UC Santa Cruz

UC Santa Cruz Previously Published Works

Title

Quorum-oriented multicast protocols for data replication

Permalink

https://escholarship.org/uc/item/89d0838q

Authors

Golding, RA
Long, DDE

Publication Date

1992

DOI

10.1109/icde.1992.213160

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/2514886

Quorum-Oriented Multicast Protocols for Data Replication

Article · August 1992

Source: CiteSeer

CITATION
1

READS
17

2 authors, including:

Richard Golding
NASA
60 PUBLICATIONS **2,163** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Shared / NAS Storage View project

All content following this page was uploaded by Richard Golding on 09 January 2013.

The user has requested enhancement of the downloaded file.

Quorum-oriented Multicast Protocols for Data Replication

Richard A. Golding* and Darrell D. E. Long †
Computer and Information Sciences Board
University of California, Santa Cruz
Santa Cruz, CA 95064

Abstract

Many wide-area distributed applications use replicated data to improve the availability of the data, and to improve access latency by locating copies of the data near to their use. This paper presents a new family of communication protocols, called quorum multicasts, *that provide efficient communication services for widely replicated data. Quorum multicasts are similar to ordinary multicasts, which deliver a message to a set of destinations. The new protocols extend this model by allowing delivery to a subset of the destinations, selected according to distance or expected data currency. These protocols provide well-defined failure semantics, and can distinguish between communication failure and replica failure with high probability. We have evaluated their performance, taking measurements communication latency and failure in the Internet. A simulation study of quorum multicasts shows that they provide low latency and require few messages. A second study that measured a test application running at several sites confirmed these results.*

1 Introduction

We are investigating a new family of communication protocols that we term *quorum multicasts*. These protocols are efficient and convenient for implementing replicated and distributed applications. They provide a communication mechanism similar to traditional multicast protocols, but unlike other multicast protocols, the quorum multicast protocols communicate with just a *subset* of the set of destinations. The quorum multicast protocols can dynamically select this subset to be the closest available destinations, limiting the portion of the internetwork affected by any particular multicast. Such protocols are useful when implementing replication protocols based on voting or when communicating with one of a group of replicas in a fault-tolerant manner.

When systems dispersed over a wide area are to access the same data, that data can be *replicated*. Several sites in the network maintain copies of the replicated data. Replicated data can be more fault-tolerant than unreplicated data, and its use can improve performance by locating copies of the data near to their use. Copies of replicated data are held at a number of *replicas* that consist of storage and a process that maintains the data copy. A *client* process can communicate with the replicas to read or update the data. Both the clients and the replicas reside on *hosts*, which are connected using an internetwork consisting of local-area networks with *gateways* and *point-to-point links*. When a host sends a message to another host, the message will be forwarded through (perhaps many) gateways and links to its destination.

Communication between the client and the replicas is performed according to a *replication protocol* that provides the client with the illusion of a single data object. This is more complex than using a single copy of the data, since operations must be coordinated among the replicas. Replication protocols hide this complexity by providing a set of operations performed by clients and a set performed by replicas: clients can read and update the data, while replicas perform operations such as failure recovery, creation of new replicas, and information propagation between replicas.

Replication protocols are constructed using lower-level communication mechanisms. Most networks provide one-to-one datagram and byte stream protocols, and many provide limited broadcast or multicast facilities. This paper is concerned with a version of multicast protocols, the *quorum multicast*, that is particularly well-suited for implementing a replication protocol.

Many existing wide-area systems, such as airline reservation systems and library card catalogues, do not use replication techniques, relying instead on large central systems. This is in part due to the beliefs that replication is inconvenient when compared to centralized solutions and that replicated data provide poor end-to-end performance. Replication can provide good performance when quorum multicasts are used. In other work [8] we have shown that they are a convenient mechanism for constructing replication protocols.

2 Performance measures

There are several measures that can be used to evaluate the performance of replicated data. These include the latency of operations, the amount of message traffic caused by an operation, and the data availability. Though the ideal protocol would provide the highest availability with the lowest traffic and latency, trade-offs between these measures must be made. The scale of the Internet further complicates protocol performance, for a protocol that performs well in a local-area network may not scale to the world-wide Internet. We will show how quorum multicast

*Supported in part by the Concurrent Systems Project at Hewlett-Packard Laboratories.

†Supported in part by the National Science Foundation under Grant NSF CCR–9111220, by the Institute for Scientific Computing Research at Lawrence Livermore National Laboratory, and by faculty research funds from the University of California, Santa Cruz

0-8186-2545-7/92 $3.00 © 1992 IEEE

protocols contribute to good availability while using fewer messages and requiring less latency than a simple multicast.

Wide-area networks contain many more systems that might share resources, implying that the potential load on highly utilized components in an internetwork is much higher than the load on components in a LAN. The potential users of an application at a local site number at most a few hundred to a few thousand, while the number of potential users of a wide-area application is orders of magnitude larger. Systems such as the Domain Name Service [12] and Usenet [15] show that the load on some applications scales with the number of users.

Sending a message between any two hosts on an internetwork requires a variable amount of time, termed the *communication latency*. The latency depends on the load on the network and the available routes. In a wide-area internetwork, communication latency can be the predominant delay for operations on the replicated data, which often involve only a few disk accesses and a small amount of computation. Communication with nearby replicas requires a few milliseconds, while access times for distant replicas often require several hundred milliseconds for communications across a continent, and as long as several seconds when satellites are involved.

The amount of *message traffic* required for an operation governs the degree the operation will interfere with other communication in the network. The number of messages, their size, and their destinations contribute to this effect. A communication protocol will cause less interference if it can send a message to a nearby host since the message will traverse fewer intermediaries. Broadcast messages on a LAN allow replication protocols to send requests to all replicas in one message, while a separate message must generally be sent to each replica in an internetwork.

The *availability* of a service can be defined as the likelihood of providing correct service at a given instant [16]. This must be contrasted with *reliability*, the probability of providing correct service during a period of time. Highly-available applications must be *fault-tolerant*: they must continue to provide service even when parts of the system have failed. Internetworks are *unreliable*, meaning they lose and duplicate messages from time to time, and may deliver them out of order. Hosts can use timeouts and acknowledgments to detect with high probability that a message has not been received.

3 Multicast protocols

We have developed a family of *quorum multicast* communication protocols that can take advantage of good replica placement. These protocols send a multicast to a subset of a group of replicas, rather than to the entire group. They first use the closest available replicas, falling back on more distant replicas when nearby ones are unavailable, and are parameterized so that the replication protocol can provide hints to further improve performance. Replication protocols can be implemented using them, limiting the cost of message traffic and using nearby replicas for low latency.

A communication protocol that is to work well in internetworks must address their particular performance characteristics: long, variable latency and occasional high message loss. These characteristics make some techniques used for replication in a local-area network inappropriate for internetwork use. The protocols should not require broadcast, but instead send messages to replicas in a more controlled fashion. The protocols should be sensitive to the communication latency of replicas, and should tend to communicate with nearby replicas, providing lower access latencies and limiting the portion of the internetwork affected by an access. They should respond to changes in network topology and performance, perhaps by communicating with different replicas. The protocols should also address the problems associated with transient failures by resending messages to replicas.

3.1 Existing multicast protocols

Quorum multicast protocols are a specialization of ordinary *multicast protocols*. Ordinary multicast sends a message to a set of destinations in one operation. Replication protocols can use multicast to send a message to all available replicas, later receiving a number of responses from some or all of them. Simple request-response multicast protocols define a simple interface:

multicast(message, replica set) → reply set

The message is sent to all replicas in the set.
The operation reports no exceptions.

Several researchers have considered the problem of providing a multicast facility on an internetwork that has no inherent broadcast capability. Boggs [2] developed *directed broadcast* capabilities for internetworks that deliver a message to all hosts on a network segment, even if the sender is not connected to that segment. Directed broadcast cannot provide significant performance gain when each replica is located on a separate subnet. Garcia-Molina and Kogan [6] extend internetwork broadcast algorithms with a novel mechanism that provides a *reliable* multicast facility on an internetwork with unreliable multicast, even if the network can partition.

In contrast to this work, the Isis system [1] provides a distributed programming environment based on reliable *atomic multicast* in a local area. It provides specialized protocols to multicast to a process group, providing strong guarantees on the ordering and atomicity of delivery and failure detection.

Many other researchers have investigated multicast protocols as part of distributed operating systems. Cheriton [3] used multicast with distributed *process groups* as a primary communication mechanism in the V system, a locally-distributed operating system at Stanford University.

Some RPC systems have provided one-to-many and many-to-many communication semantics similar to a request-response multicast protocol. The Circus replicated RPC system [5] extended RPC to include replicated calls to a group of processes, called a troupe; each process in the troupe was required to perform the same computation and issue the same RPCs in the same order. The Parallel Remote Procedure Call (PARPC) system [10] implemented one-to-many replicated procedure calls. Replication protocols could be implemented in both these system using only a few lines of code.

3.2 Quorum multicast protocols

We now present designs for a variant of multicast, the *quorum multicast* protocols. The design is guided by four goals: make use of proximity; communicate with subsets of the destinations; adapt to changing network conditions;

and minimize latency and message traffic. Some of these goals conflict, so we will present protocols that trade one for another.

Quorum multicast protocols send a message to a *subset* of the destinations. For example, many replication protocols require that a simple majority of the replicas perform an operation, while others require only one or two replicas.

The minimum number of destinations required is specified by a *reply count* parameter. In many cases the message need only be sent to enough of the closest replicas to satisfy the requirement, avoiding message traffic to the most distant replicas. 'Closeness' can be defined as 'fastest to respond' or as 'least number of hops'. Of course, this multicast should be fault-tolerant, using more distant replicas when those nearby are unavailable. The protocol may not be able to meet the reply count if some of the replicas are unavailable. Replicas can be unavailable due to host failure, replica failure (perhaps due to insufficient resources), network gateway or link failure, or controlled shutdown. Since replication protocols generally require request-response communication, the responses to a multicast serve as acknowledgment that the message was received and processed.

The semantics of quorum multicast define the interface:

quorum-multicast(message, replica set, reply count) → reply set

The message is sent to at least a reply count of the replicas. Exceptions: **reply count not met.**

The quorum multicast protocols maintain an *expected communication latency* for each possible host. When a request is issued to communicate with q members of a set of replicas, the communication protocol can order the set by expected latency and communicate with the q closest replicas. If responses are not received from all q within a certain time, then messages can be sent to more distant replicas. The delay before sending to distant replicas is determined by the parameter d. The expected latency can be determined by measuring recent performance, on the assumption that replicated operations will be performed much more often than the structure of the network changes. Many Internet protocols use *moving averages* of recent behavior to determine such expectations [4].

The two extremes of sending all messages at once or sending as few messages as possible are not always appropriate for all applications. Three of the new protocols are parameterized by a *delay parameter* $0 \leq d \leq 1$ that allows an application to specify an intermediate position, where sending more messages than strictly necessary is used to improve operation latency. When $d = 0$, the protocol will not wait to send to distant replicas. When $d = 1$, the protocol waits until a message failure is reported before sending to distant replicas. Since message failure is detected using timeout, when $d < 1$ the protocol will wait some fraction of the timeout period before sending to more distant replicas.

If the communication protocol has not received a reply from a replica after some amount of time, the protocol assumes that the message has failed. After some number of messages have failed, the protocol declares the replica unavailable and does not attempt to retransmit messages until the next communication request. Some protocols will only try sending a message to a replica once, while other protocols will try several times before giving up. This *persistence* is a tunable parameter in one of our protocols. Once a protocol has declared enough replicas unavailable, it will return a negative indication to the replication protocol and abandon the operation. Our measurements of the Internet, detailed in another report [9], show that short transient failures comprise more than three-fourths all message failures. They also show that long transient failures are uncommon, so a protocol can confidently declare host failure after observing only a few lost messages.

In the next sections we will present four quorum multicast protocols. The first, called **naive**, is a straightforward implementation of multicast that sends messages to all replicas, providing a baseline to which the other protocols can be compared. The second, called **reschedule**, uses the delay parameter to send to fewer replicas. The third and fourth, called **retry** and **count**, send to replicas according to the delay parameter, but will retry messages to replicas after a first message has failed.

3.2.1 The naive protocol

The first protocol, a simple multicast, is called **naive**. It sends one message iteratively to all replicas. Replies are counted, and when a reply count has been obtained the protocol returns, indicating success. When a reply or a failure has been observed for every replica without reaching the reply count it declares the access a failure.

There are two problems with this protocol: it neither accounts for transient communication failures nor uses proximity to improve performance. It uses more messages than are necessary, though it can quickly either meet the reply count or decide that it is unobtainable. It also has a persistence of one message, that is, the failure of just one message to a host causes the protocol to treat the host as unavailable.

3.2.2 The reschedule protocol

Reschedule addresses the second problem with **naive**. This protocol sends fewer messages than **naive**, though often at the expense of extra latency. It still has a persistence of one message, so it does not solve the transient communication failure problem. It orders replicas by expected communication latency to determine the order in which messages should be sent, causing it to communicate with the closest available replicas. It attempts to send the fewest possible messages by first sending messages to the q closest replicas, and to additional replicas as the earlier messages fail.

This approach has a problem: it will take much longer than **naive** to complete an operation when nearby replicas have failed. The protocol cannot determine that a message has failed until a timer has expired. Since timers should not expire before the acknowledgment can arrive, the timeout period is usually set to a large value – commonly chosen to cover more than 99% of all messages. Our studies of the Internet showed that this was usually about three times the average reply latency. If additional messages are sent earlier, even though it is possibly a reply is on it way, the operation can complete more rapidly without wasting large numbers of messages. The *delay* parameter d can be used to tune the protocol in this way.

When $d = 0$, **reschedule** is identical to **naive**: messages are sent to all replicas right away because the delay

timer for sending the next message expires immediately. When $d = 1$, **reschedule** only sends additional messages when communication failures are detected When $d = \frac{1}{2}$, messages are sent to additional replicas either if a failure is reported, or if at one-half of the longest message failure timeout.

This protocol meets the design goals better than the **naive** protocol. It sends to the closest replicas first, which tends to minimize message traffic if the nearest replicas are available. It also will communicate with only as many replicas as are needed to meet the reply count. The protocol will adapt somewhat to changing network conditions, in that it orders the replicas by distance, and uses timeouts to observe failures. However, since the protocol only has a persistence of one message, it cannot handle transient communication failures. We discuss the performance of this protocol, in terms of fault-tolerance, messages and latency, in §4.

3.2.3 The retry protocol

Neither **naive** nor **reschedule** accommodate transient failures. The next protocol, **retry**, is a modification of **reschedule** that retransmits lost messages in the hope that the failure was due to some transient problem and the next message will be delivered and acknowledged. It continues to retransmit until either the reply count has been met or until all replicas have been tried at least once. Such persistence improves both the success latency and the probability that the reply count will be met, though at the cost of sending more messages and possibly at the cost of having longer failure latencies.

Initially, the **retry** protocol sends messages to the q closest replicas, where q is the reply count. When the protocol receives a reply, it increments the count of successful replies. If sufficient replies have been obtained it declares the access a success. When it finds a message has failed the protocol schedules a retry for that replica. The first retry occurs immediately, but later retries are delayed. The performance simulators set each retry delay twice as long as the previous (a choice inspired by the collision-handling techniques used in Ethernet [11]). The delay helps to avoid sending vast numbers of messages to a nearby replica that has failed. As with **reschedule** the delay parameter d is used to determine when to send messages to distant replicas.

The **retry** protocol terminates with failure when it has received at least one reply or a timeout for every replica and the reply count has not yet been met. As a result this protocol has a variable persistence. Nearby replicas may be retried many times before a distant replica can reply. In the simulator, which doubles the delay after each message, the expected number of retries for a replica r is bounded above by $\log_2 (T_n/a_r)$, where a_r is the *expected communication latency* of the rth replica, and T_n is the failure timeout period for the most distant replica.

3.2.4 The count protocol

The **count** protocol is similar to **retry**, except that it has a fixed persistence. It maintains a counter for each replica and stops retrying that replica when l messages have been sent to it. The protocol terminates when all replicas have been tried l times or the reply count is met.

This protocol improves on **retry** in a number of ways. By trying each replica a fixed number of times, it will meet the reply count more often than **retry**, since distant replicas will be tried more times. This bound causes the two protocols to exhibit significantly different behaviors when message failures are likely. In addition, retrying a fixed number of times evens out the number of times messages are sent to each replica, preventing the protocol from trying a nearby failed replica many times. Our message failure measurements suggest that retrying more than a few times is usually of little value, since communication failures rarely lasted more than two or three messages. In our performance evaluation we used an arbitrary limit of $l = 5$. This protocol uses the same delaying techniques as the **retry** protocol.

4 Performance evaluation

We used discrete-event simulation and measurement of a sample application to analyze the performance of these protocols. The simulation experiments also measured the sensitivity of the results to the communication latency distribution, the length of message failure timeouts, and the overall message failure rate. Some of the simulations used traces of Internet communication behavior to determine the communication latency and failure of each message, while other used synthetic distributions derived from the traces. In this section we will summarize our findings; details on the simulation methods can be found in [8], and a more detailed report covering the performance evaluation and network measurements is available [9].

4.1 Simulation techniques

The simulations used traces of Internet communication behavior that we obtained using the **ping** program, which sends ICMP echo messages to remote hosts [14]. The remote host is expected to reply to echo messages as soon as possible. We collected traces of communication latency and message failure between a host at UC Santa Cruz and 125 randomly selected Sun-4 systems throughout the Internet. A set of several samples were taken for each host every 20 minutes, over a period of seven days. These traces do not capture any effects that are specific to quorum multicast protocols, such as congestion at the client or nearby gateways. Our measurements of an actual implementation confirm that this limitation does not invalidate the simulated results.

We used these traces to drive a simulator. The simulator performed several thousand runs, which consisted of selecting five hosts from the 125 sampled, then simulating one multicast operation with a reply count of three for each of the sets of samples recorded for those hosts. When a host was to send a message, the failure or latency was determined by looking up a sample in the trace. By performing several thousand runs, we obtained results with confidence intervals of less than 5% on all values.

The simulation also allowed us to examine the behavior of each of the four multicast protocols (**naive, reschedule, retry,** and **count**) under artificially high failure conditions that could not be created on the actual Internet. These simulations involved deriving synthetic distributions for communication latency and failure from the traces.

4.2 Direct measurement techniques

We constructed and measured a simple application running on the Internet to substantiate the simulation results. This application was structured as a client communicating with servers. The client ran on a few hosts, and sent UDP packets to the servers. The server was a simple daemon that listened for packets on a particular port, and echoed them back to their origin. The source code to the server was published on Usenet, and several people elected to run it on their system. We grouped the participating hosts into sets of five, some of which contained hosts spread evenly over Europe and North America, and some containing hosts in smaller regions. The client multicast to each group of five, using a reply count of three. The latency results have 95% confidence intervals generally between 10% and 15%, and message count results less than 5%.

The measurement experiment validated most of the results obtained by simulation, since the relative performance of each protocol is similar. The primary differences arose because the hosts showed fewer failed messages than those in the traces that drove the simulations. An error in the client invalidated the results for failed operations for the **count** protocol, but successful operations showed the expected behavior. Overall, the measurements confirm the conclusion that quorum multicast protocols can provide significant performance advantages for wide-area applications, and show that there is a trade-off among latency, traffic, and operation success.

4.3 Operation success

Operation success is measured by the fraction of all multicast operations that were successful in meeting the reply count. The **naive** and **reschedule** protocols each exhibited an approximately constant success fraction, at about 82% of all operations. Since these two protocols each attempt to send at most one message to a replica, the delay fraction has no effect on the probability of success. The **retry** protocol, however, retries nearby replicas more times when the delay parameter d is larger, since this allows more time for retries. **Retry** succeeds in more than 94% of all cases when $d \geq 0.1$, while **count** performs even better. This shows that persistence has a significant positive effect on protocols for internetworks.

The data obtained by measuring a test application show that all four protocols met the reply count more than 95% of the time. **Count** succeeded more often than the other protocols for almost all values of d, with **retry** generally succeeding more often than **naive** and **reschedule**. These results are similar to the simulation results.

4.4 Latency

For operations that are able to meet their reply count, **naive** is generally the fastest of the four protocols, since it always sends messages to every replica immediately. The communication latency for the other protocols increases approximately linearly as the delay parameter d increases, taking about the same amount of time as **naive** at $d = 0$. Of the three, **count** takes longer than **retry**, which in turn takes very slightly longer than **reschedule**. **Reschedule** takes less time than the other two because of the rare cases where the **retry** and **count** protocols must send more than one message to distant replicas to obtain the reply count.

The performance of the four protocols is quite different when the reply count cannot be met – all four protocols require several seconds to declare failure. While this is quite a long time, failures constitute only a few percent of all operations and the latency is not onerous. **Naive** is the baseline measure, requiring about 4.8 seconds to determine that a reply count cannot be obtained – almost an order of magnitude longer than was generally required for success. The latency of the other three protocols again increases roughly linearly in d. **Reschedule** requires more time than **naive** since it must detect just as many failed messages, but it may have delayed sending some of those messages. **Retry** requires more time than all the others for most values of d. **Count** performs much better than any of the other three protocols. It avoids the problem of having to communicate with the most distant replica, since it can stop when sufficient nearby replicas have failed.

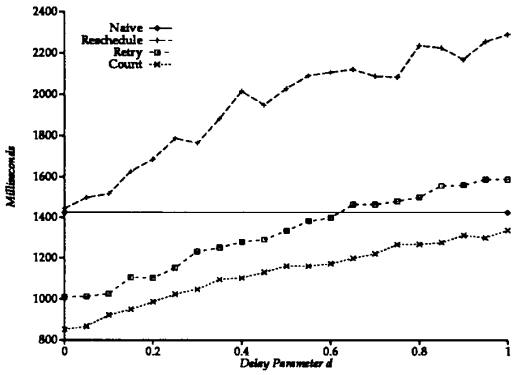

FIGURE 1: Communication latency for all operations.

The measured results differ slightly because fewer messages failed. While simulation indicated that **reschedule** takes more time than **naive** to declare failure, and that this time increases with d, the measured results show that the two have quite similar latencies. The sample size is small enough that this result is inconclusive.

Figure 1 shows the overall latency for each protocol. Since the probability of meeting the reply count is quite high, the values for successful operations predominate in these graphs. However, it is worth noting that even with a high probability of success, the low failure latency of **count** makes it the fastest of the three quorum multicast protocols, consistently faster even than **naive**. **Reschedule** has the highest latency of the three for all values of d. **Retry** is better than **naive** or **reschedule** for values of d less than about 0.6. This is the reverse of their positions for successful operations. The latency of the quorum multicast algorithms increases approximately linearly as d increases. The overall measurement results are consistent with simulation results since the overall success rate was in excess of 95%, despite the differences in failure behavior.

4.5 Messages

The **naive** protocol always sends one message to each host. For successful operations, **reschedule** sends fewer messages, except at $d = 0$ when the two algorithms are identical. This savings happens when **reschedule** avoids

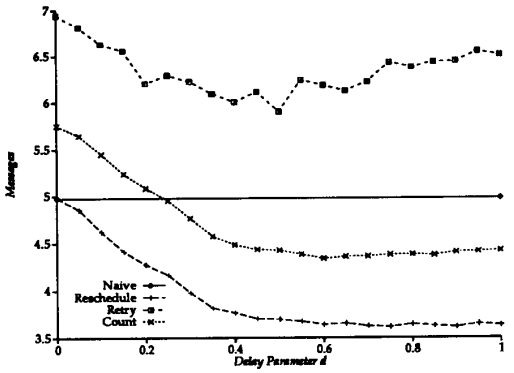

FIGURE 2: Messages for all operations.

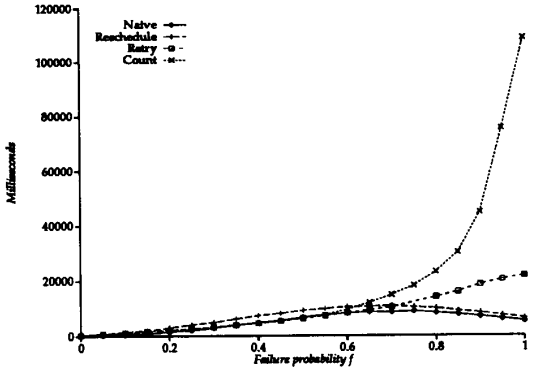

FIGURE 3: Total time, varying failure probability, $d = 0.5$.

sending messages to distant replicas. **Retry** often uses at least as many messages as **naive** since it must try each replica at least once before declaring failure, and messages to nearby hosts may be retried. **Count** uses more messages than **naive** for low values of d, behaving much like **retry**: sending messages to all replicas and occasionally resending when a message fails. When d is set to a higher value, the protocol behaves more like **reschedule**, except that it resends (on the average) about one message because of failure. The differences between the protocols were less accentuated in the measurement results. The hosts in the measurement experiment exhibited fewer message failures than did those measured for traces, and the three protocols all behave identically when no failures occur.

The four protocols perform quite differently when they are unable to obtain a reply count of responses. **Naive** requires exactly five messages. **Reschedule** also requires exactly five messages, since it will generally send to all replicas before it can determine that the operation has failed. The **retry** and **count** algorithms will generally send more than five messages before they can declare failure, but the difference between the two is dramatic. The **retry** protocol sends between three and six times as many messages as the other protocols, while **count** usually sends only one additional message. The difference is due to the extra control that **count** exercises over sending messages – no replica will be tried more than a fixed number of times. **Retry** may try nearby replicas a great many times: if a nearby replica has failed, it will have time to retry many times while waiting for a response (or timeout) from the most distant replica.

Figure 2 shows the overall number of messages sent by each protocol. Once again, since the probability of meeting the reply count is high, the values for successful operations predominate. However, the large number of messages sent by **retry** make that the least attractive quorum multicast protocol. The measurement results confirm the simulation, though the low number of message failures makes **retry** competitive with the other protocols.

4.6 Effect of failure probability

The simulation experiments also examined the performance of all four protocols under different failure conditions. The Internet measurements suggest that message failure is usually unlikely, but when a host becomes partitioned from the rest of the network, or there is a pathological condition in the Internet, it is nearly certain that a message will fail to reach its destination. The simulation allowed us to evaluate quorum multicast performance under these worst-case conditions.

For these experiments we fixed the delay parameter d at 0.5, because it was close to neither extreme. The simulations used synthetic hyperexponential distributions for communication latency and uniform message failure probability, since we could not manipulate the Internet to obtain traces with specific message failure rates. Message failures were treated as independent events occurring with a fixed probability f. Values of f in the range 0.2 to 0.3 are similar to the behavior of messages in the traces.

As expected, the **count** was able to successfully gather a reply count of responses more often than the other protocols, and **retry** succeeded less often than **count**. Both these protocols succeed more often than **reschedule** and **naive**, which only try each replica once. The data for **naive** match availability figures for data replicated using Majority Consensus Voting [7], estimated using Markov analysis and assuming reliable communication channels [13]. In that study, hosts were only checked once for availability, just as with the **naive** and **reschedule** protocols in these experiments.

Figure 3 shows the overall communication latency at different values of f. **Naive** requires the least time, as expected. At low failure probabilities, **reschedule** requires more time than the other protocols, but at high failure probabilities it does not retry failed messages and so can complete – with failure – in little more time than **naive**. At high failure probabilities, the **retry** protocol requires one-fifth the latency of the **count** protocol, since it sends only one message to the most distant host.

Figure 4 shows the overall number of messages sent by each protocol. As always, **naive** sends one message to each

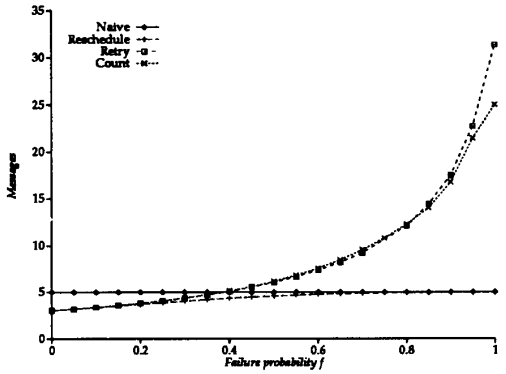

FIGURE 4: Total messages, varying failure probability, $d = 0.5$.

replica regardless of conditions. The number of messages sent by **reschedule** approaches the number of replicas as the probability of failure increases, since it becomes more likely that the protocol will have to send a message to all replicas. **Retry** sends more messages than **reschedule**, since it will retry messages that fail. This becomes increasingly important as the probability of failure increases. **Count** sends slightly fewer messages than **retry**, particularly when the probability of message failure f approaches unity. The **count** protocol is limited to sending at most 25 message (5 replicas, 5 messages per replica), while **retry** can send a nearly unbounded number of messages in the worst case.

5 Conclusions

In this paper we have presented a family of *quorum multicast* protocols, called **naive, reschedule, retry,** and **count**. We have shown that these protocols provide good availability while using fewer messages and requiring less latency than a simple multicast.

These protocols provide multicast to a subset of a group of sites. The protocols can communicate with the closest available sites and resort to more distant sites when the nearby ones fail. By varying the *reply count*, they can be used as a fault-tolerant one-to-one communication mechanism that contacts 'spare' replicas on failure, or as a one-to-many multicast for contacting several replicas at once.

Quorum multicast protocols are also useful for their clear definition of failure detection and its fault-tolerance. They can be used to approximate actual failure detection with high probability. The ability to retry communications makes quorum multicasts more robust in the face of transient network problems than a simple multicast protocol, making our protocols a convenient mechanism for building higher-level fault tolerant mechanisms.

We can choose between the protocols depending on whether the probability of success, operation latency, or message count are more important. When message failure is unlikely, the protocols all require about the same latency, though **naive** requires more messages than the other three. As the likelihood of message failure increases, **count** provides the best chance of successfully completing an operation and the lowest latency, while **reschedule** uses the fewest messages. However, under pathological conditions the protocols behave quite differently. **Count** can send many messages and take several seconds when no replicas are available. **Retry** is perhaps a more reasonable choice under pathological conditions, succeeding less often than **count** but taking between half and one-fifth as much time. If availability is not of great importance, **reschedule** and **naive** both perform much better than the other two protocols under high-failure conditions, since they do not retry messages for extended periods of time.

Acknowledgments

John Wilkes, of the Concurrent Systems Project at Hewlett-Packard laboratories, provided insight during the initial research, and helped improve the presentation of this paper. George Neville-Neil, at UC Berkeley, and the anonymous reviewers also provided helpful comments.

References

[1] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76 (February 1987).

[2] D. R. Boggs. Internet broadcasting. Technical report CSL–83–3 (October 1983). Xerox Palo Alto Research Center, CA.

[3] D. R. Cheriton and W. Zwaenepoel. One-to-many interprocess communication in the V-system. Technical report STAN–CS–84–1011 (August 1984). Computer Systems Laboratory, Stanford University.

[4] D. Comer. *Internetworking with TCP/IP: principles, protocols, and architecture* (1988). Prentice Hall, Englewood Cliffs, NJ.

[5] E. C. Cooper. Circus: a replicated procedure call facility. *Proceedings of 4th Symposium on Reliability in Distributed Software and Database Systems*, pages 11–24 (October 1984).

[6] H. Garcia-Molina and B. Kogan. An implementation of reliable broadcast using an unreliable multicast facility. *Proceedings of 7th Symposium on Reliable Distributed Systems*, pages 101–11 (10–12 October 1988).

[7] D. K. Gifford. Weighted voting for replicated data. *Proceedings of 7th ACM Symposium on Operating Systems Principles*, pages 150–62 (December 1979).

[8] R. Golding and D. D. E. Long. Accessing replicated data in a large-scale distributed system. *International Journal in Computer Simulation*, 1(4) (1991, to appear).

[9] R. A. Golding. Accessing replicated data in a large-scale distributed systems. Master's thesis; published as Technical report UCSC–CRL–91–18 (June 1991). Computer and Information Science Board, University of California at Santa Cruz.

[10] B. Martin, C. Bergan, and B. Russ. PARPC: a system for parallel procedure calls. *Proceedings of 1987 International Conference on Parallel Processing* (1987).

[11] R. M. Metcalfe and D. R. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, **19**(7):395–404 (July 1976).

[12] P. Mockapetris. Domain names – concepts and facilities. Request for comments 1034 (November 1987). ARPA Network Working Group.

[13] J.-F. Pâris. Voting with witnesses: a consistency scheme for replicated files. *6th International Conference on Distributed Computer Systems*, pages 606–12 (1986).

[14] J. Postel. *Internet control message protocol*, Request for comments 792 (September 1981). USC Information Sciences Institute.

[15] J. S. Quarterman and J. C. Hoskins. Notable computer networks. *Communications of the ACM*, **29**(10):932–71 (October 1986).

[16] K. Trivedi. *Probability and statistics with reliability, queuing, and computer science applications* (1982). Prentice-Hall, Englewood Cliffs, NJ.

View publication stats