A Cost-Benefit Approach to Fault Tolerant Communication and Information Access

Yair Amir
Department of Computer Science,
Johns Hopkins University

Final Report

Prepared for DARPA
Under contract F30602-00-2-0550

February 2004

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE 15-02-2004

2. REPORT TYPE Final Report

3. DATES COVERED

4. TITLE AND SUBTITLE

A Cost-Benefit Approach to Fault Tolerant Communication and Information Access

5a. CONTRACT NUMBER
F30602-00-2-0550

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Yair Amir

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Johns Hopkins University
3400 N. Charles Street
Computer Science Department
224 NEB
Baltimore, MD 21218

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Col. Tim Gibson
DARPA/ATO
3701 North Fairfax Drive
Arlington, VA 22203

Mr. Alan Akins
AFRL/IFGA
525 Brooks Rd.
Rome, NY 13441

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We develop the theory and algorithms required to overcome strong network faults and attacks, while providing theoretically provable performance bounds. We build a system that incorporates these algorithms, and that exhibits good performance in practice. Since we are interested in information access and communication, our work focuses on two areas: network routing, and replication. Robust network routing allows passing and accessing information even when the network is under heavy attack. Replication is required to allow information access even when there is no network connectivity

15. SUBJECT TERMS

Fault tolerant communication, routing, wide-area replication, middleware

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Yair Amir
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** 410-516-4803

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. 239.18

Abstract

Fault tolerant information access and communication are becoming crucial for almost every computer application. Even traditional computer applications that did not require network connectivity just a few years ago, rely on the network for their smooth execution. This trend will only increase with the proliferation of non-traditional networked computing devices. At the same time, attacks on the network have become more sophisticated and harder to contain. The distributed nature of network control further complicates the problem.

Traditional network protocols were developed to handle simple adversaries, such as network problems resulting from normal operational events. For example, link congestion can lead to buffer overflow, which then causes message omission. Another example is infrequent link downtime that may lead to short-term network partitions. While existing protocols can handle such problems, they will fail miserably under slightly more sophisticated attacks.

We develop the theory and algorithms required to overcome strong network faults and attacks, while providing theoretically provable performance bounds. We build a system that incorporates these algorithms, and that exhibits good performance in practice. Since we are interested in information access and communication, our work focuses on two areas: network routing, and replication. Robust network routing allows passing and accessing information even when the network is under heavy attack. Replication is required to allow information access even when there is no network connectivity

Table of Contents

1. Summary

The goal of this project was to develop a Cost-Benefit framework for fault tolerant communication and information access that addresses extremely powerful adversaries that were never handled in the past. The project develops the theory and algorithms required to overcome strong network attacks, while providing theoretically provable performance bounds. We build a system that incorporates these algorithms, and that exhibits good performance in practice.

We present a suite of novel routing protocols tailored to the above adversary models and prove that these protocols perform in a near-optimal manner. Specifically, we present novel solutions that, in case an operational path exists, will be able to find it.

We then develop an overlay network architecture that will make these protocols practical since they are too complex to have any hope to be implemented in general Internet routers anytime soon. In order to better analyze and understand the overlay networks paradigm in an environment defined by weaker semantics, we developed a stand alone prototype called Spines, using the client-daemon architecture that is able to build and automatically configure a dynamic overlay network. Our Overlay Network is very scalable, as it does not have any limitation in number of nodes or links, other than what the routing protocol used can support.

When there is no theoretic possibility of communication, say in the case of a cut in the network, one can still continue the operation by making sure that the data is replicated in most areas, or at least in the areas where disconnection is likely. We develop a suite of replication protocols that can handle a range of adversaries and can gracefully degrade performance and semantics as the network hostility increases. Our goal is to replicate an ACID database as this is the most demanding replication problem. We show how we seamlessly replicate the Postgres database without the applications (or the database) knowing that they operate in a replicated system. We demonstrate the practicality of our method by replicating Postgres over a nation-wide network.

2. Introduction

Fault tolerant information access and communication are becoming crucial for almost every computer application. Even traditional computer applications that did not require network connectivity just a few years ago, rely on the network for their smooth execution. This trend will only increase with the proliferation of non-traditional networked computing devices. At the same time, attacks on the network have become more sophisticated and harder to contain. The distributed nature of network control further complicates the problem.

Traditional network protocols were developed to handle simple adversaries, such as network problems resulting from normal operational events. For example, link congestion can lead to buffer overflow, which then causes message omission. Another example is infrequent link downtime that may lead to short-term network partitions. While existing protocols can handle such problems, they will fail miserably under slightly more sophisticated attacks.

This project was focused on designing algorithms and building protocols that will provide resilient communication and information access in the presence of strong adversaries.

Since we are interested in information access and communication, our work focuses on two areas: Fault tolerant routing, and replication. Robust network routing allows passing and accessing information even when the network is under heavy attack. A certain degree of replication is required to allow information access even when there is no network connectivity.

The remainder of this report is organized as follows. Section 2 is focused on fault tolerant routing and has two parts. The first part describes an on demand routing algorithm that is resilient to Byzantine behavior of routers. The second part describes an overlay network approach for reliable communication. Section 3 is focused on highly available information access techniques and has three parts. The first part describes the Wackamole mechanism for making servers and routers fault tolerant. The second part describes an algorithm that allows consistent replication with strong properties and. The third part applies this algorithm to replicate a database over wide area networks. Section 4 concludes the report.

3. Fault Tolerant Routing

3.1. An On-Demand Secure Routing Protocol Resilient to Byzantine Failures

Ad hoc wireless networks are self-organizing multi-hop wireless networks where all the hosts (nodes) take part in the process of forwarding packets. Ad hoc networks can easily be deployed since they do not require any fixed infrastructure, such as base stations or routers. Therefore, they are highly applicable to emergency deployments, natural disasters, military battlefields, and search and rescue missions.

A key component of ad hoc wireless networks is an efficient routing protocol, since all of the nodes in the network act as routers. Some of the challenges faced in ad hoc wireless networks include high mobility and constrained power resources. Consequently, ad hoc wireless routing protocols must converge quickly and use battery power efficiently. Traditional proactive routing protocols (link-state and distance vectors), which use periodic updates or beacons which trigger event based updates, are less suitable for ad hoc wireless networks because they constantly consume power throughout the network, regardless of the presence of network activity, and are not designed to track topology changes occurring at a high rate. On-demand routing protocols are more appropriate for wireless environments because they initiate a route discovery process only when data packets need to be routed. Discovered routes are then cached until they go unused for a period of time, or break because the network topology changes.

Many of the security threats to ad hoc wireless routing protocols are similar to those of wired networks. For example, a malicious node may advertise false routing information, try to redirect routes, or perform a denial of service attack by engaging a node in resource consuming activities such as routing packets in a loop. Furthermore, due to their cooperative nature and the broadcast medium, ad hoc wireless networks are more vulnerable to attacks in practice. Although one might assume that once authenticated, a node should be trusted, there are many scenarios where this is not appropriate. For example, when ad hoc networks are used in a public Internet access system (airports or conferences), users are authenticated by the Internet service provider. However, this authentication does not imply trust between the individual users of the service. Also, mobile devices are easier to compromise because of reduced physical security, so complete trust should not be assumed.

We focus on providing routing survivability under an adversarial model where any intermediate node or group of nodes can perform Byzantine attacks such as creating routing loops, misrouting packets on non-optimal paths, or selectively dropping packets (black hole). Only the source and destination nodes are assumed to be trusted. We propose an on-demand routing protocol for wireless ad hoc networks that operate under this strong adversarial model.

It is provably impossible under certain circumstances, for example when a majority of the nodes are malicious, to attribute a Byzantine fault occurring along a path to a specific node, even using expensive and complex Byzantine agreement. Our protocol circumvents this obstacle by avoiding the assignment of “guilt” to individual nodes. Instead it reduces the possible fault location to two adjacent nodes along a path, and attributes the fault to

the link between them. As long as a fault-free path exists between two nodes, they can communicate reliably even if an overwhelming majority of the network acts in a Byzantine manner.

Our protocol consists of the following phases:

- Route discovery with fault avoidance. Using flooding and a faulty link weight list, this phase finds a least weight path from the source to the destination.
- Byzantine fault detection. This phase discovers faulty links on the path from the source to the destination. Our adaptive probing technique identifies a faulty link after log n faults have occurred, where n is the length of the path.
- Link weight management. This phase maintains a weight list of links discovered by the fault detection algorithm. A multiplicative increase scheme is used to penalize links which are then rehabilitated over time. This list is used by the route discovery phase to avoid faulty paths.

Our protocol establishes a reliability metric based on past history and uses it to select the best path. The metric is represented by a list of link weights where high weights correspond to low reliability. Each node in the network maintains its own list, referred to as a weight list, and dynamically updates that list when it detects faults. Faulty links are identified using a secure adaptive probing technique that is embedded in the normal packet stream. These links are avoided using a secure route discovery protocol that incorporates the reliability metric. We describe this work in our "An On Demand Secure Routing Protocol Resilient to Byzantine Failures" paper by B. Awerbuch, D. Holmer, C. Nita-Rotaru and H. Rubens that appeared in the *ACM International Workshop on Wireless Security (WiSe02)*, found in Appendix A.

3.2. Reliable Communication in Overlay Networks

Reliable point-to-point communication is one of the main utilizations of the Internet, where over the last few decades TCP has served as the dominant protocol. Over the Internet, reliable communication is performed end-to-end in order to address the severe scalability and interoperability requirements of a network in which potentially every computer on the planet could participate. Thus, all the work required in a reliable connection is distributed only to the two end nodes of that connection, while intermediate nodes route packets without keeping any information about the individual packets they transfer.

Overlay networks are opening new ways to Internet usability, mainly by adding new services (e.g. built-in security) that are not available or cannot be implemented in the current Internet, and also by providing improved services such as higher availability. However, the usage of overlay networks may come with a price, usually in added latency that is incurred due to longer paths created by overlay routing, and by the need to process the messages in the application level by every overlay node on the path.

Reliable communication in overlay networks is usually achieved by applying TCP on the edges of a connection. This surely works. However, we prove that employing hop-by-hop reliability techniques considerably reduces the average latency and jitter of reliable

communication. When using such an approach one has to consider networking aspects such as congestion control, fairness, and flow control and end-to-end reliability. We demonstrate through simulation that our approach provides tremendous benefit for the application as well as for the network itself, even when very few packets are lost. Simulations usually do not take into account many practical issues such as processing overhead, CPU scheduling, and most important, the fact that overlay network processing is performed at the application level of general purpose computers. These may have considerable impact on real-life behavior and performance. Therefore, we test our approach in practice on an overlay network platform called Spines that we have built. We show that the benefit of hop-by-hop reliability greatly overcomes the overhead of overlay routing and achieves much better performance compared to standard end-to-end TCP connections deployed on the same overlay network.

An overlay network constructs a user level graph on top of an existing networking infrastructure such as the Internet, using only a subset of the available network links and nodes. An overlay link is a virtual edge in this graph and may consist of many actual links in the underlying network. Overlay nodes act as routers, forwarding packets to the next overlay link toward the destination. At the physical level, packets traveling along a virtual edge between two overlay nodes follow the actual physical links that form that edge.

Overlay networks have two main drawbacks. First, the overlay routers incur some overhead every time a message is processed, which requires delivering the message to the application level, processing it, and resending the message to the next overlay router. Second, the placement of overlay routers in the topology of the physical network is often far from optimal, because the creator of the overlay network rarely has control over the physical network (usually the Internet) or even the knowledge about its actual topology. Therefore, overlay networks provide longer paths that have higher latency than point to point Internet connections.

The easiest way to achieve reliability in Overlay Networks is to use a reliable protocol, usually TCP, between the end points of a connection. This mechanism has the benefit of simplicity in implementation and deployment, but pays a high price upon recovery from a loss. As overlay paths have higher delays, it takes a relatively long time to detect a loss, and data packets and acknowledgments are sent on multiple overlay hops in order to recover the missed packet

We propose a mechanism that recovers the losses only on the overlay hop on which they occurred, localizing the congestion and enabling faster recovery. Since an overlay link has a lower delay compared to an end-to-end connection that traverses multiple hops, we can detect the loss faster and resend the missed packet locally. Moreover, the congestion control on the overlay link can increase the congestion window back faster than an end-to-end connection, as it has a smaller round-trip time.

Hop-by-hop reliability involves buffers and processing in the intermediate overlay nodes. These nodes need to deploy a reliable protocol, and keep track of packets, acknowledgments and congestion control, in addition to their regular routing functionality. Although such an approach may not be feasible to implement at the level of the Internet routers due to scalability limitations, we can easily deploy it at the level of an

overlay network, thus allowing us to pinpoint the congestion, limiting the problem to the congested part of the network. We describe this work in our "Reliable Communication in Overlay Networks" paper by Y. Amir and C. Danilov that appeared in the *Proceedings of the International Conference on Dependable Systems and Networks (DSN03)*, found in Appendix A.

4. Highly Available Information Access

4.1. N-Way Fail-Over Infrastructure for Reliable Servers and Routers

Maintaining the availability of critical network servers is an important concern for many organizations. Server redundancy is the traditional approach to provide availability in the presence of failures. From the client perspective, a network-accessible service is resolved via a set of public IP addresses specified for this service. Therefore, the continued availability of a service via these IP addresses is a prerequisite for providing uninterrupted service to the client. In order to function correctly, each of the service's public IP addresses has to be covered by exactly one physical server at any given time. If no physical server covers a public IP address, the clients will not receive any service. On the other hand, if more than one physical server is covering the same IP address at any time, the network might not function properly and clients may not be served correctly.

A sizable market exists for hardware solutions that maintain the availability of IP addresses, usually via a gateway that hides the actual servers behind a smart switch or router in a centralized manner. We present Wackamole, a high availability tool for clusters of servers. Wackamole ensures that all the public IP addresses of a service are available to its clients. Wackamole is a completely distributed software solution based on a provably correct algorithm that negotiates the assignment of IP addresses among the available servers upon detection of faults and recoveries, and provides N-way fail-over, so that any one of a number of servers can cover for any other.

Using a simple algorithm that utilizes strong group communication semantics, Wackamole demonstrates the application of group communication to address a critical availability problem at the core of the system, even in the presence of cascading network or server faults and recoveries. We also demonstrate how the same architecture is extended to provide a similar service for highly-available routers.

Our solution has three main components:

- An IP address control (acquire and release) mechanism.
- A state synchronization algorithm (the Wackamole Algorithm).
- A membership service provided by a group communication toolkit.

The group communication toolkit maintains a membership service among the currently connected servers and notifies the synchronization algorithm of any view changes that occur due to server crashes and recoveries, or network partitions and remerges.

The synchronization algorithm manages the logical assignment of virtual IP addresses among the currently connected members, avoiding conflicts that can occur upon merges and recoveries and covering the "holes" that can arise as a result of a crash or partition. The IP address control mechanism enforces the decisions of the synchronization algorithm by acquiring and releasing the IP addresses accordingly. These mechanisms are highly specific to the operating system on which the Wackamole system runs. The correctness as well as the efficiency of the system depends on the use of a group communication system that provides reliable, totally ordered multicast and group

membership notifications for a cluster of servers. Wackamole was implemented using the Spread group communication toolkit. Spread is a general-purpose group communication system for wide-and local-area networks. It provides reliable and ordered delivery of messages (FIFO, causal, agreed ordering) as well as Virtual Synchrony and Extended Virtual Synchrony membership services.

Spread uses a client-daemon architecture. Node crashes/recoveries and network partitions/remerges are detected by Spread at the daemon level; upon detecting such an event, the Spread daemons install the new daemon membership and inform their clients of the corresponding changes in the group membership that are introduced by the failure. Clients are also notified when changes in the group membership are triggered by a graceful leave or join of any client. The Spread toolkit is optimized to support the latter situation without triggering a full daemon membership reconfiguration, but rather informing only the participating group about the new group membership.

Wackamole's state synchronization algorithm is implemented using group membership and messaging services offered by the Spread Toolkit. Immediately upon startup, the Wackamole daemon connects to a Spread daemon running on the same host and joins the wackamole group. It then relies on the regular membership messages sent by Spread to determine the current set of available hosts, and to initiate state transfer upon view-change detection. Spread is also used to ensure that messages are sent in a total order among Wackamole daemons, that old messages which must be discarded upon receipt can be identified properly, and that endian conflicts across platforms are handled correctly.

A Wackamole daemon that becomes disconnected from Spread will drop all of its virtual interfaces and enter a cycle in which it periodically attempts to reconnect to Spread, because it cannot ensure correctness without the services Spread provides. This behavior allows clusters to survive changes to the Spread daemons with which they communicate, such as version. All routing from the internal network will not be affected unless the designated router fails. In this case, Wackamole reassigns another router to control the virtual router address and the hand-off is complete as soon as Wackamole reconfigures without additional need to transfer routing information. We describe this work in our "N-Way Fail-Over Infrastructure for Reliable Servers and Routers" paper by Y. Amir, R. Caudy, A. Munjal, T. Schlossnagle and C. Tutu that appeared in the *Proceedings of the International Conference on Dependable Systems and Networks (DSN03)*, found in Appendix A.

4.2. From Total Order to Database Replication

Database replication is quickly becoming a critical tool for providing high availability, survivability and high performance for database applications. However, to provide useful replication one has to solve the non-trivial problem of maintaining data consistency between all the replicas.

The state machine approach to database replication ensures that replicated databases that start consistent will remain consistent as long as they apply the same deterministic actions (transactions) in the same order. Thus, the database replication problem is reduced to the problem of constructing a global persistent consistent order of actions.

This is often mistakenly considered easy to achieve using the Total Order service (e.g. ABCAST, Agreed order, etc) provided by group communication systems.

Early models of group communication, such as Virtual Synchrony, did not support network partitions and merges. The only failures tolerated by these models were process crashes, without recovery. Under this model, total order is sufficient to create global persistent consistent order. When network partitions are possible, total order service does not directly translate to a global persistent consistent order. Existing solutions that provide active replication either avoid dealing with network partitions, or require additional end-to-end acknowledgements for every action after it is delivered by the group communication and before it is admitted to the global consistent persistent order (and can be applied to the database).

We describe a complete and provably correct algorithm that provides global persistent consistent order in a partitionable environment without the need for end-to-end acknowledgments on a per action basis. In our approach, end-to-end acknowledgements are only used once for every network connectivity change event (such as network partition or merge) and not per action. Our algorithm builds a generic replication engine which runs outside the database and can be seamlessly integrated with existing databases and applications. The replication engine supports various semantic models, relaxing or enforcing the consistency constraints as needed by the application. We implemented the replication engine on top of the Spread toolkit and provide experimental performance results, comparing the throughput and latency of the global consistent persistent order using our algorithm, the COReL algorithm, and a two-phase commit algorithm. These results demonstrate the impact of eliminating the end-to-end acknowledgments on a per-action basis.

An action defines a transition from the current state of the database to the next state; the next state is completely determined by the current state and the action. We view actions as having a query part and an update part, either of which can be missing. Client transactions translate into actions that are applied to the database. The basic model best fits one-operation transactions, but in Section 6 we show how other transaction types can also be supported.

In the presence of network partitions, the replication layer identifies at most a single component of the server group as a primary component; the other components of a partitioned group are non-primary components. A change in the membership of a component is reflected in the delivery of a view-change notification by the group communication layer to each server in that component. The replication layer implements a symmetric distributed algorithm to determine the order of actions to be applied to the database. Each server builds its own knowledge about the order of actions in the system.

In many systems, processes exchange information only as long as they have a direct and continuous connection. In contrast, our algorithm propagates information by means of eventual path: when a new component is formed, the servers exchange knowledge regarding the actions they have, their order and color. This exchange process is only invoked immediately after a view change. Furthermore, all the components exhibit this behavior, whether they will form a primary or non-primary component. This allows the information to be disseminated even in non-primary components, reducing the amount of

data exchange that needs to be performed once a server joins the primary component. We describe this work in our "From Total Order to Database Replication" paper by Y. Amir and C. Tutu that appeared in the *Proceedings of the IEEE International Conference on Distributed Computing System (ICDCS02)*, found in Appendix A.

4.3. Practical Wide Area Database Replication

In many Internet applications, a large number of users that are geographically dispersed may routinely query and update the same database. In this environment, the location of the data can have a significant impact on application response time and availability. A centralized approach manages only one copy of the database. This approach is simple since contradicting views between replicas are not possible. The centralized approach suffers from two major drawbacks:

- Performance problems due to high server load or high communication latency for remote clients.
- Availability problems caused by server downtime or lack of connectivity. Clients in portions of the network that are temporarily disconnected from the server cannot be serviced.

The server load and server downtime problems can be addressed by replicating the database servers to form a cluster of peer servers that coordinate updates. However, communication latency and server connectivity remain a problem when clients are scattered on a wide area network and the cluster is limited to a single location. Wide area database replication coupled with a mechanism to direct the clients to the best available server (network-wise and load-wise) can greatly enhance both the response time and availability.

A fundamental challenge in database replication is maintaining a low cost of updates while assuring global system consistency. The problem is magnified for wide area replication due to the high latency and the increased likelihood of network partitions in wide area settings. We explore a novel replication architecture and system for local and wide area networks. Our architecture provides peer replication, where all the replicas serve as master databases that can accept both updates and queries. Our failure model includes network partitions and merges, computer crashes and recoveries, and message omissions, all of which are handled by our system. We rely on the lower level network mechanisms to handle message corruptions, and do not consider Byzantine faults.

Our replication architecture includes two components: a wide area group communication toolkit, and a replication server. The group communication toolkit supports the Extended Virtual Synchrony model. The replication servers use the group communication toolkit to efficiently disseminate and order actions, and to learn about changes in the membership of the connected servers in a consistent manner. Based on a sophisticated algorithm that utilizes the group communication semantics, the replication servers avoid the need for end-to-end acknowledgements on a per-action basis without compromising consistency. End-to-end acknowledgments are only required when the membership of the connected servers changes due to network partitions, merges, server crashes and recoveries. This leads to high system performance. When the membership of connected servers is stable,

the throughput of the system and the latency of actions are determined mainly by the performance of the group communication and the single node database performance, rather than by other factors such as the number of replicas. When the group communication toolkit scales to wide area networks, our architecture automatically scales to wide area replication.

We implemented the replication system using the Spread Group Communication Toolkit and the PostgreSQL database system. We then define three different environments to be used as test-beds: a local area cluster with fourteen replicas, the CAIRN wide area network that spans the U.S.A with seven sites, and the Emulab emulated wide area test bed. We conducted an extensive set of experiments on the three environments, varying the number of replicas and clients, and varying the mix of updates and queries. Our results show that sophisticated algorithms and careful distributed systems design can make symmetric, synchronous, peer database replication a reality over both local and wide area networks. We describe this work in our "Practical Wide Area Database Replication" CNDS-2002-1 technical report by Y. Amir, C. Danilov, Michal Miskin-Amir, J. Stanton and C. Tutu, found in Appendix A.

5. Conclusions

We developed the theory and algorithms required to overcome strong network faults and attacks, while providing theoretically provable performance bounds. We built a system that incorporates these algorithms, and that exhibits good performance in practice. We focused on two areas: network routing and replication. We designed and built a routing protocol that is resilient to Byzantine faults. We designed and built a general overlay network system that achieves high performance in practice. We designed a generic replication algorithm that allows information access even at times when there is no network connectivity. We implemented the algorithm in the environment of the Postgres database and deployed it over wide area networks, proving the practicality of this approach.

Appendix A.

- "An On Demand Secure Routing Protocol Resilient to Byzantine Failures", B. Awerbuch, D. Holmer, C. Nita-Rotaru and H. Rubens, In the *ACM International Workshop on Wireless Security (WiSe02)*Atlanta, Georgia, September 2002.
- "Reliable Communication in Overlay Networks", Y. Amir and C. Danilov. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN03)*, pages 511-520, San Francisco CA, June 2003.
- "N-Way Fail-Over Infrastructure for Reliable Servers and Routers", Y. Amir, R. Caudy, A. Munjal, T. Schlossnagle and C. Tutu. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN03)*, pages 403-412, San Francisco CA, June 2003
- "From Total Order to Database Replication", Y. Amir and C. Tutu In the *Proceedings of the IEEE International Conference on Distributed Computing System (ICDCS02)*, pages 494-503, Vienna, Austria, July 2002.
- "Practical Wide Area Database Replication" Y. Amir, C. Danilov, Michal Miskin-Amir, J. Stanton and C. Tutu. Technical Report CNDS-2002-1, Center for Networking and Distributed Systems, Johns Hopkins University, www.cnds.jhu.edu.

An On-Demand Secure Routing Protocol Resilient to Byzantine Failures

Baruch Awerbuch, David Holmer, Cristina Nita-Rotaru and Herbert Rubens
Department of Computer Science
Johns Hopkins University
3400 North Charles St.
Baltimore, MD 21218 USA
{baruch, dholmer, crisn, herb}@cs.jhu.edu

ABSTRACT

An ad hoc wireless network is an autonomous self-organizing system of mobile nodes connected by wireless links where nodes not in direct range can communicate via intermediate nodes. A common technique used in routing protocols for ad hoc wireless networks is to establish the routing paths on-demand, as opposed to continually maintaining a complete routing table. A significant concern in routing is the ability to function in the presence of byzantine failures which include nodes that drop, modify, or mis-route packets in an attempt to disrupt the routing service.

We propose an on-demand routing protocol for ad hoc wireless networks that provides resilience to byzantine failures caused by individual or colluding nodes. Our adaptive probing technique detects a malicious link after $\log n$ faults have occurred, where n is the length of the path. These links are then avoided by multiplicatively increasing their weights and by using an on-demand route discovery protocol that finds a least weight path to the destination.

Categories and Subject Descriptors

C.2.0 [**General**]: Security and protection; C.2.1 [**Network Architecture and Design**]: Wireless communication; C.2.2 [**Network Protocols**]: Routing protocols

General Terms

Algorithms, Design, Reliability, Security, Theory

Keywords

ad hoc wireless networks, on-demand routing, security, byzantine failures

WiSe'02, September 28, 2002, Atlanta, Georgia, USA.

1. INTRODUCTION

Ad hoc wireless networks are self-organizing multi-hop wireless networks where all the hosts (nodes) take part in the process of forwarding packets. Ad hoc networks can easily be deployed since they do not require any fixed infrastructure, such as base stations or routers. Therefore, they are highly applicable to emergency deployments, natural disasters, military battle fields, and search and rescue missions.

A key component of ad hoc wireless networks is an efficient routing protocol, since all of the nodes in the network act as routers. Some of the challenges faced in ad hoc wireless networks include high mobility and constrained power resources. Consequently, ad hoc wireless routing protocols must converge quickly and use battery power efficiently. Traditional proactive routing protocols (link-state [1] and distance vectors [1]), which use periodic updates or beacons which trigger event based updates, are less suitable for ad hoc wireless networks because they constantly consume power throughout the network, regardless of the presence of network activity, and are not designed to track topology changes occurring at a high rate.

On-demand routing protocols [2, 3] are more appropriate for wireless environments because they initiate a route discovery process only when data packets need to be routed. Discovered routes are then cached until they go unused for a period of time, or break because the network topology changes.

Many of the security threats to ad hoc wireless routing protocols are similar to those of wired networks. For example, a malicious node may advertise false routing information, try to redirect routes, or perform a denial of service attack by engaging a node in resource consuming activities such as routing packets in a loop. Furthermore, due to their cooperative nature and the broadcast medium, ad hoc wireless networks are more vulnerable to attacks in practice [4].

Although one might assume that once authenticated, a node should be trusted, there are many scenarios where this is not appropriate. For example, when ad hoc networks are used in a public Internet access system (airports or conferences), users are authenticated by the Internet service provider, but this authentication does not imply trust between the individual users of the service. Also, mobile devices are easier to compromise because of reduced physical security, so complete trust should not be assumed.

Our contribution. We focus on providing routing survivability under an adversarial model where any intermediate node or group of nodes can perform byzantine attacks such as creating routing loops, misrouting packets on non-optimal paths, or selectively dropping packets (*black hole*). Only the source and destination nodes are assumed to be trusted. We propose an on-demand routing protocol for wireless ad hoc networks that operates under this strong adversarial model.

It is provably impossible under certain circumstances, for example when a majority of the nodes are malicious, to attribute a byzantine fault occurring along a path to a specific node, even using expensive and complex byzantine agreement. Our protocol circumvents this obstacle by avoiding the assignment of "guilt" to individual nodes. Instead it reduces the possible fault location to two adjacent nodes along a path, and attributes the fault to the link between them. As long as a fault-free path exists between two nodes, they can communicate reliably even if an overwhelming majority of the network acts in a byzantine manner.

Our protocol consists of the following phases:

- *Route discovery with fault avoidance.* Using flooding and a faulty link weight list, this phase finds a least weight path from the source to the destination.
- *Byzantine fault detection.* This phase discovers faulty links on the path from the source to the destination. Our adaptive probing technique identifies a faulty link after $\log n$ faults have occurred, where n is the length of the path.
- *Link weight management.* This phase maintains a weight list of links discovered by the fault detection algorithm. A multiplicative increase scheme is used to penalize links which are then rehabilitated over time. This list is used by the route discovery phase to avoid faulty paths.

The rest of the paper is organized as follows. Section 2 summarizes related work. We further define the problem we are addressing and the model we consider in Section 3. We then present our protocol in Section 4 and provide an analysis in Section 5. We conclude and suggest directions for future work in Section 6.

2. RELATED WORK

Secure routing protocols for ad hoc wireless networks is a fairly new topic. Although routing in ad hoc wireless networks has unique aspects, many of the security problems faced in ad hoc routing protocols are similar to those faced by wired networks. In this section, we review the work done in securing routing protocols for both ad hoc wireless and wired networks.

One of the problems addressed by researchers is providing an effective public key infrastructure in an ad hoc wireless environment which by nature is decentralized. Examples of these works are as follows. Hubaux et al.[5] proposed a completely decentralized public-key distribution system similar to PGP [6]. Zhou and Haas [7] explored threshold cryptography methods in a wireless environment. Brown et al.[8] showed how PGP, enhanced by employing elliptic curve cryptography, is a viable option for wireless constrained devices.

A more general trust model where levels of security are defined for paths carrying specific classes of traffic is suggested in [9]. The paper discusses very briefly some of the cryptographic techniques that can be used to secure on-demand routing protocols: shared key encryption associated with a security level and digital signatures for data source authentication.

As mentioned in [10], source authentication is more of a concern in routing than confidentiality. Papadimitratos and Haas showed in [11] how impersonation and replay attacks can be prevented for on-demand routing by disabling route caching and providing end-to-end authentication using an HMAC [12] primitive which relies on the existence of security associations between sources and destinations. Dahill et al.[16] focus on providing hop-by-hop authentication for the route discovery stage of two well-known on-demand protocols: AODV [2] and DSR [3], relying on digital signatures. Other significant works include SEAD [13] and Ariadne [4] that provide efficient secure solutions for the DSDV [14] and DSR [3] routing protocols, respectively. While SEAD uses one-way hash chains to provide authentication, Ariadne uses a variant of the Tesla [15] source authentication technique to achieve similar security goals.

Marti et al.[18] address a problem similar to the one we consider, survivability of the routing service when nodes selectively drop packets. They take advantage of the wireless cards promiscuous mode and have trusted nodes monitoring their neighbors. Links with an unreliable history are avoided in order to achieve robustness. Although the idea of using the promiscuous mode is interesting, this solution does not work well in multi-rate wireless networks because nodes might not hear their neighbors forwarding communication due to different modulations. In addition, this method is not robust against collaborating adversaries.

Also, relevant work has been done in the wired network community. Many researchers focused on securing classes of routing protocols such as link-state [10, 19, 20, 21] and distance-vector [22]. Others addressed in detail the security issues of well-known protocols such as OSPF [23] and BGP [24]. The problem of source authentication for routing protocols was explored using digital signatures [23] or symmetric cryptography based methods: hash chains [10], chains of one-time signatures [20] or HMAC [21]. Intrusion detection is another topic that researchers focused on, for generic link-state [25, 26] or OSPF [27].

Perlman [28] designed the Network-layer Protocol with Byzantine Robustness (NPBR) which addresses denial of service at the expense of flooding and digital signatures. The problem of byzantine nodes that simply drop packets (*black holes*) in wired networks is explored in [29, 30]. The approach in [29] is to use a number of trusted nodes to probe their neighbors, assuming a limited model and without discussing how probing packets are disguised from the adversary. A different technique, flow conservation, is used in [30]. Based on the observation that for a correct node the number of bytes entering a node should be equal to the number of bytes exiting the node (within a threshold), the authors suggest a scheme where nodes monitor the flow in the network. This is done by requiring each node to have a copy of the routing table of their neighbors and reporting the incoming and outgoing data. Although interesting, the scheme does not work when two or more adversarial nodes collude.

3. PROBLEM DEFINITION AND MODEL

In this section we discuss the network and security assumptions we make in this paper and present a more precise description of the problem we are addressing.

3.1 Network Model

This work relies on a few specific network assumptions. Our protocol requires bi-directional communication on all links in the network. This is also a requirement of most wireless MAC protocols, including 802.11 [31] and MACAW [32]. We focused on providing a secure routing protocol, which addresses threats to the ISO/OSI network layer. We do not specifically address attacks against lower layers. For example, the physical layer can be disrupted by jamming, and MAC protocols such as 802.11 can be disrupted by attacks using the special RTS/CTS packets. Though MAC protocols can detect packet corruption, we do not consider this a substitute for cryptographic integrity checks [33].

3.2 Security Model and Considered Attacks

In this work we consider only the source and the destination to be trusted. Nodes that can not be authenticated do not participate in the protocol, and are not trusted. Any intermediate node on the path between the source and destination can be authenticated and can participate in the protocol, but may exhibit byzantine behavior. The goal of our protocol is to detect byzantine behavior and avoid it. We define *byzantine behavior* as any action by an authenticated node that results in disruption or degradation of the routing service. We assume that an intermediate node can exhibit such behavior either alone or in collusion with other nodes. More generally, we use the term *fault* to refer to any disruption that causes significant loss or delay in the network. A fault can be caused by byzantine behavior, external adversaries, lower layer influences, and certain types of normal network behavior such as bursting traffic.

An adversary or group of adversaries can intercept, modify, or fabricate packets, create routing loops, drop packets selectively (often referred to as a *black hole*), artificially delay packets, route packets along non-optimal paths, or make a path look either longer or shorter than it is. All the above attacks result in disruption or degradation of the routing service. In addition, they can induce excess resource consumption which is particularly problematic in wireless networks.

There are strong attacks that our protocol can not prevent. One of these strong attacks, referred to as a *wormhole* [4], is where two attackers establish a path and tunnel packets from one to another. For example, the attackers can tunnel route request packets that can arrive faster than the normal route request flood. This may result in non-optimal adversarial controlled routing paths. Our protocol addresses this attack by treating the wormhole as a single link which will be avoided if it exhibits byzantine behavior, but does not prevent the wormhole formation. Also, we do not address traditional denial of service attacks which are characterized by packet injection with the goal of resource consumption.

Whenever possible, our protocol uses efficient cryptographic primitives. This requires pairwise shared keys[1] which are established on-demand. The public-key infrastructure used for authentication can be either completely distributed (as described in [5]), or Certificate Authority (CA) based. In the latter case, a distributed cluster of peer CAs sharing a common certificate and revocation list can be deployed to improve the CA's availability.

[1]We discourage group shared keys since this is an invitation for impersonation in a cooperative environment.

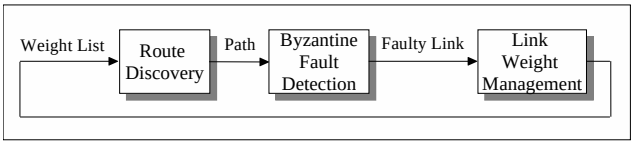

Figure 1: Secure Routing Protocol Phases

3.3 Problem Definition

The goal of this work is to provide a robust on-demand ad hoc routing service which is resilient to byzantine behavior and operates under the network and security models described in Sections 3.1 and 3.2. We attempt to bound the amount of damage an adversary or group of adversaries can cause to the network.

4. SECURE ROUTING PROTOCOL

Our protocol establishes a reliability metric based on past history and uses it to select the best path. The metric is represented by a list of link weights where high weights correspond to low reliability. Each node in the network maintains its own list, referred to as a *weight list*, and dynamically updates that list when it detects faults. Faulty links are identified using a secure adaptive probing technique that is embedded in the normal packet stream. These links are avoided using a secure route discovery protocol that incorporates the reliability metric.

More specifically, our routing protocol can be separated into three successive phases, each phase using as input the output from the previous (see Figure 1):

- *Route discovery with fault avoidance.* Using flooding, cryptographic primitives, and the source's weight list as input, this phase finds and outputs the full least weight path from the source to the destination.
- *Byzantine fault detection.* The goal of this phase is to discover faulty links on the path from the source to the destination. This phase takes as input the full path and outputs a faulty link. Our adaptive probing technique identifies a faulty link after $\log n$ faults occurred, where n is the length of the path. Cryptographic primitives and sequence numbers are used to protect the detection protocol from adversaries.
- *Link weight management.* This phase maintains a weight list of links discovered by the fault detection algorithm. A multiplicative increase scheme is used to penalize links which are then rehabilitated over time. The weight list is used by the route discovery phase to avoid faulty paths.

4.1 Route Discovery with Fault Avoidance

Our route discovery protocol floods both the route request and the response in order to ensure that if any fault free path exists in the network, a path can be established. However, there is no guarantee that the established path is free of

Procedure list:
CreateSignSend(item1, item2, ...) - creates a message of the concatenated item list, signed by the current node, and broadcasts it
Broadcast(message) - broadcasts a message
VerifySignature(node, signature) - verifies the signature and exits the procedure if the signature is not valid
Find(list, item) - returns an item in a list, or NULL if the item does not exist
InsertList(list, item) - inserts an item in a list
UpdateList(list, item) - replaces the item in a list
LinkWeight(weight_list, A, B) - returns the listed weight of the link between A and B, or one if the link is not listed

Code executed at node source **when a new route to node** destination **is needed:**
(1) CreateSignSend(REQUEST, destination, source, req_sequence, weight_list)

Code executed at node this_node **when a request message** req **is received:**
(2) **if**(Find(requests_list, req) = NULL)
(3) VerifySignature(req.source, req.signature)
(4) **if**(this_node = req.destination)
(5) CreateSignSend(RESPONSE, req.destination, req.source, req.req _sequence, req.high_weights_list)
(6) **else**
(7) Broadcast(req)
(8) **endif**
(9) InsertList(requests_list, req)
(10) **endif**

Code executed at node this_node **when a response message** res **is received:**
(11) update = **false**
(12) prev_node = res.destination
(13) total_weight = 0
(14) **for**(i = 0; i < res.no_hops; i++)
(15) total_weight += LinkWeight(res.weight_list, prev_node, res.hops[i].node)
(16) prev_node = res.hops[i].node
(17) **endfor**
(18) res.total_weight = total_weight + LinkWeight(res.weight_list, prev_node, this_node)
(19) prev_response = Find(responses_list, res)
(20) **if**(prev_response $\neq$ NULL)
(21) **if**(res.total_weight $\geq$ prev_response.total_weight)
(22) update = **true**
(23) **endif**
(24) **else**
(25) update = **true**
(26) **endif**
(27) **if**(update)
(28) VerifySignature(res.destination, res.signature)
(29) **for**(i = 0; i < res.no_hops; i++)
(30) VerifySignature(res.hops[i].node, res.hops[i].signature)
(31) **endfor**
(32) **if**(this_node = source)
(33) UpdateList(path_list, res)
(34) **else**
(35) CreateSignSend(res, this_node)
(36) UpdateList(responses_list, res)
(37) **endif**
(38) **endif**

Figure 2: Route Discovery Algorithm

adversarial nodes. The initial flood is required to guarantee that the route request reaches the destination. The response must also be flooded because if it was unicast, a single adversary could prevent the path from being established. If an adversary was able to prevent routes from being established, the fault detection algorithm would be unable to detect and avoid the faulty link since it requires a path as input in order to operate.

A digital signature is used to authenticate the source. This is required to prevent unauthorized nodes from initiating resource consuming route requests. An unauthorized route request would fail verification and be dropped by each of the requesting node's immediate neighbors, preventing the request from flooding through the network.

At the completion of the route discovery protocol, the source is provided with the complete path to the destination. Many on-demand routing protocols use route caching by intermediate nodes as an optimization; we do not consider it in this work because of the security implications. We intend to address route caching optimizations with strong security semantics in a future work.

Our route discovery protocol uses link weights to avoid faults. A weight list is provided by the link weight management phase (Section 4.3). The route discovery protocol chooses a route that is a minimum weight path between the source and the destination. This path is found during a flood by accumulating the cost hop by hop and forwarding the flood only if the new cost is less than the previously forwarded cost. The protocol uses digital signatures at each hop to prevent an adversary from specifying an arbitrary path. For example, it can stop an adversary from inventing a short path in an attempt to draw packets into a black hole. Since the cost associated with signing a message at each hop is very high, the weights are accumulated as part of the response flood instead of the request flood in order to minimize the cost of route requests to unreachable destinations.

If only the source verifies all of the weights and signatures, then the protocol becomes vulnerable to attacks on the response flood propagation. The adversaries could block correct information from reaching the source by propagating low cost fabricated responses. The source can ignore non-authentic responses, however, since intermediate nodes only re-send lower cost information, a valid response would never reach the source. Therefore, each intermediate node must verify the weights and the signatures carried by a response, in order to guarantee that a path will be established.

An adversary can still influence the path selection by creating what we refer to as *virtual links*. A virtual link is formed when adversaries form wormholes, as described in Section 3.2, or any other type of shortcuts in the network. A virtual link can be created by deleting one or more hops from the end of the route response. Our detection algorithm (Section 4.2) can identify and avoid virtual links if they exhibit byzantine behavior, but our route discovery algorithm does not prevent their formation. We present a detailed analysis of the effect of virtual links in Section 5.

As part of the route discovery protocol, each node maintains a list of recent requests and responses that it has already forwarded. The following five steps comprise the route discovery protocol (see also Figure 2):

I. Request Initiation. The source creates and signs a request that includes the destination, the source, a sequence number, and a weight list (see Line 1, Figure 2). The source then broadcasts this request to its neighbors. The source's signature allows the destination and intermediate nodes to authenticate the request and prevents an adversary from creating a false route request.

II. Request Propagation. The request propagates to the destination via flooding which is performed by the intermediate nodes as follows. When receiving a request, the node first checks its list of recently seen requests for a matching request (one with the exact same destination, source, and request identifiers). If there is no matching request in its list, and the source's signature is valid, it stores the request in its list and rebroadcasts the request (see Lines 2-10, Figure 2). If there is a matching request, the node does nothing.

III. Request Receipt / Response Initiation. Upon receiving a new request from a source for the first time, the destination verifies the authenticity of the request, creates and signs a response that contains the source, the destination, a response sequence number and the weight list from the request packet. The destination then broadcasts this response (see Lines 2-10, Figure 2).

IV. Response Propagation. When receiving a response, the node computes the total weight of the path by summing the weight of all the links on the specified path to this node (Lines 12-18, Figure 2). If the total weight is less than any previously forwarded matching response (same source, destination and response identifiers), the node verifies the signatures of the response header and every hop listed on the packet so far[2] (Lines 28-31, Figure 2). If the entire packet is verified, the node appends its identifier to the end of the packet, signs the appended packet, and broadcasts the modified response (Lines 35-36, Figure 2).

V. Response Receipt. When the source receives a response, it performs the same computation and verification as the intermediate nodes as described in the response propagation step. If the path in the response is better than the best path received so far, the source updates the route used to send packets to that specific destination (see Line 33, Figure 2).

4.2 Byzantine Fault Detection

Our detection algorithm is based on using acknowledgments (*acks*) of the data packets. If a valid ack is not received within a timeout, it is assumed that the packet has been lost. Note that this definition of loss includes both malicious and non-malicious causes. A loss can be caused by packet drop due to buffer overflow, packet corruption due to interference, a malicious attempt to modify the packet contents, or any other event that prevents either the packet or the ack from being received and verified within the timeout.

A network operating "normally" exhibits some amount of loss. We define a *threshold* that sets a bound on what is considered a tolerable loss rate. In a well behaved network the loss rate should stay below the threshold. We define a *fault* as a loss rate greater than or equal to the threshold.

[2]To maximize the performance of multiple verifications we use RSA keys with a low public exponent.

The value of the threshold also specifies the amount of loss that an adversary can create without being detected. Hence, the threshold should be chosen as low as possible, while still greater than the normal loss rate. The threshold value is determined by the source, and may be varied independently for each route to accommodate different situations, but this work uses a fixed threshold.

While this threshold scheme may seem overly "simple", we would like to emphasize that our protocol provides fault avoidance and never disconnects nodes from the network. Thus, the impact of false positives, due to normal events such as bursting traffic, is drastically reduced. This provides a much more flexible solution than one where nodes are declared faulty and excluded from the network. In addition, this avoidance property allows the threshold to be set very low, where it may be periodically triggered by false positives, without severely impacting network performance or affecting network connectivity.

A substantial advantage of our protocol is that it limits the overhead to a minimum under normal conditions. Only the destination is required to send an ack when no faults have occurred. If losses exceed the threshold, the protocol attempts to locate the faulty link. This is achieved by requiring a dynamic set of intermediate nodes, in addition to the destination node, to send acks to the source.

Normal topology changes occur frequently in ad hoc wireless networks. Although our detection protocol locates "faulty links" that are caused by these changes, an optimized mechanism for detecting them would decrease the overhead and detection time. Any of the mechanisms described in the route maintenance section of the DSR protocol [3], for instance MAC layer notification, can be used as an optimized topology change detector. When our protocol receives notification from such a detector, it reacts by creating a route error message that is propagated along the path back to the source. The node that generates this message, signs it, in order to provide integrity and authentication. Upon receipt of an authenticated route error message, the source passes the faulty link to the link weight management phase. Note that an intermediate node exhibiting byzantine behavior can always incriminate one of its links, so adding a mechanism that allows it to explicitly declare one of its links faulty, does not weaken the security model.

Fault Detection Overview. Our fault detection protocol requires the destination to return an ack to the source, for every successfully received data packet. The source keeps track of the number of recent losses (acks not received over a window of recent packets). If the number of recent losses violates the acceptable threshold, the protocol registers a fault between the source and the destination and starts a binary search on the path, in order to identify the faulty link. A simple example is illustrated in Figure 3.

The source controls the search by specifying a list of intermediate nodes on data packets. Each node in the list, in addition to the destination, must send an ack for the packet. We refer to the set of nodes required to send acks as probed nodes, or for short *probes*. Since the list of probed nodes is specified on legitimate traffic, an adversary is unable to drop traffic without also dropping the list of probed nodes and eventually being detected.

The list of probes defines a set of non-overlapping intervals that cover the whole path, where each interval covers the sub-path between the two consecutive probes that form its endpoints. When a fault is detected on an interval, the interval is divided in two by inserting a new probe. This new probe is added to the list of probes appended to future packets. The process of sub-division continues until a fault is detected on an interval that corresponds to a single link. In this case, the link is identified as being faulty and is passed as input to the link weight management phase (see Figure 1). The path sub-division process is a binary search that proceeds one step for each fault detected. This results in the identification of a faulty link after $\log n$ faults are detected, where n is the length of the path.

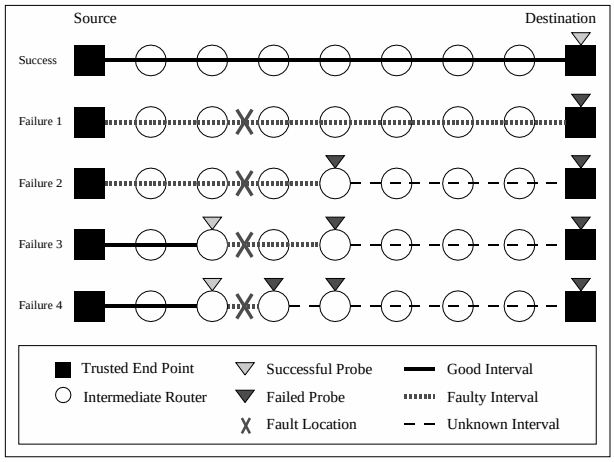

Figure 3: Byzantine Fault Detection

We use shared keys between the source and each probed node as a basis for our cryptographic primitives in order to avoid the prohibitively high cost of using public key cryptography on an per packet basis. These pairwise shared keys can be established on-demand via a key exchange protocol such as Diffie-Hellman [34], authenticated using digital signatures. The on-demand key exchange must be fully integrated into the fault detection protocol in order to maintain the security semantics. The integrated key exchange operates similarly to the probe and ack specification discussed below (see also Figure 4), but it is not described in detail in this work.

Probe Specification. The mechanism for specifying the probe list on a packet is essential for the correct operation of the detection protocol. The probes are specified in the list in the same order as they appear on the path. The list is "onion" encrypted [17]. Each probe is specified by the identifier of the node, an HMAC of the packet (not including the list), and the encrypted remaining list (see Lines 3-6, Figure 4). Both the HMAC and the encrypted remaining list are computed with the shared key between the source and that node. An HMAC [12] using a one-way hash function such as SHA1 [35] and a standard block cipher encryption algorithm such as AES [36] can be used.

A node can detect if it is required to send acks by checking the identifier at the beginning of the list (see Lines 8-12, Figure 4). If it matches, then it verifies the HMAC of the packet and replaces the list on the packet with the decrypted version of the remaining list. This mechanism forces the

Procedure list:
Cat(a, b, ...) - returns the concatenation of a, b, etc.
Hmac(data, key) - compute and return the hmac of data using key
Encrypt/Decrypt(data, key) - encrypt/decrypt data with key and return result
Report_Loss_and_Return(node) - reports that a loss was detected on the interval before *node* and exit the procedure

Code executed at source **when sending a packet with the contents** data **to** destination **:**
(1) body = Cat(destination.id, source.id, destination.counter++, Encrypt (data, destination.key))
(2) tail = Hmac(body, destination.key)
(3) **for**(i = probe_list.length - 1, i $\geq$ 0, i−−)
(4) tail = Encrypt(tail, probe_list[i].key)
(5) tail = Cat(probe_list[i].id, Hmac(body, probe_list[i].key), tail)
(6) **endfor**
(7) Send(Cat(body, tail))

Code executed at this_node **when receiving a packet with the contents** source, destination, enc_data, id, hmac, enc_remainder **:**
(8) **if**(id = this_node **and** hmac = Hmac(enc_data, source.key)
(9) Send(Cat(source, destination, enc_data, Decrypt(enc_remainder, key))
(10) waiting_for_ack = **true**
(11) Schedule_ack_timer()
(12) **endif**

Code executed at destination **when receiving a packet with the contents** source, destination, counter, enc_data, hmac:
(13) **if**(counter > prev_counter **and** hmac = Hmac(Cat(source, destination, counter, enc_data)))
(14) Deliver(Decrypt(enc_data, source.key))
(15) Send(Cat(source.id, destination.id, counter, Hmac(Cat(destination.id, counter), source.key)))
(16) **endif**

Code executed at probed_node **when receiving an ack with the contents** source, ack_node, counter, enc_remainder:
(17) **if**(waiting_for_ack)
(18) encrypted_ack = Encrypt(Cat(ack_node, counter, enc_remainder), source.key)
(19) Send(Cat(source.id, probed_node.id, counter, enc_ack, Hmac(Cat(probed_node.id, counter, enc_ack), source.key)))
(20) waiting_for_ack = **false**
(21) Unschedule_ack_timer()
(22) **endif**

Code executed at this_node **when ack timer expires:**
(23) waiting_for_ack = **false**
(24) Send(Cat(source, this_node.id, counter, Hmac(Cat(this_node.id, counter), source.key)))

Code executed at source **when ack timer expires:**
(25) waiting_for_ack = **false**
(26) Report_Loss_and_Return(probe_list[0])

Code executed at source **when receiving an ack with the contents** source, ack_node, counter, enc_remainder, hmac:
(27) **if**(wait_for_ack **and** ack_node = probe_list[0].id **and** hmac = Hmac(Cat(ack_node, counter, enc_remainder), ack_node.key))
(28) waiting_for_ack = **false**
(29) Unschedule_ack_timer()
(30) **for**(i = 1, i < probe_list.length, i++)
(31) **if**(enc_remainder = NULL)
(32) Report_Loss_and_Return(probe_list[i])
(33) **endif**
(34) ack_node, counter, enc_remainder, hmac = Decrypt(enc_remainder, probe_list[i-1].key)
(35) **if**(ack_node $\neq$ probe_list[i].id **or** hmac $\neq$ Hmac(Cat(ack_node, counter, enc_remainder), ack_node.key))
(36) Report_Loss_and_Return(probe_list[i])
(37) **endif**
(38) **endfor**
(39) **if**(enc_remainder = NULL)
(40) Report_Loss_and_Return(destination)
(41) **endif**
(42) ack_node, counter, hmac = Decrypt(enc_remainder, probe_list[i-1].key)
(43) **if**(ack_node $\neq$ destination **or** hmac $\neq$ Hmac(Cat(ack_node, counter), destination.key))
(44) Report_Loss_and_Return(destination)
(45) **return**
(46) **endif**
(47) Success()
(48) **endif**

Figure 4: Probe and Acknowledgement Specification

packet to traverse the probes in order, which verifies the route taken. Additionally, it verifies the contents of the packet at every probe point. The onion encryption prevents the adversary from incriminating other links by removing specific nodes from the probe list. Note that the adversary is able to remove the entire probe list, but this will incriminate one of its own links.

Acknowledgment Specification. If the adversary can drop individual acks, it can incriminate any arbitrary link along the path. In order to prevent this, each probe does not send its ack immediately, but waits for the ack from the next probe and combines them into one ack. Each ack consists of the identifier of the probe, the identifier of the data packet that is being acknowledged, the ack received from the next probe encrypted with the key shared by this probe and the source, and an HMAC of the new combined ack (see Lines 15 and 18-19, Figure 4).

If no ack is received within a timeout, the probe gives up waiting, and creates and sends its ack (see Line 24, Figure 4). The timeouts are set up in such a way that if there is a failure, all the acks before the failure point can be combined without other timeouts occurring. This is accomplished by setting the timeout for each probe to be the upper bound of the round-trip from it to the destination.

Upon receipt of an ack, the source checks the acks from each probe by successively verifying the HMACs and decrypting the next ack (see Lines 27-54, Figure 4). The source either verifies all the acks up through the destination, or discovers a loss on the interval following the last ack.

Interval and Probe Management. Let τ be the acceptable threshold loss rate. By using the above probe and acknowledgment specifications, it is simple to attribute losses to individual intervals. A loss is attributed to an interval between two probes when the source successfully received and verified an ack from the closer probe, but does not from the further probe. When the loss rate on an interval exceeds τ, the interval is divided in two.

Maintaining probes adds overhead to our protocol, so it is desirable to retire probes when they are no longer needed. The mechanism for deciding when to retire probes is based on the loss rate τ and the number of lost packets. The goal is to amortize the cost of the lost packets over enough good packets, so that the aggregate loss rate is bounded to τ.

Each interval has an associated counter C that specifies its lifetime. Initially, there is one interval with a counter of zero (there are initially no losses between the source and destination). When a fault is detected on an interval with a counter C, a new probe is inserted which *divides* the interval. Each of the two new intervals have their counters initialized to $\mu/\tau + C$, where μ is the number of losses that caused the fault. The counters are decremented for every ack that is successfully received, until they reach zero. When the counters of both intervals on either side of a probe reach zero, the probe is retired *joining* the two intervals.

In the worst case scenario, a dynamic adversary can cause enough loss to trigger a fault, then switch to causing loss just under τ in order to wait out the additional probe, and then repeat when the probe is removed. This results in a loss rate bounded to 2τ. If the adversary attempts to create a higher loss rate, the algorithm will be able to identify the faulty link.

4.3 Link Weight Management

An important aspect of our protocol is its ability to avoid faulty links in the process of route discovery by the use of link weights. The decision to identify a link as faulty is made by the detection phase of the protocol. The management scheme maintains the weight list using the history of faults that have been detected. When a link is identified as faulty, we use a multiplicative increase scheme to double its weight.

The technique we use for reseting a link weight is similar to the one we use for retiring probes (see Section 4.2). The weight of a link can be reset to half of the previous value after the counter associated with that link returns to zero. If μ is the number of packets dropped while identifying a faulty link, then the link's counter is increased by μ/τ where τ is the threshold loss rate. Each non-zero counter is reduced by $1/m$ for every successfully delivered packet, where m is the number of links with non-zero counters. This bounds the aggregate loss rate to 2τ in the worst case.

5. ANALYSIS

Our protocol ensures that, even in a highly adversarial controlled network, as long as there is one fault-free path, it will be discovered after a bounded number of faults have occurred. As defined in Section 4.2, a fault means a violation of the threshold loss rate. We consider a network of n nodes of which k exhibit adversarial behavior. The adversaries cooperate and create the maximum number of virtual links possible in order to slow the convergence of our algorithm.

We provide an analysis of the upper bound for the total number of packets lost while finding the fault free path. This bound is defined by the number of losses that result in an increase of the costs of all adversarial controlled paths above the cost of the fault free path.

Let q^- and q^+ be the total number of lost packets and successfully transmitted packets, respectively. Ideally, $q^- - \rho \cdot q^+ \leq 0$, where ρ is the transmission success rate, slightly higher than the original threshold. This means the number of lost packets is a ρ-fraction of the number of transmitted packets. While this is not quite true, it is true "up to an additive constant", i.e. ignoring a bounded number ϕ of packets lost. Specifically, we prove that there exists an upper bound ϕ for the previous expression. We show that:

$$q^- - \rho \cdot q^+ \leq \phi \tag{1}$$

Assume that there are k adversarial nodes, $k < n$. We denote by $\tilde{E}$ the set of "virtual links" controlled by adversarial nodes. The maximum size of $\tilde{E}$ is kn.

Consider a faulty link e, convicted j_e times and rehabilitated a_e times. Then, its weight, w_e, is at most n, $w_e = n$ means that the whole path is adversarial. By the algorithm, w_e is given by the formula:

$$w_e = 2^{j_e - a_e} \tag{2}$$

The number of convictions is at least $\frac{q^-}{\mu}$, so

$$\frac{q^-}{\mu} - \sum_{e \in \tilde{E}} j_e < 0. \tag{3}$$

Also, the number of rehabilitations is at most $\frac{q^+}{\mu/\rho}$, so

$$\sum_{e \in \tilde{E}} a_e - \frac{q^+}{\mu/\rho} < 0 \quad (4)$$

where μ is the number of lost packets that exposes a link. Thus

$$\frac{q^-}{\mu} - \frac{q^+}{\mu/\rho} \leq \sum_{e \in \tilde{E}} (j_e - a_e) \quad (5)$$

From Eq. 2 we have $j_e - a_e = \log w_e$. Therefore:

$$\sum_{e \in \tilde{E}} (j_e - a_e) = \sum_{e \in \tilde{E}} \log w_e \quad (6)$$

By combining Eq. 5 and 6, we obtain

$$q^- - \rho \cdot q^+ \leq \mu \sum_{e \in \tilde{E}} \log w_e \leq \mu \cdot kn \cdot \log n \quad (7)$$

and since $\mu = b \log n$, where b is the number of lost packets per window, Eq. 7 becomes

$$q^- - \rho \cdot q^+ \leq b \cdot kn \cdot \log^2 n \quad (8)$$

Therefore, the amount of disruption a dynamic adversary can cause to the network is bounded. Note that kn represents the number of links controlled by an adversary. If there is no adversarial node Eq. 8 becomes the ideal case where $q^- - \rho \cdot q^+ \leq 0$.

6. CONCLUSIONS AND FUTURE WORK

We presented a secure on-demand routing protocol resilient to byzantine failures. Our scheme detects malicious links after $\log n$ faults occurred, where n is the length of the routing path. These links are then avoided by the route discovery protocol. Our protocol bounds logarithmically the total amount of damage that can be caused by an attacker or group of attackers.

An important aspect of our protocol is the algorithm used to detect that a fault has occurred. However, it is difficult to design such a scheme that is resistant to a large number of adversaries. The method suggested in this paper uses a fixed threshold scheme. We intend to explore other methods, such as adaptive threshold or probabilistic schemes which may provide superior performance and flexibility.

In order to further enhance performance, we would like to investigate ways of taking advantage of route caching without breaching our security guarantees.

We also plan to evaluate the overhead of our protocol with respect to existing protocols, in normal, non-faulty conditions as well as in adversarial environments. Finally, we are interested in investigating means of protecting routing against traditional denial of service attacks.

Acknowledgments

We are grateful to Giuseppe Ateniese, Avi Rubin, Gene Tsudik and Moti Yung for their comments. We would like to thank Jonathan Stanton and Ciprian Tutu for helpful feedback and discussions. We also thank the anonymous referees for their comments.

We would like to thank the Johns Hopkins University Information Security Institute for providing the funding that made this research possible.

7. REFERENCES

[1] J. Kurose and K. Ross, *Computer Networking, a top down approach featuring the Internet*. Addison-Wesley Longman, 2000.

[2] C. E. Perkins and E. M. Royer, *Ad hoc Networking*, ch. Ad hoc On-Demand Distance Vector Routing. Addison-Wesley, 2000.

[3] D. B. Johnson, D. A. Maltz, and J. Broch, *DSR: The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks. in Ad Hoc Networking*, ch. 5, pp. 139–172. Addison-Wesley, 2001.

[4] Y.-C. Hu, A. Perrig, and D. B. Johnson, "Ariadne: A secure on-demand routing protocol for ad hoc networks," in *The 8th ACM International Conference on Mobile Computing and Networking*, September 2002. To appear.

[5] J.-P. Hubaux, L. Buttyan, and S. Capkun, "The quest for security in mobile ad hoc networks," in *The 2nd ACM Symposium on Mobile Ad Hoc Networking and Computing*, October 2001.

[6] P. Zimmermann, *The Official PGP User's Guide*. MIT Press, 1995.

[7] L. Zhou and Z. Haas, "Securing ad hoc networks," *IEEE Network Magazine*, vol. 13, November/December 1999.

[8] M. Brown, D. Cheung, D. Hankerson, J. Hernandez, M. Kirkup, and A. Menezes., "PGP in constrained wireless devices," in *The 9th USENIX Security Symposium*, USENIX, August 2000.

[9] S. Yi, P. Naldurg, and R. Kravets, "Security-aware ad hoc routing for wireless networks," in *The 2nd ACM Symposium on Mobile Ad Hoc Networking and Computing*, October 2001.

[10] R. Hauser, T. Przygienda, , and G. Tsudik, "Reducing the cost of security in link-state routing," in *Symposium of Network and Distributed Systems Security*, 1997.

[11] P. Papadimitratos and Z. Haas, "Secure routing for mobile ad hoc networks," in *SCS Communication Networks and Distributed Systems Modeling and Simulation Conference*, pp. 27–31, January 2002.

[12] *The Keyed-Hash Message Authentication Code (HMAC)*. No. FIPS 198, National Institute for Standards and Technology (NIST), 2002. http://csrc.nist.gov/publications/fips/index.html.

[13] Y.-C. Hu, D. B. Johnson, and A. Perrig, "SEAD: Secure efficient distance vector routing for mobile wireless ad hoc networks," in *The 4th IEEE Workshop on Mobile Computing Systems and Applications*, IEEE, June 2002.

[14] C. E. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers," in *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, 1994.

[15] A. Perrig, R. Canetti, D. Song, and D. Tygar, "Efficient and secure source authentication for multicast," in *Network and Distributed System Security Symposium*, February 2001.

[16] B. Dahill, B. Levine, C. Shields, and E. Royer, "A secure routing protocol for ad hoc networks," Tech. Rep. 01-37, Department of Computer Science,

University of Massachusetts, August 2001.

[17] P. F. Syverson, D. M. Goldschlag, and M. G. Reed, "Anonymous connections and onion routing," in *IEEE Symposium on Security and Privacy*, 1997.

[18] S. Marti, T. Giuli, K. Lai, and M. Baker, "Mitigating routing misbehavior in mobile ad hoc networks," in *The 6th ACM International Conference on Mobile Computing and Networking*, August 2000.

[19] S. Cheung, "An efficient message authentication scheme for link state routing," in *The 13th Annual Computer Security Applications Conference*, pp. 90–98, December 1997.

[20] K. Zhang, "Efficient protocols for signing routing messages," in *Symposium on Networks and Distributed Systems Security*, 1998.

[21] M. T. Goodrich, "Efficient and secure network routing algorithms." Provisional patent filing., January 2001.

[22] B. R. Smith, S. Murthy, and J. Garcia-Luna-Aceves, "Securing distance-vector routing protocols," in *Symposium on Networks and Distributed Systems Security*, 1997.

[23] S. L. Murphy and M. R. Badger, "Digital signature protection of the OSPF routing protocol," in *Symposium on Networks and Distributed Systems Security*, 1996.

[24] B. Smith and J. Garcia-Luna-Aceves, "Efficient security mechanisms for the border gateway routing protocol," *Computer Communications (Elsevier)*, vol. 21, no. 3, pp. 203–210, 1998.

[25] S. F. Wu, F. yi Wang, B. M. Vetter, W. R. Cleaveland, Y. F. Jou, F. Gong, and C. Sargor, "Intrusion detection for link-state routing protocols," in *IEEE Symposium on Security and Privacy*, 1997.

[26] D. Qu, B. M. Vetter, F. Wang, R. Narayan, S. F. Wu, Y. F. Jou, F. Gong, and C. Sargor, "Statistical anomaly detection for link-state routing protocols," in *IEEE Symposium on Security and Privacy (5 Minutes)*, May 1997.

[27] S. Wu, H. Chang, D. Qu, F. W. F. Jou, F. Gong, C. Sargor, and R. Cleaveland, "JiNao: Design and implementation of a scalable intrusion detection system for the OSPF routing protocol," *Journal of Computer Networks and ISDN Systems*, 1999.

[28] R. Perlman, *Network Layer Protocols with Byzantine Robustness*. PhD thesis, MIT LCS TR-429, October 1988.

[29] S. Cheung and K. Levitt, "Protecting routing infrastructures from denial of service using cooperative intrusion detection," in *New Security Paradigms Workshop*, 1997.

[30] K. A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R. A. Olsson, "Detecting disruptive routers: A distributed network monitoring approach," in *IEEE Symposium on Security and Privacy*, 1998.

[31] *ANSI/IEEE Std 802.11, 1999 Edition*. 1999. http://standards.ieee.org/catalog/olis/lanman.html.

[32] V. Bharghavan, A. J. Demers, S. Shenker, and L. Zhang, "MACAW: A media access protocol for wireless LAN's," in *SIGCOMM*, pp. 212–225, 1994.

[33] J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," in *ACM SIGCOM*, August/September 2000.

[34] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Trans. Inform. Theory*, vol. IT-22, pp. 644–654, November 1976.

[35] *Secure Hash Standard (SHA1)*. No. FIPS 180-1, National Institute for Standards and Technology (NIST), 1995. http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[36] *Advanced Encryption Standard (AES)*. No. FIPS 197, National Institute for Standards and Technology (NIST), 2001. http://csrc.nist.gov/encryption/aes/.

Reliable Communication in Overlay Networks

Yair Amir and Claudiu Danilov
Johns Hopkins University
{yairamir, claudiu}@cs.jhu.edu

Abstract

Reliable point-to-point communication is usually achieved in overlay networks by applying TCP on the end nodes of a connection. This paper presents a hop-by-hop reliability approach that considerably reduces the latency and jitter of reliable connections. Our approach is feasible and beneficial in overlay networks that do not have the scalability and interoperability requirements of the global Internet.

The effects of the hop-by-hop reliability approach are quantified in simulation as well as in practice using a newly developed overlay network system that is fair with the external traffic on the Internet. The experimental results show that the overhead associated with overlay network processing at the application level does not play an important factor compared with the considerable gain of the approach.

1 Introduction

Reliable point-to-point communication is one of the main utilizations of the Internet, where over the last few decades TCP has served as the dominant protocol. Over the Internet, reliable communication is performed end-to-end in order to address the severe scalability and interoperability requirements of a network in which potentially every computer on the planet could participate. Thus, all the work required in a reliable connection is distributed only to the two end nodes of that connection, while intermediate nodes route packets without keeping any information about the individual packets they transfer.

Overlay networks are opening new ways to Internet usability, mainly by adding new services (e.g. built-in security) that are not available or cannot be implemented in the current Internet, and also by providing improved services such as higher availability [2]. However, the usage of overlay networks may come with a price, usually in added latency that is incurred due to longer paths created by overlay routing, and by the need to process the messages in the application level by every overlay node on the path.

Reliable communication in overlay networks is usually achieved by applying TCP on the edges of a connection. This surely works. However, this paper argues that employing hop-by-hop reliability techniques considerably reduces the average latency and jitter of reliable communication. When using such an approach one has to consider networking aspects such as congestion control, fairness, flow control and end-to-end reliability. We discuss these aspects and our design decisions in Section 2.

In Section 3, we demonstrate through simulation that our approach provides tremendous benefit for the application as well as for the network itself, even when very few packets are lost. Simulations usually do not take into account many practical issues such as processing overhead, CPU scheduling, and most important, the fact that overlay network processing is performed at the application level of general purpose computers. These may have considerable impact on real-life behavior and performance. Therefore, we test our approach in practice on an overlay network platform called Spines that we have built.

We introduce Spines in Section 4. Spines [16] is an open source research platform that allows deployment of overlay networks in the Internet. We run the same experiments that were simulated, on a Spines overlay network. The results are presented in Section 5. We show that the benefit of hop-by-hop reliability greatly overcomes the overhead of overlay routing and achieves much better performance compared to standard end-to-end TCP connections deployed on the same overlay network.

We describe existing related work and compare it with our approach in Section 6, and end the paper, concluding that hop-by-hop reliability is a viable and beneficial approach to reliable communication in overlay networks.

2 Hop-by-hop reliable communication in overlay networks

An overlay network constructs a user level graph on top of an existing networking infrastructure such as the Internet, using only a subset of the available network links and nodes. An overlay link is a virtual edge in this graph and

may consist of many actual links in the underlying network. Overlay nodes act as routers, forwarding packets to the next overlay link toward the destination. At the physical level, packets traveling along a virtual edge between two overlay nodes follow the actual physical links that form that edge.

Overlay networks have two main drawbacks. First, the overlay routers incur some overhead every time a message is processed, which requires delivering the message to the application level, processing it, and resending the message to the next overlay router. Second, the placement of overlay routers in the topology of the physical network is often far from optimal, because the creator of the overlay network rarely has control over the physical network (usually the Internet) or even the knowledge about its actual topology. Therefore, overlay networks provide longer paths that have higher latency than point to point Internet connections.

The easiest way to achieve reliability in Overlay Networks is to use a reliable protocol, usually TCP, between the end points of a connection. This mechanism has the benefit of simplicity in implementation and deployment, but pays a high price upon recovery from a loss. As overlay paths have higher delays, it takes a relatively long time to detect a loss, and data packets and acknowledgments are sent on multiple overlay hops in order to recover the missed packet.

2.1 Hop-by-hop reliability

We propose a mechanism that recovers the losses only on the overlay hop on which they occurred, localizing the congestion and enabling faster recovery. Since an overlay link has a lower delay compared to an end-to-end connection that traverses multiple hops, we can detect the loss faster and resend the missed packet locally. Moreover, the congestion control on the overlay link can increase the congestion window back faster than an end-to-end connection, as it has a smaller round-trip time.

Hop-by-hop reliability involves buffers and processing in the intermediate overlay nodes. These nodes need to deploy a reliable protocol, and keep track of packets, acknowledgments and congestion control, in addition to their regular routing functionality. Although such an approach may not be feasible to implement at the level of the Internet routers due to scalability limitations, we can easily deploy it at the level of an overlay network, thus allowing us to pinpoint the congestion, limiting the problem to the congested part of the network.

Let's consider a simple overlay network composed of five 10 millisecond links in a chain, as shown in Figure 1. Such a network may span a continent such as North America or Europe. Every time a packet is lost (say on link C-D), it will take at least 50 milliseconds from the time that packet was sent until the receiver detects the loss, and at least 50 additional milliseconds until the sender learns about

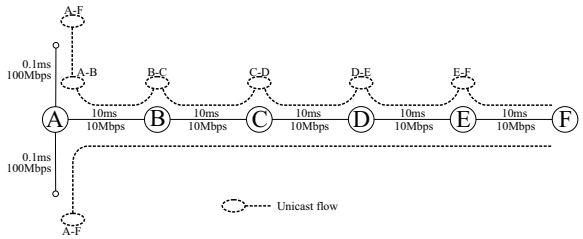

Figure 1. Chain Network Setup

it. The sender will retransmit the lost packet that will travel 50 more milliseconds until the receiver will get it. This accounts for a total of at least 150 milliseconds to recover a packet. If the sender continues to send packets during the recovery period, even if the new packets arrive at the receiver in time (assuming no loss for them), they will not be delivered at the receiver until the missing packet is recovered, as they are not in order. Our experimental results presented in Sections 3 and 5 show that the number of packets delayed is much higher than the number of packets lost.

Let us assume that we use five reliable hops of 10 milliseconds each instead of one end-to-end connection. Suppose the same message is lost on the same intermediate link, as in the above scenario. On that particular link (with 10 milliseconds delay) it will take only about 30 milliseconds for the receiver to recover the missed packet. Moreover, as the recovery period is smaller, a smaller number of out of order packets will be delayed. This effect is more visible as the throughput increases.

2.2 End-to-end reliability and congestion control

Simply having reliable overlay links does not guarantee end-to-end reliability. Intermediate nodes may crash, overlay links may get disconnected. However, such events are not likely to happen and most of the reliability problems (generated by network losses) are indeed handled locally at the level of each hop. Therefore we still need to send some end-to-end acknowledgments from the end-receiver to the initial sender, at least once per round-trip time, but not for every packet. This means that for some of the packets we will pay the price of sending two acknowledgments, one on each of the overlay hops for local reliability, and one end-to-end, that will traverse the entire path. However, acknowledgments are small and are piggy-backed on the data packets whenever possible. We believe that the penalty of sending double acknowledgments for some of the packets is drastically reduced by resending the missed data packets (which are much bigger than the acknowledgments) only locally, on the hop where the loss occurred, and not on the entire end-to-end path.

Intermediate overlay nodes handle reliability and con-

gestion control only for the links to their immediate neighbors and do not keep any state for individual flows in the system. Packets are forwarded and acknowledged per link, regardless of their originator. This is essensial for the scalability with the number of reliable sessions in the system.

Since the packets are not needed in order at the intermediate overlay nodes, but only at the final destination, in case of a loss there is no need to delay the following packets locally on each link in order to forward them FIFO on the next link. We choose to forward the packets even if out of order on intermediate hops, and reestablish the initial order at the end receiver.

Our tests show that out of order forwarding reduces the burstiness inside the network. It also contributes to the reduction of the end-to-end latency (although that contribution is not as significant as the latency reduction achieved by the hop-by-hop reliability). The latency effect of out of order forwarding is magnified when multiple flows use the same overlay link. In that case, they do not need to reorder packets with respect to each other but only according to their own packets. The same occurs when more than one overlay link is congested and looses packets.

Overlay links are seen as individual point-to-point connections by the underlying network. Since overlay flows coexist with external traffic, each overlay link needs to have a congestion control mechanism in place. Our approach uses a window-based congestion control on each overlay link, that very closely follows the slow start and congestion avoidance of TCP [11].

The available bandwidth is different on each overlay link, depending on the underlying network characteristics, and is also dynamic, as the overlay link congestion control adjusts to provide fairness with the external traffic. If, at an intermediate node, the incoming traffic is bigger than the outgoing available bandwidth of the overlay link, that node will buffer the incoming packets, but if the condition persists it will either store an infinite number of packets or will start dropping them. Since end-to-end recovery is expensive, there needs to exist a congestion control mechanism that will limit, or even better, avoid packet losses at the overlay level. As opposed to the regular mechanism in TCP that uses packet losses to signal congestion, we use an explicit congestion notification scheme [15] where congested routers stamp the header of the data packets. Upon receiving such a stamped packet, the end receiver will send an end to end acknowledgment signaling the congestion immediately, and the sender's congestion control will treat that acknowledgment as a loss, even though the sender will not resend the corresponding packet. Note that the initial sender still sends retransmissions if necessary (e.g. in case of node failures and rerouting).

Since end-to-end acknowledgments are not sent for every packet, the end-to-end window may advance in big chunks once a cummulative acknowledgment is received. If the network path is not congested, this phenomenon does not affect the burstiness of the traffic, as the sending throughput is anyway smaller than the size of the window. However, in case of congestion the receiver sends end-to-end acknowledgments for every packet (stamped by an intermediate overlay router) until the congestion is resolved.

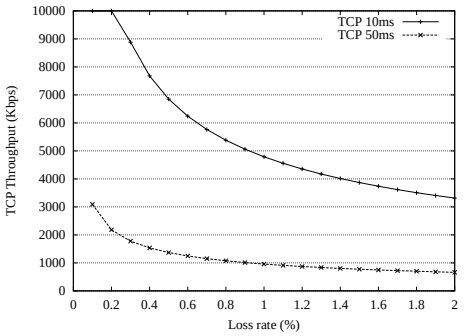

Figure 2. TCP throughput (analytical model)

2.3 Fairness

Since we intend to deploy our protocols on the Internet we need to share the global resources fairly with the external TCP traffic. A "TCP-compatible" flow is defined in [3] as one that is responsive to congestion notification, and in steady state, it uses no more bandwidth than a conformant TCP running under comparable conditions (loss rate, round-trip time, packet size, etc.).

The throughput obtained by a conformant TCP flow is evaluated analytically in [13], where the authors approximate the bandwidth B of a TCP flow as a function of packet size s, loss rate p and round-trip time RTT, where T_0 is the retransmission timeout and b is the number of packets that have to be received before sending an acknowledgment.

$$B = \frac{s}{RTT\sqrt{\frac{2bp}{3}} + T_0 3\sqrt{\frac{3bp}{8}}p(1+32p^2)}$$

Considering $b = 1$ and $T_0 = RTT$ in the ideal case, on a network topology such as in in Figure 1 the throughput obtained by an end-to-end TCP connection (50 millisecond delay) and by a short one hop TCP connection (10 millisecond delay on link CD) sending 1000 byte packets are shown in Figure 2 as a function of loss rate.

Clearly, an end-to-end reliable connection with a delay of 50 milliseconds will achieve less bandwidth than a hop-by-hop flow that will be limited only by the short bottleneck link C-D with 10 milliseconds delay, where the losses occur. This phenomenon happens because TCP throughput is biased against long connections. Analytically, RTT

appears at the denominator of the throughput formula, and in practice it will take more time for the long connection to recover its congestion window (the congestion avoidance protocol adds one to the congestion window for each RTT).

Note that achieving more throughput by a hop-by-hop flow does not happen with respect to external TCP connections that run outside of the overlay traffic. Each of the overlay links provides fairness and congestion control with respect to the external flows. A comparison of the throughput obtained by a single flow traversing multiple hops on the overlay network with one that uses the Internet directly cannot be done because of several factors:

- Flows that run within the overlay network usually have longer paths (higher delay) than direct Internet connections (due to the overlay routing which is usually far from optimal), and therefore achieve less throughput.
- In general, multiple connections coexist within an overlay network, so there is more than one stream using a single overlay link. In that case, multiple streams will share a single overlay link using only a part of what they could get if each of them used the Internet directly by opening a separate TCP connection. One way to overcome this problem is to open multiple connections between two overlay nodes depending on the number of internal flows using that overlay link. However, we see an overlay network as a single distributed application, no matter how many internal flows it carries; therefore, it should get only one share of the available bandwidth.

Some mechanisms can be deployed in order to limit the internal hop-by-hop throughput to the one obtained by an end-to-end connection that uses the overlay network. Such mechanisms can evaluate the loss rate and round-trip time of a path and adjust the sending rate accordingly, in a way similar to [7]. We believe such mechanisms are not necessary in our case - since we provide end-to-end congestion control, obtaining more throughput is just an effect of pinpointing the congestion and resolving it locally. However, in all the experiments of this paper we choose a conservative approach and limit the sending throughput to values achievable by both end-to-end and hop-by-hop flows, and focus only on the latency of the connections.

3 Simulation Environment and Results

In this section we analyze the multihop reliability behavior using the ns2 simulator [12]. We run a simple end-to-end TCP connection from node A to node F on a network setup as shown in Figure 1, while changing the packet loss rate on link C-D. Since this paper focuses on the latency of reliable connections, we limit the sending throughput to the same value for end-to-end and hop-by-hop flows in order to keep the same network parameters for our latency measurements.

We record the delay of each packet for the different sending rates and packet loss for both end-to-end and hop-by-hop reliability approaches. We define the delay of a packet as the difference between the time the packet was received at the destination, and the time it was initiated by a constant rate sending application. Note that there is a difference between the time a packet is sent by an application and the time that packet is actually put on the network by the reliable protocol (in our case, TCP). If TCP shrinks its window or reaches a timeout, it will not accept or send new packets until it has enough room for them. During this time, the new packets generated by the application will be stored in a buffer owned either by the host operating system or by the application itself. We believe that a delay measurement that is fair to the application would count the time spent by packets in these buffers as well.

The ns2 simulator offers a variety of TCP implementations. Out of these, we used TCP-Fack - TCP with forward acknowledgments - as we believe it resembles a behavior closest to the actual TCP implementation in the Linux Redhat 7.1, that we use in Section 5. The Linux kernel allows adjustment of different TCP parameters (for example, turning off forward acknowledgments would give us a version similar to TCP-SACK), however we opted for leaving the default protocol in the kernel unaltered.

Table 1 shows the average packet delay given by different TCP variations in ns2, as well as the Linux TCP implementation and the Spines link protocol (described in Section 4) when a 500Kbps stream is sent on an end-to-end A-F connection in the network showed in Figure 1, with link C-D experiencing 1% loss. The Redhat 7.1 TCP and the Spines link protocol delays were measured on an emulated network setup described in Section 5.

We compare the performance of the standard end-to-end approach to that of our hop-by-hop approach, where we forward packets reliably on each link, A-B, B-C, ... up to link E-F. For hop-by-hop reliability we use a modified version of TCP-Fack: the initial sender (at node A) adds its original sequence number in an additional packet header, intermediate receivers deliver packets out of order, and the destination delivers packets FIFO according to the original sequence number available in the new header. We did not change the congestion control or the send and acknowledge mechanisms in any way. We verified that our modified TCP and the original TCP-Fack in ns2 behave identically with respect to each packet on a point-to-point connection under different loss rates. All the simulations in this section were run for 5000 seconds, sending 1000 byte messages.

Figure 3 shows that the average delay for a 500 Kbps data stream increases faster with an end-to-end connection while a hop-by-hop flow maintains a low average delay even

Table 1. Average latency for under loss

Protocol	Tahoe	Reno	NewReno	SACK	Fack	Vegas	Redhat 7.1	Spines
Avg. delay (ms)	407.49	217.52	155.76	144.70	84.66	74.07	90.06	117.55

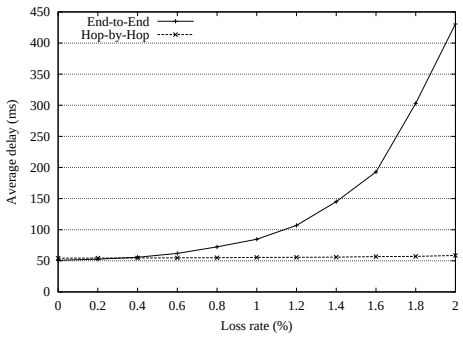

Figure 3. Average delay for a 500 Kbps stream (simulation)

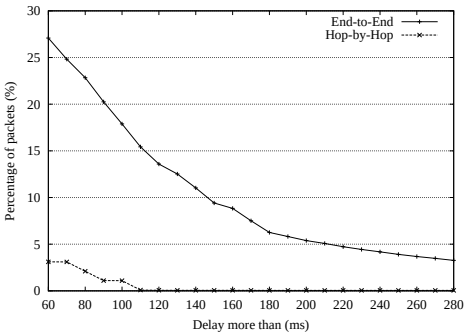

Figure 6. Packet delay distribution for a 500 Kbps stream (simulation)

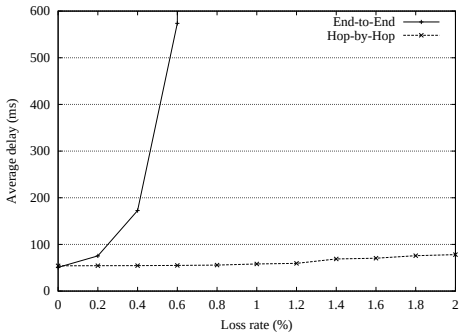

Figure 4. Average delay for a 1000 Kbps stream (simulation)

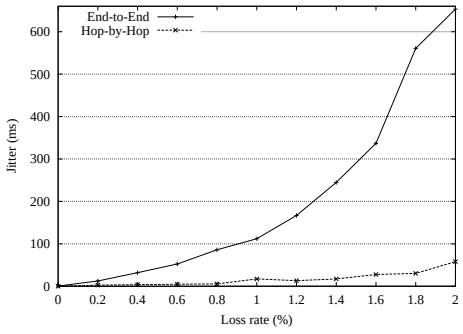

Figure 5. Average jitter for a 500 Kbps stream (simulation)

when it experiences a considerable loss rate. This phenomenon is magnified as the throughput required by the flow increases, as depicted by Figure 4 for a 1000 Kbps data stream.

Jitter is an important aspect of network protocols behavior due to its impact both on other flows at the network level and on the application served by the flow. Figure 5 shows that the jitter of an end-to-end connection is considerably higher and increases faster than the jitter of a hop-by-hop connection for a 500 Kbps stream. We computed the jitter as the standard deviation of the packet delay.

It is interesting to see the distribution of the packet delay for a certain loss rate. In Figure 6, we see that for a 500 Kbps data stream under 1% loss rate, over 27% of the packets are delayed more than 60 milliseconds (including the 50 milliseconds network delay) for an end-to-end connection, while for a hop-by-hop connection only about 3% of the packets are delayed more than 60 milliseconds. Similarly, about 18% of the packets are delayed more than 100 milliseconds by the end-to-end connection, while for a hop-by-hop connection only 1% of the packets are delayed as much. Note that the actual number of packets delayed is much higher than the number of packets lost.

We studied how the performance is affected by the number of intermediate reliable hops in an overlay network. We consider the same network of 50 milliseconds delay, and we measure the percentage of packets that are delayed as we increase the number of intermediate hops from 1 to 10, while keeping the total path latency constant. First, we use two hops of 25 milliseconds each, then three hops of 16.66 milliseconds each, and so forth. Figure 7 shows the per-

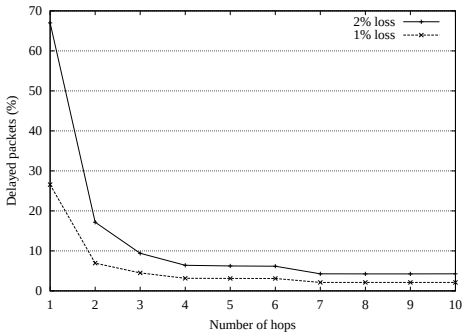

Figure 7. Increasing the number of hops (simulation)

centage of packets delayed more than 60 milliseconds (10 milliseconds more than the path latency) for a 500 Kbps data stream with 1% and 2% packet loss as the number of hops increases. It is interesting to note that two to four hops appear to be sufficient to capture almost all of the benefit associated with hop-by-hop reliability. This is encouraging as small overlay networks are relatively easy to deploy.

The important factor in obtaining better performance with hop-by-hop reliability is the latency of the lossy link rather than the number of hops in the end-to-end connection. The reason for the phenomenon depicted in Figure 7 is that increasing the number of hops from one to two reduces the latency of the lossy link by approximately 50 percent (25 milliseconds in our case), while increasing the number of hops from nine to ten reduces the latency of the lossy link only by approximately 1 percent (0.55 milliseconds).

It is important how well we can isolate a potentially lossy or congested Internet link in an overlay link that is as short as possible. This can be achieved in practice by placing a few overlay nodes such that we create close to equal latency overlay links, as we do not usually know in advance which Internet connections will be congested.

We believe that the simulation results are promising. The reminder of the paper will investigate whether the same behavior is not limited to our simulation environment but is in fact achieved in practice.

4 The Spines Overlay Network

In this section we introduce Spines, an open source research platform that allows the deployment of an overlay network in the Internet. We use Spines to evaluate the hop-by-hop reliability properties in practice.

Spines instantiates overlay nodes on participating computers and creates virtual links between these nodes. Once a message is sent on a Spines overlay network it will be forwarded on the overlay links until it reaches the destination. Many Spines overlays can coexist in the Internet, and even overlap on some of the nodes or links. Both the source and the destination of a connection should be part of the same Spines overlay network.

Spines runs a software daemon on each of the overlay nodes. The daemon acts both as a router, forwarding packets toward other nodes, and as a server, providing network services to client applications.

Clients use a library to connect to a daemon through an API very similar to the Unix Socket interface. A *spines_socket()* call will return a socket, which is actually a TCP/IP connection to the daemon. The application can use that socket to bind, listen, connect, send and receive, using Spines library calls (e.g. a *spines_bind()* call is the equivalent to the regular *bind()*, etc.). The interface is almost transparent, and virtually any socket-based application can be easily adapted to work with Spines. In addition to the TCP-like interface, the Spines API also provides UDP-like functions for unreliable, best effort communication.

The Spines daemon communicates with clients through a Session layer as seen in Figure 8. There is one session for each client connection, and if the client requests a reliable connection, the daemon will instantiate an end-to-end Reliable Session module that will take care of end-to-end reliability, FIFO ordering, and end-to-end congestion control.

An overlay link consists of three logical components.

- An *Unreliable Data Link* sends and receives data packets with no regard to ordering and reliability. It is used for unreliable, best effort, fast communication as it has no buffering other than the ones provided by the operating system.

- A *Reliable Data Link* provides link reliability through a selective repeat protocol and congestion control, but does not provide FIFO ordering. Packets are buffered before being sent on a *Reliable Data Link* only in case the congestion control or available link capacity limit the outgoing bandwidth to a lower value than the incoming throughput. The explicit congestion notification mentioned in Section 2 is based on the size of these buffers. The link congestion control allows the deployment of Spines in the Internet, providing fairness with external TCP traffic. Figure 9 shows the throughput obtained by an end-to-end TCP stream and by the Spines link protocol for a 10 and a 50 millisecond delay link of 10 Mbps capacity under different levels of losses, and compares it to the analytical TCP model from [13]. The throughput achieved by Spines is very close to that of a TCP connection under similar conditions. Note that for a 10 millisecond link, as the throughput of both TCP and Spines approaches the maximum capacity of 10Mbps, they start developing

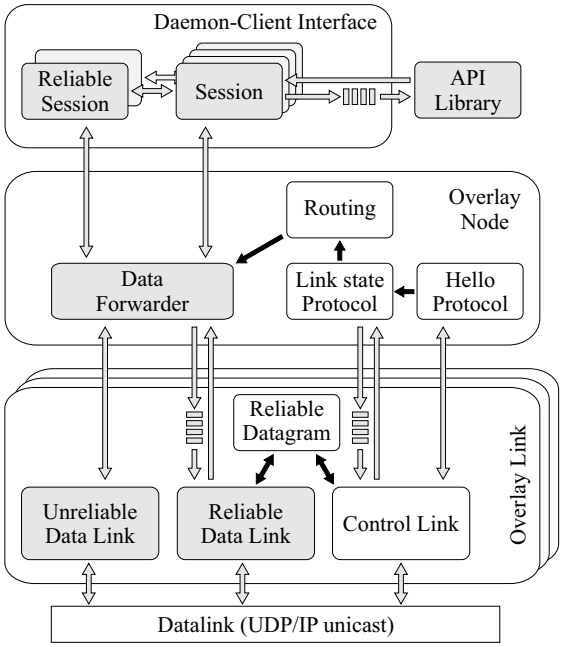

Figure 8. Spines daemon architecture

their own additional losses in order to probe the available bandwidth. This is why they appear to achieve less than the analytical model that takes into account only the original losses we enforced on the link.

- A *Control Link* is used for sending and receiving control information between two neighbor daemons. It provides both reliable and unreliable communication. In case of buffering for the reliable data, the unreliable packets will bypass the buffer and go directly on the network.

The overlay node is responsible for maintaining connections to its neighbors and forwarding data packets either on the overlay links or to its own clients. A *Data Forwarder* parses the header of each message and sends it on the next link or to the daemon-client interface. The *Data Forwarder* allows any combination of reliable and unreliable session and reliable and unreliable link in order to experiment with different forwarding mechanisms. The type of Session and Data Link requested are stamped in the header of each message. For example, one can create a reliable end-to-end session using either unreliable links or reliable links.

Neighboring overlay nodes ping each other periodically using unreliable hello packets. The Spines *Hello Protocol* is responsible for creating, destroying and monitoring overlay links between neighbor daemons. Each Spines daemon sends information about the links to its neighbors through a reliable link state protocol, only when the state of its links change, or periodically at large intervals for garbage collection. The link state protocol provides a complete information about the existing overlay links, out of which a *Routing* module chooses the neighbor providing the shortest path to each destination. The choice of link state routing is purely arbitrary, any other routing protocol could have been used without afecting the hop by hop reliablility mechanisms.

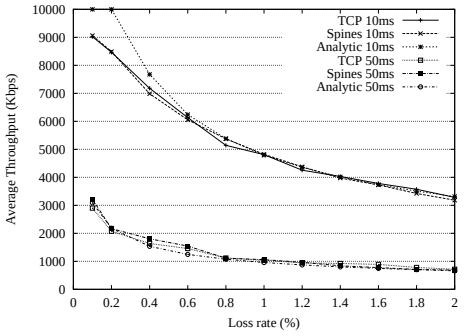

Figure 9. Spines congestion control (Emulab)

In addition to the IP and UDP headers, Spines adds its own headers for routing and reliability. Also, for reliable connections Spines sends acknowledgments for every packet at the level of each link for hop reliability and at least four acknowledgments per end-to-end window for end-to-end reliability and congestion control. When possible acknowledgments are piggybacked with data packets. The control traffic is relatively small, being composed only

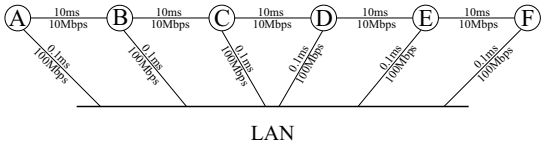

Figure 10. Emulab Network Setup

by hello packets (currently, two 28 byte packets per second on each link) and link state packets that are sent only when the network conditions change. A single link state packet can contain information about up to 90 links, depending on the dispersion of the network. Due to this overhead, our experiments show that when compared with a standard TCP connection running alone on a network link with capacity ranging from 500 Kbps to 100 Mbps, the Spines link protocol achieves about 3.5% less data throughput, and the end-to-end connection that uses both levels of reliability and congestion control (on the hop and end-to-end) shows an overhead of at most 5.7%. The best effort, unreliable protocol in Spines has an overhead of about 2.3%.

5 Experimental results

In this section we evaluate the hop-by-hop reliability behavior using the Spines overlay network deployed on the Emulab testbed. Emulab [1] [5] is a network facility that allows real instantiation in a hardware network (composed of actual computers and network switches) of a given topology, simply by using an ns script in the configuration setup. Link latencies, loss rates and bandwidths are emulated with additional nodes that delay packets or drop them according to specified link characteristics.

We instantiated on Emulab the network setup presented in Figure 10 that follows the topology used in our Section 3 simulations. In addition to the five links A-B, B-C,... E-F we also connected the nodes through a fast, local area network that was used to obtain accurate clock measurements between the overlay nodes.

The routing was set up such that all the experiment traffic went on the 10 millisecond links, while on the local area network we continuously measured (every 100 milliseconds) the clock difference between the computers making the end nodes of a connection. The one-way delay of the data packets was calculated as the difference between the timestamp at the sender and the current time at the receiver, adjusted with the clock difference between the end nodes.

On the overlay network, the round-trip delay between nodes A and F measured with ping under no traffic was 99.96 milliseconds, and the throughput achieved by a TCP connection on each of the 10 millisecond links was about 9.59 Mbps. On the local area network the round-trip delay between any two nodes was about 0.135 milliseconds, which gave us a very good accuracy in measuring the clock difference and one-way delay of the packets. For each experiment in this section we sent 200000 messages of 1000 bytes each.

[1]The Utah Network Emulation Testbed (www.emulab.net) is primarily supported by NSF grant ANI-00-82493 and Cisco Systems

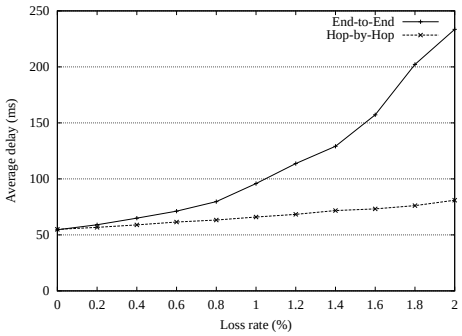

Figure 11. Average delay for a 500 Kbps stream (Emulab)

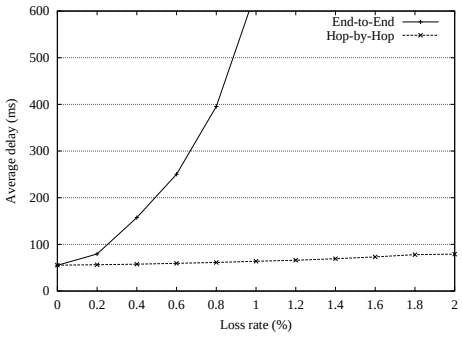

Figure 12. Average delay for a 1000 Kbps stream (Emulab)

We compared the packet delay of a data stream using an end-to-end TCP connection between nodes A and F, with that of a hop-by-hop connection using Spines on the overlay nodes, while varying the sending rate (at node A) and the loss rate on the intermediate link C-D. Note that the end-to-end TCP connection does not go through the Spines application-level routers, but only through the overlay nodes A, B, ... F - so it is not affected in any way by the Spines overhead in user-level processing and added headers.

Figure 11 and Figure 12 show that the low latency effect of hop-by-hop reliability is very signifficant also in the experimental setting, overcoming by far the overhead of user-level processing at the level of the intermediate overlay network nodes. The latency of a real TCP connection is lower than the simulation result (presented in Figure 3

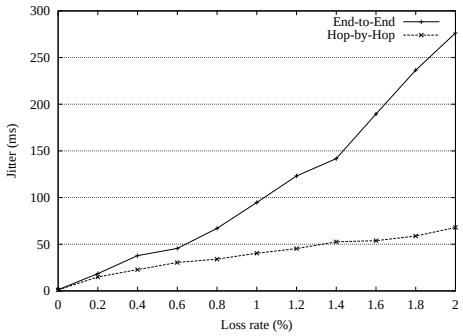

Figure 13. Average jitter for a 500 Kbps stream (Emulab)

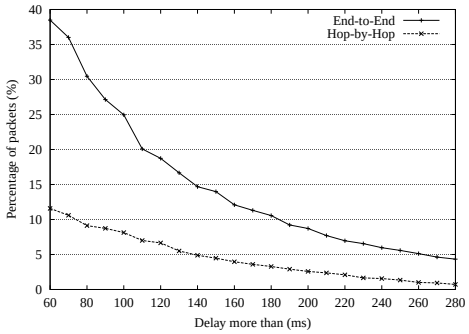

Figure 14. Packet delay distributuin for a 500 Kbps stream (Emulab)

and Figure 4), especially at high loss rates, which shows us that the TCP model we used in the simulation (TCP-Fack), even though the closest, does not resemble exactly the Linux kernel implementation. The latency achieved by Spines hop-by-hop reliability is slightly higher than the latency obtained in the simulator, mainly due to simplifying assumptions of the simulation. However, the hop-by-hop latency remains very low, and increases much slower compared to the latency of the end-to-end TCP connection.

Jitter follows a similar pattern, as seen in Figure 13 (and compared with Figure 5). Packets sent through the Spines overlay network arrive at the destination with a jittter up to three to four times smaller than the jitter of an end-to-end connection. In Figure 14, although the delay distribution for the end-to-end TCP connection is almost identical to the result of the simulation (Figure 6), the overhead of the application-level routing is clearly visible in the hop-by-hop delay distribution. However, even with this overhead, the number of packets delayed by Spines is significantly (more than three times) lower than the number of packets delayed by the end-to-end connection.

6 Related Work

The idea of using reliable intermediate links is not new. In 1976 the International Committee for Telegraph and Telephony (CCITT) recommended X.25 as a store-and-forward connection oriented protocol between end-nodes (DTE) and routers (DCE). In [14], the authors give a detailed description of the X.25 protocol. However, since the Internet was developed as a conectionless, best-effort network (which allows better scalability and interoperability), it did not incorporate the X.25 specifications, but relied on end-to-end protocols such as TCP to provide reliable connections.

One of the early uses of overlay networks in the Internet was in a proposed overlay network called EON (Experimental OSI-based Network) [10] on top the IP network, that would allow experimentation with the OSI network layer. The scheme was only experimental and did not specify hop-by-hop reliability. More recently, overlay networks emerged mainly by providing new services to the application. The Mbone [6] is a routing mechanism that creates an overlay infrastructure over the global Internet and extends the use of IP multicast by creating virtual tunnels between the networks that support native IP multicast. The Mbone facilitates the use of multicast services on the global Internet but does not provide reliability by itself.

TRAM [4] is a tree-based reliable multicast protocol that uses repair trees to localize recoveries, and aggregates end-to-end acknowlegenents at intermediate nodes. TRAM was designed specifically for single-source multicast. If applied to multiple flows (unicast or multicast), TRAM requires intermediate nodes to keep packet-based state for each end-to-end session in order to provide end-to-end reliability and congestion control. Since we use two completely separated levels of reliability (hop-by-hop and end-to-end) our approach allows an unlimited number of reliable sessions, as per flow information is only handled at the end nodes. SRM [8] provides a form of localized recovery for reliable multicast by using randomized timeouts for sending retransmission requests and the retransmissions themselves. SRM does not guarantee recovery from the nearest node, as the closest one may set its timeout to be higher than that of an upstream node. Its probabilistic algorithm allows for double retransmission requests and recovery messages to be sent. The Spread system [1] uses a network of daemons to provide wide area group communication, where missed messages are recovered from the nearest daemon on the path, localizing message recovery in a way similar to ours. The system is confined to group communication and does not provide a generic service such as ours.

Yoid [9] is a set of protocols that allows host-based content distribution using unicast tunnels and, where available, IP multicast. Yoid has the option of using TCP as the link

protocol on the overlay network, but does not guarantee either end-to-end congestion control or end-to-end reliability. In addition to these guarantees, our approach uses an out of order forwarding mechanism that provides less burstiness at the network level, and lower packet latency and jitter.

The X-Bone [17] is a system that uses a graphical user interface for automatic configuration of IP-based overlay networks. RON [2] creates a fully connected graph between several nodes, monitors the connectivity between them, and, in case of Internet route failures, re-directs packets through alternate overlay nodes. Both X-Bone, and RON are implemented at the IP level, do not provide reliability other than the regular end-to-end offered by TCP, and are complementary to our work.

7 Conclusion

This paper presented a hop-by-hop reliability approach that considerably reduces the latency and jitter of reliable connections in overlay networks. We first quantified these effects in simulation.

Overlay networks pay a performance price due to the need to process each message at the application level, and to maintain the overlay. The paper presented experimental results with a new overlay network software we have built. These results resemble the simulation results and show that the overhead associated with overlay network processing does not play an important factor compared with the considerable gain of the approach. We also learned that having a small number of approximately equal hops (two to four) is sufficient to capture most of the performance benefit.

While network bandwidth increases exponentially over time, latency is very slow to improve. This work shows how coupling cheap processing and memory with the programmable platform provided by overlay networks and paying a small price in throughput overhead, can considerably improve the latency characteristics of reliable connections.

Acknowledgment

The authors would like to thank Mike Dahlin for insightful comments and discussions.

This work was partially funded by DARPA grant F30602-00-2-0550 to Johns Hopkins University.

References

[1] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceeding of International Conference on Dependable Systems and Networks*, pages 327–336. IEEE Computer Society Press, Los Alamitos, CA, June 2000.

[2] D. G. Andersen, H. Balakrishnan, and M. F. K. R. Morris. Resilient overlay networks. In *Operating Systems Review*, pages 131–145, December 2001.

[3] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partrige, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroklawski, and L. Zhang. Recommendations on queue management and congestion avoidance in the internet. RFC 2309, April 1998.

[4] D. M. Chiu, M. Kadanski, J. Provino, J. Wesley, H.-P. Bischof, and H. Zhu. A congestion control algorithm for tree-based reliable multicast protocols. In *Proceeding of IEEE Infocom*, pages 1209–1217, June 2002.

[5] The Utah network emulation facility. http://www.emulab.net/.

[6] H. Eriksson. Mbone: the multicast backbone. In *Communications of the ACM*, volume 37, pages 54–60, August 1994.

[7] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *ACM Computer Communications Review: Proceedings of SIGCOMM 2000*, volume 30, pages 43–56, Stockholm, Sweden, August 2000.

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

[9] P. Francis. Yoid: Extending the internet multicast architecture. http://www.icir.org/yoid/docs/yoidArch.ps, April 2000.

[10] R. Hagens, N. Hall, and M. Rose. Use of the internet as a subnetwork for experimentation with the osi network layer. RFC 1070, February 1989.

[11] V. Jacobson. Congestion avoidance and control. *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, 18, 4:314–329, 1988.

[12] ns2 network simulator. Available at http://www.isi.edu/nsnam/ns/.

[13] J. Padhye, V. Firoiu, D. Towsley, and J. Krusoe. Modeling TCP throughput: A simple model and its empirical validation. In *ACM Computer Communications Review: Proceedings of SIGCOMM 1998*, pages 303–314, Vancouver, CA, 1998.

[14] R. Perlman. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*. Addison- Wesley Professional Computing Series, second edition, 1999.

[15] K. K. Ramakrishnan and S. Floyd. A proposal to add explicit congestion notification (ECN) to IP. RFC 2481, January 1999.

[16] The Spines overlay network. http://www.spines.org/.

[17] J. Touch and S. Hotz. X-bone: a system for automatic network overlay deployment. *Third Global Internet Mini Conference in conjunction with Globecom98*, November 1998.

N-Way Fail-Over Infrastructure for Reliable Servers and Routers

Yair Amir Ryan Caudy Ashima Munjal Theo Schlossnagle Ciprian Tutu

Johns Hopkins University
Computer Science Department
{yairamir, wyvern, munjal, theos, ciprian}@cnds.jhu.edu

Abstract

Maintaining the availability of critical servers and routers is an important concern for many organizations. At the lowest level, IP addresses represent the global namespace by which services are accessible on the Internet.

We introduce Wackamole, a completely distributed software solution based on a provably correct algorithm that negotiates the assignment of IP addresses among the currently available servers upon detection of faults. This reallocation ensures that at any given time any public IP address of the server cluster is covered exactly once, as long as at least one physical server survives the network fault. The same technique is extended to support highly available routers.

The paper presents the design considerations, algorithm specification and correctness proof, discusses the practical usage for server clusters and for routers, and evaluates the performance of the system.

1 Introduction

Maintaining the availability of critical network servers is an important concern for many organizations. Server redundancy is the traditional approach to provide availability in the presence of failures. From the client perspective, a network-accessible service is resolved via a set of public IP addresses specified for this service. Therefore, the continued availability of a service via these IP addresses is a prerequisite for providing uninterrupted service to the client. In order to function correctly, each of the service's public IP addresses has to be covered by exactly one physical server at any given time. If no physical server covers a public IP address, the clients will not receive any service. On the other hand, if more than one physical server is covering the same IP address at any time, the network might not function properly and clients may not be served correctly.

A sizable market exists for hardware solutions that maintain the availability of IP addresses, usually via a gateway that hides the actual servers behind a smart switch or router in a centralized manner. We present Wackamole, a high availability tool for clusters of servers. Wackamole ensures that all the public IP addresses of a service are available to its clients. Wackamole is a completely distributed software solution based on a provably correct algorithm that negotiates the assignment of IP addresses among the available servers upon detection of faults and recoveries, and provides N-way fail-over, so that any one of a number of servers can cover for any other.

Using a simple algorithm that utilizes strong group communication semantics, Wackamole demonstrates the application of group communication to address a critical availability problem at the core of the system, even in the presence of cascading network or server faults and recoveries. We also demonstrate how the same architecture is extended to provide a similar service for highly-available routers.

The remainder of this paper is organized as follows. Section 2 introduces the system architecture. Section 3 describes the system model and the core algorithm behind the engine of Wackamole and discusses its correctnes. Section 5 analyzes practical considerations and presents two applications for the system. Section 6 presents performance results concerning the reconfiguration time of Wackamole clusters. We discuss related work in Section 7 and conclude in Section 8.

2 System Overview

Our solution has three main components, presented in Figure 1:

- An IP address control (acquire and release) mechanism.

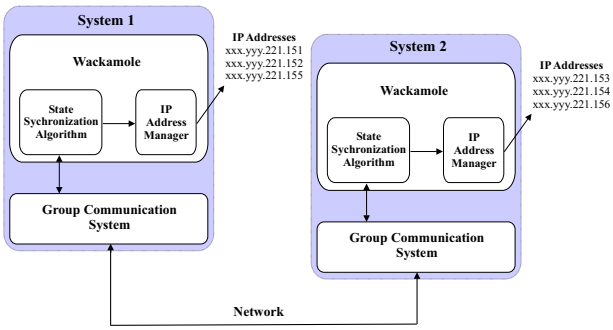

Figure 1. Wackamole Architecture

- A state synchronization algorithm (the Wackamole Algorithm).
- A membership service provided by a group communication toolkit.

The group communication toolkit maintains a membership service among the currently connected servers and notifies the synchronization algorithm of any view changes that occur due to server crashes and recoveries, or network partitions and remerges.

The synchronization algorithm manages the logical assignment of virtual IP addresses among the currently connected members, avoiding conflicts that can occur upon merges and recoveries and covering the "holes" that can arise as a result of a crash or partition.

The IP address control mechanism enforces the decisions of the synchronization algorithm by acquiring and releasing the IP addresses accordingly. These mechanisms are highly specific to the operating system on which the Wackamole system runs.

The correctness of the system is dependent on the assumption that the group communication system provides an accurate view of the current network connectivity. If there is additional connectivity beyond that reported by the group communication system, there may be conflicts in the assignment of IP addresses. On the other hand, if the group communication system does not detect the disconnection of a server from the current membership in a timely manner, the IP addresses that were covered by that server will be unavailable to the clients, since the system will not reconfigure without the proper notification.

3 The Wackamole Algorithm

In this section we present the state synchronization algorithm that forms the core of the Wackamole system and discuss its correctness, given the assumption that the membership notifications issued by the group communication system reflect the actual network status.

3.1 System Model

In order to formally identify the problem that Wackamole attempts to solve, we define the system model and introduce the correctness properties that the algorithm and the implemented system need to maintain.

We consider a set S=$\{s_1, s_2, ..., s_m\}$ of servers that provide service to outside client applications. The servers are all located in the same Local Area Network (LAN), but are susceptible to crashes and temporary network partitions[1] . During a network partition, the servers are separated into two or more components that are unable to communicate with each other.

The client applications access the services through IP addresses in the set I=$\{i_1, i_2, ..., i_n\}$. The servers in S are responsible for covering the set I of *virtual* IP addresses. We refer to the IP addresses in I as *virtual* in order to distinguish them from the stationary default IP addresses, that do not change, used by the servers for intercommunication.

The client applications are oblivious to the stationary IP addresses of the servers in S or to the possible partitioning that may exist among the servers.

In order to guarantee correct service, the following properties need to be maintained.

Property 1 (Correctness) *Every IP address in the set I is covered exactly once by a server in each subset S_k, where S_k is a maximal connected component whose servers are in the operational (RUN) state.*

Property 2 (Liveness) *If there is a time t from which a set of connected servers does not experience any crashes/recoveries or network partitions/merges, the servers will switch to the operational (RUN) state.*

In order to guarantee these properties we rely on the group communication system to follow the Virtual Synchrony properties [6, 14] in partitionable systems and to provide Agreed message delivery. The Virtual Synchrony property specifies that any two servers that advance *together* from one membership to the next one, will deliver an identical set of messages in the first membership. The Agreed delivery property guarantees that additionally, the messages will be delivered in the same order at all servers. Furthermore we assume that the group communication system provides a membership service that provides each server in the group a uniquely ordered list of the currently connected participants.

3.2 Algorithm Specification

The algorithm runs according to the state machine presented in Figure 2.

[1]Partitions can occur even in LAN environments due, for instance, to a switch failure in one of the subnetworks.

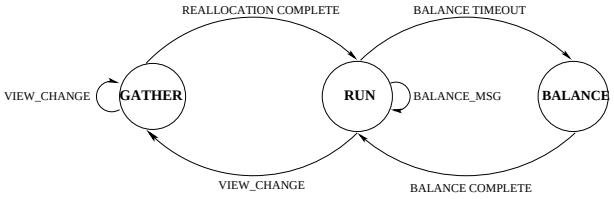

Figure 2. Wackamole Algorithm

Each server maintains a table *current_table* that contains the virtual IP allocation during the current membership. During normal operation, the algorithm is in the RUN state. In this state, each server is responsible for a set of virtual IP addresses and will answer all the requests directed to those IP addresses. While in the RUN state, the *current_table* information is conflict-free and the complete IP set is covered, maintaining the correctness guarantees of the algorithm. When the group communication system delivers a VIEW_CHANGE event, a backup of the IP table is created and a STATE_MSG is sent to every member of the new view containing the information about the IP addresses managed by the server and the identifier of the view in which it is initiated. The algorithm then moves to the GATHER state.

Algorithm 1 RUN State

```
1: when: VIEW_CHANGE do
2:     old_table = current_table
3:     send STATE_MSG
4:     state = GATHER
5: when: receive BALANCE_MSG do
6:     Change_IPs()
```

In the GATHER state each server incorporates the information received through the STATE_MSGs in its *current_table* variable and checks for the existence of conflicts in the IP allocation; if members from previously partitioned components are merged together, conflicts are expected since each component covers the full IP address set. ResolveConflicts() is a deterministic procedure invoked as soon as a new STATE_MSG is received, that checks whether the server that sent this message introduces any conflict with respect to the information already gathered about the set of covered IP addresses. If a conflict is detected, the server drops the addresses that are overlapping, thus restoring consistency at the network level as soon as possible.

When all the state messages (STATE_MSG) have been received, each server invokes a deterministic procedure Reallocate_IPs(). During the Reallocate_IPs() procedure, the servers make sure that all the virtual IPs are covered by a server in the current configuration. In particular, the procedure relies on the uniquely ordered membership list provided by the group communication system to distributely decide which server covers which IP address.

If the GATHER state is interrupted by a cascading VIEW_CHANGE event, the server clears its *current_table*, discarding the information already collected and reverting to the information in the *old_table*, then sends a new STATE_MSG to all members of the new configuration.

Algorithm 2 GATHER State

```
1: when: receive STATE_MSG with current view id do
2:     update current_table
3:     ResolveConflicts()
4:     if (received STATE_MSG from all cur-
       rent_table.members) then
5:         Reallocate_IPs()
6:         state = RUN
7: when: VIEW_CHANGE do
8:     clear current_table
9:     send STATE_MSG
10: when: BALANCE_MSG do
11:    ignore
```

3.3 Correctness of the Algorithm

We consider a subset of servers S'$<$ S that are in the operational RUN state. Between the servers in S', each virtual IP address is covered exactly once. We consider a VIEW_CHANGE event that is detected by members of S'. According to the group communication guarantees, all members of S' will receive VIEW_CHANGE notifications, even though they are possibly disconnected from each other. Following the algorithm, each server will proceed to the GATHER state and send a state message containing its local knowledge base. Let's consider a server *s* that was part of S' and is now part of S" as indicated by the group communication. The group communication system provides every *s* in S" an identically ordered list of all the servers in S".

Lemma 1 *For every connected set S' of servers in the RUN state, every IP address is covered at most once by a server in S'.*

Proof:

We consider a set of servers that are connected after a view change event, as indicated by the group communication notification. In order for the servers to advance to the RUN state, the state transfer algorithm, executed in the GATHER state, needs to complete. Therefore we consider the situation where a set of servers S' does not detect further VIEW_CHANGE events until they exit the GATHER state.

Let's assume that upon receiving the last STATE message in the GATHER state (line 4 in Algorithm 2) there

exists a virtual IP address *vip* that is covered by two servers *p* and *q* in S'. According to the Virtual Synchrony and Agreed delivery guarantees of the group communication, both *p* and *q* received all the state messages that were sent during the GATHER state, therefore they received their own state messages. According to the management of the *current_table* variable from the algorithm (line 2 Algorithm 1 and lines 2,8 Algorithm 2) and the fact that only STATE_MSGs generated in the current view are considered (line 1 of Algorithm 2), the variable will accurately reflect at this point the state of the currently connected component. During this stage, following the algorithm, the servers don't acquire new IP addresses, therefore both *p* and *q* were already covering *vip* from their previous memberships. From the algorithm, in the Resolve_Conflicts() procedure (line 3 in Algorithm 2), when *p* receives the state message from *q*, it will notice the conflict in the coverage of *vip* and will adjust its IP coverage table and release *vip* if *p* appears in the membership list of S' before *q*. The same reasoning applies for *q*; therefore it is impossible for *vip* to be covered by both *p* and *q* at this point. Furthermore, both *p* and *q* will have the same view of the virtual ip coverage. Note that reaching agreement does not assume any particular relation between the initial states of *p* and *q* or of the other members of S'.

When all the state messages have been received each server will execute the Reallocate_IPs() procedure. During this procedure a server may acquire new IP addresses only if there is a virtual IP that is not covered by any server in S'. Since all the servers have the same view of the coverage table, they will all detect the same set of IP addresses that need to be covered. Furthermore, since they all have the same uniquely ordered list of the membership of S' the procedure Reallocate_IPs() will guarantee that each unallocated virtual IP address will be covered by exactly one server in S'. This concludes the proof of the lemma.

□

Lemma 2 *During the RUN state, every virtual IP address in the set R is covered by at least one server.*

Proof:

According to the algorithm, after a view-change, if the connectivity remains stable allowing the GATHER procedure to complete, all the connected servers will execute the Reallocate_IPs() procedure. As shown above, all servers that start this procedure in the same component will have identical views of the IP coverage and will detect the same "holes" (IP addresses that are not covered by any server in the current component). Following the algorithm, these IP's are covered at the end of the Reallocate_IPs() procedure, ensuring the complete coverage during the RUN state.

□

From the two lemmas above, we obtain the correctness property as specified in section 3.1.

We will now prove the liveness property.

Proof:

Due to the properties of the group communication delivery specification, if there is a time t from which no view-change notifications occur, then every server is guaranteed to deliver all the state messages that were sent in that membership. At that time, each server will execute the finite procedure Reallocate_IPs() and will switch to the RUN state.

□

3.4 Practical Considerations

The algorithm presented so far satisfies the correctness guarantees but can be further optimized in order to improve its performance.

From a practical perspective we want to minimize the amount of time that an IP address is covered by two or more servers in the same connected component in order to avoid network level conflicts. This is ensured by the fact that the ResolveConflicts() procedure is invoked as soon as a conflict is detected and one of the involved parties will drop the offending IP.

Algorithm 3 BALANCE State

```
1: Balance_IPs()
2: send BALANCE_MSG
3: state = RUN
4: when: VIEW_CHANGE or BALANCE_MSG or
   STATE_MSG do
5:     delay event
```

Of similar importance to the system is the fast completion of reconciliation during the GATHER state. The minimal task that needs to be executed in the Reallocate_IPs() procedure is the acquisition of non-allocated IP addresses in order to guarantee the complete coverage. However, after several partitions/merges, the system may end up with a very unbalanced allocation of IP addresses among the set of connected servers. To avoid this we can modify the Reallocate_IPs() procedure to perform load-based reallocation of IP addresses. However, this would extend the time the system is in a non-operational state. We introduce a re-balancing procedure which is triggered from the RUN state by a set timeout and is executed only by one member (representative) of the connected component, selected based on the order in the membership list provided by the group communication system. The representative decides on the new IP allocation based on load balancing considerations and explicit preferences specified by each server at startup and passed along through state messages. The representative broadcasts a BALANCE_MSG

containing the new IP allocation and switches back to the RUN state. Upon receiving a BALANCE_MSG, a server in the RUN state acquires or releases the necessary IP addresses. Note that the BALANCE state executes as an atomic procedure with the server ignoring any potential VIEW_CHANGE notification from the group communication until it returns to the RUN state. Furthermore, even when a VIEW_CHANGE is detected before all servers receive and apply the BALANCE_MSG the correctness of the algorithm is not endangered since the GATHER procedure does not assume anything about the starting state of the participating servers and treats any conflict as it is discovered.

Another optimization was added in order to gracefully bootstrap the system. A server *s* starts with the local variable *mature* unset and without being responsible for any IP addresses. Upon receiving a view change notification, *s* switches to the GATHER state. If during the GATHER state *s* receives a state message from a *mature* server, it will mark itself as mature and continue the normal algorithm execution. If all the servers that *s* can contact are not mature, *s* will remain "immature" until a certain timeout expires after which it automatically sets itself as mature, notifies the other servers, and starts managing the IP addresses. The reason for this optimization is to avoid quick IP reallocations as the cluster is rebooted.

4 Implementation

Wackamole [23] has been implemented with cross-platform interoperability in mind; it currently supports FreeBSD, Linux, and Solaris systems. To more readily accommodate its use on multiple platforms, the implementation is separated into two clearly delineated parts. The first, comprised of generic ANSI C code, implements the core algorithm presented above. The second, which contains platform-specific code, implements the functionality needed to manage multiple interfaces and spoof ARP caches on each supported operating system.

4.1 The Spread Toolkit

The correctness as well as the efficiency of the system depends on the use of a group communication system that provides reliable, totally ordered multicast and group membership notifications for a cluster of servers. Wackamole was implemented using the Spread group communication toolkit [20, 3].

Spread is a general-purpose group communication system for wide- and local-area networks. It provides reliable and ordered delivery of messages (FIFO, causal, agreed ordering) as well as Virtual Synchrony and Extended Virtual Synchrony membership services. These properties match the algorithm requirements specified in Section 3.1

Spread uses a client-daemon architecture. Node crashes/recoveries and network partitions/remerges are detected by Spread at the daemon level; upon detecting such an event, the Spread daemons install the new daemon membership and inform their clients of the corresponding changes in the group membership that are introduced by the failure. Clients are also notified when changes in the group membership are triggered by a graceful leave or join of any client. The Spread toolkit is optimized to support the latter situation without triggering a full daemon membership reconfiguration, but rather informing only the participating group about the new group membership. The impact of this optimized approach will become apparent in section 6.

The Spread toolkit is publicly available and is being used by several organizations in both research and production settings. It supports cross-platform applications and has been ported to several Unix platforms as well as to Windows and Java environments.

4.2 Implementation Considerations

Wackamole's state synchronization algorithm is implemented using group membership and messaging services offered by the Spread Toolkit. Immediately upon startup, the Wackamole daemon connects to a Spread daemon running on the same host and joins the wackamole group. It then relies on the regular membership messages sent by Spread to determine the current set of available hosts, and to initiate state transfer upon view-change detection. Spread is also used to ensure that messages are sent in a total order among Wackamole daemons, that old messages which must be discarded upon receipt can be identified properly, and that endian conflicts across platforms are handled correctly.

As a consequence of Wackamole's tightly-coupled relationship with Spread, some of the fine-tuning decisions that can be made to improve Wackamole's response time to network events are dependent on the way Spread is configured. Modifying the Spread network-failure probing timeouts must be, however, done on a system-specific basis. If not done properly, this tuning can be detrimental to the performance of a Wackamole cluster by increasing the number of false-positive network failures. The impact of this tuning is analyzed in Section 6.

A Wackamole daemon that becomes disconnected from Spread will drop all of its virtual interfaces and enter a cycle in which it periodically attempts to reconnect to Spread, because it cannot ensure correctness without the services Spread provides. This behavior allows clusters to survive changes to the Spread daemons with which they communicate, such as version changes and re-

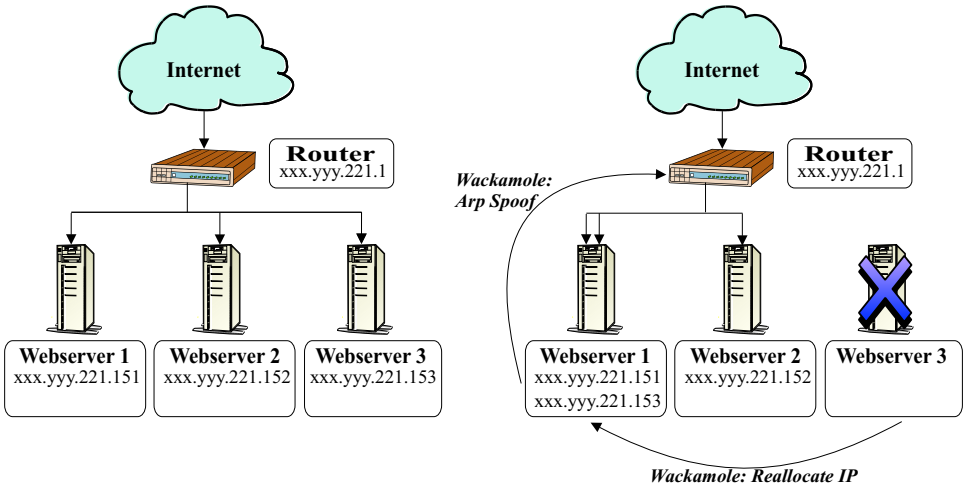

Figure 3. N-Way Fail-Over for Web Clusters: The IP of the failed server is reassigned to one of the available servers and the router is informed of the ARP change.

initializations for configuration modification, taking into account the fact that Spread may be used for multiple applications concurrently.

In order to provide continuous service to the Wackamole daemons, Spread must bind to IP addresses that are not subject to Wackamole's management. Consequently, it is possible (although not required) to run Spread on a separate Network Interface Card (NIC) than the one being used for the virtual IP addresses managed by Wackamole. Also, Wackamole does not provide failure detection of any of the applications that may be relying on its management, e.g. HTTP servers. Either of these two situations can cause failures that are not detectable by the Spread membership service. This problem is not directly addressed by Wackamole's implementation, but a possible solution is to perform run-time checks on the availability of the NIC or of the specific applications that use Wackamole, and trigger the virtual IP migration when a failure is detected.

Another practical aspect of the Wackamole implementation is the addition of an input channel to allow administrative control of a cluster's behavior. Also, the way Wackamole handles network failures can be modified, such that all decisions are made by a deterministically chosen representative and imposed upon the other daemons, rather than made independently by each daemon through a deterministic decision process. This will enable changing the way virtual address allocation decisions are made without breaking version compatibility.

5 Practical Applications

The two primary applications for which Wackamole was developed are clusters and fail-over routers. The implementation of Wackamole takes these applications into account and can be fine-tuned to make appropriate tradeoffs in either situation. We show how Wackamole provides availability for these applications.

5.1 N-Way Fail-over for Clusters

Web clustering is the application that drove the creation of Wackamole. In combination with Domain Name Service (DNS), Wackamole provides the functionality to enable websites served by multiple IP addresses and/or hosted on a cluster of machines to be highly available. The generic management of virtual addresses has already been discussed. However, this class of application requires Wackamole to perform an additional task: ARP spoofing.

While IP addresses are used for routing on wide area networks, on local area networks Media Access Control (MAC) addresses are used. An IP address is resolved to a MAC address using the Address Resolution Protocol (ARP). In and of itself, this is not a problem for Wackamole. However, ARP data is cached on an IP address basis. This cache must be updated for any virtual address that is moved from one host to another, on each host that has cached an <IP address, MAC address> pair for that virtual address.

Since we assume that we are managing a local area cluster, all requests to the server must come through a router. That router's ARP cache must be updated in order to ensure that it correctly forwards packets to the appropriate machine whenever Wackamole alters the allocation of virtual addresses within the cluster. Consequently, part of Wackamole's platform-specific code deals with spoofing of ARP reply packets to force updates to the router ARP cache.

An example layout for a Wackamole-assisted web cluster (Figure 3) consists of a number of web servers and

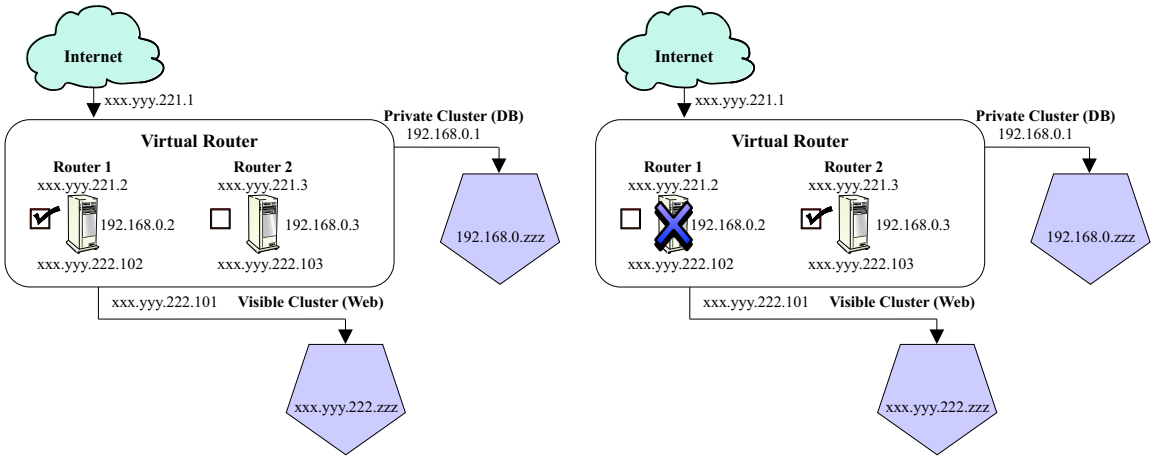

Figure 4. N-Way Fail-Over for Routers: At any point, a single physical router acts as the virtual router, managing the virtual addresses xxx.yyy.221.1, xxx.yyy.222.101, 192.168.0.1.

a single router through which outside requests are made. Each of the web servers must be running a Spread daemon, likely on a private IP address, and must be running a Wackamole daemon, to ensure that virtual IP addresses are correctly allocated. Each server must also be responsible for notifying the router to update its ARP cache when it assumes responsibility for a new virtual address.

5.2 N-Way Fail-Over for Routers

Router management is another application that has emerged as a common use for Wackamole. An example layout for this application of Wackamole consists of multiple physical routers that act as a single virtual router as depicted in Figure 4. An indivisible set of virtual addresses on different interfaces is allocated to the physical router currently acting as the fail-over router. In the figure, these IP's are xxx.yyy.222.101, 192.168.0.1, and xxx.yyy.221.1 which represent the logical IP addresses of the router in the three networks that it serves. The picture also shows the stationary IP addresses of each physical router, on each of the three networks. These IPs are depicted in the figure inside the Virtual Router box.

If the interface through which the machine is connected to Spread fails, or the machine itself crashes, the set of virtual IP addresses will be reallocated to a different machine. The set of physical routers running Wackamole, each of which is potentially "the" router, can be conceptualized as a single virtual router.

For the most part, the presented Wackamole architecture can support this application without additional modifications beyond what is needed for web clustering. However, a router needs to simultaneously exist on multiple networks in order to route packets between said networks. Therefore a set of virtual IP addresses must be considered as a single entity. As a result, Wackamole was modified to support grouping of multiple IP addresses, possibly on different interfaces, as an indivisible set of virtual addresses. This enables the correct handling of situations where a single host being managed by Wackamole must be accessible on multiple virtual addresses.

Furthermore, the notification mechanism for ARP spoofing must be enhanced to update the ARP cache of every host which has resolved the MAC address of the virtual router. To facilitate the necessary notification, each Wackamole daemon periodically sends data from its ARP cache to all other daemons. This makes it possible for a daemon to approximately know the set of machines that must be notified when it assumes responsibility for a virtual IP address. Obviously, this approach does not scale well to very large LANs. We are investigating the potential of applying garbage collection techniques to make the ARP spoof notification more accurately targeted towards hosts that require such notification.

The method described above incurs additional delays when the router is using any dynamic routing protocol such as OSPF [15] or RIP [18] on one or more of its interfaces. The fail-over router in such a case needs to be updated with the current state of the relevant dynamic routing tables before it is able to route messages properly. This usually takes around 30 seconds. A different setup can be used to avoid this delay. In this alternate setup, all the participating fail-over routers act as separate entities in the dynamic routing protocol and all advertise the same internal networks to the external dynamic networks. Therefore, all of the fail-over routers are continuously updated with route changes. On the internal network only one of the fail-over routers actively routes, and Wackamole will ensure that its IP address is always covered by one of the fail-over routers.

Using this setup, a failure of any of the routers will only cause a minor service interruption, noticeable only

by the fraction of the external routing queries that are directed to the failed router. All routing from the internal network will not be affected unless the designated router fails. In this case, Wackamole reassigns another router to control the virtual router address and the hand-off is complete as soon as Wackamole reconfigures without additional need to transfer routing information. Essentially, this setup is closer to the setup described in Section .

Our solution, in both scenarios, provides the additional benefit of allowing a heterogenous set of physical routers to collaborate in forming a virtual router. Using a variety of architectures and operating systems for the routers provides increased protection of the virtual router against security exploits that may target specific platforms.

6 Performance Results

In order to assess the performance of Wackamole, the most relevant measurement is the length of the service interruption perceived by a client when the server with which it is communicating is made unavailable by a fault. For this reason, we report results for the average availability interruption time when a computer running Wackamole fails and its virtual addresses must be reallocated to another computer, as measured from a client.

In our experiment we place a simple server process on each computer using Wackamole. The server responds to UDP packets by sending a packet containing its hostname. A client process on another computer is instructed to continuously access a specific virtual address by sending UDP request packets at a specified interval, and record the hostname of the server that responds as well as the time since the last response was received. For our experiments, we used a 10ms interval between requests. The value is the smallest that can be practically used, and is determined by the linux context-switch times. When a fault is induced by disconnecting the interface through which Spread, Wackamole, and the experimental server access the network, the client will stop receiving responses to its requests. When Wackamole completes the IP address reallocation procedure and the client's ARP cache is updated, the client resumes receiving responses to its queries, this time from the computer that has aquired its target address. The time elapsed between the receipt of the last response from the disabled computer and the first response from the new server is the *availability interruption time* from the experimental client's point of view. While there is a small possibility for error in this measurement due to the interval between requests and fluctuations in the network, our measurements represent an upper bound on the actual interruption time.

As discussed, Wackamole depends upon the Spread group communication toolkit for notification of membership changes. For this reason, the availability interruption time measures the total time to complete four actions: Spread's detection of membership changes, Spread's daemon and process group membership installation, Wackamole's state transfer and virtual address reallocation, and Wackamole's ARP spoofing.

Parameter Name	Default Spread	Tuned Spread
Fault-detection timeout	5	1
Distributed Heartbeat timeout	2	0.4
Discovery timeout	7	1.4

Table 1. Spread timeout tuning (seconds)

In light of this dependency, we performed two sets of experiments. The first set uses the default Spread settings with timeout intervals designed to perform adequately on most networks, for a variety of applications. The second set uses a fine-tuned version of Spread, in which we adjusted the relevant timeout intervals specifically for the Wackamole application and our network setup. Both experiments were run on a 100Mbit Ethernet LAN cluster, maintaining 10 virtual IP addresses in a cluster, and varying the number of servers from 2 to 12.

Table 1 shows the differences between the two experiment setups. The timeouts presented in the table cover the major components of the time it takes Spread to notify Wackamole of network faults. The distributed heartbeat timeout specifies an interval after which a Spread daemon notifies other daemons that it is still in operation. The fault-detection timeout begins at approximately the same time as the distributed heartbeat timeout; after the fault-detection timeout expires, if a daemon has not specified that it is operating, Spread assumes a fault has taken place and attempts to reconfigure. Because a fault could occur at any time during the heartbeat interval, the actual time to detect a failure ranges from *failure-detection timeout - distributed heartbeat timeout* to *failure-detection timeout*. The discovery timeout is the time spent performing this reconfiguration by determining the currently available set of Spread daemons and installing this configuration view at each daemon. Thus the time it takes the default Spread to notify Wackamole of a failure (ignoring the minor overhead of Spread's group membership procedure) ranges from 10 seconds to 12 seconds. For the tuned Spread, this time ranges from 2 seconds to 2.4 seconds.

Figure 5 displays the average availability interruption time when varying the cluster size, for each version of Spread. We notice that the Spread timeouts account for the majority of the interruption time recorded in our experiments.

These results were obtained using a cluster of servers under low average load. Both Wackamole and Spread can be used in production on highly-loaded machines as well. However, it is recommended that both daemon processes be run with high priority (real-time priority under Linux)

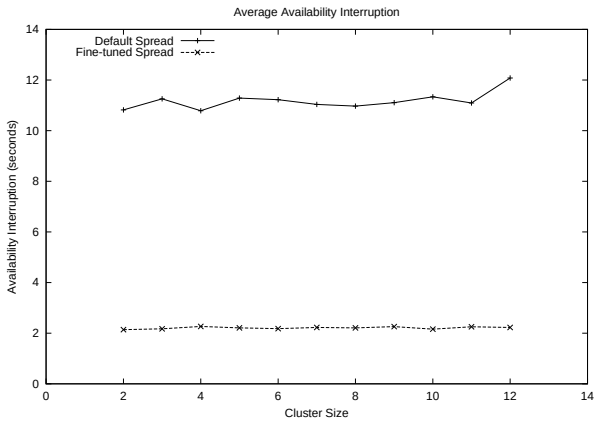

Figure 5. Average Availability Interruption with Varying Cluster Size

in these types of environments in order to avoid false positive errors. This has no adverse impact on the cluster performance as Spread and Wackamole hardly consume resources when used for this application.

Also relevant is the availability interruption time when a Wackamole daemon leaves voluntarily, not as the result of a failure. This is experienced when Wackamole daemons are taken offline for administrative or policy reasons. However, we found that this time interval is difficult to measure precisely, because it is more susceptible to context switch times and other low-level fluctuations. In general, our measurements suggest a conservative upper bound of 250 milliseconds of availability interruption on our experimental cluster; most of our measurements actually recorded an interruption time as small as 10ms.

7 Related Work

Wackamole, in its current state, has evolved from an idea first introduced in [1]. There are two areas that relate to the work presented in this paper. On one hand our solution benefits from extensive research in the areas of group communication and distributed algorithms. On the other hand, various other techniques have been employed to provide availability for critical services. Additionally our IP fail-over solution is usually used in conjunction with load-balancing mechanisms. Wackamole is often used together with the mod-backhand load-balancing module for web servers. We do not further address the coupling between Wackamole and various load-balancing techniques as it is outside the scope of the paper.

Research in the group communication area has lead to the implementation of several systems which provide properties similar to those required by the Wackamole algorithm. Among such systems we mention Horus [21], Ensemble [10], Totem [2]. The Wackamole algorithm uses a design similar to the state machine approach for maintaining consistent state in distributed systems [19, 16]

Wackamole as a fail-over solution is designed to preserve the IP presence of a service. The Virtual Router Redundancy Protocol (VRRP) was designed to perform a similar task for routers. VRRP specifies an election protocol that dynamically assigns responsibility for a virtual router to one of the VRRP routers on a local area network. VRRP design is chaired by an IETF working group and has been formalized into an Internet Standard RFC 2338 [22]. A similar protocol is the Hot Standby Router Protocol (HSRP) developed by Cisco [11]. In essence, HSRP elects one router to be the active router and another to be the standby router. The active and the standby routers send hello messages. The standby router is the candidate to take over the active role if the active router faults. All other routers are monitoring the hello messages sent by the active and standby routers. Routers may be assigned priorities. The router with the highest priority will become the active router after initialization. After an certain Active timeout elapses without hearing hello messages from the active router, the standby router takes over. Similarly if a Standby timeout elapses, a monitoring router (if such exist) with the lower IP address takes over the standby role. By default, hello messages are sent every 3 seconds and the Active and Stanby timeouts are set to 10 seconds.

Aside from IP fail-over, front-end high-availability and load-balancing devices are often used in front of mission critical networked services to provide uninterrupted service in the event of a system failure. These devices perform application level checks against machines in the cluster and keep track of which machines are providing service. They present a virtual IP address to which clients connect, and then dynamically set the local endpoint of the IP connection to an active machine in the local cluster. These devices are in common use today to support most large Internet sites and are provided by a variety of vendors. Such devices include Cisco's Arrowpoint [4], Foundry's ServerIron [12], F5's BIG/ip [5], Coyote-Point's Equalizer [7], and Linux Virtual Server [13].

While these components may provide more than just high-availability (specifically load balancing), they themselves must be made highly available – by itself, any such component is a single point of failure. Each vendor has its own method of providing High availability between two of their devices, but an application independent protocol such as VRRP or Wackamole could just as easily be used to accomplish this.

Many services need high availability and only remedial load-balancing techniques such as multiple DNS A records. For these architectures, using an IP fail-over protocol directly on the machine providing the service

in question reduces the need for complicated, expensive and otherwise unnecessary high-availability/load-balancing components.

The Linux Fake project [8] provides IP fail-over through service-probing and ARP-spoofing. The availability of the main server is probed regularly and upon failure detection a backup server instantiates a virtual IP interface that will take over the failed one and send a gratuitous ARP request to accelerate the transition.

The PolyServe Matrix HA [17] product provides a service similar to Wackamole. The technical details of the implementation or the soundness of the protocols cannot be assessed as the product and procotols are unreleased. Until recently the Polyserve solution only offered pair-wise fail-over, where every server is covered by one other specific server. The latest version of the software is reporting use of the Spread Toolkit and provides N:M, N:N, and N:1 IP failover.

In their presentation of the Raincore Distributed Session Service infrastructure [9], the authors mention a Virtual IP Manager application that similarly to Wackamole exploits underlying group communication guarantees to provide fail-over for servers and also indicate that the technology can be applied to firewalls or routers.

8 Conclusions

This paper presented a software-based distributed solution for providing high availability for clusters and routers at the IP addressing level. The core algorithm relies on a group communication service to monitor the currently connected membership and reallocate virtual IP addresses that are accessible to client machines, between the avaible servers. We presented the algorithm and discussed its correctness. We discussed two classes of practical applications of the system and provided experimental performance results.

The Wackamole system has been available as an open-source tool since August 2001 (www.wackamole.org). During the past 20 months the system was downloaded more than 1000 times and is actively used in production environments for both the web-cluster and router availability applications described in this paper. This work demonstrates how sound academic research can readily make an impact in production environments.

8.1 Acknowledgements

We thank Jim Burns, Brian Coan, Gary Levin and Sanjai Narain from Telcordia Technologies for their insightful discussions and suggestions.

References

[1] Y. Amir, Y. Gu, T. Schlossnagle, and J. Stanton. Practical cluster applications of group communication. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, New York, NY, 2000.

[2] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the 13th IEEE International Conference on Distributed Computing Systems*, May 1993.

[3] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS 98-4, Johns Hopkins University, Center for Networking and Distributed Systems, 1998.

[4] Cisco/arrowpoint. http://www.cisco.com/en/US/products/hw/contnetw/ps792/index.html.

[5] BIG/ip. http://www.f5.com/f5products/bigip/.

[6] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the ACM Symposium on OS Principles*, pages 123–138, 1987.

[7] Equalizer. http://www.coyotepoint.com/equalizer.htm.

[8] The Linux Fake Project. http://www.vergenet.net/linux/fake.

[9] C. Fan and J. Bruck. The raincore distributed session service for networking elements. In *Workshop on Communication Architecture for Clusters. International Parallel and Distributed Processing Symposium*, 2001.

[10] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[11] Hot Standby Router Protocol. http://www.cisco.com/ univercd/cc/td/doc/cisintwk/ics/cs009.htm.

[12] Foundry/ServerIron. http://www.foundrynet.com/products/webswitches/serveriron/.

[13] Linux Virtual Server. http://www.linuxvirtualserver.org/.

[14] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *International Conference on Distributed Computing Systems*, pages 56–65, 1994.

[15] Open Shortest Path First. http://www.ietf.org/html.charters/ospf-charter.html.

[16] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[17] Polyserver Matrix HA. http://polyserve.com/.

[18] Routing Information Protocol. http://www.ietf.org/html.charters/rip-charter.html.

[19] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[20] The Spread Toolkit. http://www.spread.org.

[21] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr. 1996.

[22] Virtual Router Redundancy Protocol. http://www.ietf.org/html.charters/vrrp-charter.html.

[23] Wackamole. http://www.wackamole.org.

From Total Order to Database Replication

Yair Amir and Ciprian Tutu
Johns Hopkins University
Department of Computer Science
3400 N.Charles Street, Baltimore, MD 21218
{yairamir, ciprian}@cnds.jhu.edu

Abstract

This paper presents in detail an efficient and provably correct algorithm for database replication over partitionable networks. Our algorithm avoids the need for end-to-end acknowledgments for each action while supporting network partitions and merges and allowing dynamic instantiation of new replicas. One round of end-to-end acknowledgments is required only upon a membership change event such as a network partition. New actions may be introduced to the system at any point, not only while in a primary component. We show how performance can be further improved for applications that allow relaxation of consistency requirements. We provide experimental results that demonstrate the efficiency of our approach.

1 Introduction

Database replication is quickly becoming a critical tool for providing high availability, survivability and high performance for database applications. However, to provide useful replication one has to solve the non-trivial problem of maintaining data consistency between all the replicas.

The state machine approach [25] to database replication ensures that replicated databases that start consistent will remain consistent as long as they apply the same deterministic actions (transactions) in the same order. Thus, the database replication problem is reduced to the problem of constructing a global persistent consistent order of actions. This is often mistakenly considered easy to achieve using the Total Order service (e.g. ABCAST, Agreed order, etc) provided by group communication systems.

Early models of group communication, such as Virtual Synchrony, did not support network partitions and merges. The only failures tolerated by these models were process crashes, without recovery. Under this model, total order is sufficient to create global persistent consistent order.

When network partitions are possible, total order service does not directly translate to a global persistent consistent order. Existing solutions that provide active replication either avoid dealing with network partitions [27, 23, 22] or require additional end-to-end acknowledgements for every action after it is delivered by the group communication and before it is admitted to the global consistent persistent order (and can be applied to the database) [16, 12, 26].

In this paper we describe a complete and provably correct algorithm that provides global persistent consistent order in a partitionable environment without the need for end-to-end acknowledgments on a per action basis. In our approach, end-to-end acknowledgements are only used once for every network connectivity change event (such as network partition or merge) and not per action. The basic concept was first introduced as part of a PhD thesis [2]. This paper presents our newly developed insight into the problem and goes beyond [2] by supporting online additions of completely new replicas and complete removals of existing replicas while the system executes.

Our algorithm builds a generic replication engine which runs outside the database and can be seamlessly integrated with existing databases and applications. The replication engine supports various semantic models, relaxing or enforcing the consistency constraints as needed by the application. We implemented the replication engine on top of the Spread toolkit [4] and provide experimental performance results, comparing the throughput and latency of the global consistent persistent order using our algorithm, the COReL algorithm introduced in [16], and a two-phase commit algorithm. These results demonstrate the impact of eliminating the end-to-end acknowledgments on a per-action basis.

The rest of the paper is organized as follows. The following subsection discusses related work. Section 2 describes the working model. Section 3 introduces a conceptual solution. Section 4 addresses the problems exhibited by the conceptual solution in a partitionable system and introduces the Extended Virtual Synchrony model as a tool to provide global persistent order. Section 5 describes the de-

Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)
1063-6927/02 $17.00 © 2002 IEEE

tailed replication algorithm and extends it to support online removals and additions to the set of participating replicas. Section 6 shows how the global persistent order guarantees of the algorithm can be used to support various relaxed consistency requirements. Section 7 evaluates the performance of our prototype and Section 8 concludes the paper.

1.1 Related Work

Two-phase commit protocols [12] remain the main technique used to provide a consistent view in a distributed replicated database system over an unreliable network. These protocols impose a substantial communication cost on each transaction and may require the full connectivity of all replicas to recover from some fault scenarios. Three-phase-commit protocols [26, 17] overcome some of the availability problems of two-phase-commit protocols, paying the price of an additional communication round.

Some protocols optimize for specific cases: limiting the transactional model to commutative transactions [24]; giving special weight to a specific processor or transaction [28]. Explicit use of timestamps enables other protocols [6] to avoid the need to claim locks or to enforce a global total order on actions, while other solutions settle for relaxed consistency criteria [11]. Various groups investigated methods to implement efficient lazy replication algorithms by using epidemic propagation [8, 14] or by exploiting application semantics [21].

Atomic Broadcast [13] in the context of Virtual Synchrony [7] emerged as a promising tool to solve the replication problem. Several algorithms were introduced [27, 23] to implement replication solutions based on total ordering. All these approaches, however, work only in the context of non-partitionable environments.

Keidar [16] uses the Extended Virtual Synchrony (EVS) [20] model to propose an algorithm that supports network partitions and merges. The algorithm requires that each transaction message is end-to-end acknowledged, even when failures are not present, thus increasing the latency of the protocol. In section 7 we demonstrate the impact of these end-to-end acknowledgements on performance by comparing this algorithm with ours. Fekete, Lynch and Shvartsman [9] study both [16] and [2] (which is our static algorithm) to propose an algorithm that translates View Synchrony, another specification of a partitionable group service defined in the same work, into a global total order.

Kemme, Bartoli and Babaoglu[19] study the problem of online reconfiguration of a replicated system in the presence of network events, which is an important building block for a replication algorithm. They propose various useful solutions to performing the database transfer to a joining site and provide a high-level description of an online reconfiguration method based on Enriched Virtual Synchrony allowing new replicas to join the system if they are connected with the primary component. Our solution can leverage from these database transfer techniques and adds the ability to allow new sites to join the running system without the need to be connected to the primary component.

Kemme and Alonso [18] present and prove the correctness for a family of replication protocols that support different application semantics. The protocols are introduced in a failure-free environment and then enhanced to support server crashes and recoveries. The model does not allow network partitions, always assuming that disconnected sites have crashed. In their model, the replication protocols rely on external view-change protocols that provide uniform reliable delivery in order to provide consistency across all sites. Our work shows that the transition from the group communication uniform delivery notification to the strict database consistency is not trivial, we provide a detailed algorithm for this purpose and prove its correctness.

2 System Model

The system consists of a set of nodes (servers) S=$\{S_1, S_2, ..., S_n\}$, each holding a copy of the entire database. Initially we assume that the set S is fixed and known in advance. Later, in Section 5.1, we will show how to deal with online changes to the set of potential replicas[1].

2.1 Failure and Communication Model

The nodes communicate by exchanging messages. The messages can be lost, servers may crash and network partitions may occur. We assume no message corruption and no Byzantine faults.

A server that crashes may subsequently recover retaining its old identifier and stable storage. Each node executes several processes: a database server, a replication engine and a group communication layer. The crash of any of the components running on a node will be detected by the other components and treated as a global node crash.

The network may partition into a finite number of disconnected components. Nodes situated in different components cannot exchange messages, while those situated in the same component can continue communicating. Two or more components may subsequently merge to form a larger component.

We employ the services of a *group communication layer* which provides reliable multicast messaging with ordering guarantees (FIFO, causal, total order). The group communication system also provides a membership notification service, informing the replication engine about the nodes that

[1] Note that these are changes to the system setup, not view changes caused by temporary network events.

can be reached in the current component. The notification occurs each time a connectivity change, a server crash or recovery, or a voluntary join/leave occurs. The set of participants that can be reached by a server at a given moment in time is called a *view*. The replication layer handles the server crashes and network partitions using the notifications provided by the group communication. The basic property provided by the group communication system is called Virtual Synchrony [7] and it guarantees that processes moving together from one view to another deliver the same (ordered) set of messages in the former view.

2.2 Service Model

A *Database* is a collection of organized, related data. Clients access the data by submitting *transactions.* consisting of a set of commands that follow the ACID properties. A replication service maintains a replicated database in a distributed environment. Each server from the server set maintains a private copy of the database. The initial state of the database is identical at all servers. Several models of consistency can be defined, the strictest of which is *one-copy serializability* that requires that the concurrent execution of transactions on a replicated data set is equivalent to a serial execution on a non-replicated data set. We focus on enforcing the strict consistency model, but we also support weaker models (see Section 6).

An *action* defines a transition from the current state of the database to the next state; the next state is completely determined by the current state and the action. We view actions as having a *query* part and an *update* part, either of which can be missing. Client *transactions* translate into actions that are applied to the database. The basic model best fits one-operation transactions, but in Section 6 we show how other transaction types can also be supported.

3 Replication Algorithm

In the presence of network partitions, the replication layer identifies at most a single component of the server group as a *primary component*; the other components of a partitioned group are *non-primary components*. A change in the membership of a component is reflected in the delivery of a view-change notification by the group communication layer to each server in that component. The replication layer implements a symmetric distributed algorithm to determine the order of actions to be applied to the database. Each server builds its own knowledge about the order of actions in the system. We use the coloring model defined in [1] to indicate the knowledge level associated with each action. Each server marks the actions delivered to it with one of the following colors:

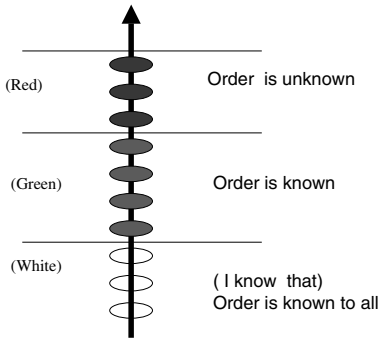

Figure 1. Action coloring

Red Action An action that has been ordered within the local component by the group communication layer, but for which the server cannot, as yet, determine the global order.

Green Action An action for which the server has determined the global order.

White Action An action for which the server knows that all of the servers have already marked it as *green*. These actions can be discarded since no other server will need them subsequently.

At each server, the *white* actions precede the *green* actions which, in turn, precede the *red* ones. An action can be marked differently at different servers; however, no action can be marked *white* by one server while it is missing or is marked *red* at another server.

The actions delivered to the replication layer in a primary component are marked green. Green actions can be applied to the database immediately while maintaining the strictest consistency requirements. In contrast, the actions delivered in a non-primary component are marked red. The global order of these actions cannot be determined yet, so, under the strong consistency requirements, these actions cannot be applied to the database at this stage.

3.1 Conceptual Algorithm

The algorithm presented in this section should, intuitively, provide an adequate solution to the replication problem. This is not actually the case, as the algorithm is not able to deal with some of the more subtle issues that can arise in a partitionable system. We present this simplified solution to provide a better insight into some of the problems the complete solution needs to cope with and to introduce the key properties of the algorithm.

Figure 2 presents the state machine associated with the conceptual algorithm. A replica can be in one of the following four states:

- **Prim State.** The server belongs to the primary component. When a client submits a request, it is multicast

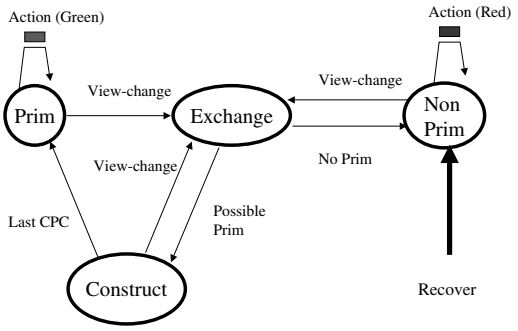

Figure 2. Conceptual Replication Algorithm

using the group communication to all the servers in the component. When a message is delivered by the group communication system to the replication layer, the action is immediately marked green and is applied to the database.

- **NonPrim State.** The server belongs to a non-primary component. Client actions are ordered within the component using the group communication system. When a message containing an action is delivered by the group communication system, it is immediately marked red.
- **Exchange State.** A server switches to this state upon delivery of a view change notification from the group communication system. All the servers in the new view will exchange information allowing them to define the set of actions that are known by some of them but not by all. These actions are subsequently exchanged and each server will apply to the database the green actions that it gained knowledge of. After this exchange is finished each server can check whether the current view has a quorum to form the next primary component. This check can be done locally, without additional exchange of messages, based on the information collected in the initial stage of this state. If the view can form the next primary component the server will move to the Construct state, otherwise it will return to the NonPrim state.
- **Construct State.** In this state, all the servers in the component have the same set of actions (they synchronized in the Exchange state) and can attempt to install the next primary component. For that they will send a Create Primary Component (CPC) message. When a server has received CPC messages from all the members of the current component it will transform all its red messages into green, apply them to the database and then switch to the Prim state. If a view change occurs before receiving all CPC messages, the server returns to the Exchange state.

In a system that is subject to partitioning we must ensure that two different components do not apply contradictory actions to the database. We use a quorum mechanism to allow the selection of a unique primary component from among the disconnected components. Only the servers in the primary component will be permitted to apply actions to the database. While several types of quorums could be used, we opted to use *dynamic linear voting* [15]. Under this system, the component that contains a (weighted) majority of the last primary component becomes the new primary component.

In many systems, processes exchange information only as long as they have a direct and continuous connection. In contrast, our algorithm propagates information by means of *eventual path*: when a new component is formed, the servers exchange knowledge regarding the actions they have, their order and color. This exchange process is only invoked immediately after a view change. Furthermore, all the components exhibit this behavior, whether they will form a primary or non-primary component. This allows the information to be disseminated even in non-primary components, reducing the amount of data exchange that needs to be performed once a server joins the primary component.

4 From Total Order to Database Replication

Unfortunately, due to the asynchronous nature of the system model, we cannot reach complete common knowledge about which messages were received by which servers just before a network partition occurs or a server crashes. In fact, it has been proven that reaching consensus in asynchronous environments with the possibility of even one failure is impossible [10]. Group communication primitives based on Virtual Synchrony do not provide any guarantees of message delivery that span network partitions and server crashes. In our algorithm it is important to be able to tell whether a message that was delivered to one server right before a view change, was also delivered to all its intended recipients.

A server p cannot know, for example, whether the last actions it delivered in the Prim state, before a view-change event occurred, were delivered to all the members of the primary component; Virtual Synchrony guarantees this fact only for the servers that will install the next view together with p. These messages cannot be immediately marked *green* by p, because of the possibility that a subset of the initial membership, big enough to construct the next primary component, did not receive the messages. This subset could install the new primary component and then apply other actions as green to the database, breaking consistency with the rest of the servers. This problem will manifest itself in any algorithm that tries to operate in the presence of network partitions and remerges. A solution based on Total Order cannot be correct in this setting without further enhancement.

4.1 Extended Virtual Synchrony

In order to circumvent the inability to know who received the last messages sent before a network event occurs we use an enhanced group communication paradigm called *Extended Virtual Synchrony* (EVS) [20] that reduces the ambiguity associated with the decision problem. Instead of having to decide on two possible values, as in the consensus problem, EVS will create three possible cases. To achieve this, EVS splits the view-change notification into two notifications: a *transitional* configuration change message and a *regular* configuration change message. The transitional configuration message defines a reduced membership containing members of the next regular configuration coming directly from the same regular configuration. This allows the introduction of another form of message delivery, *safe delivery*, which maintains the total order property but also guarantees that every message delivered to any process that is a member of a configuration is delivered to every process that is a member of that configuration, unless that process fails. Messages that do not meet the requirements for safe delivery, but are received by the group communication layer, are delivered in the transitional configuration. No new messages are sent by the group communication in the transitional configuration.

The safe delivery property provides a valuable tool to deal with the incomplete knowledge in the presence of network failures or server crashes. We distinguish now three possible cases:

1. A safe message is delivered in the regular configuration. All guarantees are met and everyone in the configuration will deliver the message (either in the regular configuration or in the following transitional configuration) unless they crash.
2. A safe message is delivered in the transitional configuration. This message was received by the group communication layer just before a partition occurs. The group communication layer cannot tell whether other components that split from the previous component received and will deliver this message.
3. A safe message was sent just before a partition occurred, but it was not received by the group communication layer in some detached component. The message will, obviously, not be delivered at the detached component.

The power of this differentiation lies in the fact that, with respect to the same message, it is impossible for one server to be in case 1, while another is in case 3. To illustrate the use of this property consider the Construct phase of our algorithm: If a server p receives all CPC messages in the regular configuration, it knows that every server in that configuration will receive all the messages before the next regular

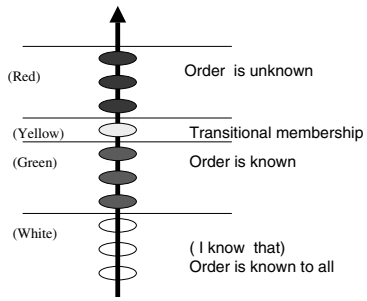

Figure 3. Updated coloring model

configuration is delivered, unless they crash; some servers may, however, receive some of the CPC messages in a transitional configuration. Conversely, if a server q receives a configuration change for a new regular configuration before receiving all of the CPC messages, then no server could have received a message that q did not receive as safe in the previous configuration. In particular, no server received all of the CPC messages as safe in the previous regular configuration. Thus q will know that it is in case 3 and no other server is in case 1. Finally, if a server r received all CPC messages, but some of those were delivered in a transitional configuration, then r cannot know whether there is a server p that received all CPC messages in the regular configuration or whether there is a server q that did not receive some of the CPC messages at all; r does, however, know that there cannot exist both p and q as described.

5 Replication Algorithm

Based on the above observations the algorithm skeleton presented in Section 3.1 needs to be refined. We will take advantage of the Safe delivery properties and of the differentiated view change notification that EVS provides. The two delicate states are, as mentioned, Prim and Construct.[2]

In the **Prim** state, only actions that are delivered as safe during the regular configuration can be applied to the database. Actions that were delivered in the transitional configuration cannot be marked as green and applied to the database before we know that the next regular configuration will be the one defining the primary component of the system. If an action a is delivered in the transitional membership and is marked directly as green and applied to the database, then it is possible that one of the detached components that did not receive this action will install the next primary component and will continue applying new actions

[2]While the same problem manifests itself in any state, it is only these two states where knowledge about the message delivery is critical, as it determines either the global total order (in Prim) or the creation of the new primary (Construct).

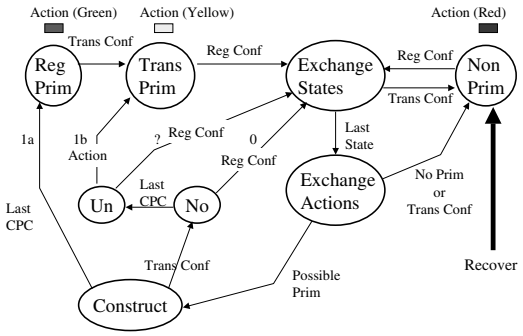

Figure 4. Replication State Machine

to the database, without applying a, thus breaking the consistency of the database. To avoid this situation, the **Prim** state was split into two states: ***RegPrim*** and ***TransPrim*** and a new message color was introduced to the coloring model:

Yellow Action An action that was delivered in a transitional configuration of a primary component.

A *yellow* action becomes *green* at a server as soon as this server learns that another server marked the action *green* or when this server becomes part of a primary component. As discussed in the previous section, if an action is marked as *yellow* at some server p, then there cannot exist r and s, in this component, such that one marked the action as *red* and the other marked it *green*.

In the presence of consecutive network changes, the process of installing a new primary component can be interrupted by another configuration change. If a transitional configuration is received by a server p while in the **Construct** state, before receiving all the CPC messages, the server will not be able to install the new primary and will switch to a new state: ***No***. In this state p expects to receive the delivery of the new regular configuration which will trigger the initiation of a new exchange round. However, if p receives all the rest of the CPC messages in **No** (i.e. in the transitional configuration), it means that it is possible that some server q has received all the CPC messages in **Construct** and has moved to **RegPrim**, completing the installation of the new primary.

To account for this possibility, p will switch to another new state: ***Un*** (undecided). If an action message is received in this state then p will know for sure that there was a server q that switched to **RegPrim** and even managed to generate new actions before noticing the network failure that caused the cascaded membership change. Server p, in this situation (1b), has to act as if installing the primary component in order to be consistent, mark its old yellow/red actions as green, mark the received action as yellow and switch to **TransPrim**, "joining" q who will come from **RegPrim** as it will also eventually notice the new configuration change. If the regular configuration message is delivered without any message being received in the **Un** state (transition marked ? in Figure 4), p remains uncertain whether there was a server that installed the primary component and will not attempt to participate in the formation of a new primary until this dilemma is cleared through exchange of information with one or, in the worst case, all of the members that tried to install the same primary as p.

Figure 4 shows the updated state machine. Aside from the changes already mentioned, the Exchange state was also split into **ExchangeStates** and **ExchangeActions**, mainly for clarity reasons. From a procedural point of view, once a view change is delivered, the members of each view will try to establish a maximal common state that can be reached by combining the information and actions held by each server. After the common state is determined, the participants proceed to exchange the relevant actions. Obviously, if the new membership is a subset of the old one, there is no need for action exchange, as the states are already synchronized.

5.1 Dynamic Replica Instantiation and Removal

As mentioned in the description of the model, the algorithm that we presented so far works under the limitation of a fixed set of potential replicas. It is of great value, however, to allow for the dynamic instantiation of new replicas as well as for their deactivation. Moreover, if the system does not support permanent removal of replicas, it is susceptible to blocking in case of a permanent failure or disconnection of a majority of nodes in the primary component.

However, dynamically changing the set of servers is not straightforward: the set change needs to be synchronized over all the participating servers in order to avoid confusion and incorrect decisions such as two distinct components deciding they are the primary, one being the rightful one in the old configuration, the other being entitled to this in the new configuration. Since this is basically a consensus problem, it cannot be solved in a traditional fashion. We circumvent the problem with the help of the persistent global total order that the algorithm provides.

When a replica r wants to permanently leave the system, it will broadcast a PERSISTENT_LEAVE message that will be ordered as if it was an action message. When this message becomes *green* at replica s, s can update its local data structures to exclude r from the list of potential replicas. The PERSISTENT_LEAVE message can also be administratively inserted into the system to signal the permanent removal, due to failure, of one of the replicas. The message will be issued by a site that is still in the system and will contain the server id of the dead replica.

A new replica r that wants to join the replicated system will first need to connect to one of the members (s of the system, without joining the group. s will act as a representative for the new site to the existing group by creating a

PERSISTENT_JOIN message to announce *r*'s intention to join the group. This message will be ordered as a regular action, according to the standard algorithm. When the message becomes *green* at a server, that replica will update its data structures to include the newcomer's server id and set the green line (the last globally ordered message that the server has) for the joining member as the action corresponding to the PERSISTENT_JOIN message. Basically, from this point on the servers acknowledge the existence of the new member, although *r* is still not connected to the group. When the PERSISTENT_JOIN message becomes green at the peer server (the representative), the peer server will take a snapshot of the database and start transferring it to the joining member. If the initial peer fails or a network partition occurs before the transfer is finished, the new server will try to establish a connection with a different member of the system and continue its update. If the new peer already ordered the PERSISTENT_JOIN message sent by the first representative, it will know about *r* and the state that it has to reach before joining the system, therefore will be able to resume the transfer procedure. If the new peer has not yet ordered the PERSISTENT_JOIN message it will issue another PERSISTENT_JOIN message for the *r*. PERSISTENT_JOIN messages for members that are already present in the local data structures are ignored by the existing servers, so only the first ordered PERSISTENT_JOIN will define the entry point of the new site into the system. Since the algorithm guarantees global total ordering, this entry point is uniquely defined. Finally, when the transfer is complete, *r* will set the action counter to the last action that was ordered by the system and will join the group of replicas. This will be seen as a view change by the existing members and they will go through the EXCHANGE states and continue according to the algorithm.

Another method for performing online reconfiguration is described in [19]. This method requires the joining site to be permanently connected to the primary component while being updated. We maintain the flexibility of the engine and we allow joining replicas to be connected to non-primary components during their update stage. It can even be the case that a new site is accepted into the system without **ever** being connected to the primary component, due to the eventual path propagation method. The insertion of a new replica into the system in a non-primary component, can be useful to certain applications as is shown in Section 6.

The static algorithm code was presented in [2], while the complete algorithm code, including the dynamic capabilities can be found in the extended version of this paper [5].

5.2 Proof of Correctness

The algorithm in its static form was proven correct in [2]. The correctness properties that were guaranteed were liveness, FIFO order and Total global order. Here, we prove that the enhanced dynamic version of the algorithm still preserves the same guarantees.

Lemma 1 (Global Total Order (static)) *If both servers s and r performed their ith actions, then these actions are identical.*

Lemma 2 (Global FIFO Order (static)) *If server r performed an action a generated by server s, then r already performed every action that s generated prior to a.*

These are the two properties that define the **Safety** criterion in [2]. These specifications need to be refined to encompass the removal of servers or the addition of new servers to the system.

Theorem 1 (Global Total Order (dynamic)) *If both servers s and r performed their ith action, then these actions are identical.*

Proof: Consider the system in its start-up configuration set. Any server in this configuration will trivially maintain this property according to Lemma 1. Consider a server *s* that joins the system. The safety properties of the static algorithm guarantee that after ordering the same set of actions, all servers will have the same consistent database. This is the case when a PERSISTENT_JOIN action is ordered. According to the algorithm *s* will set its global action counter to the one assigned by the system to the PERSISTENT_JOIN action. From this point on the behavior of *s* is indistinguishable from a server in the original configuration and the claim is maintained as per Lemma 1. □

Theorem 2 (Global FIFO Order (dynamic)) *If server r performed an action a generated by server s, then r already performed every action that s generated prior to a, or it inherited a database state which incorporated the effect of these actions.*

Proof: According to Lemma 2, the theorem holds true from the initial starting point until a new member is added to the system. Consider *r,* a member who joins the system. According to the algorithm, the joining member transfers the state of the database as defined by the action ordered immediately before the PERSISTENT_JOIN message. All actions generated by *s* and ordered before the PERSISTENT_JOIN will be incorporated in the database that *r* received. From Theorem 1, the PERSISTENT_JOIN message is ordered at the same place at all servers. All actions generated by *s* and ordered after the PERSISTENT_JOIN message will be ordered similarly at every server, including *r*, according to Theorem 1. Since Lemma 2 holds for any other member, this is sufficient to guarantee that *r* will order all other actions generated by *s* prior to a, and ordered after *r* joined the system. □

Lemma 3 (Liveness (static)) *If server s orders action a and there exists a set of servers containing s and r, and a*

time from which on that set does not face any communication or process failures, then server r eventually orders action a.

This is the liveness property defined in [2] and proven to be satisfied by the static replication algorithm. This specification needs to be refined to include the notion of servers permanently leaving the system.

Theorem 3 (Liveness (dynamic)) *If server s orders action a in a configuration that contains r and there exists a set of servers containing s and r, and a time from which on that set does not face any communication or process failures, then server r eventually orders action a.*

Proof: The theorem is a direct extension of Lemma 3, which acknowledges the potential existence of different server-set configurations. An action that is ordered by a server in one configuration will be ordered by all servers in the same configuration as a direct consequence of Theorem 1. Servers that leave the system or crash do not meet the requirements for the liveness property, while servers that join the system will order the actions generated in any configuration that includes them, unless they crash. □

6 Supporting Various Application Semantics

The presented algorithm was designed to provide strict consistency semantics by applying actions to the database only when they are marked green and their global order is determined. In the real world, where incomplete knowledge is unavoidable, many applications would rather have an immediate answer, than incur a long latency to obtain a complete and consistent answer. Therefore, we provide additional service types for clients in a non-primary component. The result of a *weak query* is obtained from a consistent, but possibly obsolete state of the database, as reflected by the green actions known to the server at the time of the query, even while in a non-primary component. Other applications prefer getting an immediate reply based on the latest information available, although possibly inconsistent. In the primary component the state of the database reflects the most updated situation and is always consistent. In a non-primary component, however, red actions must be taken into account in order to provide the latest, though not consistent, information. We call this type of query a *dirty query*.

Different semantics can be supported also with respect to updates. In the *timestamp* semantics case, the application in interested only in the most recent information/ Location tracking is a good example of an application that would employ such semantics. Similarly, *commutative* semantics are used in applications where the order of action execution is irrelevant as long as all actions are eventually applied. In an inventory management application all operations on the stock would be commutative. For both semantics, the one-copy serializability property is not maintained in the presence of network partitions. However, after the network is repaired and the partitioned components merge, the databases states converge.

The algorithm can be significantly optimized if the engine has the ability to distinguish a query-only action from an action that contains updates. A query issued at one server can be answered as soon as all previous actions generated by this server were applied to the database, without the need to generate and order an action message.

Modern database applications exploit the ability to execute a procedure specified by a transaction. These are called *active* transactions and they are supported by our algorithm, provided that the invoked procedure is deterministic and depends solely on the current database state. The procedure will be invoked at the time the action is ordered, rather than before the creation of the update.

Finally, we mentioned that our model best fits one-operation transactions. Actually, any non-interactive transactions that do not invoke triggers are supported in a similar way. However, some applications need to use interactive transactions which, within the same transaction, read data and then perform updates based on a user decision, rather than a deterministic procedure. Such behavior, cannot be modeled using one action, but can be mimicked with the aid of two actions. The first action reads the necessary data, while the second one is an active action as described above. This active action encapsulates the update dictated by the user, but first checks whether the values of the data read by the first action are still valid. If not, the update is not applied, as if the transaction was aborted in the traditional sense. Note that if one server "aborts", all of the servers will abort that (trans)action, since they apply an identical deterministic rule to an identical state of the database.

7 Performance Analysis

In this section we evaluate our replication engine and compare its performance to that of two existing solutions: two-phase commit (2PC) and COReL by Keidar [16]. 2PC is the algorithm adopted by most replicated systems that require strict consistency. 2PC requires two forced disk writes and 2n unicast messages per action. COReL exploits group communication properties to improve on that. COReL requires one forced disk write and n multicast messages per action. In contrast, our engine only requires 1/n forced disk write and one multicast message per action on average (only the initiating server needs to force the action to disk).

We implemented all three algorithms and compared their performance in normal operation, without view changes. Our 2PC implementation does not perform the locking required to guarantee the unique order of transaction execution, as this is usually the task of the database. Therefore

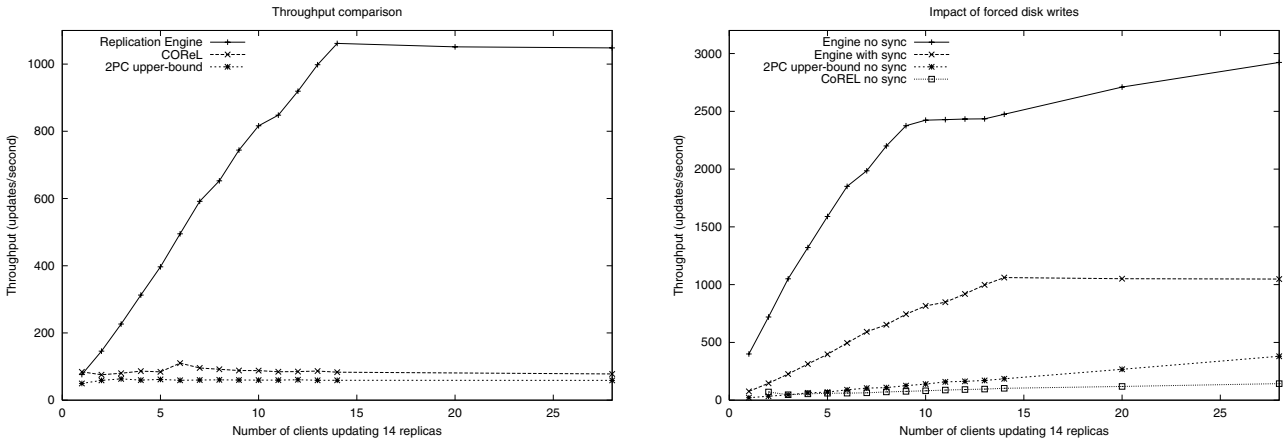

Figure 5. Throughput Comparison

a complete 2PC will perform strictly worse than our upper-bound implementation.

Since we are interested in the intrinsic performance of the replication metods, clients receive responses to their actions as soon as the actions are globally ordered, without any interaction with a database. A follow-up work [3] evaluates a complete solution that replicates a Postgres database over local and wide area networks using our engine.

All the tests were conducted with 14 replicas, each running on a dual processor Pentium III-667 with Linux connected by a 100Mbits/second local area switch. Each action is 200 bytes long (e.g. an SQL statement).

Figure 5(a) compares the maximal throughput that a system of 14 replicas can sustain under each of the three methods. We vary the number of clients that simultaneously submit requests into the system between 1 and 28, evenly spread between the replicas as much as possible. The clients are constantly injecting actions into the system, the next action from a client being introduced immediately after the previous action from that client is completed and its result reported to the client.

Our engine achieves a maximum throughput of 1050 updates/second once there are sufficient clients to saturate the system, outperforming the other methods by at least a factor of 10. COReL outperforms the uppper-bound 2PC as expected, mainly due to the saving in disk writes reaching a maximum of 110 updates/second as opposed to 63 updates/second for the upper-bound 2PC.

High-performance database environments commonly use superior storage technology (e.g flash disks). In order to estimate the performance that the three methods would exhibit in such environment, we used asynchronous disk writes instead of forced disk writes. Figure 5(b) shows that our engine tops at processing 3000 updates/second. Under the same conditions, the upper-bound 2PC algorithm achieves 400 updates/second. COReL reaches a throughput of approximately 100 updates/second with 28 clients, but more clients are needed in order to saturate the COReL system due to its higher latency. With 50 clients, CORel saturates the system with about 200 updates/second. 2PC outperforms COReL in this experiment because of two reasons: the fact that we use an upper-bound 2PC as mentioned above, and the particular switch that serves our local area network that is capable of transmitting multiple unicast messages between different pairs in parallel.

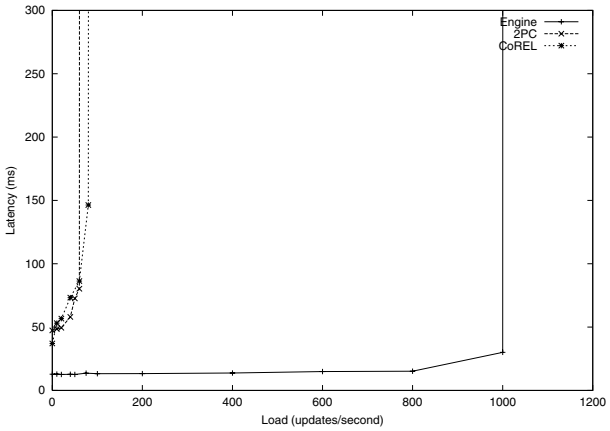

Figure 6. Latency variation with load

We also measured the response time a client experiences under different loads (Figure 6). Our Engine maintains an average latency of 15ms with load increasing up to 800 updates/second and breaks at the maximum supported load of 1050 updates/second. COReL and 2PC experience latencies of 35ms up to 80ms under loads up to 100 updates/second with COReL being able to sustain more throughput.

8 Conclusions

We presented a complete algorithm for database replication over partitionable networks sophistically utilizing

group communication and proved its correctness. Our avoidance of the need for end-to-end acknowledgment per action contributed to superior performance. We showed how to incorporate online instantiation of new replicas and permanent removal of existing replicas. We also demonstrated how to efficiently support various types of applications that require different semantics.

Acknowledgements

We thank Jonathan Stanton for his numerous technical ideas and support that helped us optimize the overall performance of the system. We also thank Michal Miskin-Amir and Jonathan Stanton for their insightful suggestions that considerably improved the presentation of this paper. This work was partially funded by grant F30602-00-2-0550 from the Defense Advanced Research Projects Agency (DARPA). The views expressed in this paper are not necessarily endorsed by DARPA.

References

[1] O. Amir, Y. Amir, and D. Dolev. A highly available application in the Transis environment. *Lecture Notes in Computer Science*, 774:125–139, 1993.

[2] Y. Amir. *Replication Using Group Communication over a Partitioned Network*. PhD thesis, Hebrew University of Jerusalem, Jerusalem, Israel, 1995. http://www.cnds.jhu.edu/publications/yair-phd.ps.

[3] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. Practical wide-area database replication. Technical Report CNDS 2002-1, Johns Hopkins University, Center for Networking and Distributed Systems, 2002.

[4] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS 98-4, Johns Hopkins University, Center for Networking and Distributed Systems, 1998.

[5] Y. Amir and C. Tutu. From total order to database replication. Technical Report CNDS 2002-3, Johns Hopkins University, Center for Networking and Distributed Systems, 2002.

[6] P. Bernstein, D. Shipman, and J. Rothnie. Concurrency control in a system for distributed databases (sdd-1). *ACM Transactions on Database Systems*, 5(1):18–51, Mar. 1980.

[7] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the ACM Symposium on OS Principles*, pages 123–138, Austin, TX, 1987.

[8] A. Demers et al. Epidemic algorithms for replicated database maintenance. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, BC, Canada, Aug. 1987.

[9] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, May 2001.

[10] M. H. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[11] R. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, UC Santa Cruz, 1992.

[12] J. N. Gray and A. Reuter. *Transaction Processing: concepts and techniques*. Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA, 1993.

[13] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5. Addison-Wesley, second edition, 1993.

[14] J. Holliday, D. Agrawal, and A. E. Abbadi. Database replication using epidemic update. Technical Report TRCS00-01, University of California Santa-Barbara, 19, 2000.

[15] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–280, 1990.

[16] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, 1994.

[17] I. Keidar and D. Dolev. Increasing the resilience of atomic commit at no additional cost. In *Symposium on Principles of Database Systems*, pages 245–254, 1995.

[18] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333 – 379, 2000.

[19] B. Kemme, A. Bartoli, and O. Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the Internationnal Conference on Dependable Systems and Networks*, Göteborg, Sweden, 2001.

[20] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *International Conference on Distributed Computing Systems*, pages 56–65, 1994.

[21] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proceedings of* 14^{th} *International Symposium on DIStributed Computing (DISC'2000)*, 2000.

[22] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[23] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, Sept. 1998.

[24] C. Pu and A. Leff. Replica control in distributed systems: an asynchronous approach. *ACM SIGMOD Record*, 20(2):377–386, 1991.

[25] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[26] D. Skeen. A quorum-based commit protocol. Berkley Workshop on Distributed Data Management and Computer Networks, February 1982.

[27] I. Stanoi, D. Agrawal, and A. E. Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the* 18^{th} *IEEE International Conference on Distributed Computing Systems '98*, pages 148–155, Amsterdam, May 1998.

[28] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, SE-5:188–194, May 1979.

Technical Report CNDS-2002-1 Johns Hopkins University, http://www.cnds.jhu.edu/publications

Practical Wide-Area Database Replication[1]

Yair Amir[*], Claudiu Danilov[*], Michal Miskin-Amir[†], Jonathan Stanton[*], Ciprian Tutu[*]

[*] Johns Hopkins University
Department of Computer Science
Baltimore MD 21218
{yairamir, claudiu, jonathan, ciprian}@cnds.jhu.edu

[†] Spread Concepts LLC
5305 Acacia Ave
Bethesda MD 20814
michal@spreadconcepts.com

Abstract

This paper explores the architecture, implementation and performance of a wide and local area database replication system. The architecture provides peer replication, supporting diverse application semantics, based on a group communication paradigm. Network partitions and merges, computer crashes and recoveries, and message omissions are all handled. Using a generic replication engine and the Spread group communication toolkit, we provide replication services for the PostgreSQL database system. We define three different environments to be used as test-beds: a local area cluster, a wide area network that spans the U.S.A, and an emulated wide area test bed. We conduct an extensive set of experiments on these environments, varying the number of replicas and clients, the mix of updates and queries, and the network latency. Our results show that sophisticated algorithms and careful distributed systems design can make symmetric, synchronous, peer database replication a reality for both local and wide area networks.

1. Introduction

Database management systems are among the most important software systems driving the information age. In many Internet applications, a large number of users that are geographically dispersed may routinely query and update the same database. In this environment, the location of the data can have a significant impact on application response time and availability. A centralized approach manages only one copy of the database. This approach is simple since contradicting views between replicas are not possible. The centralized approach suffers from two major drawbacks:

- Performance problems due to high server load or high communication latency for remote clients.
- Availability problems caused by server downtime or lack of connectivity. Clients in portions of the network that are temporarily disconnected from the server cannot be serviced.

The server load and server downtime problems can be addressed by replicating the database servers to form a cluster of peer servers that coordinate updates. However,

[1] This work was partially funded by grant F30602-00-2-0550 from the Defense Advanced Research Projects Agency (DARPA). The views expressed in this paper are not necessarily endorsed by DARPA.

communication latency and server connectivity remain a problem when clients are scattered on a wide area network and the cluster is limited to a single location. Wide area database replication coupled with a mechanism to direct the clients to the best available server (network-wise and load-wise) [APS98] can greatly enhance both the response time and availability.

A fundamental challenge in database replication is maintaining a low cost of updates while assuring global system consistency. The problem is magnified for wide area replication due to the high latency and the increased likelihood of network partitions in wide area settings.

In this paper, we explore a novel replication architecture and system for local and wide area networks. We investigate the impact of latency, disk operation cost, and query versus update mix, and how they affect the overall performance of database replication. We conclude that for many applications and network settings, symmetric, synchronous, peer database replication is practical today.

The paper focuses on the architecture, implementation and performance of the system. The architecture provides peer replication, where all the replicas serve as master databases that can accept both updates and queries. The failure model includes network partitions and merges, computer crashes and recoveries, and message omissions, all of which are handled by our system. We rely on the lower level network mechanisms to handle message corruptions, and do not consider Byzantine faults.

Our replication architecture includes two components: a wide area group communication toolkit, and a replication server. The group communication toolkit supports the Extended Virtual Synchrony model [MAMA94]. The replication servers use the group communication toolkit to efficiently disseminate and order actions, and to learn about changes in the membership of the connected servers in a consistent manner.

Based on a sophisticated algorithm that utilizes the group communication semantics [AT01, A95], the replication servers avoid the need for end-to-end acknowledgements on a per-action basis without compromising consistency. End-to-end acknowledgments are only required when the membership of the connected servers changes due to network partitions, merges, server crashes and recoveries. This leads to high system performance. When the membership of connected servers is stable, the throughput of the system and the latency of actions are determined mainly by the performance of the group communication and the single node database performance, rather than by other factors such as the number of replicas. When the group communication toolkit scales to wide area networks, our architecture automatically scales to wide area replication.

We implemented the replication system using the Spread Group Communication Toolkit [AS98, ADS00, Spread] and the PostgreSQL database system [Postgres]. We then define three different environments to be used as test-beds: a local area cluster with fourteen replicas, the CAIRN wide area network [CAIRN] that spans the U.S.A with seven sites, and the Emulab emulated wide area test bed [Emulab].

We conducted an extensive set of experiments on the three environments, varying the number of replicas and clients, and varying the mix of updates and queries. Our results show that sophisticated algorithms and careful distributed systems design can make

symmetric, synchronous, peer database replication a reality over both local and wide area networks.

The remainder of this paper is organized as follows. The following subsection discusses related work. Section 2 presents our architecture for transparent database replication. Section 3 presents the Spread group communication toolkit and the optimization we implemented to support efficient wide area replication. Section 4 details our replication server. Section 5 presents the experiments we constructed to evaluate the performance of our system, and Section 6 concludes the paper.

Related Work

Despite their inefficiency and lack of scalability, two-phase commit protocols [GR93] remain the principal technique used by most commercial database systems that try to provide synchronous peer replication. Other approaches investigated methods to implement efficient lazy replication algorithms using epidemic propagation [Demers87, HAE00].

Most of the state-of-the-art commercial database systems provide some level of database replication. However, in all cases, their solutions are highly tuned to specific environment settings and require a lot of effort in their setup and maintenance. Oracle [Oracle], supports both asynchronous and synchronous replication. However, the former requires some level of application decision in conflict resolution, while the latter requires that all the replicas in the system are available to be able to function, making it impractical. Informix [Informix], Sybase [Sybase] and DB2 [DB2] support only asynchronous replication, which again ultimately rely on the application for conflict resolution.

In the open-source database community, two database systems emerge as clear leaders: MySQL [Mysql] and PostgreSQL [Postgres]. By default both systems only provide limited master-slave replication capabilities. Other projects exist that provide more advanced replication methods for Postgres such as Postgres Replicator, which uses a trigger-based store and forward asynchronous replication method [Pgrep].

The more evolved of these approaches is Postgres-R [Postgres-R], a project that combines open-source expertise with academic research. This work implements algorithms designed by Kemme and Alonso [KA00] into the PostgreSQL database manager in order to provide synchronous replication. The current work focuses on integrating the method with the upcoming 7.2 release of the PostgreSQL system.

Kemme and Alonso introduce the Postgres-R approach in [KA00] and study its performance on Local Area settings. They use an eager-replication method that exploits group communication ordering guarantees to serialize write conflicts at all sites. The initial work was done on version 6.4.2 of PostgreSQL. In similar approaches, Jimenez-Peris and Patino-Martinez, together with Kemme and Alonso analyze various other methods and their performance in local area settings [JPKA00, PJKA00].

Research on protocols to support group communication across wide area networks such as the Internet has begun to expand. Recently, new group communication protocols designed for such wide area networks have been proposed [KSMD00, KK00, AMMB98, ADS00] which continue to provide the traditional strong semantic properties such as

reliability, ordering, and membership. The only group communication systems we are aware of that currently exist, are available for use, and can provide the Extended Virtual Synchrony semantics are Horus[RBM96], Ensemble[H98], and Spread[AS98]. The JGroups[Mon00] system provides an object-oriented group communication system, but its semantics differ in substantial detail from Extended Virtual Synchrony.

2. An Architecture for Transparent Database Replication

Our architecture provides peer replication, supporting diverse application semantics, based on a group communication paradigm. Peer replication is a symmetric approach where each of the replicas is guaranteed to invoke the same set of actions in the same order. In contrast with the common Master/Slave replication model, in peer replication each replica acts as a master. This approach requires the next state of the database to be determined by the current state and the next action, and it guarantees that all of the replicas reach the same database state.

The architecture is structured into two layers: a replication server and a group communication toolkit (Figure 1).

Each of the replication servers maintains a private copy of the database. The client application *requests* an action from one of the replication servers. The replication servers agree on the order of actions to be performed on the replicated database. As soon as a replication server knows the final order of an action, it *applies* this action to the database. The replication server that initiated the action returns the database *reply* to the client application. The replication servers use the group communication toolkit to disseminate the actions among the servers group and to help reach an agreement about the final global order of the set of actions.

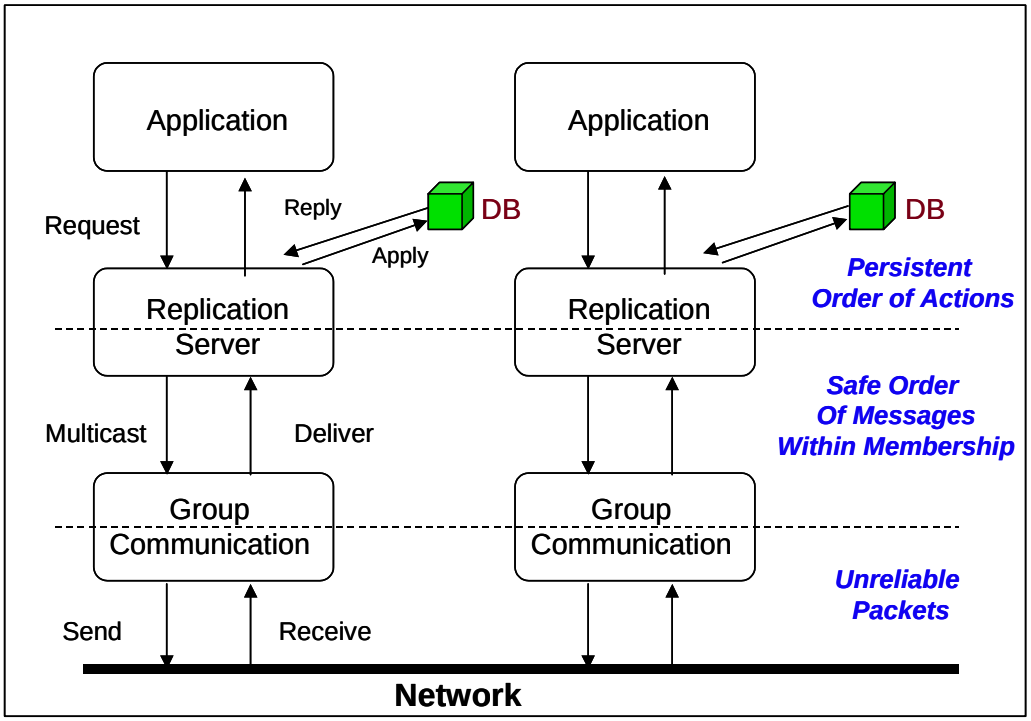

Figure 1: A Database Replication Architecture

In a typical operation, when an application submits a request to a replication server, this server logically *multicasts* a message containing the action through the group communication. The local group communication toolkit *sends* the message over the network. Each of the **currently connected** group communication daemons eventually *receives* the message and then *delivers* the message in the same order to their replication servers.

The group communication toolkit provides multicast and membership services according to the Extended Virtual Synchrony model [MAMA94]. For this work, we are particularly interested in the Safe Delivery property of this model. Delivering a message according to Safe Delivery requires the group communication toolkit both to determine the total order of the message and to know that every other daemon in the membership already has the message.

The group communication toolkit overcomes message omission faults and notifies the replication server of changes in the membership of the currently connected servers. These notifications correspond to server crashes and recoveries or to network partitions and re-merges. On notification of a membership change by the group communication layer, the replication servers exchange information about actions sent before the membership change. This exchange of information ensures that every action known to any member of the currently connected servers becomes known to all of them. Moreover, knowledge of final order of actions is also shared among the currently connected servers. As a consequence, after this exchange is completed, the state of the database at each of the connected servers is identical. The cost of such synchronization amounts to one message exchange among all connected servers plus the retransmission of all updates that at least one connected server has and at least one connected server does not have. Of course, if a site was disconnected for an extended period of time, it might be more efficient to transfer a current snapshot [KBB01].

The careful use of Safe Delivery and Extended Virtual Synchrony allows us to eliminate end-to-end acknowledgments on a per-action basis. As long as no membership change takes place, the system eventually reaches consistency. End to end acknowledgements and state synchronization are only needed once a membership change takes place. A detailed description of the replication algorithm we are using is given in [AT01, A95].

Advanced replication systems that support a peer-to-peer environment must address the possibility of conflicts between the different replicas. Our architecture eliminates the problem of conflicts because updates are always invoked in the same order at all the replicas.

The latency and throughput of the system for updates is obviously highly dependent on the performance of the group communication Safe Delivery service. Read-only queries will not be sent over the network.

An important property our architecture achieves is transparency - it allows replicating a database without modifying the existing database manager or the applications accessing the database. The architecture does not require extending the database API and can be implemented directly above the database or as a part of a standard database access layer (e.g. ODBC or JDBC).

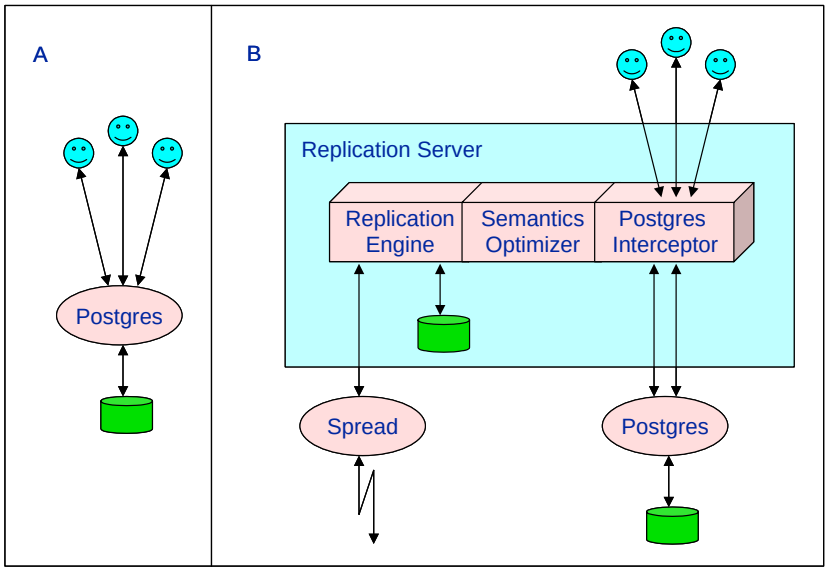

Figure 2: Modular Software Architecture

Figure 2.A presents a non-replicated database system that is based on the Postgres database manager. Figure 2.B. presents the building blocks of our implementation, replicating the Postgres database system. The building blocks include a replication server and the Spread group communication toolkit. The Postgres clients see the system as in figure 2.A., and are not aware of the replication although they access the database through our replication server. Similarly, any instance of the Postgres database manager sees the local replication server as a client.

The replication server consists of several independent modules that together provide the database integration and consistency services (Figure 2.B). They include:

- A provably correct generic Replication Engine that includes all of the replication logic, and can be applied to any database or application. The engine maintains a consistent state and can recover from a wide range of network and server failures. The replication engine is based on the algorithm presented in [AT01, A95].
- A Semantics Optimizer that decides whether to replicate transactions and when to apply them based on the required semantics, the actual content of the transaction, and whether the replica is in a primary component or not.
- A database specific interceptor that interfaces the replication engine with the DBMS client-server protocol. To replicate Postgres, we created a Postgres specific interceptor. Existing applications can transparently use our interceptor layer to provide them with an interface identical to the Postgres interface, while the Postgres database server sees our interceptor as a regular client. The database itself does not need to be modified nor do the applications. A similar interceptor could be created for other databases.

To optimize performance, the replication server could be integrated with the database manager, allowing more accurate determination of which actions could be parallelized internally.

The flexibility of this architecture enables the replication system to support heterogeneous replication where *different* database managers from different vendors replicate the same logical database.

3. The Spread Toolkit

Spread is a client-server group messaging toolkit that provides reliable multicast, ordering of messages and group membership notifications under the Extended Virtual Synchrony semantics [MAMA94] over local and wide-area networks to clients that connect to daemons.

To provide efficient dissemination of multicast messages on wide-area networks, all Spread daemons located in one local area network are aggregated into a group called a site, and a dissemination tree is rooted at each site, with other sites forming the nodes of the tree. Messages that originate at a Spread daemon in a site will first be multicast to all the daemons of the site. Then, one of those daemons will forward the message onto the dissemination tree where it will be forwarded down the tree until all of the sites have received the message. Therefore, the expected latency to deliver a message is the latency of the local area network plus the sum of the latencies of the links on the longest path down the tree.

The total order of messages is provided by assigning a sequence number and a Lamport time stamp to each message when it is created by a daemon. The Lamport time stamps provide a global partial causal order on messages and can be augmented to provide a total causal order by breaking ties based on the site identifier of the message [L78]. To know the total order of a message that has been received, a server must receive a message from every site that contains a Lamport time stamp at least equal to the Lamport time stamp of the message that is to be delivered. Thus, each site must receive a message from every other site prior to delivering a totally ordered message.

The architecture and reliability algorithms of the Spread toolkit [AS98, ADS00], and its global flow control [AADS01] provide basic services for reliable, FIFO, total order and Safe delivery. However, our architecture requires a high-performance implementation of the Safe Delivery service for wide-area networks, something not developed in previous work.

Delivering a message according to Safe Delivery requires the group communication toolkit to determine the total order of the message and to know that every other daemon in the membership already has the message. The latter property (sometimes called message stability) was traditionally implemented by group communication systems for garbage collection purposes, and therefore was not optimized. For this work, we designed and implemented scalable, high performance wide-area Safe Delivery for Spread.

Scalable Safe Delivery for Wide Area Networks

A naive algorithm will have all the daemons immediately acknowledge each Safe message. These acknowledgements are sent to all of the daemons, thus reaching all the possible destinations of the original Safe message within one network diameter. Overall, this leads to a latency of twice the network diameter. However, this algorithm is

obviously not scalable. For a system with *N* sites, each message requires *N* broadcast acknowledgements, leading to *N* times more control messages than the data messages.

Our approach avoids this ack explosion problem by aggregating information into cumulative acknowledgements. However, minimizing the bandwidth at all costs will cause extremely high latency for Safe messages, which is also undesirable. Therefore our approach permits tuning the tradeoff between bandwidth and latency.

The structure of our acknowledgements, referred to as *ARU_updates* (all received up-to), is as follows for an *ARU_update* originating at site *A*:

- *Site_Sequence* is the sequence number of the last message originated by any daemon at site *A* and forwarded in order to other sites (Spread may forward messages even out of order to other sites to optimize performance). This number guarantees that no messages will be originated in the future from site *A* with a lower sequence number.
- *Site_Lts* is the Lamport timestamp that guarantees that site *A* will never send a message with a lower Lamport timestamp and a sequence number higher then *Site_Sequence*.
- *Site_Aru* is the highest Lamport timestamp such that all the messages with a lower Lamport timestamp that originated from any site are already received at site *A*.

Each daemon keeps track of these three values received from each of the currently connected sites. In addition, each daemon updates another variable, *Global_Aru*, which is the minimum of the *Site_Aru* values received from all of the sites. This represents the Lamport timestamp of the last message received in order by all the daemons in the system. A Safe message can be delivered to the application (the replication server, in our case) when its Lamport timestamp is smaller or equal to the *Global_Aru*.

In order to achieve minimum latency, an *ARU_update* message should be sent immediately upon a site receiving a Safe message at any of the daemons in the site. However, if one *ARU_update* is sent for every message by every site, the traffic on the network will increase linearly with the number of sites. Spread optimizes this by trading bandwidth for improved latency when the load of messages is low, and by sending the *ARU_update* after a number of messages have been received when the message load is high. The delay between two consecutive *ARU_updates* it is bounded by a certain timeout *delta*. For example, if five Safe messages are received within one *delta* time, only one *ARU_update* will be sent for an overhead of 20%, however, if a message is received and no *ARU_update* has been sent in the last *delta* interval, then an *ARU_update* is sent immediately. Note that this is a simplification of the actual implementation which piggybacks control information on data messages.

In practical settings, *delta* will be selected higher than the network diameter *Dn*. Therefore, the Safe Delivery latency is between $3 * Dn$ and $2 * delta + 2 * Dn$.

This could be optimized by including in the *ARU_update* the complete table with information about **all** of the currently connected sites, instead of just the three local values described above. This would reduce the Safe Delivery latency to be between

$2 * Dn$ and $delta + 2 * Dn$. However, this scheme is not scalable, since the size of the *ARU_update* will grow linearly with the number of sites. For a small number of sites (e.g. 10), this technique could be useful, but we elected not to report results based on it in order to preserve the scalability of the system (e.g. 100 sites).

A detailed example of how Safe Delivery is performed, is provided in Appendix 1.

4. The Replication Server

A good group communication system is unfortunately not sufficient to support consistent synchronous peer database replication. The replication server bridges the gap. The replication server is constructed with three main modules, as depicted in Figure 2.B in Section 2. Each of the modules is responsible for the following tasks: the Replication Engine consistently and efficiently orders actions; the database interceptor manages the user connections and the replication server connections to the database; the Semantics Optimizer optimizes the handling of actions based on the required application semantics and the current connectivity. Below we discuss these modules as they are used to replicate the Postgres database.

The Replication Engine

The specifics of the Replication Engine and its algorithm are discussed elsewhere [AT01, A95] and are not part of this paper. However, it is important to summarize the properties of the engine in order to understand how the replication server uses it.

The Replication Engine implements a complete and provably correct algorithm that provides global persistent consistent order of actions in a partitionable environment without the need for end-to-end acknowledgements on a per action basis. End-to-end acknowledgements are only used once for every network connectivity change event such as network partition or merge, or server crash or recovery. The engine uses Spread as the group communication toolkit.

The Replication Engine minimizes synchronous disk writes: it requires only a single synchronous disk write per action at the originating replica, just before the action is multicast (see Figure 1) via the group communication. Any peer replication scheme will have to include at least this operation in order to cope with a failure model that includes crashes and recoveries as well as network partitions and merges, and to allow a recovered database to work without connecting to the primary component first.

The above properties lead to high performance of the replication server: the throughput and latency of actions while in the primary component are mainly determined by the Safe Delivery performance of the group communication and are less influenced by other factors such as the number of replicas and the performance of the synchronous disk writes. Of course, the performance of actually applying updates to the database depends on the quality of the database and the nature of the updates.

In the presence of network partitions, the replication servers identify at most a single component of the servers group as the primary component. The other components of the partitioned server group are non-primary components. While in the primary component, actions are immediately applied to the database upon their delivery by the group communication according to the Safe Delivery service. While in a non-primary

component, they will be applied according to the decision made by the Semantic Optimizer, described below. Updates that are generated in non-primary components will propagate to other components as the network connectivity changes. These updates will be ordered in the final persistent consistent order upon the formation of the first primary component that includes them.

Every architecture that separates the replication functionality from the database manager needs to guarantee the *exactly once* semantics for updates. The difficulty occurs when the database manager crashes and the replication server does not know if the update was committed. One solution to this problem is to use the ordinal of the update assigned by the consistent persistent order. This ordinal will be stored in the database and updated as part of each update transaction. Upon recovery, the replication server can query the database for the highest ordinal transaction committed and re-apply actions with higher ordinal values. Since the database guarantees all-or-none semantics, this solution guarantees exactly once semantics.

The Replication Engine uses the Dynamic Linear Voting [JM90] quorum system to select the primary component. Dynamic Linear Voting allows the component that contains a weighted majority of the last primary component to form the next primary component. This quorum system performs well in practice [PL88], establishing an active primary component with high probability (better than weighted majority, for example).

The Replication Engine employs the propagation by eventual path technique, where replication servers that merge into a network component exchange their knowledge immediately upon the delivery of a membership notification. If the servers stay connected long enough to complete a re-conciliation phase, they share the same knowledge and will appear identical to clients. This technique is optimal in the sense that if no eventual path exists between two servers, no algorithm can share information between these servers. If such an eventual path exists between two servers, the Replication Engine will be able to propagate the information between them.

Employing the Dynamic Linear Voting quorum system and the propagation by eventual path techniques contributes to the high availability of our system.

The Postgres Interceptor

The interceptor module provides the link between the replication server and a specific database. Therefore, it is the only non-generic module of the architecture. The interceptor facilitates handling client connections, sends client messages to the database, gets back results from the database, and forwards the results back to the relevant clients.

Our interceptor was implemented for the Postgres version 7.1.3 database system. We intercept the clients that use the TCP interface to connect to the database. Once a client message is received, it is passed to the Semantics Optimizer that decides the next processing step, as described in the following subsection. The Postgres Interceptor reuses, with minor modifications, parts of the Postgres libpq library implementation in order to communicate with the database and the database clients.

In order to capture the client-database communication, the interceptor listens for client connections on the standard Postgres database port, while Postgres itself listens on a different private port, known only to the interceptor. The clients that attempt to connect

to Postgres will transparently connect to the interceptor, which will take care of redirecting their queries to the database and passing back the response. The interceptor can handle multiple clients simultaneously, managing their connections and identifying transactions initiated by a local client from transactions initiated by a client connected to a different replica. This method can be applied to any database manager with a documented client-server communication protocol.

When the interceptor receives an action from the Replication Engine, it submits the action to the database. Under normal execution, Postgres creates one process for each client connected to the database, taking advantage of the parallelism of transaction execution when there are no conflicts between the transactions issued by different clients.

Our architecture however, has to make sure that the order assigned to the transactions by the replication engine is **not** modified by this parallelization. Since we do not control the Postgres scheduler, as our implementation resides entirely outside of the database, in order to maintain the established ordering we need to know which actions can be parallelized. Our current implementation, benchmarked in the next section, ensures a correct execution by maintaining a single connection to the database on each replica, serializing the execution of transactions.

The performance could be improved by implementing a partial SQL parser that can allow us to parallelize the execution of local queries. Any such action, that does not depend on an update transaction issued by the same client and that was not yet executed, can be sent to the database independently of other transactions submitted by other clients (local or remote). For this purpose, the interceptor can maintain a pool of connections to the database where one connection is used for all update transactions (and thus maintains their order) and the others are shared by all local independent queries. This optimization is not implemented yet in our system.

A lower-level integration of our replication server inside the database manager could exploit the same parallelization as the database exploits. Of course, that would incur the price of losing the database transparency.

The Semantics Optimizer

The Semantics Optimizer provides an important contribution to the ability of the system to support various application requirements as well as to the overall performance.

In the strictest model of consistency, updates can be applied to the database only while in a primary component, when the global consistent persistent order of the action has been determined. However, read-only actions (queries) do not need to be replicated. A query can be answered immediately by the local database if there is no update pending generated by the same client. A significant performance improvement is achieved because the system distinguishes between queries and actions that also update the database. For this purpose the Semantics Optimizer implements a very basic SQL parser that identifies the queries from the other actions.

If the replica handling the query is not part of the primary component, it cannot guarantee that the answer of its local database reflects the current state of the system, as determined by the primary component. Some applications may require only the most updated information and will prefer to block until that information is available, while

others may be content with outdated information that is based on a prior consistent state (*weak consistency*), preferring to receive an immediate response. Our system allows each client to specify its required semantics individually, upon connecting to the system. Our system can even support such specification for each action but that will require the client to be aware of our replication service.

In addition to the strict consistency semantics and the standard weak consistency, our implementation supports, but is not limited to, two other types of consistency requirements: *delay updates* and *cancel actions,* where both names refer to the execution of updates/actions in a non-primary component. In the *delay updates* semantics, transactions that update the database are ordered locally, but are not applied to the database until their global order is determined. The client is not blocked and can continue submitting updates or even querying the local database, but needs to be aware that the responses to its queries may not yet incorporate the effect of its previous updates. In the *cancel actions* semantics a client instructs the Semantics Optimizer to immediately abort the actions that are issued in a non-primary component. This specification can also be used as a method of polling the availability of the primary component from a client perspective. These decisions are made by the Semantics Optimizer based on the semantic specifications that the client or the system setup provided.

The following examples demonstrate how the Semantics Optimizer determines the path of the action as it enters the replication server. After the Interceptor reads the action from the client, it passes it on to the Semantics Optimizer. The optimizer detects whether the action is a query and, based on the desired semantics and the current connectivity of the replica, decides whether to send the action to the Replication Engine, send it directly to the database for immediate processing, or cancel it altogether.

If the action is sent through the Replication Engine, the Semantics Optimizer is again involved in the decision process once the action is ordered. Some applications may request that the action is optimistically applied to the database once the action is locally ordered. This can happen either when the application knows that its update semantics is commutative (i.e. order is not important) or when the application is willing to resolve the possible inconsistencies that may arise as the result of a conflict. Barring these cases, an action is applied to the database when it is globally ordered.

5. Performance Evaluation

We evaluated the performance of our system in three different environments.

- A local area cluster at our lab at Johns Hopkins University. The cluster contains 14 Linux computers, each of which has the characteristics described by the Fog machine in Figure 3.
- The CAIRN wide-area network [CAIRN]. We used seven sites spanning the U.S.A. as depicted in Figure 4. CAIRN machines generally serve as routers for networking experiments. As a consequence, many of the CAIRN machines are weak machines with slow disks and not a lot of memory. The characteristics of each of the different CAIRN machines used in our evaluation is described in Figure 3. Especially note the low Postgres performance achieved by some of these machines without replication.

- The Emulab wide-area test-bed [Emulab]. Emulab[2] (the Utah Network Test-bed) provides a configurable test-bed where the network setup sets the latency, throughput and link-loss characteristics of each of the links. The configured network is then emulated in the Emulab local area network, using actual routers and in-between computers that emulate the required latency, loss and capacity constraints. We use 7 Emulab Linux computers, each has the characteristics described by the Emu machine in Figure 3. Most of our Emulab experiments emulated the CAIRN network depicted in Figure 4.

Our experiments were run using PostgreSQL version 7.1.3 standard installations. We use a database and experiment setup similar to that introduced by [KA00, JPKA00].

The database consists of 10 tables, each with 1000 tuples. Each table has five attributes (two integers, one 50 character string, one float and one date). The overall tuple size is slightly over 100 bytes, which yields a database size of more than 10MB. We use transactions that contain either only queries or only updates in order to simplify the analysis of the impact each poses on the system. We control the percentage of update versus query transactions for each experiment. Each action used in the experiments was of one of the two types described below, where `table-i` is a randomly selected table and the value of `t-id` is a randomly selected number:

```
update table-i set attr1="randomtext", int_attr=int_attr+4
                where t-id=random(1000);

select avg(float_attr), sum(float_attr) from table-i;
```

Before each experiment, the Postgres database was "vacuumed" to avoid side effects from previous experiments.

Machine	Processor	Memory (MB)	HDD (GB)	Postgres Updates/sec	Postgres Queries/sec
Local Area cluster					
Fog [1-14]	Dual PIII 667	256	9G SCSI	119.9	181.8
Cairn wide area testbed					
TISWPC	PII 400	128	4G IDE	37.26	77.49
ISIPC4	Pentium 133	64	6G IDE	25.44	11.97
ISIPC	PII 450	64	19G IDE	42.34	90.41
ISIEPC3	PII 450	64	6G IDE	50.10	93.41
ISIEPC	PII 450	64	19G IDE	43.52	92.72
MITPC2	Ppro 200	128	6G IDE	48.22	40.40
UDELPC	Ppro 200	128	4G SCSI	23.75	39.07
Emulab emulated wide-area testbed					
Emu[1-7]	PIII 850	512	40 G IDE	118.3	211.4

Figure 3: System Specification for the Three Test-beds

[2] Emulab is available at www.emulab.net and is primarily supported by NSF grant ANI-00-82493 and Cisco Systems.

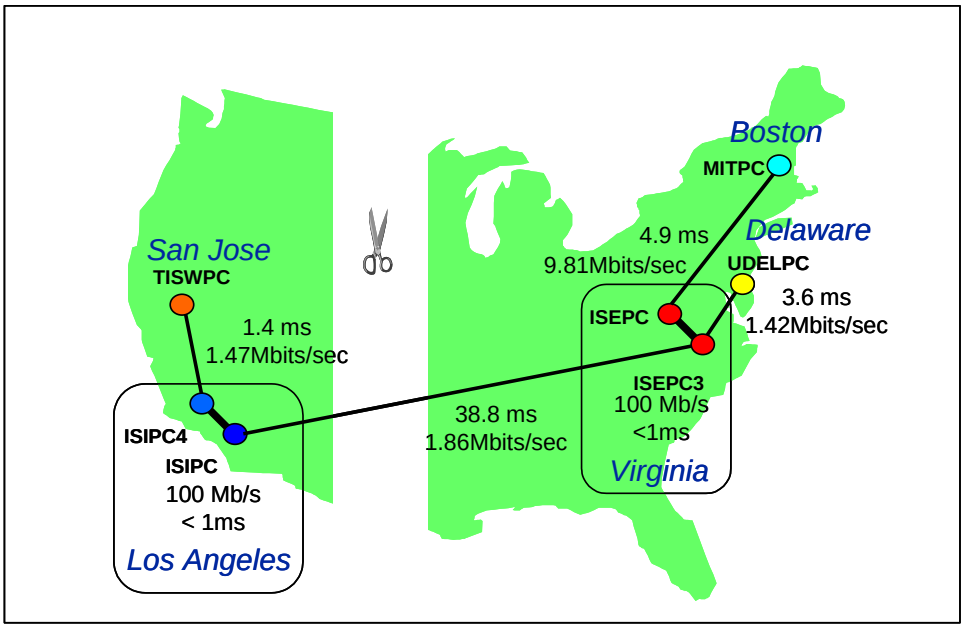

Figure 4: Layout of The CAIRN Test-bed

One of the difficulties in conducting database experiments is that real production database servers are very expensive and are not always available for academic research. We conducted our experiments on standard, inexpensive Intel PCs whose disk and memory performance is significantly poorer and that lack specialized hardware such as flash RAM logging disks. To evaluate the potential performance of our system on such hardware we conducted several tests either with Postgres not syncing its writes to disk (forced writes that the OS must not cache), or with both Postgres and our Replication Server not syncing the writes to disk. In all of these tests we also report the full-sync version. All tests that do not specify *no-sync*, or *replication server sync*, were conducted with full data sync on all required operations.

Each client only submits one action (update or query) at a time. Once that action has completed, the client can generate a new action.

Note that **any** database replication scheme has to guarantee that any query can be answered **only after** the effects of all of the updates preceding the query are reflected in the database. If the replication scheme is transparent, all the above updates have to be applied to the database before answering the query. Therefore, any such scheme is limited by the native speed of the database to execute updates. In our experiments, the local Fog and Emulab machines are limited to about 120 updates/sec as can be seen in Figure 3. Therefore any transparent replication method is limited to less than 120 updates/sec in a similar setting. Using our scheme to replicate a database with better performance could achieve better results as is indicated by the *replication server sync* tests, since our replication scheme is not the bottleneck.

First, we evaluate our system over the local area cluster defined in Figure 3. The first experiment tested the scalability of the system as the number of replicas of the database increased. Each replica executes on a separate Fog machine with one local client. Figure 5 shows five separate experiments. Each experiment used a different proportion of updates to queries.

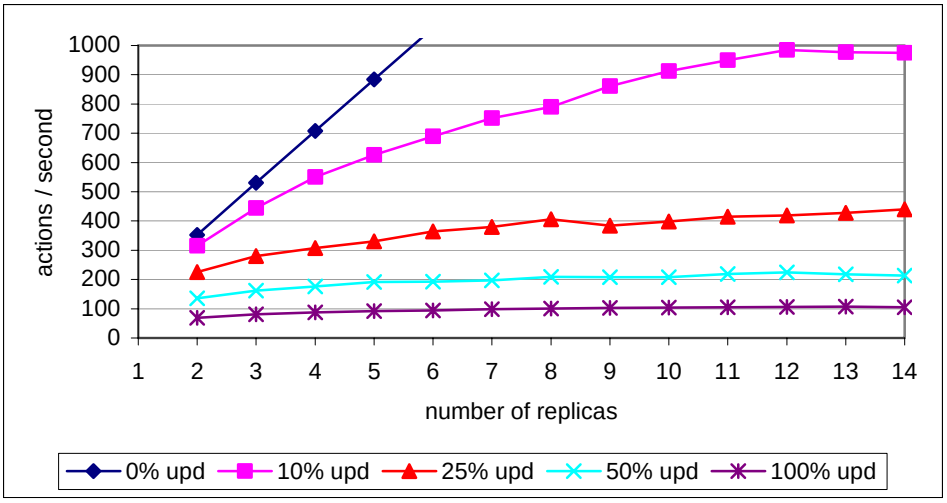

Figure 5: Max Throughput under a Varying Number of Replicas (LAN)

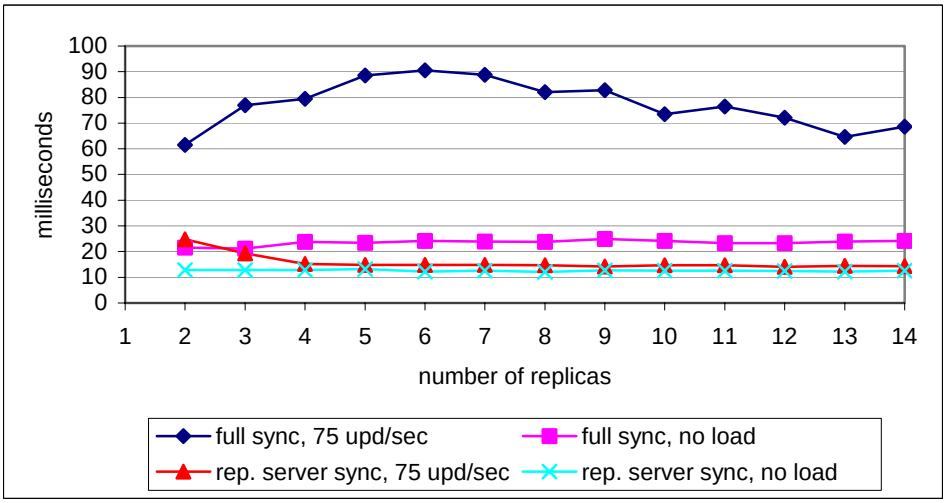

Figure 6: Latency of Updates (LAN)

The 100% update line shows the disk bound performance of the system. As replicas are added, the throughput of the system increases until the maximum updates per second the disks can support is reached – about 107 updates per second with replication (which adds one additional disk sync for each *N* updates, *N* being the number of replicas). Once the maximum is reached, the system maintains a stable throughput. The achieved updates per second, when the overhead of the Replication Server disk syncs are taken into account, matches the potential number of updates the machine is capable of as specified in Figure 3. The significant improvement in the number of sustained actions per second when the proportion of queries to updates increases is attributed to the Semantics Optimizer which executes each query locally, without any replication overhead. The maximum throughput of the entire system actually improves because each replica can handle an additional load of queries. The throughput with 100% queries increases linearly, reaching 2473 queries per second with 14 replicas.

In addition to system throughput, we evaluated the impact of replication on the latency of each update transaction both under no load and medium load of 75 updates per second. Figure 6 shows that the latency of update transactions does not increase as the number of replicas increases. When the system becomes more loaded, the latency per transaction caused by Postgres increases from around 23ms to an average of 77ms. It is interesting to note that the latency does not increase under higher load when Postgres sync is disabled, but the Replication Server does sync.

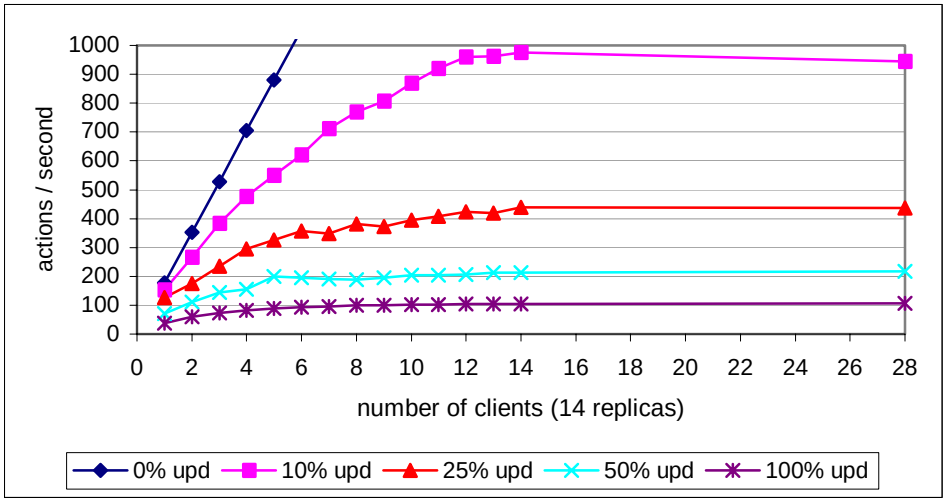

Figure 7: Throughput under Varied Client Set and Action Mix (LAN)

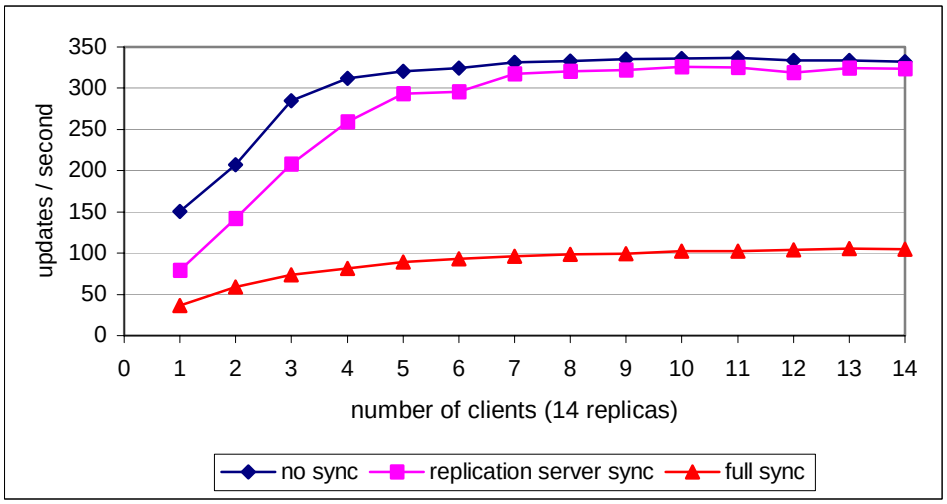

Figure 8: Throughput under Varied Client Set and Disk Sync Options (LAN)

The next two experiments fixed the number of replicas at 14, one replica on each Fog machine. The number of clients connected to the system was increased from 1 to 28, evenly spread among the replicas. In Figure 7 one sees that a small number of clients cannot produce maximum throughput for updates. The two reasons for this are: first, each client can only have one transaction active at a time, so the latency of each update limits the number of updates each client can produce per second. Second, because the Replication Server only has to sync updates generated by locally connected clients, the

work of syncing those updates is more distributed when there are more clients. Again, the throughput for queries increases linearly up to 14 clients (reaching 2450 in the 100% queries case), and is flat after that as each database replica is already saturated.

To test the capacity of our system for updates, we reran the same experiment as Figure 7, but only with 100% updates, under the three sync variations mentioned at the beginning of this section. This is depicted in Figure 8. Under full-sync the maximum throughput of 107 updates per second is reached, while with sync disabled for both Postgres and the Replication Server, we reach a maximum throughput of 336 updates per second. The interesting point is that when the Replication Server does updates with full-sync, the system still achieves 326 updates per second. This shows that as the number of replicas and clients increase, the Replication Server overhead decreases considerably. The cost of disk syncs when only a few clients are used is still noticeable.

We next evaluated our system on the CAIRN wide-area network depicted in Figure 4. The diameter of the network as measured by *ping*, is approximately 45ms involving seven sites and a tree of six links. These experiments validated that the system works as expected on a real operational network. The system was able to achieve the potential performance the hardware allowed, as presented in Figure 3.

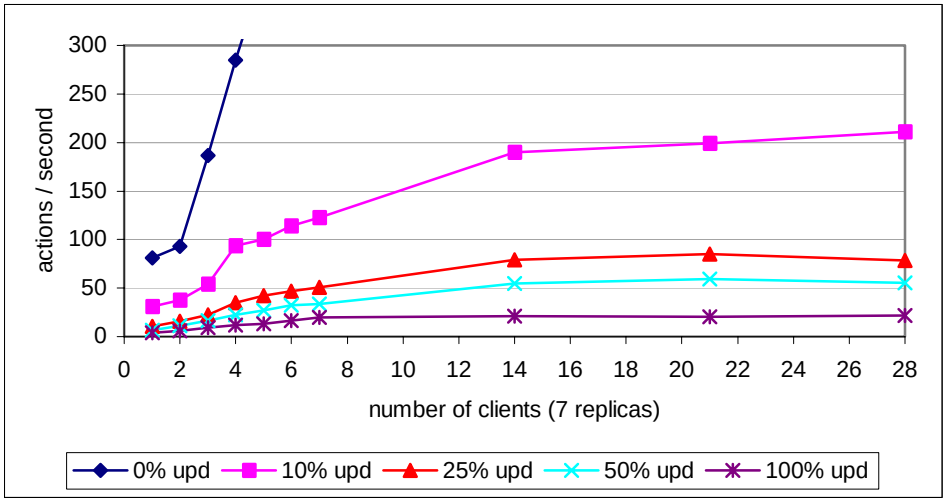

Figure 9: Throughput under Varied Client Set and Action Mix (CAIRN)

We ran the same basic experiment as done for Figure 7. In this experiment, however, the number of replicas is seven, because only seven sites were available. Figure 9 shows the throughput of the system as the number of clients connected to the system increases. Clients are evenly spread among the replicas. Because of the high network latency, more clients are required to achieve the potential maximum throughput when updates are involved. Also, the lines are not as smooth because the machines are very heterogeneous in their capabilities (ranging from an old Pentium 133 up to a Pentium II 450).

The latency each update experiences in the CAIRN network is 267ms under no load (measured at TISWPC in San Jose) and reaches 359ms at the point the system reaches maximum throughput of 20 updates per second. Once the throughput limit of the system is reached, the latency experienced by each client increases because more clients are sharing the same throughput.

As explained in the beginning of this section, the CAIRN machines were not adequate for our database performance tests because of their hardware limitations. We extended our wide-area tests by utilizing the Emulab facility. As accurately as possible, we emulated the CAIRN network on Emulab.

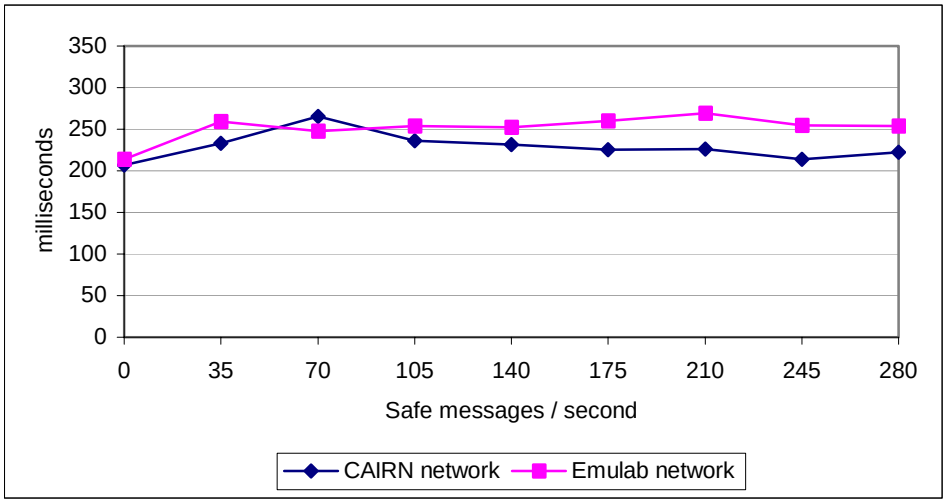

Figure 10: Latency of Safe Delivery on CAIRN and Emulab

Figure 10 shows the validation of Emulab against CAIRN by presenting the latency of Safe Delivery in the system under different Safe message load that resembles updates and queries in size. Under all loads both networks provided very similar message latency. Therefore, we believe that Emulab results are comparable to equivalent real-world networks running on the same machines.

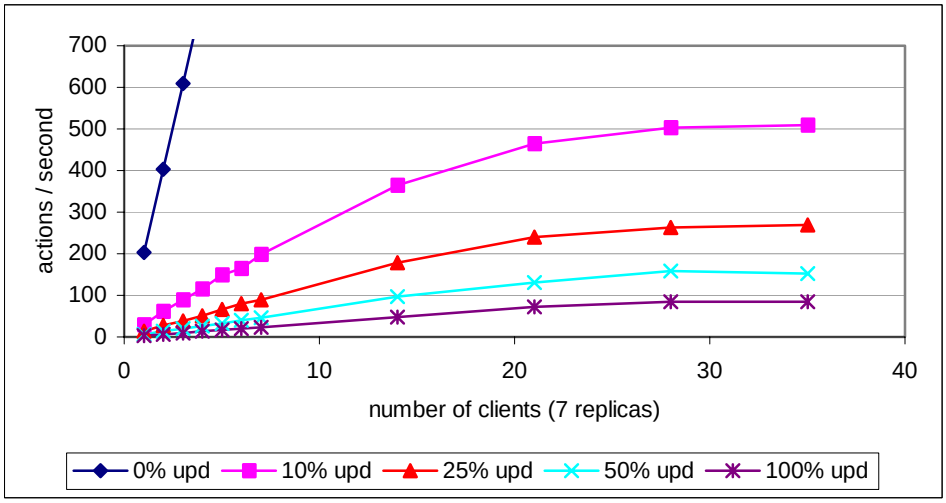

Figure 11: Throughput under Varying Client Set and Action Mix (Emulab)

The first experiment conducted on Emulab duplicated the experiment of Figure 9. In this case, the system was able to achieve a throughput close to that achieved on a local area network with similar number of replicas (seven), but with more clients as depicted in Figure 11. For updates, the maximum throughput achieved on Emulab is 85 updates per second (with about 28 clients in the system), compared with 98 updates per second on

LAN for the same number of replicas (with about 10 clients in the system) as can be seen in Figure 5. Although these results show very useful performance on a wide area network, we are not entirely happy with them since the algorithm predicts that the Emulab system should be able to reach similar maximum throughput for updates (as long as enough clients inject updates). We attribute the 14% difference to the fact that the Fog machines used in the LAN test are dual processor machines with SCSI disks, while the Emulab machines are single (though somewhat stronger) processor machines with IDE disks.

The latency each update experiences in this Emulab experiment is 268ms when no other load is present (almost identical to the corresponding CAIRN experiment above) and reaches 331ms at the point the system reaches maximum throughput of 85 updates per second. Again, once the throughput limit of the system is reached, the latency experienced by each client increases because more clients are sharing the same throughput.

For queries, similarly to the LAN tests in Figure 7, the performance increases linearly with the number of clients until the seven servers are saturated with seven clients at 1422 queries per second. The results for in-between mixes are as expected.

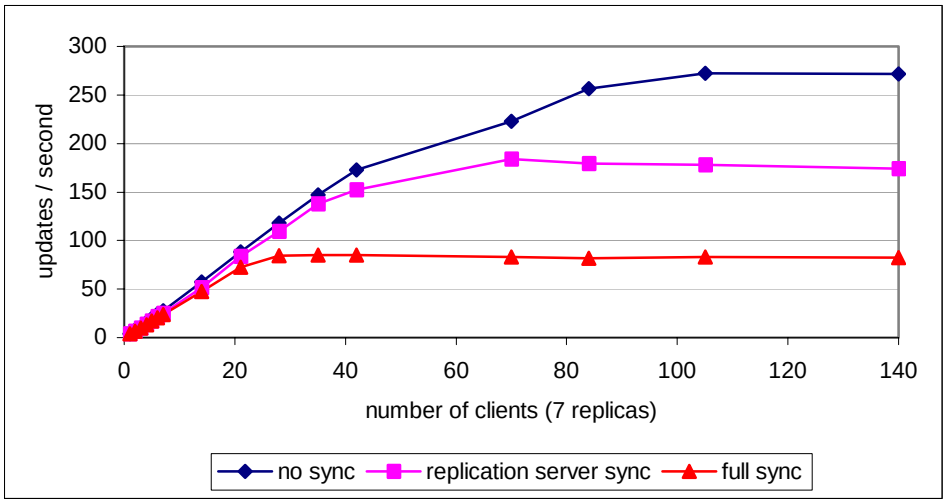

Figure 12: Throughput under Varied Client Set and Disk Sync Options (Emulab)

To test the capacity of our system for updates, we reran the same experiment as Figure 11, but just with 100% updates, under the three sync variations mentioned at the beginning of this section. Figure 12 shows that under full-sync the maximum throughput of 85 updates per second is reached, while with sync disabled for both Postgres and the Replication Server a maximum throughput of 272 updates per second is reached (with 105 clients in the system). When only the Replication Server syncs to disk, the system achieves 183 updates per second (with 70 clients in the system). The system performance does not degrade as clients are added.

To test the impact of network latency, we used Emulab to construct two additional networks, identical in topology to the CAIRN network, but with either one half or double the latency on each link. In effect, this explores the performance of the system as the

diameter of the network changes, as the original CAIRN network has a diameter of about 45ms, and the additional networks have about 22ms and 90ms diameters respectively. Figure 13 illustrates that under any of these diameters the system can achieve the same maximum throughput of 85 updates per second. However, as the diameter increases, more clients are required to achieve the same throughput.

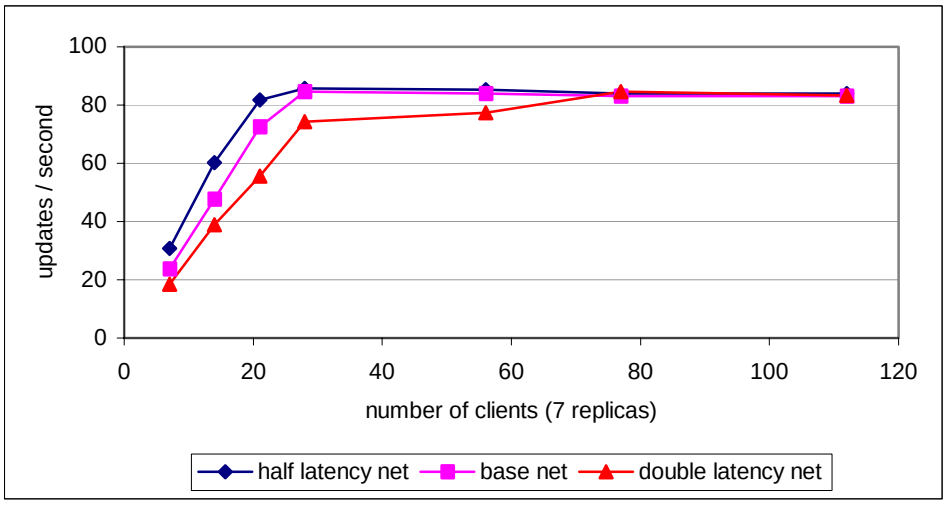

Figure 13: Throughput under Varied Network Latency (Emulab)

We conclude that the above results demonstrate that for many applications, peer synchronous database replication is becoming an attractive option not only for local area networks, but also for wide area networks.

6. Conclusions

One of the difficulties in providing efficient and correct wide area database replication is that it requires integrating different techniques from several research fields including distributed systems, databases, network protocols and operating systems. Not only does each of these techniques have to be efficient by itself, they all have to be efficient in concert with each other.

Some highlights of our results, replicating the Postgres database (that can perform about 120 updates per second without replication): For a local-area cluster with 14 replicas, the latency each update experiences is 27ms under zero throughput and 50ms under a throughput of 80 updates per second. The highest throughput in this setting is 106 updates per second.

For a wide-area network with a 45ms diameter and 7 replicas, the latency each update experiences is 268ms under zero throughput and 281ms under a throughput of 73 updates per second. The highest throughput in this setting is 85 updates per second, which is achieved with 28 clients and a latency of 331ms per update. When the diameter of the network is doubled to 90ms, the 85 updates per second throughput is maintained (although more clients are needed).

In all cases, the throughput for read-only actions approaches the combined power of all the replicas.

These results demonstrate the practicality of local and wide area peer (synchronous) database replication. Using our scheme to replicate a database with better performance could achieve better results since our replication architecture was not the bottleneck.

To achieve these results optimizations had to take place at various levels: First, at the network level, we had to optimize the latency of Safe Delivery on wide area networks. Second, we had to avoid end-to-end acknowledgments for each transaction while not compromising correctness of the system even in partitionable and crash-prone environments, and not delaying transaction execution. Third, we had to minimize the synchronous disk writes the replication server requires in addition to those performed by the database manager. Fourth, we had to obtain some semantic knowledge to correctly avoid replicating transactions that do not require it (e.g. read-only queries).

We show the feasibility of a database replication architecture that is transparent to both client and database manager, correctly handles arbitrary failures, and supports a number of different semantic guarantees for transactions such as one-copy serializability, weak consistency and delayed updates.

References

[A95] Y. Amir. Replication Using Group Communication Over a Partitioned Network. Ph.D. thesis, The Hebrew University of Jerusalem, Israel 1995. www.cs.jhu.edu/~yairamir.

[AADS01] Yair Amir, Baruch Awerbuch, Claudiu Danilov, and Jonathan Stanton. Flow control for many-to-many multicast: A cost-benefit approach. Technical Report CNDS--2001--1, Johns Hopkins University, Center for Networking and Distributed Systems, 2001.

[ADS00] Yair Amir, Claudiu Danilov, and Jonathan Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 327--336. IEEE Computer Society Press, Los Alamitos, CA, June 2000. FTCS 30.

[AMMB98] D.A. Agarwal, L.~E. Moser, P.~M. Melliar-Smith, and R.~K. Budhia. The totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems, 16(2):*93--132, May 1998.

[APS98] Yair Amir, Alec Peterson, and David Shaw. Seamlessly Selecting the Best Copy from Internet-Wide Replicated Web Servers. *Proceedings of the International Symposium on Distributed Computing (Disc98), LNCS 1499*, pages 22-33 Andros, Greece, September 1998.

[AS98] Yair Amir and Jonathan Stanton. The Spread wide area group communication system. Technical Report 98-4, Johns Hopkins University, CNDS, 1998.

[AT01] Yair Amir and Ciprian Tutu. From total order to database replication. *Proceedings of the International Conference on Distributed Computing Systems*, July 2002, to appear. Also, Technical Report CNDS-2001-6, Johns Hopkins University, November 2001, www.cnds.jhu.edu./publications.

[AW96] Y.Amir and A.Wool. Evaluating quorum systems over the internet. *Symposium on Fault-Tolerant Computing*, pages 26--35, 1996.

[CAIRN] http://www.cairn.net

[Demers87] A. Demers et al. Epidemic algorithms for replicated database maintenance. Fred B. Schneider, editor, *Proceedings of the 6th Annual {ACM} Symposium on Principles of Distributed Computing*, pages 1--12, Vancouver, BC, Canada, August 1987. ACM Press.

[DB2] http://www.ibm.com/software/data/db2/

[EMULAB] http://www.emulab.net

[GR93] J. N. Gray and A. Reuter. Transaction Processing: concepts and techniques. Data Management Systems. Morgan Kaufmann Publishers, Inc., 1993.

[H98] Mark Hayden. The Ensemble System. PhD thesis, Cornell University, 1998.

[HAE00] J. Holliday, D. Agrawal, and A. El Abbadi. Database replication using epidemic update.Technical Report TRCS00-01, University of California Santa-Barbara, 19, 2000.

[Informix] www.informix.com

[JM90] S. Jajodia and D. Mutchler. Dynamic Voting Algorithms for Maintaining the Consistency of Replicated Database. *ACM Trans. on Database Systems*, 15(2):230-280, June 1990.

[JPKA00] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters, Technical Report 2000

[KA00] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication. *Proceedings of the 26th International Conference on Very Large Databases (VLDB),* Cairo, Egypt, September 2000.

[KBB01] B. Kemme, A. Bartoli and O. Babaoglu. Online Reconfiguration in Replicated Databases Based on Group Communication. *In Proceedings of the International Conference on Dependable Systems and Networks (IC-DSN),* pages 117-126, Sweden, July 2001.

[KK00] Idit Keidar and Roger Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. *In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, p. 344--355, Taipei, Taiwan, April 2000.

[KSMD00] Idit Keidar, Jeremy Sussman, Keith Marzullo, and Danny Dolev. A client-server oriented algorithm for virtually synchronous group membership in {WAN}s. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems,* pages 356--365, Taipei, Taiwan, April 2000

[L78] L. Lamport. Time, Clocks, and The Ordering of Events in a Distributed System. *Comm. ACM*, 21(7), pages 558-565. 1978.

[MAMA94] L. E. Moser, Y. Amir, P. M. Melliar-Smith and D. A. Agarwal. Extended Virtual Synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56-65, June 1994.

[Mon00] Alberto Montresor. System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems.PhD thesis, Dept. of Computer Science, University of Bologna, Februrary 2000.

[MySQL] http://www.mysql.com/doc/R/e/Replication.html

[Oracle] http://www.oracle.com.

[PJKA00] M. Patino-Martinez and R. Jimenez-Peris and B. Kemme and G. Alonso. Scalable replication in database clusters. *Proceedings of 14th International Symposium on DIStributed Computing (DISC2000),* 2000

[PL88] J. F. Paris and D. D. E. Long. Efficient Dynamic Voting Algorithms. In *Proceedings of the 4th International Conference on Data Engineering*, pages 268-275, February 1988.

[Pgrep] pgreplicator.sourceforge.net

[Postgres] www.postgresql.com.

[Postgres-R] http://gborg.postgresql.org/project/pgreplication/projdisplay.php

[RBM96] Robbert~Van Renesse, Kenneth Birman, and S.~Maffeis. Horus: A flexible group communication system. *Communications of the ACM, 39(4):*76--83, April 1996

[Spread] www.spread.org.

[Sybase] www.sybase.com.

Appendix 1. Example of Safe Delivery in Action

Let's consider a network of five daemons N_1 – N_5 where each of the daemons represents a site. Figure A1 shows how the Safe Delivery mechanism works at daemon N_4, when a Safe message is sent by N_1, considering the worst case scenario when the latency from N_1 to N_4 is the diameter of the network. Initially all the daemons have all their variables initialized with zero (Figure A1.a).

Upon receiving the Safe message from N_1, the daemon N_4 updates its *Site_lts* (Figure A1.b). Assuming there are no losses and no other messages in the system, all the daemons will behave similarly, updating their *Site_lts* value. It takes one network diameter *Dn* for the message to get from N_1 to all the other daemons.

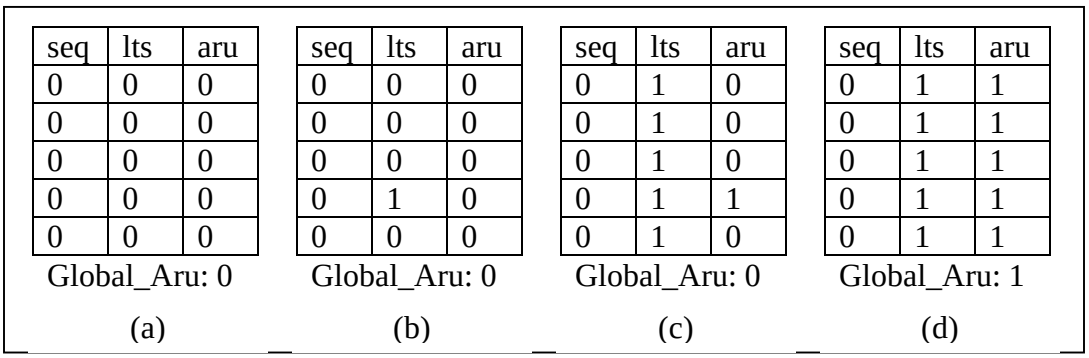

Figure A1: Scalable Safe Delivery for Wide Area Networks

After at most a *delta* interval, every daemon sends an *ARU_update* containing the row representing themselves in their matrix. Depending on the time each daemon waits until sending its Aru update, the daemons will receive information from all the other daemons in between *Dn* and *delta* + *Dn* time. Upon receiving these updates, daemon N_4 knows that all of the daemons increased their *Site_lts* to 1, and since it did not detect any loss, it can update its *Site_Aru* to 1 (Figure A1.c). Similarly, all the other daemons will update their *Site_Aru* values to 1, assuming there are no losses. At this point, N_4 knows that no daemon will create a message with a Lamport timestamp lower than 1 in the future, so according to total order delivery, it could deliver this message to the upper layer. However, this is not enough for Safe Delivery; N_4 does not know yet whether all of the daemons received all of the *ARU_updates*.

Already, at least *Dn* time has elapsed at N_1 (the farthest daemon from N_4) between the time it sent its last *Aru_update* and the time N_4 got it. Therefore, after waiting at most *delta* – *Dn* time, N_1 (as well as the other daemons) can send another *ARU_update* containing their *Site_Aru* value advanced to 1. Finally, after one more *Dn* time, when N_4 receives all these *Aru_updates*, it can advance its *Global_Aru* to 1 (Figure A1.d), and deliver the Safe message. The total latency in the worst case is *Dn* + (*delta* + *Dn*) + ((*delta* - *Dn*) + *Dn*), which is equal to 2 * *delta* + 2 * *Dn*.

Note that *ARU_updates* are periodic and cumulative, and as more Safe messages are sent in a *delta* interval, this delay will be amortized between different messages. However, the expected latency for a Safe message is at least 3 * *Dn*, as the delivery mechanism includes three rounds in this scalable approach.