistency of data reporting.

- Publish-subscribe is a very natural interface to reliable multicast technologies in settings where very large numbers of users or oriented primarily towards monitoring streams of data.

Our observations also motivate enumeration of a spectrum of possible reliability properties:

- Quick restart of the infrastructure itself,

- Low loss rates, overcoming any message-loss problems occurring on the wire,

- Steady data throughput (a form of weak real-time guarantee), predictably low latencies, isochrony,

- Stable behavior as the system is scaled to large numbers of users,

- Guarantees that all messages reach all users in identical order despite failure (virtual synchrony),

3.3 Real-Time Fault-Tolerance in Air Traffic Control Systems

3.3.1 The American AAS

Many readers will be familiar with the American Air Traffic Control (ATC) system (the Advanced Automation System, or AAS), and also with its problems. Architecturally, the system was supposed to make use of reliable multicast – specifically, the Δ-T atomic broadcast [CASD85, CDD96]. The basic idea is to view the ATC system as a pipeline, and to replace the various computational components of the system with replicated ones, using atomic broadcast to relay the output of each process-pair to the next pair (processes are modified to detect and discard duplicates). For example, radar data (tracks) are “piped” into a trajectory computation, which produces input to an aircraft identification and labeling computation, which in turn produces input for a flight conflict computation, and so forth.

Fault-tolerant realizations of this piprelined architecture recall Cooper’s Circus system [Cooper85], developed at Berkeley in the 1980’s, or the approach used by Kopetz in the MARS system [Kopetz97]. Both replace components of a distributed system with groups of 2 or more objects replicating the function of the original component according to what is known as the State Machine methodology. However, Cooper’s approach lacked real-time properties, and Kopetz’s work was at a very low level (indeed, it closely resembles hardware fault-tolerance solutions such as the Stratus “pair and a spare” architecture). At any rate, such observations are moot because these aspects of the AAS architecture, and in fact most aspects dealing with distributed fault-tolerance, had to be scaled back or eliminated after serious delays and major budget overruns.

This is not the setting for a serious post-mortum of the AAS. Many reports argue that the most serious problem was managerial and that the technical issues were solvable, or at least would have been if the project had simply pursued a sensible schedule and adopted feasible goals. But the author has also been shown an internal review [Tham90] of the development status of the fault-tolerance architecture in 1990 (4 years after the project started). That review found that little work had been done on translating the ideas in the Δ-T atomic broadcast paper into practice, and doing so would require resolving a long list of technical questions. In fact, solving this problem in a manner responsive to the very stringent fault-tolerance concerns seen in the AAS is a very significant undertaking. One reason is that the Δ-T reliable multicast protocols can become unstable in settings which also demand a mixture of low latency and high performance [Birman97]. The analysis suggests that the protocols can lose their reliability guarantees when such requirements are imposed. Thus, if the project stumbled for managerial problems, they may simply have concealed even more serious technical problems lying down the road[5].

3.3.2 The French PHIDEAS Project

In France, an ATC project based on a different set of technologies has been successful. Called PHIDEAS [Phideas], the project is part of a broad ATC modernization effort named CAUTRA. PHIDEAS, per-se, is much less ambitious than the AAS system. It limits itself to the console subsystem, which supports teams of controllers who cooperate to control a single ATC sector[6]. The technical challenge is to build such a controller position out of a cluster of workstations, guaranteeing the high levels of availability and consistency required by the application. PHIDEAS must maintain identically replicated state for the sector being managed, recover from failures within seconds, and ensure that in any imaginable situation, the team can keep working. The system will be used at 5 ATC centers throughout the country, each having 50-100 ATC control teams. There could be as many as 40 centers throughout Europe if PHIDEAS is adopted by other countries: the current EURESCON plan will offer each ATC center a choice of two or more technologies, and each center will make an independent selection.

Other parts of CAUTRA, to be built in the coming years, include a wide-area database architecture for tracking flights through the system (each database component being replicated for extremely high availability). An X.500-style registry gives, for each flight, information on who is currently responsible for that flight, the radar tracking system itself, and various interconnections to other ATC systems in Europe and internationally. IIn contrast to the AAS system, which attacked all of these problems at the same time, CAUTRA stages them over at least two decades.

PHIDEAS is based on a reliable multicast subsystem, which does the work of state replication for each ATC sector. The development methodology was painstaking and the approach to testing exhaustive. To summarize, the French (Thomson Aerospace, working under supervision of STNA, the French equivalent of the US FAA) started by experimentally determining the performance limits of the reliability technology with which they worked (the commercial version of the Isis Toolkit). This experimental work permitted the group to confirm that even considering the long lifetime of ATC systems and expected evolution of traffic loads, the demands placed on the multicast technology would remain very far from its performance limits. Moreover, the experimental work demonstrated that in this range of loads, performance of the technology was highly predictable, and that latencies were very low for small groups of 3-5 participating processes. But they also confirmed the observation of the Swiss Exchange that these stability properties were less certain in configurations involving large numbers of overlapping process groups or for large systems including hundreds of processes. Accordingly, a decision was made to design the system as conservatively as possible.

The development team used reliable multicast sparingly. They employed Isis, but found a way to configure it using a single process-group per control position. Moreover, they instantiated Isis to isolate each

[5] At the time of this writing, Britain was considering an inquiry into delays and technical deficiencies associated with their new ATC system, which is described in [CDD96] as using the same architecture as did the AAS system.

[6] French ATC is done by teams of three to six controllers who share the responsibility for an ATC sector and cover for one-another if necessary. Other countries, notably the United States, favor different approaches to this human-factors element of the control problem.

controller position from all others. In effect, each controller position of 3-5 machines is a complete, self-contained Isis application, with no direct interaction at all with other controller positions. (Isis can be told what UDP port numbers to use, and the system designers simply assigned disjoint port numbers to the different positions). Isis is thus used on hundreds of machines, yet needs to scale only to 5 at one time. At peak load, PHIDEAS generates no more than a few messages per second in such a cluster.

The nice property of this solution is that it avoids the scalability limits of the technology. No matter what happens to a given team of controllers on its system, since there is no direct communication with other positions, the chance of a problem rippling to other positions is minimized. Clusters do share the same LAN, and they connect to the same database, and for that matter depend upon the same external system services and power supply. But there is a secondary system of last resort, hence if the risk of catestrophic failure is low enough, such shared dependencies can be tolerated.

The machines within a control position do share the database, though, and this raised a problem. Recall that CAUTRA stages the replacement of components of the system. An implication is that the *new* PHIDEAS system must talk to the *old* database system. Yet the old database was designed to support just a single connection with each control position, and might be overloaded by establishing connections to 3 or even 5 times as many machines. Accordingly, the PHIDEAS developers were forced to chose between upgrading the database simultaneous with the roll-out of the control position technology, or limiting each position to have a single communication link to the database. They adopted the latter approach.

To solve this problem, it is necessary to designate a cluster leader which makes the connection and relays any database updates on behalf of the machines composing the position. If the leader fails, a new leader must be selected to replace it. This problem is solved by the virtual synchrony model supported by Isis and by other reliable process-group technologies. Virtual synchrony places a series of conditions on the reporting of group membership to process group members, and on the synchronization of membership changes with respect to message delivery [Birman97]. Given such guarantees, the cluster leader can be elected from the current list of group members[7], and reconfiguration will be triggered when the group communication system reports that the current leader has failed. The idea is that the reliable multicast informs the application not just about incoming messages, but also about changing membership of the cluster of machines comprising the position, and that it even orchestrates the transfer of system state to machines joining the cluster (presumably, after being rebooted for some reason). The application itself is responsible for acting upon each of these events and for determining what constitutes its "state" (much as for a checkpoint).

One might wonder how messages from the remainder of the ATC system into the position are handled. PHIDEAS accepts two kinds of messages: those from the database, which are relayed by the position leader to the other members, and radar updates. The reliability of the first class of information comes from the reliable multicast protocol. Radar updates are distributed using hardware broadcast, which turns out to have very good basic reliability. Only if several radar messages were lost in a row could the lack of radar data become an issue for the controller (who would then be warned of the problem). Although different computers in the cluster might not receive the identical sequence of radar updates, analysis by the development team yielded a convincing argument that the tiny inconsistencies visible on the screen have no impact on the consistency of the software state of the system replicated within the cluster. Intuitively, this is because none of the software that "looks" at radar data modifies the replicated system state.

Figure 3 illustrates this pictorially. Time advances from left to right. The top-most process is initially the cluster leader, and we can see multicasts being transmitted to update the state within the position and heavy black arrows corresponding to state transfers when new members join. When the cluster leader fails, the reliable multicast system creates a new membership list for the position, and the new top-most process assumes this role, making a new connection to the ATC database. At the bottom, the radar system uses the

[7] Specifically, the cluster leader is selected as the longest-running member of the system. If two processes are tied, the tie is broken using a ranking on their machine addresses. Since age only increases with time, this approach ensures that the cluster leader will only change if the current cluster leader actually fails. When a machine fails and then restarts, its "age" is reset to zero.

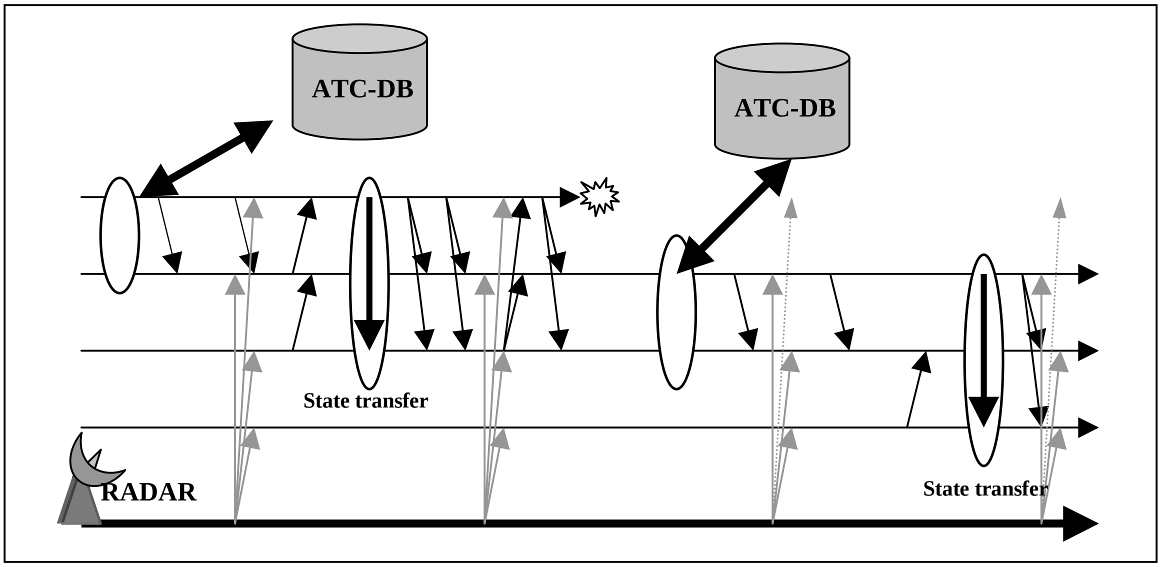

Figure 3: Virtual Synchrony in PHIDEAS. The leader is responsible for connections to the database. Communication includes multicasts of events affecting the cluster state and state transfer to a joining (recovering) machine. Membership changes are reported by the reliable multicast technology to the application layer. As processes join, the state of the system is transferred to them.

same hardware (a fault-tolerant token ring) to broadcast radar sweeps and basic tracking information, but is oblivious to the operational status of individual computers and makes no effort to provide fault-tolerance. (Radar traces change slowly, and the reliability of the broadcast is high. A machine which is dropping a high percentage of radar data would be taken offline, but infrequent loss of a radar data packet is tolerable.)

Notice that the approach depends upon consistency of event reporting; if two different machines within a position detect failures inconsistently, or employ different rankings on the members, they might both try to act as leader, or a run might arise in which there was no leader at all. These problems are all avoided by the virtual synchrony model. Similarly, although one might wonder whether state transfer is actually “well defined”, in this model there is a specific point in the execution where a current member can be asked to write down its state for transfer, and the guarantee that if the application does so, the state reflects every message received up to the time when the transfer was done, and that every subsequent message will be delivered to the new member.

To deal with the threat of severe failures, an independent backup system which simply presents the raw data. However, in testing (which has been realistic, using real data and real control actions), the basic reliability of the system has been found to be extremely good. Indeed, the project expects to eventually achieve “10 in 10” reliability: no more than 10 seconds of “denied service” in 10 years, for a typical control position. Officials with STNA, the equivalent of the American FAA, also stress that their team approach provides a substantial margin of safety even if technology support for control is briefly lost while the system is reconfiguring after an equipment or software failure. This reduces the degree to which the software system must be viewed as a life-critical component of the overall ATC problem.

PHIDEAS could also have been built using a replicated database product: in principle, each controller workstation could be treated as a server for a replicated database tracking the state of the sector. However, this wouldn’t have been very practical. Database systems can have extended outages when reconfiguring to drop a failed system or to recover one that is restarting, whereas PHIDEAS requires nearly continuous availability. In fact, one can *prove* that database consistency is incompatible with high availability, at least in the absence of special hardware (see [Birman97]). Obviously, DBMS systems work around such restrictions, but the complexity and heavyweight nature of the resulting architecture makes the actual use of a database for this purpose cumbersome and costly. The near match of the technologies, though, does point to a deeper issue, which is that replicated DBMS technology and reliable multicast technologies have a great deal in common. In the future, one might hope for a greater integration of these complementary technologies.

One last observation. Readers familiar with the architecture of the American ATC project will recall that it stressed the importance of a reliable multicast technology with *guarantees* of real-time behavior [CASD85, CDD96]. The French architecture manages well with a very fast technology but does not assume guarantees of the same sort. Used in the limited way that multicast arises in PHIDEAS, a fast multicast system with observably predictable behavior under the range of possible failure and load conditions evaluated during testing is "close enough" to real-time for the purposes of the application. This is interesting, because it reinforces the suggestion made earlier that real-time properties really fall into two categories: real-time in the sense of guarantees supported by formal proofs, and real-time in the sense of "fast and predictable", supported by extensive experimental work. As it happens, the former type of real-time is hard to achieve in fast systems. Weaker guarantees are another matter, as illustrated by the work reported in [BH98, Birman97].

The French experience seems not to be a fluke. In the United States, the Hiper-D project developed a prototype integrated battle management and radar system for a rebuild of the AEGIS (the top-line Naval radar platform). This effort, which was based on architectural principles similar to those of the AAS but employing roughly the same technologies as were used in PHIDEAS was a success. The outcome shaped the emerging SC-21 standards for communication within and between Naval vessels.

To summarize:

Used in a cautious, restrictive manner, reliable multicast can be a successful component of a modernized ATC system or similar critical control applications. The key to success is to treat each process group as a completely separate, free-standing system (thus avoiding the danger of a common-mode failure which might bring down the whole ATC network), and to architect the system so that communication loads and scaling will be modest.

- Performance of modern multicast technologies is adequate to accommodate the loads seen in a replicated console architecture for air traffic control. Indeed, performance is good enough so that special real-time protocols and architectures can be avoided.

- If reliable multicast is used inappropriately the technology can emerge as a serious stumbling block. Rather than promoting fault-tolerance, such an approach can significantly increase project complexity and may even emerge as a single point of failure, in the sense that the reliable multicast technology itself becomes basic to overall system reliability.

- Merely replicating computational components does not constitute a solution to the fault-tolerance problem; considerable additional work is needed to make such replication practical.

3.4 Other Uses of Reliable Multicast

Reliable multicast has been successfully hidden within a number of other tools. The most important in a commercial sense are tools for cluster management, tools for conferencing and collaborative work, and tools for file replication.

3.4.1 Cluster Management

Cluster management tools have not traditionally been presented in terms of reliable group multicast and membership management, but recent software systems for cluster management make the connection more clear. A good example is NT Clusters [VD98], which combines a reliable group communications architecture with a component-ware presentation to provide users with a very simple collection of reliability options. To make an application fault-tolerant, the developer simply drags the application onto an icon representing the cluster, and the cluster manager copies the binary to the appropriate places and wires itself to standardized management interfaces associated with application monitoring and restart.

Technology	**Pros**	**Cons**
IP multicast	Widely available, standard Respects end-to-end design philosophy	In wide-area settings, limited to mbone. No reliability guarantees of any sort. Not “router-friendly”
Scalable Reliable Multicast (SRM)	Useful for replicating “soft” or unchanging state Optimizes behavior to exploit latency and properties of IP multicast diffusion Places onus for reliability on receiver. No ack/nack implosions	Limited to best-effort scalability Can’t track group membership Throughput unstable if overall network loss rate is non-negligible
Reliable Multicast Transport Protocol (RMTP)	Hierarchical structure with servers gives predictable behavior Self-adaptive, automatically reconfigures as servers crash or restart or become unreachable Useful for multicast file transfer	Shares most characteristics of SRM
Virtual Synchrony	Strong model supports consistency, application-level fault tolerance Supports replication of changing data or computation with guaranteed coherency, provides state transfer High performance when used under good conditions	Hard to design scalable applications. Single process groups scale poorly: either latency becomes high, or they suffer from ack/nack-implosion Throughput typically becomes unstable as group size and data rate grow
Probabilistic multicast	Probabilistic model gives guarantees similar to virtual synchrony Scales with very low overheads Stable throughput even under stress	Model is weaker than virtual synchrony and this is visible to the application In large configurations, protocols need to schedule gossip communication so as to avoid overloading centralized routers and to limit communication over high-latency links

Table I: Summary of major styles of reliable multicast protocol, their best-known positive attributes, and their limitations. No single technology dominates all other: each has settings in which it performs well and settings in which it has problems.

A second example arises in work done by Berkeley's NOW project, which employs scalable reliable multicast protocols to replicate "soft" state (unchanging or slowly changing data) on cluster-style servers [FG97]. The idea behind this work is to view certain machines as caching data which has a primary home on a single server. Updates are directed to this server, but queries are load-balanced across the machines with replicas. The applications of interest to the Berkeley group involve web servers in which slightly stale data is generally acceptable to the user, hence this type of state replication is adequate. Reliable multicast is also used for load balancing and other system management uses.

Indeed, if one looks beneath the hood of scalable cluster solutions, one usually finds a reliable multicast architecture with properties matched to the specific uses arising in cluster management. Roles for the multicast software include maintaining some sort of replicated checkpoint data (for use in restarting programs), replicating load information (for load-balancing), and orchestrating (coordinating) restart actions after a node fails or joins. Typically, this is done by using a totally ordered multicast to drive a form of replicated state machine. The multicast technology may also play a role in avoiding "split brain syndrome", which can arise if a node is somehow cut off from the rest of the cluster (by a communications interconnect failure) but remains connected to a shared disk. One wants such a node to disconnect from the disk, so that the remainder of the cluster can take control. Cluster fault-tolerance can also provide additional guarantees. For example, Roy Friedman has shown that cluster technology can be extended to provide real-time fault-tolerance properties [FB96].

We should note that although most cluster management systems have similar internal structures [Pfister95], the group communications aspect is not always obvious. For example, VaxClusters [KLS85] and Tandem NonStop [Barlett91] have architectures presented in terms of basic cluster services – the lock manager and shared disk in the former case; process-pair mechanisms in the latter. The mechanisms implementing these features, however, involve protocols very similar to group communication protocols. NT Clusters similarly has a set of basic cluster management services (centered around the cluster registry, the "quorum resource" and an associated group update protocol), but the mapping from these to group communication is made relatively explicit by [VD98].

3.4.2 Scalable Scientific Computing Environments

Readers familiar with PVM and MPI, the favored communication tools of the parallel computing community, might wonder whether these should also be viewed as reliable multicast implementations. Both include reliable multicast primitives, although neither supports tools for cluster management. Indeed, both PVM and MPI assume a static system configuration, and their reliable multicast interfaces are better understood as a small part of a much larger application development environment and runtime support package oriented towards master-slave architectures. Reliable multicast and multicast-based barrier synchronization arise in such settings, but are just a corner of the overall picture.

Within the context of the present review, we see PVM and MPI as application-level solutions which make use of reliable multicast (and other forms of communication), but that have objectives of their own, aimed at a specialized community. A successful reliable multicast tool should be capable of supporting scientific computing environments, but resides at a lower layer of the computing system where it can be highly optimized just for communication.

3.4.3 Collaboration and Conferencing Tools

Group communication has a good match with collaboration and conferencing. Best known among research projects in this area is the Wb system, an electronic "whiteboard" that runs over the Internet Mbone using a protocol called SRM [FJ95] (scalable reliable multicast), making heavy use of IP multicast. Wb works well when the Internet is relatively idle, but has been observed to degrade when overall packet loss rates rise, a phenomenon studied in some detail in [Liu97, Lucas98]. The problem is that as noise rates rise, SRM begins to send high rates of duplicated packets and duplicated retransmission requests. SRM sends these over IP multicast, and although an effort is made to use the time-to-live option of IP multicast to limit propagation of retransmissions, these studies suggest that participants still experience an overhead that grows with the size of the system and loss rate. Liu proposes SRM extensions which, in simulation studies, reduce this problem.

Although Wb runs over a reliable multicast protocol, SRM's reliability goals differ from what we discussed previously. In SRM there is no notion of end-to-end membership in the group application – applications join and leave anonymously, and senders do not in general "know" who the receivers will be. Reliability is achieved by receiver-initiated retransmission solicitations, and the sender, in general, cannot determine when a message has reached all its destinations because there is no way for it to know the destination set. The resulting guarantees are weak in comparison with the ones needed in "critical" applications such as stock market and ATC systems, or systems such the ones used for cluster management, where coordinated reaction to failures and other events is required.

3.4.4 File Replication

RMTP is a scalable reliable multicast protocol developed at Lucent technologies [LP96, PS97]. Like SRM the protocol employs an anonymous membership mechanism; unlike SRM it operates hierarchically and (at least in principle) should not suffer from the retransmission "storms" observed by Liu and Lucus. A major use of RMTP is in reliable file transfer (RMFTP), although the system is also used for other applications in telecommunications settings.

3.4.5 Summary

Again, we conclude with a review of major lessons which can be extracted from these examples:

- Specialized applications are often best served by specialized solutions matched to the problems encountered in those settings and presented in the paradigm of expected developers.
- Enumeration of such applications points to a broader use of reliable multicast than one might have expected, although each application area is itself isolated and small.
- Some of the successful solutions for specific areas include reliable multicast functionality, which would probably be implemented as a "wrapper" over a standard O/S supported solution, if one was present and optimized to offer very good performance.

4 Integrating Reliable Multicast with Software Engineering Tools

The previous section focused on applications and the manner in which they use reliable multicast. This section focuses on the embeddings of multicast into presentation technologies, with a software-engineering emphasis. Several times we noted that the various systems that have been most successful employ one style or another of presentation. Here, we ask how well the major presentations have worked, and also how each is limited. We consider toolkit and object-oriented embeddings of reliable multicast, treating message bus technologies briefly at the end as an example of a possible object-oriented presentation of reliable multicast.

4.1 Reliable Multicast in Toolkit Settings

Many multicast systems are presented as "toolkits" – layers of standardized algorithms that operate over the multicast subsystem to solve standard problems in standard ways. Examples of tools normally found in toolkits are interfaces to the basic membership reporting and communication services, locking or token passing, a means for replicating data, and a state transfer tool[8]. Recent toolkits have gone further, offering tools for load-balancing, adapting the protocol stack to changing conditions (perhaps by adding or removing protocol features or by changing parameters), facilities for forming subset groups or lightweight groups, interfaces by which one process group can manage other groups, real-time tools, and other

[8] It is not clear that a state transfer tool, per se, is really needed. By packaging state transfer as a tool, many toolkits impose a single way of obtaining state on the developer. Experience with Isis and other systems suggests that different applications often have very different state transfer requirements [Birman97]. To offer flexibility, it may be preferable to just provide basic tools covering the major events by which state transfer can be orchestrated. The developer needs examples of how to do state transfer, but can then be left to customize the mechanism to the needs of the application.

mechanisms packaged as toolkit components. Among systems mentioned previously, MPI and PVM are toolkits, as is the Isis Toolkit [BJ87, Birman93]. Cornell's successors to Isis, two systems called Horus [RBM96] and Ensemble [Hayden98], both support Maestro, an object-oriented toolkit for C++ and Java [Vaysburd98]. Kaashoek's Ameoba multicast and Cristian's Δ-T atomic broadcast are examples of reliable multicast solutions presented independent of toolkits [Kaa94, CASD95].

For developers working in the programming language to which a toolkit is directed, it can bring major benefits. The average developer of a distributed application wouldn't want to solve the state transfer problem, except by providing procedures to marshal state into a suitable representation and to de-marshal received state. But toolkits can be hard to use because they often presume that the developer began building the application using the toolkit right at the outset. Developers of complex, large-scale applications often encounter reliability and distributed security problems late in the game, and only then consider grafting on a mechanism supported by the toolkit. Stand-alone, that application might be easy to build, but suddenly it needs to operate covertly side-by-side with some large pre-existing application. Doing so often involves introducing multithreading into applications that were previously single-threaded, providing new kinds of interrupt handling mechanisms, and providing means by the which the new tool can intercept certain classes of events in the older legacy application. Such efforts often spiral out of control, introducing bugs and reducing, rather than improving, reliability.

In contrast, the component-ware trends sweeping the distributed computing industry treat entire applications as "components" that implement a small set of standard operations and register the corresponding handlers by exporting an appropriate interface. Toolkits don't fit well with this philosophy: by operating at the programming language level, the toolkit denies its functionality to component programmers, even if the toolkit has a tool well-matched to some component-level need. For example, toolkits like Isis and Maestro offer mechanisms for restarting a computation if the node where it is running crashes. But rather than doing this by reinstantiating the failed system component at some new location, they assume that the developer will pre-configure the system with standby components at the locations where restart is feasible, and write code to shift them into action. The mechanism needed for component-level fault-tolerance is thus secondary to an assumption both about programming style and choice of language.

A further limitation of many toolkits is that even when used precisely as intended from the outset, some tools suffer from "excessive transparency." The danger is that a mechanism may look simpler than it really is, inviting careless use that will not perform well. The most serious (and common) problem arises when the tool was really designed or tested under conditions which do not hold in general. The application developer who decides to use such a tool may be doomed to a frustrating failure, through ignorance of constraints which were never documented or by placing the tool under a load or usage pattern unanticpated by the original designer. Many problems with the Isis Toolkit fit this pattern.

Thus while toolkits are among the most successful presentations of reliable multicast, they demand a degree of sophistication from the user that matches only a small class of likely developers. For the builder of a system such as PHIDEAS, a toolkit solution makes good sense. Such a developer is likely to be well informed about limitations of the toolkit, cautious and very deliberate in their development effort. With this background, the project is likely to succeed. But a developer working at a large database company who wishes to graft a new feature onto a major database product will face constraints not seen in the former class of applications. Such a developer must, in effect, add new properties to a complex black-box that cannot be changed and for which even the interface to the environment may be extremely subtle and largely inaccessible.

To summarize:

1. Some applications depend upon the fine-grained control offered by toolkits.

2. The major "tools" are concerned with replicated data, initialization when a new replica is created, and the synchronization needed to cope with dynamically changing membership. These are often supplanted by additional tools for related tasks such as load-balancing, checkpointing and recovery, parallel processing, etc.

3. Toolkits often express explicit or implicit language biases that limit their applicability.

4. Toolkits often reflect explicit (or hidden) environmental assumptions. Even when explicit, these may surprise users, particularly with respect to scale or performance.

Object Orientation and Component Systems

Although "object oriented" and "component" technologies are common terms, some readers may wonder precisely how they differ from one-another: object-oriented systems are often referred to as component systems in the literature.

In this paper, an object-oriented system is one internally structured using objects: software-defined data types that encapsulate the implementation of a set of methods together with the data representing the data type. Object orientation is a programming tool, and languages can be characterized with respect to the degree of their object support. Examples include O'Caml, Java, and to a lesser degree, C++. Object oriented systems often support means by which programs can export interfaces, permitting their methods to be remotely invoked.

Component systems also treat each program as an object with an external interface. However, such systems focus on what might be called "external" programming rather than "internal" programming. Think of an operating system such as UNIX or NT which interacts with running programs by delivering events which cause methods to be invoked, for example to ask the program to redisplay itself. Standards can be defined: "all programs will support a print operation", for example. The world of components is thus one in which system components are viewed as objects, and are assumed to support a certain basic degree of runtime functionality. The component perspective of programs has become popular because of it fits well with iconic user interfaces and support for drag-and-drop. This style of programming is widely credited for a dramatic increase in developer productivity.

One can talk about programming a component system from the perspective of a programmer starting from scratch, but these kinds of systems also support programming at a scripting level. In the past, a scripting language such as Perl or Tcl/Tk was traditionally used to create a file containing the script, which was then stored as an executable command. Component systems typically offer the additional capability of associating executable scripts with *events* that can be triggered by components. In this paper, we extend that idea to one in which scripts might be associated with groups of components. This is best understood by viewing the entire distributed system as having a single component-monitoring application. These multi-component scripts are stored within it. The monitor senses component-level events, determines what actions are needed and then runs the corresponding script in the right place with information specific to the event that triggered the action.

In practice, of course, component scripting systems are themselves distributed programs, but we will not treat this issue in the present paper. However, we do talk about the integration of group multicast functionality with object-oriented and component-system designs. In the former case, the functions of the multicast system are provided to a programmer but are expressed in terms of objects: a message, for example, is viewed as an object containing other objects. In the latter case, however, the reliable multicast technology is used to trigger actions within the component scripting layer, as a way of supporting sophisticated distributed system management and control functionality.

4.2 Reliable Multicast in Object-Oriented Programming Environments

Among the applications surveyed above, there are two dominant object-oriented embeddings of reliable multicast. The more successful one involves message bus architectures matching a publish/subscribe model. We discussed this in Section 3.2 and will not repeat that material here. However, it should be noted that publish/subscribe is really an "application", not a core technology. In particular, while message-bus (or object-bus, as the case may be) systems make very good sense in settings where the goal is to distribute sensory data to large numbers of clients, they are ill-matched with applications such as cluster-management. One can imagine elaborate message-bus architectures in which the underlying group-structure of an application might be selectively available to users, but this would make the mapping of subjects to multicast groups explicit and hence would remove a useful degree of freedom from the object-bus developer. Moreover, the resulting system might be unreasonably complex.

4.2.1 Limitations of Transparent Fault-Tolerance

With the exception of Maestro, there has been little work on object-oriented toolkits for high reliability applications. Instead, the prevailing trend has been to focus on transparent object replication. The basic concept involves identifying an object or service which is critical to the application, and then replacing it with a group-implementation that preserves the interface of the original version, but provides high availability to the user. Orbix+Isis and Electra [RBM96] are two well known implementations of object replication technologies (the former also included an event notification system). Both operate at the level of the object request broker (ORB). A third such technology, Eternal [NMM97] intercepts object requests at the interface to the operating system. Each maps intercepted invocations into group multicasts, collects the replies, collapses them down to a single reply, and passes the result back to the application.

It is hardly surprising that the object oriented community would have seized upon this approach. If one goes back to the ANSA system, a precursor to CORBA, the concept of object orientation was explicitly identified with transparency: The goal of an object, in effect, is to conceal the nature of its implementation, including the mechanisms employed for reliability and consistency. It is a small step from such a mindset to an expectation that reliability should take the form of a magic wand, which one waives over an initial implementation of an object to obtain a replicated version, which is precisely the approach adopted in Orbix+Isis, Electra, Eternal, and other such systems. However, there is a serious conflict between this form of transparency and the types of successes identified in Section 3. Moreover, there are practical problems with implementing transparency which impose serious limitations upon the programmer.

Notice that among the success stories reviewed earlier, not many involve replicating a critical component of a larger application in order to achieve fault-tolerance. PHIDEAS, for example, uses replication to support n-way redundant positions, but the controllers at the different consoles in the position don't perform identical tasks, and there are aspects of the "replication" scheme that are not symmetric, such as the use of a leader to manage interactions with the ATC database. The Swiss Exchange is certainly a form of replicated object, but one that required tremendous care to implement. What magic wand would ever have the sophistication to start with a single-site version of a stock exchange server and produce the Swiss Exchange automatically?

Additionally, the decision to provide transparent object replication brings with it some serious practical limitations. To see this, it will be useful to digress momentarily by discussing the manner in which exception handling occurs in CORBA, COM and Java RMI. All three share a common legacy in CORBA, and CORBA generally treats failure recovery as a problem of restarting components out of persistent storage – a sort of checkpoint/rollback philosophy. Now, checkpoint and rollback are easy to understand and for many applications they represent a very good approach to fault-tolerance. But the ability to make and restart from a checkpoint are just one part of the solution, and there are applications for which the time needed to restart from a checkpoint might be prohibitive. Moreover, unless checkpoint/restart is used consistently throughout a system, the ability to restart individual components may not be adequate to provide system-wide reliability.

Consider an application that discovers that some server it is talking to has failed. Now, suppose that this application needs to fail-over to some other server (Figure 4, 5) rapidly. None of the technologies listed here really provides a clean way to do so, because of the complexity of binding to a server and the manner in which the server may have cached information within client-side stubs. To fail-over would normally require some cleanup of information associated with the old server instance, selection of an appropriate new server instance, initialization of the connection (to "catch up" with the state presumed by the remainder of the application), and finally a resumption of normal computing[9].

To make this concrete, here are two examples. In a distributed brokerage application, one often employs servers that compute theoretical pricing, for example the price predicted for a particular bond in light of current interest rate trends, market volatility, quality of the issuer, etc. The client program would capture various bond parameters from the broker or from a database, then perform a remote method invocation on the pricing service, which returns the theoretical price. This is an example of a "stateless" computation in the sense that the same request can be issued several times; the exact answer may change (because the underlying market fundamentals are changing) but any answer should make sense.

A second example can be drawn from our ATC example. Suppose that we want to transform the ATC database into a fault-tolerant server by replicating it. The client system – what we called a PHIDEAS position – will need to perform update operations that notify the database of route changes. The status code returned in response to these updates is important to the controller, who will want to know that an update can't be "lost" by the system as a consequence of a failure/restart sequence.

Now, consider the fail-over problem on the server side. Even if we relax the availability goal, so that the server is permitted to briefly go offline while restarting from a checkpoint, many questions remain to be resolved. How will the client-side software realize that the service is only temporarily offline and should be back up again soon? Will there be some timeout involved, so that a server that is a little slow to restart would seem to be down after all? And if so, how can we call this a fault-tolerant solution?

What will happen if the checkpoint isn't exactly up to date? For our two examples, we get different answers. In the first example (the bond pricing service) it should be possible to simply reissue the request. A minor complication arises if the server caches some data within the client, which in fact is a common way to overcome performance problems. Here, when failing over, there may be a non-trivial cleanup issue. Nonetheless, for a stateless server, it seems plausible that simply garbage collecting the old connection, reconnecting, and reissuing the most recent request would be possible. This matches the CORBA model. Nonetheless, every ORB of which the author is aware would throw an exception if the failure occurred *during* the remote method invocation (for example, if a partial reply message was received, containing the first half of the data associated with the real reply), limiting transparent failure masking to the case where the failure occurs when no invocation is pending. The application itself would thus need to be tolerant of error returns on method invocations, and would need to be designed to reissue the request. In effect, the model is inadequate if failure masking is the goal.

The ATC version of the problem is quite a bit more difficult. Here, it is not obvious that one can simply reissue the request. Instead, it may be necessary to query the ATC database to find out if the pending request was completed successfully or not. Since we also wish to support fault-tolerance within the PHIDEAS position, we'll also need to treat the decision to issue the request as an update to the position state, and the reception of the reply as a second update, multicasting both. Moreover, there is a timing consideration: recall that the ATC system must guarantee nearly continuous availability, so this fail-over must not take more than a few milliseconds – surely too quickly to restart a server from a backup (see step 9 in Figure 4). We arrive at a very elaborate problem statement which is obviously not matched to the CORBA model. The developer of an object-oriented database application of this sort would in fact see this as a situation embodying major obstacles.

An oft-cited option is to require that such systems use transactional interfaces. Since the ATC database is, presumably, transactional, this seems like an excellent match with the need. Yet it isn't difficult to see that if we do so, the client program would *still* need to be aware of the fail-over, and indeed all of the questions

[9] Some CORBA ORB's support an automatic rebind operation, but only in cases where there is no significant client-side state associated with the server. Even so, the mechanism is only a partial solution (Figure 4).

just posed remain. We still need a way to learn the status of an interrupted remote operation. Moreover, transactional rollback is incompatible with taking external actions while the transaction is running, and most transactional systems are restricted to short-lived transactions, whereas many applications perform what would logically be considered a "single" transaction over a very long period of time. Indeed, the more one studies this alternative, the less it seems to resolve any of the issues just raised.

Bruce Lindsey, the guru of transactional systems, has often commented that the only problem with the transaction model is the one we have been discussing. When the server unilaterally aborts the client's request, for whatever reason, we can be sure that the server state is "clean" in the sense of the transactional model. But what exactly should the client do? The model offers no insight, and the kinds of questions raised above are typical. Masking server failures from the client is simply a very hard problem, and doing it in a way that guarantees rapid response is even harder.

We've assumed that the server will restart from a checkpoint. An alternative is to create a group of servers, keeping them synchronized using active replication [BvR96]. One might hope to do this using object groups, particularly because CORBA includes a server group specification. Unfortunately, this CORBA feature turns out to be intended for use at binding time. A CORBA server group is a set of servers, any of which is equally suitable for bindings by application programs needing the associated service. The resulting abstraction lacks standard mechanisms for fail-over while a connection between an application and a server is active, with the exception of mechanisms like the "rebind" for cases where the server is stateless. Even here a failure might be visible to the client in the form of an exception return from a method invocation.

1. **Client is bound to a healthy server instance, issues method invocation**
2. **Server fails, potentially garbling or losing a partially transmitted response**
3. **Client detects loss of connection, may first receive a garbled response**
4. **Client discards incomplete response, if any, without notifying the application**
5. **Client waits until backup server is launched and running. This may take some time; client must distinguish between "normal" delay and "abnormal" delay (server that cannot restart)**
6. **Client rebinds to a backup server instance, re-establishes any connection state associated with previous connection (for example, many servers require "registration" by clients)**
7. **Client frees (garbage collects) old server stub**
8. **If method invoked in (1) was not idempotent, client contacts the server to determine the outcome of its request (this implies a unique "request identification" scheme)**
9. **Server waits until interrupted requests have been completed (committed) or aborted, determines the outcome, and returns this to the caller.**
10. **If the request was idempotent, or if the server has no record of completion of the request, the client reissues the method invocation**
11. **Client returns the result to the application layer**

Figure 4: Steps in recovering from a server failure. Current object-oriented architectures only support recovery for idempotent requests, which can be reissued without ill effect. Such systems typically do not provide for re-registration of a client, will not wait for a server to recover if this step takes any time, and will throw an exception if a failure occurs while an invocation is pending

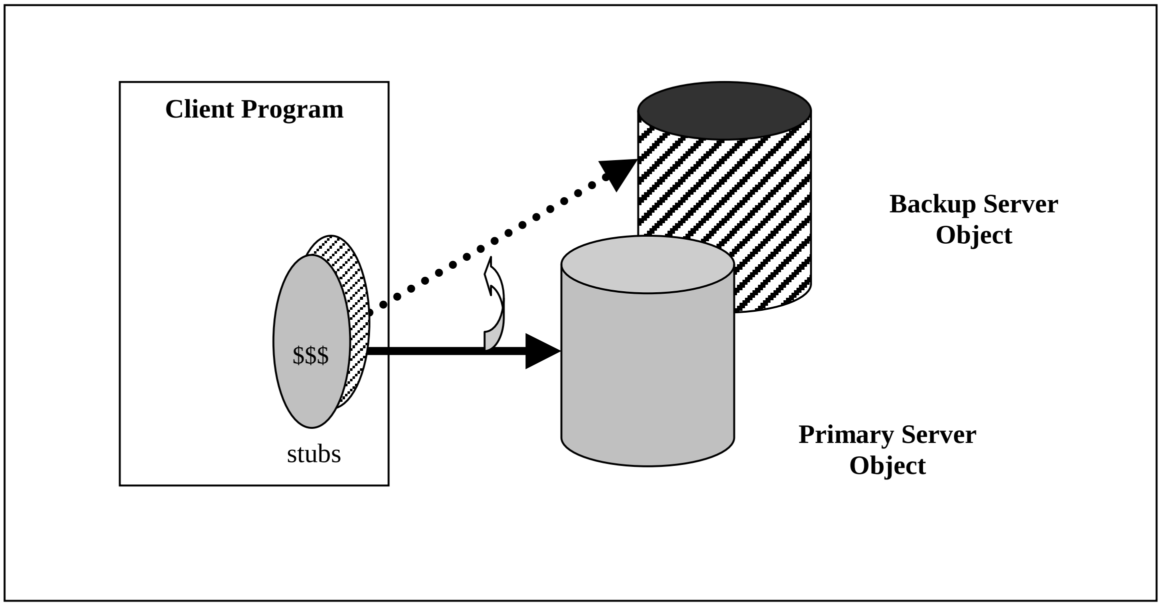

Figure 5: If a primary server fails, the client may wish to "fail-over" to a backup. However, the standard object-oriented architectures lack standards for trapping the associated exceptions, cleaning up state and cached data ($$$) from the initial binding, and transparently creating a new binding using a new instance of the server stub. Life cycle services lack support for restarting server objects with complex persistent state and dependencies on other objects. This creates obstacles for the developer of a client program which must tolerate failures.

For these reasons, developers of object oriented systems using CORBA or its counterparts tend to reject fail-over solutions that require rebinding to the server. The loss of the connection is too disruptive, and it is too likely that in some cases, exceptions will be thrown up to the end-user. The usual alternative is to seek a completely transparent form of failure handling which masks server reconfigurations from the client. If we could make failure handling completely transparent to the client of a CORBA or COM or Java RMI server, the resulting solution would fit naturally into those architectures (Figure 6). In effect, the architectures *presume* that we can offer transparent fail-over on the server side, and they encourage client-side programming in a style that really leaves no alternative.

What this means is that we are expected to solve the following problem. A single, non-replicated client program invokes a method on what appears to be a single, non-replicated remote server. The server is actually implemented by a group of servers (or by a single server that can be automatically restarted). Should the server fail, even right in the middle of a method invocation, the client sees nothing out of the ordinary at all. At worst, the behavior of the server should be behavior that could also arise if the network lost a few messages, or experienced a transient partitioning failure. Our need is for a magic wand that can be waived over a server to transform it into such a replicated fault-tolerant server.

Unfortunately, this problem is not solvable in a general sense. The best-known solutions are highly restrictive. They typically require that the server be deterministic: its actions and state are completely determined by the sequence of invocations that have been performed upon it. Determinism of this sort may be at odds with techniques as simple as checking the time, using multithreading, or exploiting asynchronous I/O. Sophisticated servers often depend upon multithreading for performance. Indeed, the class of sophisticated servers that are also deterministic may be empty.

Yet a further problem arises because connections from client systems to servers are often made over TCP. If a server fails and a new server takes over on its behalf, the TCP connection would normally break, since one of its endpoints is gone. However, if we wish to hide server failures from the client, such connection loss must be concealed. This leads to the idea of moving one endpoint of the TCP connection from the failed server to the one taking over – a mechanism sometimes called TCP "connection stealing". The necessary steps are discussed in [Birman97]. To summarize, TCP connection stealing can sometimes be

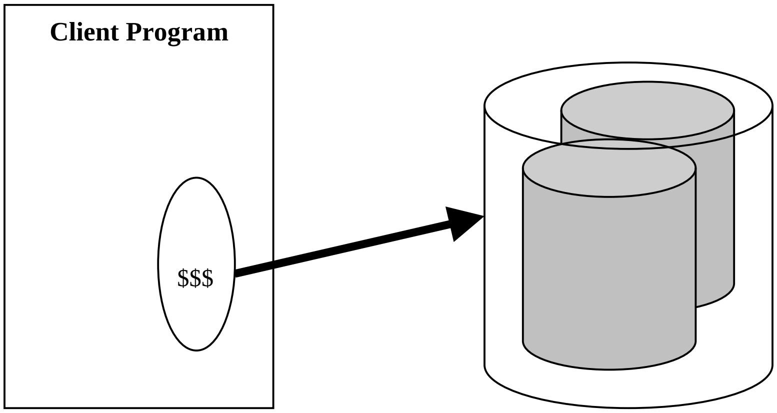

Figure 6: Transparent failure handling, a "holy grail" for the object-oriented computing community. The idea is to conceal the failure from the client. But in solving this problem, systems accept restrictions on what servers can do and how they are implemented, which often unacceptably limits the flexibility of the developer. For example the servers are required to be determinstic, may not be permitted the use of callback interfaces, etc.

supported, the general version of the problem is extremely complex and very hard to support. The problem is easier with web servers, which use a separate TCP connection for each HTTP request (Friedman implemented a solution to this version of TCP connection stealing and demonstrated that servers using this approach can even provide guaranteed real-time failover [Friedman97]).

But that's not all. Transparent replication schemes also prohibit the server from doing things which servers often do, such as handling out a reference for use in callbacks. The problem here is that the service was coded as a non-replicated object, and was then automatically transformed into a replicated object group. Thus, the server instance which hands out the reference won't "know" it is really a replica of the server group as a whole. The reference it is likely to hand out will probably be a reference to itself – a single instance of the service – rather than one on the server group as a whole. There is no transparent way to intercept the object reference as it travels back to the client application so as to substitute the correct one, hence the replication scheme simply prohibits such callback handles, or forces the developer to use a very unnatural one. A further restriction involves dynamic invocation interfaces (also known as "factory" interfaces) which are incompatible with the replication scheme used in the ORB's listed above.

Finally, typical object management (life-cycle) services lack means by which a server can request that it be automatically restarted from a checkpoint after a failure. This can be a non-trivial problem, since there are often dependencies between servers, so that one must be running before another can be launched. Although modern clustering technologies such as NT Clusters address such issues, in doing so NT/DCOM goes well beyond the features of most standard ORBs.

Thus while there is a clear need to make replication and reliable multicast available in object oriented settings, the need seems to be for a kind of object-oriented or component-ware based solution which differs from the systems developed to date, and which is probably not realizable. Of course, one can also argue the contrary: for example, in [Ghormley95] the authors observe that trying to catch and mask errors in every module of a complex system simply increases complexity without necessarily improving robustness. But the ability to *selectively* catch and mask failures in a manner supported by the architecture is clearly a positive thing, provided that the mechanism is not thrust upon the user even when it serves no useful role.

Action	Reliable Multicast	Object-Oriented	Component
general style	Coded directly, often using the programming language in which protocols were implemented.	A class of “replicated data” objects is supported. Instances are required to implement standard interfaces, but can omit undesired functionality.	Defines a new class of “managed components”. Each component must be assigned to a class. A scripting language catches events corresponding to various transitions and specifies responses. Components registered using drag-and-drop in a graphical management GUI.
join	Joining process invokes the group join primitive. Leader receives an *join_req* upcall, does a *join_ok* downcall. A *new_view* upcall occurs in all members (new and old).	Each instance of a replicated data object class has a hidden group-name associated with it. When the class is instantiated at a new location, the new member is automatically joined to the existing members.	Each system component has an associated “type” and is automatically joined to a corresponding instances group when launched. This may trigger a scripted sequence of acceptance actions within existing components.
state transfer to a joining member	Implemented as an algorithm triggered by delivery of new view. The reliable multicast system handles synchronization so that *prepare_state* and *receive_state* upcalls occur at appropriate times in the execution (see Figure 2).	Each object instance must implement a method to form a persistent state object and to load state from one (in effect, a checkpoint method). These methods are automatically involved at appropriate times. The OO subsystem moves a copy of the state from an old member to a new joining member to initiate it.	Expressed in terms of checkpoint objects from which new components are initialized. Component systems also support dependency hierarchies: *before launching B, A must be online.* Components are viewed as transitioning through states: *launch, initialize, waiting, online, checkpointing, offline*, etc.
leave **fail**	Failure detectors place real-time requirements on programs; a program that fails to complete required actions on time is considered faulty. After failure a new group view is computed and reported to the application.	Failure detection is by some form of liveness probe. Delivery of a new view is an event and the object is notified by invocation of the corresponding event handler.	Failure detection may be done by the operating system or by liveness probes. View change events trigger the execution of scripted responses. For example, if a critical component fails, the script could launch a new instance. Actions may also be supported when a component-group is first created or when it has no remaining members.
send multicast	Application prepares a message invokes the *cast* downcall interface.	Object supports *read* and *update* interfaces. Each update operation is multicast to all replicas, which independently and concurrently perform the operation.	Used to notify components of state transitions. However, individual components would not normally invoke multicast interfaces directly.
receive multicast	*receive* upcall invoked, with the message as an argument.	The update interface of the object replicas are invoked with appropriate arguments.	State transitions trigger scripted response actions.

Table II: Major events in a group communication environment

To summarize:

1. Object-oriented presentations of some reliable multicast abstractions are viable and likely to be successful. The iBus message bus technology is a good example. However, successes have been limited by a mixture of architectural constraints associated with standards and unrealistic expectations for transparent failure handling.

2. Object-oriented toolkits have tended to adopt a C++ language orientation with procedural interfaces, rather than offering "replicated objects" or supporting a component-ware orientation. Both represent interesting opportunity areas.

3. Existing architectural frameworks, such as CORBA, COM and RMI have inadequate mechanisms for catching exceptions thrown when servers fail and rebinding the client application to alternatives. The necessary steps in recovery are non-trivial and poorly matched to the existing standards. This forces users to seek technologies promising transparency.

4. Providing transparent server replication is only possible if extremely restrictive server-side assumptions are made. This stems both from the presumption that the client-side will not participate in failure handling (which would otherwise not be very transparent), and from the need to assume complete determinism. The resulting server replication technology is not matched to developer requirements, and has not been popular.

5. Similar problems are well known from the transactional community. In these systems, the analogous problem arises on the client-side when a connection to a server fails: this broken connection nay not imply that a pending transaction was aborted, and there is no simple and general way to mask the uncertainty from the application.

6. Architectures in which server failure is treated as an exception which can be caught and handled in a standardized manner, if desired, seem feasible but have received little (if any) study.

5 Conclusions

Our review leads to a series of conclusions concerning the embedding of reliable multicast into future middleware environments, particularly object-oriented ones. Table I summarizes the practical implications of these point, which are as follows:

1. Reliable multicast has been applied successfully in a number of very demanding settings, but success has often come at the cost of considerable design complexity and required great care by developers. The simplest successes correspond to application with very weak reliability goals.

2. Scaling represents an ongoing challenge, particularly for the strongest fault-tolerance model (virtual synchrony). Although hierarchical group structures can provide scalability for virtually synchronous process groups, the more promising option may be to move towards systems which use a mixture of styles of reliable multicast. For example, Bimodal Multicast, SRM and RMTP all scale well, but each comes with limitations of its own. One can imagine hybrid systems which use a mixture of virtual synchrony where there is an especially stringent consistency requirement (such as the French ATC system), but employ a more scalable protocol for high volume, large-scale data distribution. The New York Stock Exchange can be understood as an example of such a hybrid structure, since data is distributed using different protocols than the ones used to track membership and system configuration data. The success of this system may point to a larger family of such solutions.

Application	Success Story	Failure
Stock Market	Hierarchical deployments of virtual synchrony offer full replication of stock market or, in case of NYSE, reliable management of data diffusion architecture	Systems are complex, potential scaling limits. Concerns about throughput stability force system to aggressively drop machines which malfunction even in limited ways Virtual synchrony may not be appropriate for the most demanding aspects of data distribution because of scalaibility problems.
Brokerage Systems	Mixture of IP multicast with unicast and self-administration give high reliability and rapid, automated restart. Very scalable.	Individual applications may see drop-outs or other problems. No guarantee of consistency. Architecture is complex
Air Traffic Control Systems	Cautious partitioning into console replication groups yield flexible, fault-tolerant architecture in PHIDEAS effort. Virtual synchrony guarantees were valuable in assuring correctness.	American ATC effort stumbled, perhaps in part because of difficulties using Δ-T atomic multicast to build fault-tolerant pipeline-style applications.
Data Replication	For unchanging data (soft state), SRM is widely cited as a success. Virtual synchrony can be used to replicate changing data and supports a way to initialize new copies (state transfer)	Stability of solutions under load when scaled to a large setting demands care in application design. Works best for small numbers of replicas and loads far from the performance limits of the network
Replication of Computation or Abstract Data Types	CORBA-compatible object replication demonstrated for deterministic servers. In general, replication is possible when the computation can be reproduced by playing identical event sequences into multiple copies of the program or object.	Non-standard ORB and many restrictions on server represent severe limitations. Yet CORBA architecture makes other forms of fault-tolerance very hard to support because of transparency assumptions implicit in client-side architecture. Determinism constraints are very severe.
Toolkits	Underlie most successes, they embed reliable multicast into general-purpose, easily used interfaces and libraries.	Tools often have hidden limitations; when used in ways not expected by original tool developers, they malfunction or become very costly.

Table III: Some successes and failures identifiable from review of applications.

3. Any successful system must preserve flexibility, so that the intended users can customize the reliability characteristics of the technology to match the application. Fortunately, there is a substantial body of research on precisely this problem and the feasibility of providing a solution with the necessary properties is well established. Broadly, this possibility involves separating the reliable multicast interface from its implementation, so that the user can plug in an implementation with the desired semantics. In a generalization of this idea, the plug-ins are designed to stack, permitting properties to be composed together at runtime in the manner of the old UNIX streams architecture.

4. Where possible, reliable multicast should be presented through a natural system abstraction, such as system management "environments", one-to-many file copying tools and object-oriented publish-subscribe tools. Not all applications make use of detailed information concerning process group membership and, for such applications, concealing the details also leaves the middleware tool with freedom to optimize the mapping from application-visible functionality to communication groups and multicast.

5. Object oriented systems that support replication need to look more closely at exposing issues of replication and failure-handling, particularly where a critical service is replicated so that multiple servers can provide the desired functionality. At present, most object oriented architectures are designed with the implicit assumption that once an application binds to a server, the server object will not fail during the session. But after nearly ten years of research on supporting this abstraction with server replication tools, we need to explore alternatives which relax transparency to increase functionality. Although a server can be replicated transparently, the constraints upon the class of servers amenable to such techniques are too limiting. In particular, transparent replication of a server can only be done if the server is deterministic, and all the most widely used object-oriented programming tools support multi-threading and other sources of non-determinism. Even so, transparent replication may be problematic if TCP is used to connect to the server.

6 Future Prospects

Reliable multicast is clearly an accepted technology in modern networked computing systems. A great variety of uses can be identified, and some represent notable successes. One could argue, in fact, that the air traffic control and stock exchange systems surveyed here represent some of the most striking successes to date in demonstrating demanding, mission-critical functionality on network architectures.

To succeed, reliable multicast technology packages need to be flexible enough to match their costs and properties to the needs of the application. Although all reliable multicast systems share some notion of a group of participants, a means to join and leave that group, and a means for communicating to it, the details of how groups behave, how group membership is synchronized and reported to members (if at all), and the properties of the reliable multicast primitives available vary widely. A one-size-fits-all mentality is unlikely to succeed, but interface standards which focus on the common features of these solutions and can be customized with plug-in protocol modules make good sense.

With regard to presentation to the developer, we can identify several layers of potential interface. One is the layer just cited: an interface below which reliable multicast can be plugged in as a module (or a stack of modules). Such an interface provides syntax without semantics, and could become a standard component of modern networked operating systems.

A second layer arises when we talk about component-ware or object-oriented embeddings of the technology into standard distributed computing architectures such as CORBA, RMI and COM. Provided that we don't insist that the purpose of reliable multicast is purely to support transparent server replication, there seems to be excellent options for integrating reliable multicast with standard architectures in ways that seem natural and compatible with other forms of non-group-oriented software.

The next layer is concerned with middleware functionality, such as publish-subscribe message buses, PVM or MPI, toolkits, and technologies for automating server replication. Solutions of these sorts are extremely popular with the classes of application developer they target, and would easily be able to make use of O/S provided reliable multicast functionality if there were performance or standardization benefits to doing so.

A final layer is concerned with end-user application development tools and standardized end-user functionality, such as multicast file transfers, scripts or web-based management interfaces for control of networked or cluster-based applications. At this very high level, reliable multicast enables desired functionality, but is not directly evident to the user.

This paper does not offer a new generation of technology in which reliable multicast might play the roles just identified, although recent research on projects such as the author's Ensemble effort certainly is exploring the question. The intent of the paper, is simply to tease out the lessons that can be learned from prior successes and failures. We believe that in the area of reliable multicast, the body of experience is finally achieving the form of critical mass needed to support the sort of inquiry which has long been common in fields such as operating systems research or database research.

Broadly, we see reason to believe that reliable multicast is finally converging on its most natural roles in mainstream distributed computing environments. The evolution of those environments takes them closer and closer to the model that seems most likely to work well for users. The success stories for the technology are impressive in the sense that they involve extremely important, difficult problems, which have not be solved in other ways. The failures point to limitations, but not show-stoppers. Meanwhile, with the growing public and government attention on problems of reliability, there is pressure or the major vendors to offer solutions of the sorts suggested here.

7 Acknowledgements

Jim Gray's, Marc Shapiro's and Robbert van Renesse's extensive and detailed comments on earlier versions of this paper are gratefully acknowledged. The author is also grateful for feedback and suggestions by David Bakken, Carl Lagoze, Fred Schneider and Joe Sventek.

8 References

[Bartlett91] J. Bartlett. A Non-Stop Kernel. Proceedings of the 8th ACM Symposium on Operating Systems Principles (Dec. 1981), 22-29.

[Birman97] K.P. Birman. *Building Secure and Reliable Network Applications.* Manning Publishing Company and Prentice Hall, Greenwich, CT. Jan. 1997. URL: http://www.browsebooks.com/Birman/index.html

[BH98] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, Yaron Minsky. Bimodal Multicast. Cornell University Dept. of Computer Science Technical Report TR-98-1683 (May 1998).

[BJ867] Kenneth P. Birman and Thomas A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. 11th ACM Symposium on Operating Systems Principles, Dec 1987.

[BvR94] Kenneth P. Birman and Robbert van Renesse, eds. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press 1994, Los Alamitos.

[BvR96] Kenneth P. Birman and Robbert van Renesse . Software for Reliable Networks. Scientific American, 274(5): 64-69 (May, 1996).

[CASD85] Flaviu Cristian, H. Aghili, Ray Strong and Danny Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *Proc. 15th International FTCS (1985),* 200-206. See also "Atomic Broadcast in a Real-Time Environment" in *Fault Tolerant Distributed Computing*, Springer-Verlag LNCS 448 (1990), 51-71.

[CDD96] Flaviu Cristian, Bob Dancey, Jon Dehn. Fault-Tolerance in Air Traffic Control Systems. *ACM Transactions on Computer Systems* 14:3 (Aug. 1996), 265-286.

[Cooper85] Eric Cooper. Replicated Distributed Programs. Proceedings of the 10th ACM Symposium on Operating Systems Principles (Orcas Island), Dec. 1985, 63-78.

[CS93] David Cheriton and Dale Skeen. Understanding the Limitations of Causal and Totally Ordered Multicast. *Proceedings of the 1993 Symposium on Operating Systems Principles*, Dec. 1993.

[CZ85] David R. Cheriton and Willy Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems* 3:2 (May 1985), 77-107.

[FB96] Roy Freidman and Ken Birman. Using Group Communication Technology to Implement a Reliable and Scalable Distributed IN Coprocessor. Proceedings Telecommunications Information Network Architecture 96, Heidleberg. VDE-Verlag 1996, 25-42.

[FG97] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, P. Gauthier. Cluster-Based Scalable Network Services. Proceedings of the 16th ACM Symposium on Operating Systems Principles. Dec. 1997, 78-91.

[FJ95] Sally Floyd, Van Jacobson, Steven McCanne, Ching-Gung Liu, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *Proceedings of the '95 Symposium on Communication Architectures and Protocols (SIGCOMM).* ACM. August 1995, Cambridge MA. http://www-nrg.ee.lbl.gov/floyd/srm.html

[Freidman97] Roy Friedman. TCP connection-stealing support for a scalable cluster-style web server. Unpublished technical report, 1997. Code available in Cornell's Ensemble distribution.

[Glade98] Brad Glade. *A Scalable Architecture for Reliable Publish/Subscribe Communication in Distributed Systems.* Ph.D. dissertation, Cornell University Dept. of Computer Science, May 1998.

[Ghormley95] Ghormley, et. al. GLUnix: A Global Layer Unix for a Network of Workstations, Software Practice and Experience (24)9, 929–961, 1995

[Guo98] Katie Guo. *Scalable Message Stability Detection Protocols.* Ph.D. dissertation, Cornell University Dept. of Computer Science. May 1998.

[Hayden97] Mark G. Hayden. *The Ensemble System.* Ph.D. dissertation, Cornell University Dept. of Computer Science. December 1997.

[iBus98] *Need a reference to Silvano Maffeis and the iBus architecture.*

[Kaa94] Kaashoek, M.F., Tanenbaum, A.S., and Verstoep, K.: "Group Communication in Amoeba and its Applications," Distributed Systems Engineering Journal, vol 1, pp. 48-58, July 1993

[KLS85] N. Kronenberg, H. Levy and W. Strecker. VAXClusters: A Closely-Coupled Distributed System. *ACM Trans. on Computer Systems*, 4:2 (May 1986), 130-146.

[Kopetz97] Hermann Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer International, volume 395 (April 1997).

[Liu97] Ching-Gung Liu. *Error Recovery in Scalable Reliable Multicast* (Ph.D. dissertation), University of Southern California, Dec 1997.

[LMJ97] Craig Labovitz, G. Robert Malan, Farnam Jahanian, "Internet Routing Instability". Proc. SIGCOMM `97, Sept. 1997, 115-126.

[LP96] J. C. Lin and Sanjoy Paul, "A Reliable Multicast Transport Protocol" Proceedings of IEEE INFOCOM '96, Pages 1414-1424. http://www.bell-labs.com/user/sanjoy/rmtp.ps

[Lucas98] Matt Lucas. *Efficient Data Distribution in Large-Scale Multicast Networks* (Ph.D. dissertation), Dept. of Computer Science, University of Virginia, May 1998.

[NMM97] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith. Replica Consistency of CORBA Objects in Partitionable Distributed Systems. Distributed Systems Engineering Journal, vol. 4, no. 3 (September 1997), pp 139-150.

[OP93] Brian Oki, Manfred Pfluegel, Alex Siegel and Dale Skeen. The Information Bus – An Architecture for Extensive Distributed Systems. Proceedings of the 1993 ACM Symposium on Operating Systems Principles, Dec. 1993, 44-57.

[OSR94] *Various authors.* Rebuttals to [CS93] appearing in *Operating Systems Review,* January 1994.

[Pfister95] G. F. Pfister. *In Search of Clusters.* Prentice Hall (Englewood Cliffs), 1995.

[PS97] Rico Piantoni and Constantin Stancescu. Implementing the Swiss Exchange Trading System. FTCS 27 (Seattle, WA), June 1997, 309-313.

[PHIDEAS] http://www.stna.dgac.fr/projects/, http://aigle.stna7.stna.dgac.fr/

[PSLB97] Sanjoy Paul, K. K. Sabnani, J. C. Lin, S. Bhattacharyya "Reliable Multicast Transport Protocol (RMTP)", IEEE Journal on Selected Areas in Communications, special issue on Network Support for Multipoint Communication, April 97, Vol 15, No. 3, http://www.bell-labs.com/user/sanjoy/rmtp2.ps

[RBM96] Robbert van Renesse, Kenneth P. Birman, Silvano Maffeis. Horus: A Flexible Group Communication System. Communications of the ACM 39:4 (April 1996), 76-83.

[Tham90] Philip Thambidurai. *State of the Fault-Tolerance Documentation.* IBM Internal Memo, Systems Integration Division, January 8, 1990.

[Vaysburd98] Alexey Vaysburd. *Building Reliable Interoperable Distributed Objects with the Maestro Tools*. Ph.D. dissertation, Cornell University Dept. of Computer Science, May 1998.

[VD98] Vogels, W., Dumitriu, D., Birman, K. Gamache, R., Short, R., Vert, J., Massa, M., Barrera, J., and Gray, J., The Design and Architecture of the Microsoft Cluster Service -- A Practical Approach to High-Availability and Scalability, Proceedings of the 28th symposium on Fault-Tolerant Computing, Munich, Germany, June 1998. Available through http://www.cs.cornell.edu/rdc/mscs/ftcs28

[vR96] Robbert van Renesse, Masking the Overhead of Protocol Layering. Proceeding of the 1996 ACM SIGCOMM Conference, Stanford, Sept. 1996