Computing in the RAIN: A Reliable Array of Independent Nodes

Vasken Bohossian, Chenggong C. Fan, *Student Member*, *IEEE*, Paul S. LeMahieu, Marc D. Riedel, Lihao Xu, *Member*, *IEEE*, and Jehoshua Bruck, *Fellow*, *IEEE*

Abstract—The RAIN project is a research collaboration between Caltech and NASA-JPL on distributed computing and data storage systems for future spaceborne missions. The goal of the project is to identify and develop key building blocks for reliable distributed systems built with inexpensive off-the-shelf components. The RAIN platform consists of a heterogeneous cluster of computing and/or storage nodes connected via multiple interfaces to networks configured in fault-tolerant topologies. The RAIN software components run in conjunction with operating system services and standard network protocols. Through software-implemented fault tolerance, the system tolerates multiple node, link, and switch failures, with no single point of failure. The RAIN technology has been transfered to Rainfinity, a start-up company focusing on creating clustered solutions for improving the performance and availability of Internet data centers. In this paper, we describe the following contributions: 1) fault-tolerant interconnect topologies and communication protocols providing consistent error reporting of link failures, 2) fault management techniques based on group membership, and 3) data storage schemes based on computationally efficient error-control codes. We present several proof-of-concept applications: a highly-available video server, a highly-available Web server, and a distributed checkpointing system. Also, we describe a commercial product, Rainwall, built with the RAIN technology.

Index Terms—Distributed computing, scalable architectures, interconnection networks, fault tolerance, data storage, cluster computing.

1 Introduction

The Reliable Array of Independent Nodes (RAIN) project is a research collaboration between Caltech's Parallel and Distributed Computing Group and the Jet Propulsion Laboratory's Center for Integrated Space Microsystems, in the area of distributed computing and data storage systems for future spaceborne missions. The goal of the project is to identify and develop key building blocks for reliable distributed systems built with inexpensive off-the-shelf components.

The general hardware platform for the RAIN system is a heterogeneous cluster of computing and/or storage nodes connected via multiple interfaces through a network of switches. A diagram of a possible system configuration is shown in Fig. 1. Our testbed at Caltech consists of 10 Pentium workstations running the Linux operating system, each with two network interfaces. These are connected via four eight-way Myrinet switches [10]. Note, however, that the RAIN software is not tied to a particular hardware platform, operating system, or network type.

- *V. Bohossian is with Rainfinity, 87 N. Raymond Ave., Suite 200, Pasadena, CA 91103. E-mail: vincent@rainfinity.com.*
- *C.C. Fan, P.S. LeMahieu, M.D. Riedel, and J. Bruck are with the California Institute of Technology, Mail Code 136-93, Pasadena, CA 91125. E-mail: {fan, lemahieu, riedel, bruck} @paradise.caltech.edu.*
- *L. Xu is with the Department of Computer Science, Washington University, One Brookings Drive, St. Louis, MO 63130. E-mail: lihao@cs.wustl.edu.*

Manuscript received 1 Mar. 2000; revised 1 Aug. 2000; accepted 15 Aug. 2000.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 112988.

The RAIN system consists of a collection of software modules that run in conjunction with operating system services and standard network protocols, as illustrated in Fig. 2. Through software-implemented fault tolerance, the RAIN system tolerates multiple node, link, and switch failures with no single point of failure. In addition to reliability, the RAIN architecture is scalable and dynamically reconfigurable, and permits the efficient use of network resources, such as multiple data paths and redundant storage, with graceful degradation in the presence of faults.

We have identified the following key building blocks for distributed computing systems:

- *Communication*: fault-tolerant interconnect topologies and reliable communication protocols. We describe network topologies that are resistant to partitioning, and a protocol guaranteeing a consistent history of link failures. We also describe an implementation of the MPI standard [49] on the RAIN communication layer.
- *Fault Management*: techniques based on group membership. We describe an efficient token-based protocol that tolerates node and link failures.
- *Storage*: distributed data storage schemes based on error-control codes. We describe schemes that are optimal in terms of storage, as well as encoding/decoding complexity.

We present three proof-of-concept applications based on the RAIN building blocks:

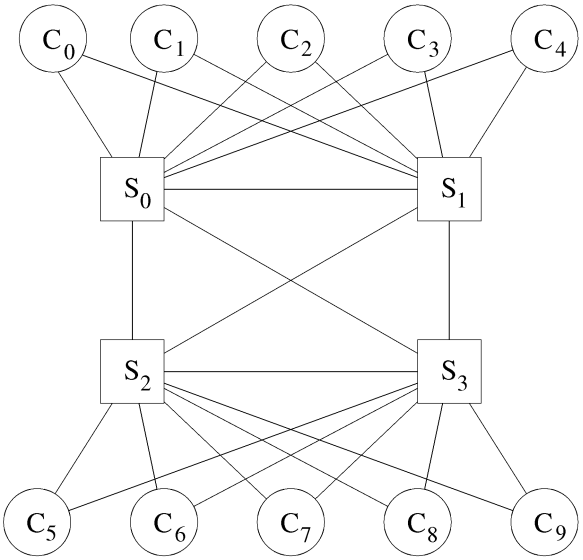

Fig. 1. Example of the RAIN hardware platform (C = Computer, S = Switch.)

- A video server based on the RAIN communication and data storage components.
- A Web server based on the RAIN fault management component.
- A distributed checkpointing system based on the RAIN storage component, as well as a leader election protocol.

This paper is intended as an overview of our work on the RAIN system. Further details of our work on fault-tolerant interconnect topologies may be found in [36] and [44]; on a consistent-history protocol in [37]; on the group membership problem in [30]; on a leader election protocol in [29]; and on data storage schemes in [55], [56], and [57].

We note that the RAIN technology has been transfered to Rainfinity, a start-up company focusing on creating clustered solutions for improving the performance and availability of Internet data centers [41].

1.1 Related Work

Cluster computing systems, such as the NOW project at the University of California, Berkeley [2] and the Beowulf project [3], have shown that networks of workstations can rival supercomputers in computational power. Packages, such as PVM [51], [52], and MPI [31], [49], are widely used for parallel programming applications. There have been numerous projects focusing on various aspects of fault management and reliability in cluster computing systems. Well-known examples are the Isis [6] and Horus [54] systems at Cornell University, the Totem system at the University of California, Santa Barbara [1], [40], and the Transis system at the Hebrew University of Jerusalem [19], [20]. Projects focusing on fault tolerance through process replication and rollback-recovery include the Manetho project at Rice University [25] and the DOME project at Carnegie Mellon University [4], [18]. RAID techniques are widely used for performance and reliability in storage systems [17]. Well-known projects in reliable distributed storage include the Zebra, CODA, and Scotch file systems [32], [47], [26].

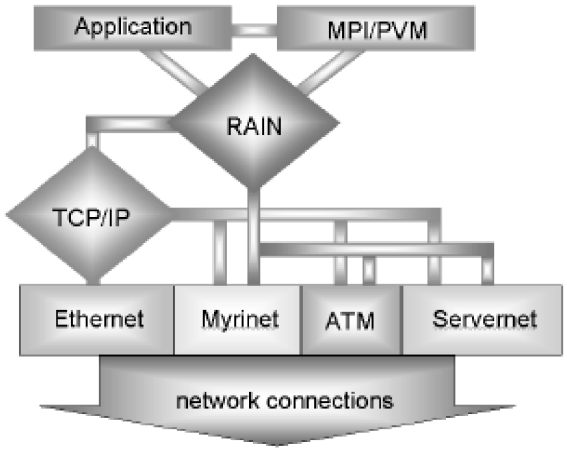

Fig. 2. RAIN software architecture.

1.2 Novel Features of RAIN

The RAIN project incorporates many novel features in an attempt to deal with faults in nodes, networks, and data storage.

1. **Communication**: Since the network is frequently a single point of failure, RAIN provides fault tolerance in the network via the following mechanisms:
   - *Bundled interfaces*: Nodes are permitted to have multiple interface cards. This not only adds fault tolerance to the network, but also gives improved bandwidth.
   - *Link montioring*: To correctly use multiple paths bewteen nodes in the presence of faults, we have developed a link-state monitoring protocol that provides a consistent history of the link state at each endpoint.
   - *Fault-tolerant interconnect topologies*: Network partitioning is always a problem when a cluster of computers must act as a whole. We have designed network topologies that are resistant to partitioning as network elements fail.
2. **Group membership**: A fundamental part of fault management is identifying which nodes are healthy and participating in the cluster. We give a new protocol for establishing group membership.
3. **Data storage**: Fault tolerance in data storage over multiple disks is achieved through redundant storage schemes. Novel error-correcting codes have been developed for this purpose. These are *array codes* that encode and decode using simple XOR operations. Traditional RAID codes generally only allow mirroring or parity (i.e., one degree of fault tolerance) as options. Array codes can be thought of as data partitioning schemes that allow one to trade storage requirements for fault tolerance. These codes exhibit optimality in the storage requirements, as well as in the number of update operations needed. Although some of the original motivation for these codes came from traditional RAID systems, these schemes apply equally well to partitioning data over disks on distinct nodes (as in

our project), or even partitioning data over disks at remote geographic locations.

1.3 Organization

This paper is organized as follows: In Section 2, we discuss fault-tolerant interconnect topologies and the communication protocol. In Section 3, we present the group membership protocol. In Section 4, we discuss error-control codes and reliable storage. In Sections 5 and 6, we describe proof-of-concept applications and a commercial product, Rainwall, built with the RAIN technology. Finally, in Section 7, we present conclusions and directions of future work.

2 Communication

The RAIN project addresses fault tolerance in the network with fault-tolerant interconnect topologies and with bundled network interfaces.

2.1 Fault-Tolerant Interconnect Topologies

We were faced with the question of how to connect compute nodes to switching networks in order to maximize the network's resistance to partitioning. Many distributed computing algorithms face trouble when presented with a large set of nodes that have become partitioned from the others. A network that is resistant to partitioning should lose only some constant number of nodes (with respect to the total number of nodes) given that we do not exceed some number of failures. After additional failures, we may see partitioning of the set of compute nodes, i.e., some fraction of the total number of compute nodes may be lost. By carefully choosing how we connect our compute nodes to the switches, we can maximize a system's ability to resist partitioning in the presence of faults.

Our main contributions are:

1. a construction for degree-2 compute nodes connected by a ring network of switches of degree 4 that can tolerate any 3 switch failures without partitioning the nodes into disjoint sets,
2. a proof that this construction is optimal in the sense that no construction can tolerate more switch failures while avoiding partitioning, and
3. generalizations of this construction to arbitrary switch and node degrees and to other switch networks, in particular, to a fully-connected network of switches.

See [36] for our full paper on the work described in this section.

2.1.1 Previous Fault-Tolerant Interconnect Work

The construction of fault-tolerant networks was studied in 1976 by Hayes [33]. This paper looked primarily at constructing graphs that would still contain some target graph as a subgraph even after the introduction of some number of faults. For example, it explored the construction of k-FT rings that still contain a ring of the given size after the introduction of k faults.

Other papers that address the construction of fault-tolerant networks are [14] for fault-tolerant rings, meshes, and hypercubes, [11], [12], [13] for rings and other circulants, and [21], [22], [23] for trees and other fault-tolerant systems.

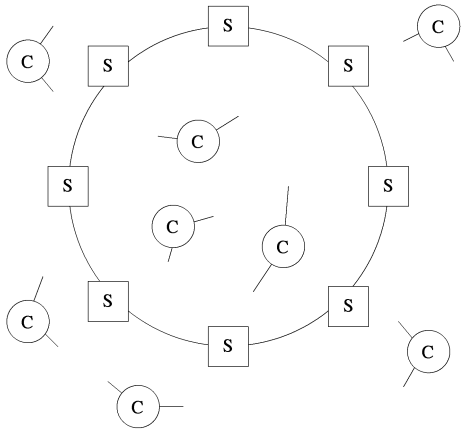

Fig. 3. How to connect n compute nodes to a ring of n switches?

A recent paper by Ku and Hayes [35] looks at an issue similar to the one covered in this paper. In particular, it discusses maintaining connectivity among compute nodes connected by buses. This is equivalent to not permitting any switch-to-switch connections in our model. We are looking at permitting such switch-to-switch connections to allow the creation of useful switch topologies and then connecting compute nodes to this network of switches.

2.1.2 The Problem

We look at the following problem: Given n switches of degree d_s connected in a ring, what is the best way to connect n compute nodes of degree d_c to the switches to minimize the possibility of partitioning the compute nodes when switch failures occur? Fig. 3 illustrates the problem.

2.1.3 A Naïve Approach

At first glance, Fig. 4a may seem like a solution to our problem. In this construction, we simply connect the compute nodes to the nearest switches in a regular fashion. If we use this approach, we are relying entirely on fault tolerance in the switching network. A ring is 1-fault-tolerant for connectivity so we can lose one switch without upset. A second switch failure can partition the switches and, thus, the compute nodes, as in Fig. 4b. This prompts the study of whether we can use the multiple connections of the compute nodes to make the network more resistant to partitioning. In other words, we want a construction where the connectivity of the nodes is maintained even after the switch network has become partitioned.

2.1.4 Diameter Construction $d_c = 2$

The intuitive, driving idea behind this construction is to connect the compute nodes to the switching network in the most *nonlocal* way possible. That is, connect a compute node to switches that are maximally distant from each other. This idea can be applied to arbitrary compute node degree d_c, where each connection for a node is as far apart as possible from its neighbors.

We call this the *diameter solution* because the maximally distant switches in a ring are on opposite sides of the ring, so a compute node of degree 2 connected between them

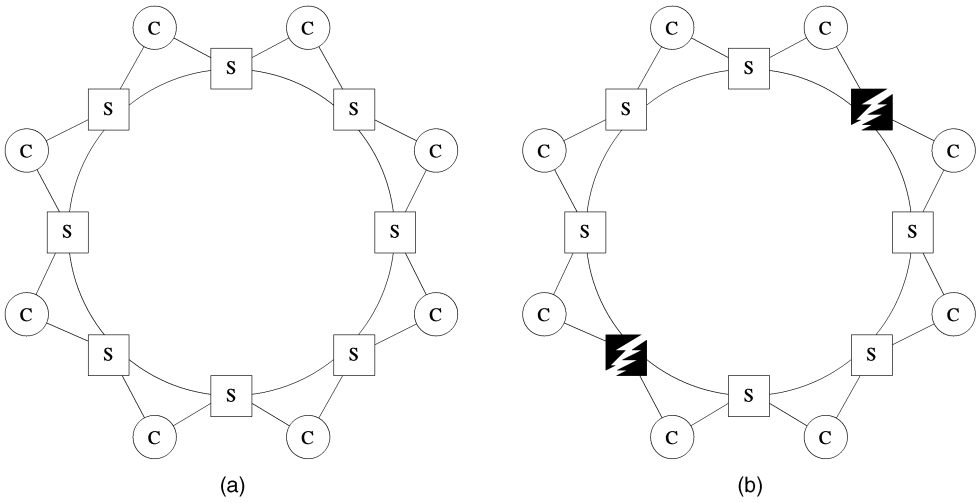

Fig. 4. (a) A naïve approach, $d_c = 2$. (b) Notice that it is easily partitioned with two switch failures.

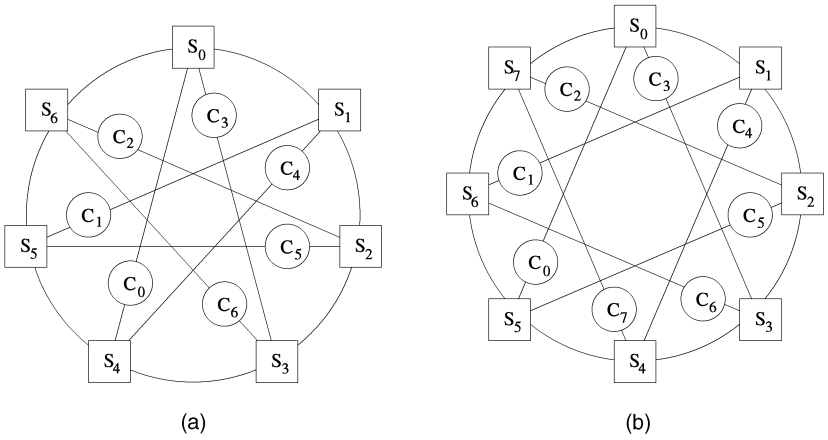

Fig. 5. (a) Diameter construction for n odd. (b) Diameter construction for n even.

forms a diameter. We actually use the switches that are one less than the diameter apart to permit n compute nodes to be connected to n switches with each compute node connected to a unique pair of switches.

Construction 2.1 (Diameters). *Let $d_s = 4$ and $d_c = 2$. $\forall i$, $0 \leq i < n$, label all compute nodes c_i and switches s_i. Connect switch s_i to $s_{(i+1) \bmod n}$, i.e., in a ring. Connect node c_i to switches s_i and $s_{(i+\lfloor n/2 \rfloor + 1) \bmod n}$. See Fig. 5 for an example for n odd and n even.*

Note: Although Construction 2.1 is given for an identical number of compute nodes and switches, we can add additional compute nodes by repeating the above process. In this case, we would connect node c_j to the same switches as node $c_{j \bmod n}$. All the following results still hold, with a simple change in constants. For example, when we connect 10 nodes to 10 switches, we have a maximum loss of six nodes with the occurrence of three faults. Increasing the number of nodes to $3n = 30$ triples the maximum number of nodes lost with three faults to 18. This is also true of the generalized diameters construction in [36]. The maximum number of lost nodes is still constant with respect to n, the number of switches. The addition of extra nodes to the ring constructions affects only this constant in our claims. The asymptotic results about resistance to partitioning are all still valid.

Theorem 2.1. *Construction 2.1 creates a graph of n compute nodes of degree $d_c = 2$ connected to a ring of n switches of degree $d_s = 4$ that can tolerate three faults of any kind (switch, link, or node) without partitioning the network. Thus, only a constant number of nodes (with respect to n) will be lost. In this case, that constant is $\min(n, 6)$ lost nodes. This construction is optimal in the sense that no construction connecting n compute nodes of degree $d_c = 2$ to a ring of switches of degree $d_s = 4$ can tolerate any arbitrary four faults without partitioning the nodes into sets of nonconstant size (with respect to n).*

The proof of this theorem is given in [36]. The latter paper also presents a generalization to nodes of degree larger than two, as well as a clique network instead of the ring shown above.

2.2 Consistent-History Protocol for Link Failures

When we bundle interfaces together on a machine and allow links and network adapters to fail, we must monitor available paths in the network for proper functioning. In [36], we give a modified *ping* protocol that guarantees that each side of the communication channel sees the same

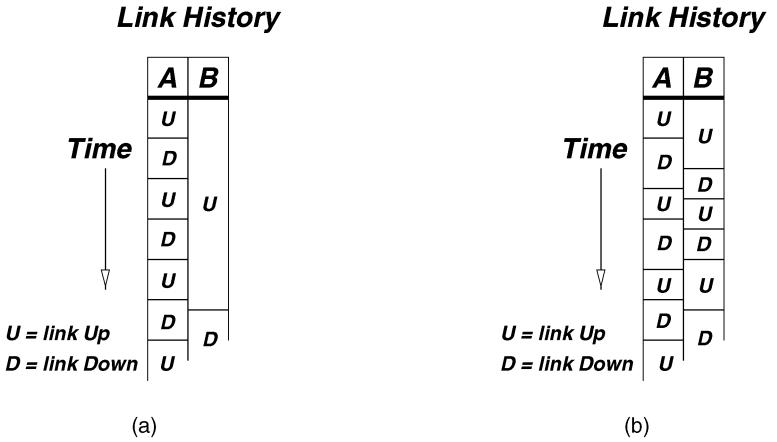

Fig. 6. (a) Node A sees many more transactions than node B. (b) Node A and B see tightly coupled views of the channel.

history. Each side is limited in how much it may lead or lag the other side of the channel, giving the protocol *bounded slack*. This notion of identical history can be useful in the development of applications using this connectivity information. For example, if an application takes error recovery action in the event of lost connectivity, it knows that both sides of the channel will see the exact same behavior on the channel over time and, thus, will take the same error recovery action. Such a guarantee may simplify the writing of applications using this connectivity information.

Our main contributions are: 1) a simple, *stable* protocol for monitoring connectivity that maintains a *consistent history* with *bounded slack* and 2) proofs that this protocol exhibits *correctness*, *bounded slack*, and *stability*.

See [37] for our full paper on the work described in this section.

2.2.1 Previous Link-State Work

Although this is not the consensus problem, it is still useful to look at past work on consensus, such as Fischer et al. in [28], or in Lynch's book [38]. The connectivity problem has been addressed with different goals by Rodeheffer and Schroeder in the Autonet system [45], [46]. They were concerned with adaptive rates and skepticism in judging the quality of a link, whereas we are concerned with consistency in reporting the quality of a link. Birman [7] gives general motivation for consistency in failure reporting for the purpose of improving reliability of distributed systems.

2.2.2 Precise Problem Definition

We now present all the requirements of the protocol:

- *Correctness*: The protocol will eventually correctly reflect the true state of the channel. If the channel ceases to perform bidirectional communication (at least one side sees time-outs), both sides should eventually mark the channel as *Down*. If the channel resumes bidirectional communication, both sides should eventually mark the channel as *Up*.
- *Bounded Slack*: The protocol will ensure that a maximum slack of N exists between the two sides. Neither side will be allowed to lag or lead the other by more than N transitions. (See Fig. 6.)
- *Stability*: Each real channel event (i.e., time-out) will cause at most some bounded number of observable state transitions, preferably, one at each endpoint.

The system model is one in which nodes do not fail, but links intermittently fail. The links must be such that a sliding window protocol can function. See the discussion on data link protocols by Lynch in [38]. Note that this protocol may still be used in a system where node failures are allowed. However, it is the job of the application using the protocol to deal with the concept of node failure via checkpointing and roll-back, or some other mechanism.

2.2.3 The Protocol

This protocol uses *reliable message passing* to ensure that nodes on opposing ends of some faulty channel see the same state history of link failure and recovery. The reliable message passing can be implemented using a sliding window protocol, as mentioned above. At first, it may seem odd to discuss monitoring the status of a link using reliable messages. However, it makes the description and proof of the protocol easier, preventing us from essentially reproving sliding window protocols in a different form. For an implementation, there is no reason to actually build the protocol on an existing reliable communication layer. The protocol can be easily implemented on top of ping messages (sent unreliably) with only a sequence number and acknowledge number as data (in other words, we can easily map reliable messaging on top of the ping messages).

The protocol consists of two parts:

- First, we have the sending and receiving of tokens using reliable messaging. Tokens are conserved, neither lost nor duplicated. Tokens are sent whenever a side sees an observable channel state transition. The observable channel state is whether the link is seen as *Up* or *Down*. The token-passing part of the protocol essentially *is* the protocol. Its job is to ensure that a consistent history is maintained.
- Second, we have the sending and receiving of ping messages using unreliable messaging. The sole purpose of the pings is to detect when the link can

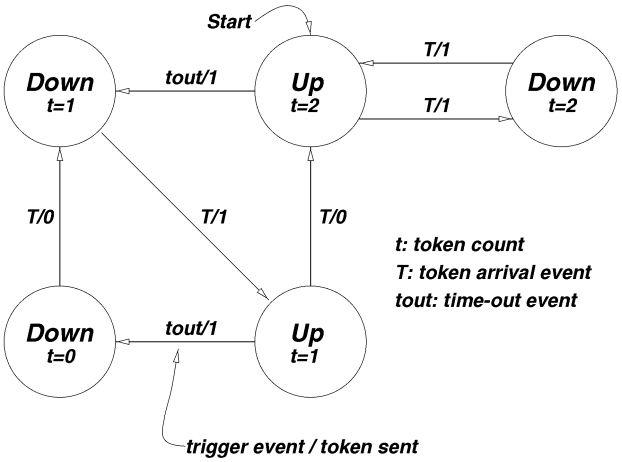

Fig. 7. State machine for the connectivity protocol, slack $N = 2$.

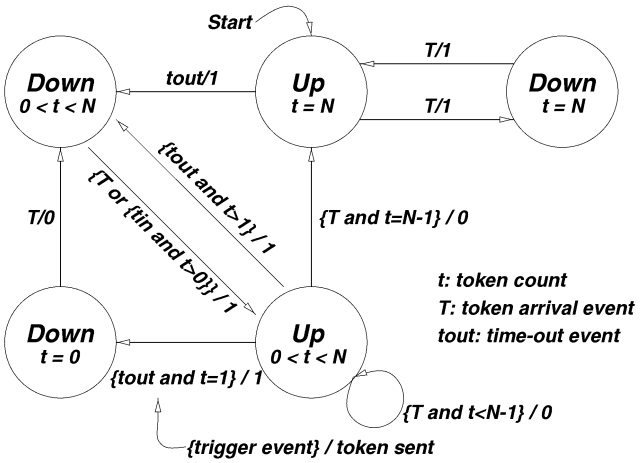

Fig. 8. State machine for the connectivity protocol, general slack N.

be considered Up or $Down$. This part of the protocol would not necessarily have to be implemented with pings but could be done using other hints from the underlying system. For example, hardware could give instant feedback about its view of link status. For all the proofs to be valid, we must have that a t_{out} is generated when bidirectional communication has (probably) been lost, and a t_{in} is generated when bidirectional communication has (probably) been reestablished.

The token-passing part of the protocol maintains the consistent history between the sides, and the pings give information on the current channel state. The token-passing protocol can be seen as a filter that takes raw information about the channel and produces channel information guaranteed to be (eventually) consistent at both ends. The state machines in Fig. 7 and Fig. 8 describe how each side of the protocol functions in the total system for $N = 2$.

In Section 2.3, we describe the protocol for a slack of $N = 2$ and, in 2.4, we do so for a general slack of N.

2.3 Slack $N = 2$

Here, we describe the protocol for the base case where we have slack of $N = 2$. This is a significant case since it is the smallest value of slack for which any such protocol can work. Its description is somewhat simpler than the general case. A state machine, as described in Fig. 7, runs at each end of the link, at each node.

Intuitively, the state machine of Fig. 7 shows the reaction to t_{out} events and T (token-receipt) events by the node at one end of the communication channel. The number of tokens currently held is t, and $2 - t$ is then the number of unacknowledged transitions the node has made. Note that $2 - t$ is at most 2, corresponding to the slack bound of two. The states can be described as follows:

1. $Up(t = 2)$: The node is in the stable state. No unacknowledged transitions have been made by this node.
2. $Down(t = 2)$: The node is catching-up with a transition seen by the other node that it itself did not see via a time-out. No unacknowledged transitions have been made by this node.
3. $Down(t = 1)$: The node has seen a time-out and marked the channel as down. One unacknowledged transition has been made by this node ($Up \rightarrow Down$).
4. $Up(t = 1)$: The node has received acknowledgement (via a received token) for the $Up \rightarrow Down$ transition. One unacknowledged transition has been made by this node ($Down \rightarrow Up$).
5. $Down(t = 0)$: The node has seen a time-out and marked the channel as down and is now blocked from further transitions by the bounded-slack constraint. Two unacknowledged transitions have been made by this node ($Down \rightarrow Up \rightarrow Down$).

In Fig. 7, each state is characterized by whether the node sees the channel as Up or $Down$, and how many tokens t are held by the node. The state transitions are labeled by the *action triggering the transition* and the *action taken upon transition*. A trigger event is either a time-out t_{out} or receipt of a token T. The action taken is always whether a token is sent (1) or not (0). Note that a token T is sent whenever a transition for an Up state to a $Down$ state or from a $Down$ state to an Up state is made.

2.4 General Slack N

Here, we describe the protocol for the general case where we have some arbitrary slack value, N. A state machine, as described Fig. 8, runs at each end of the communication at each node. This description of the state machine tries to preserve the same structure as the $N = 2$ machine described above for the purpose of the proofs.

Each state in Fig. 8 is characterized by whether the node sees the channel as Up or $Down$, and how many tokens t are held by the node. There is an implicit action for each transition: If a token is received, the token count t is incremented; if a token is sent, the token count t is decremented. The state transitions are labeled by the pair *{action triggering the transition} / {action taken upon transition}*. A trigger event is either a time-out t_{out}, a time-in t_{in}, or receipt of a token T. The action taken is a combination of sending tokens and adjusting the token count t.

Notice that here we've introduced the t_{in} event, which was not present in the $N = 2$ case. This is the complement to the t_{out} event that becomes meaningful for higher slack. A

t_{in} corresponds to a hint from some lower level that the communication link is up and running. For an implementation where tokens are mapped on top of pings, we would never explicitly see a t_{in} event since the latest token information comes with each ping response. However, if tokens were not mapped on top of pings or, if other sources of information about the communication link were also possible, t_{in} events make sense and as such are allowed.

For the protocol details and proofs of correctness, bounded slack, and stability, see [37].

2.5 A Port of MPI

A port of MPI [49] (using the MPICH implementation from Argonne Labs [31]) was done on the RAIN communication layer. This port involved creating a new communications device in the MPICH framework, essentially adapting the standard communication device calls of MPICH to those presented by the RAIN communication layer called RUDP (Reliable UDP). RUDP is a datagram delivery protocol that monitors connectivity to remote machines using the consistent history link protocol explained in this paper and presented in detail in [37]. The port to MPI was done to facilitate our own analysis and use of the RAIN communication layer.

MPI is *not* a fault-tolerant API and, as such, the best we can do is mask network errors to the extent redundant hardware has been put in place. For example, if all machines have two network adaptors and one link fails, the MPI program will proceed as if nothing had happened. If a second link fails, the MPI application may hang until the link is restored. There is no possibility of returning errors related to link connectivity in the MPI communications API. Thus, although the RUDP communication layer knows of the loss of connectivity, it can do nothing about it and must wait for the problem to be resolved.

The implementation itself has a few notable features:

- It allows individual networking components to fail up to the limit of the redundancy put into the network.
- It provides increased network bandwidth by utilizing the redundant hardware.
- It runs entirely in user space. This has the important impact that all program state exists entirely in the running process, its memory stack, and its open file descriptors. The result is that if a system running RUDP has a checkpointing library, the program state (including the state of all communications) can be transparently saved without having to first synchronize all messaging. The communications layer only uses the kernel for unreliable packet delivery and does not rely on any kernel state for reliable messaging.
- It illustrates that an experimental communication library can make the step to a practical piece of software easily in the presence of standards such as MPI.

The MPI port to RUDP has helped us use our own communication layer for real applications, has helped us argue the importance of keeping program state out of the kernel for the purposes of transparent checkpointing, and has highlighted the importance of programming standards such as MPI.

3 Group Membership

Tolerating faults in an asynchronous distributed system is a challenging task. A reliable group membership service ensures that the processes in a group maintain a consistent view of the global membership.

In order for a distributed application to work correctly in the presence of faults, a certain level of agreement among the nonfaulty processes must be achieved. There are a number of well-defined problems in an asynchronous distributed system, such as consensus, group membership, commit, and atomic broadcast that have been extensively studied by researchers. In the RAIN system, the group membership protocol is a critical building block. It is a difficult task, especially when a change in the membership occurs, either due to failures or to voluntary joins and withdrawals.

In fact, with the classical definition of an asynchronous environment, the group membership problem has been proven impossible to solve in the presence of any failures [15], [28]. The underlying reason for the impossibility is that according to the classical definition of an asynchronous environment, processes in the system share no common clock and there is no bound on the message delay. With this definition, it is impossible to implement a reliable fault detector, for no fault detector can distinguish between a *crashed* node and a *very slow* node. Since the establishment of this theoretic result, researchers have been striving to circumvent this impossibility. Theorists have modified the specifications [5], [16], [42], while practitioners have built a number of real systems that achieve a level of reliability in their particular environment [1], [6].

3.1 Novel Features

The group membership protocol in the RAIN system differs from that of other systems, such as the Totem [1] and Isis [6] projects, in several respects. First, it is based exclusively on unicast messages, a practical model given the nature of the Internet. With this model, the total ordering of packets is not relevant. Compared to broadcast messages, unicast messages are more efficient in terms of CPU overhead. Second, the protocol does not require the system to freeze during reconfiguration. We do make the assumption that the mean time to failure of the system is greater than the convergence time of the protocol. With this assumption, the RAIN system tolerates node and link failures, both permanent and transient. In general, it is not possible to distinguish a slow node from a dead node in an asynchronous environment. It is inevitable for a group membership protocol to exclude a live node, if it is slow, from the membership. Our protocol allows such a node to rejoin the cluster automatically.

The key to this fault management service is a token-based group membership protocol. The protocol consists of two mechanisms, a token mechanism and a 911 mechanism. The two mechanisms are described in greater detail in the next two sections.

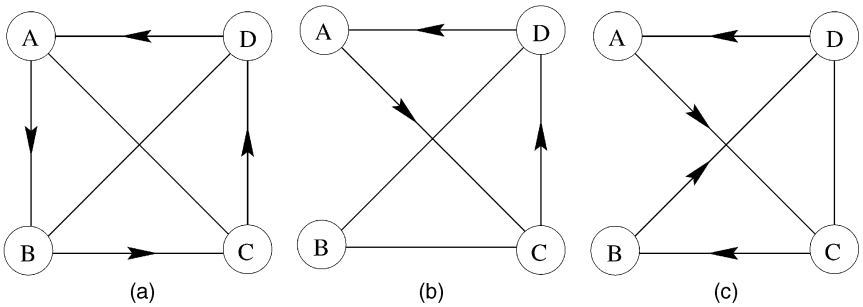

Fig. 9. (a) Token movement with no link failure. (b) Token movement with one link failure and aggressive failure detection. (c) Token movement with one link failure and conservative failure setection.

3.2 The Token Mechanism

The nodes in the membership are ordered in a logical ring. A *token* is a message that is being passed at a regular interval from one node to the next node in the ring. The reliable packet communication layer is used for the transmission of the token and guarantees that the token will eventually reach the destination. The token carries the authoritative knowledge of the membership. When a node receives a token, it updates its local membership information according to the token.

The token is also used for failure detection. There are two variants of the failure detection protocol in this token mechanism. The aggressive detection protocol achieves fast detection time, but is more prone to incorrect decisions, namely, it may temporarily exclude a partially disconnected node in the presence of link failures. The conservative detection protocol excludes a node only when its communication has failed from all nodes in the connected component. The conservative detection protocol has a slower failure detection time.

3.2.1 Aggressive Failure Detection

When the aggressive failure detection protocol is used, after a node fails to send a token to the next node, the former node immediately decides that the latter node has failed or is disconnected, and removes that node from the membership. The node updates the token with the adjusted membership information, and passes the token to the next live node in the ring. This protocol does not guarantee that all nodes in the connected component are included in the membership at all times. If a node loses a connection to part of the system because of a link failure, it could be excluded from the membership. The excluded node will automatically rejoin the system, however, via the 911 mechanism, which we will describe in the next section. For example, for the situation in Fig. 9b, the link between A and B is broken. After node A fails to send the token to node B, the aggressive failure detection protocol excludes node B from the membership. The ring changes from ABCD to ACD until node B rejoins the membership when the 911 mechanism is activated.

3.2.2 Conservative Failure Detection

In comparison, when the conservative failure detection protocol is used, partially disconnected nodes will not be excluded. When a node detects that another node is not responding, the former node does not remove the latter node from the membership. Instead, it changes the order of the ring. In the example in Fig. 9c, after node A fails to send the token to node B, it changes the ring from ABCD to ACBD. Node A then sends the token to node C and node C to node B. In the case when a node has indeed failed, all the nodes in the connected component fail to send the token to this node. When a node fails to send a token to another node twice in a row, it removes that node from the membership.

3.2.3 Uniqueness of Token

The token mechanism is the basic component of the membership protocol. It guarantees that there exists no more than one token in the system at any one time. This single token detects the failures, records the membership, and updates all live nodes as it travels around the ring. After a node is determined to have failed, all live nodes in the membership are unambiguously informed within one round of token travel. Group membership consensus is therefore achieved.

3.3 911 Mechanism

Having described the token mechanism, a few questions remain. What if a node fails when it possesses the token and, consequently, the token is lost? Is it possible to add a new node to the system? How does the system recover from a transient failure? These questions can be answered by the 911 mechanism.

3.3.1 Token Regeneration

To deal with the token loss problem, a time-out has been set on each node in the membership. If a node does not receive a token for a certain period of time, it enters the STARVING mode. The node suspects that the token has been lost and sends out a 911 message to the next node in the ring. The 911 message is a request for the right to regenerate the token, and is to be approved by all the live nodes in the membership. It is imperative to allow one and only one node to regenerate the token when a token regeneration is needed. To guarantee this mutual exclusivity, we utilize the sequence number on the token.

Every time a token is being passed from one node to another, the sequence number on it is increased by one. The primary function of the sequence number is to allow the receiving node to discard out-of-sequence tokens. The sequence number also plays an important role in the token

TABLE 1a
A Data Placement Scheme for a (6,4) Array Code

a	b	c	d	e	f
A	B	C	D	E	F
$B+D+e+f$	$C+E+f+a$	$D+F+a+b$	$E+A+b+c$	$F+B+c+d$	$A+C+d+e$

Each column represents one symbol of the encoded data. Addition (+) is the (bit-wise) binary XOR operation.

regeneration mechanism. Each node makes a local copy of the token every time that the node receives it. When a node needs to send a 911 message to request the regeneration of the token, it adds this message to the sequence number that is on its last local copy of the token. This sequence number will be compared to all the sequence numbers on the local copies of the token on the other live nodes. The 911 request will be denied by any node which possesses a more recent copy of the token. In the event that the token is lost, every live node sends out a 911 request after its STARVING time-out expires. Only the node with the latest copy of the token will receive the right to regenerate the token.

3.3.2 Dynamic Scalability

The 911 message is not only used as a token regeneration request, but also as a request to join the group. When a new node wishes to participate in the membership, it sends a 911 message to any node in the cluster. The receiving node notices that the originating node of this 911 is not a member of the distributed system and, therefore, treats it as a join request. The next time that it receives the token, it adds the new node to the membership and sends the token to the new node. Thus, the new node becomes a part of the system.

3.3.3 Link Failures and Transient Failures

The unification of the token regeneration request and the join request facilitates the treatment of link failures in the aggressive failure detection protocol. Using the example in Fig. 9b, node B has been removed from the membership because of the link failure between A and B. When node B does not receive the token for a while, it enters the STARVING mode and sends out a 911 message to node C. Node C notices that node B is not part of the membership and, therefore, treats the 911 as a join request. The ring is changed to ACBD and node B joins the membership.

Transient failures are treated with the same mechanism. When a transient failure occurs, a node is removed from the membership. After that node recovers, it sends out a 911 message. The 911 message is treated as a join request, and the node is added back into the cluster. In the same fashion, wrong decisions made in a local failure detector can also be corrected, guaranteeing that all nonfaulty nodes in the primary connected component eventually stay in the membership.

Putting together the token and 911 mechanisms, we have a reliable group membership protocol. Using this protocol, it is easy to build a fault management service. It is also possible to attach to the token application-dependent synchronization information. For example, in the SNOW project, described in Section 5.2, an HTTP request queue is attached to the token to ensure mutual exclusion of service.

4 Data Storage

Much research has been done on improving reliability by introducing data redundancy (also called information dispersity) [34], [50]. The RAIN system provides a distributed storage system based on a class of error-control codes called array codes. In Section 4.2, we describe the implementation of distributed store and retrieve operations based upon this storage scheme.

4.1 Array Codes

Array codes are a class of error-control codes that are particularly well-suited to be used as erasure-correcting codes. Erasure-correcting codes are a mathematical means of representing data so that lost information can be recovered. With an (n,k) erasure-correcting code, we represent k symbols of the original data with n symbols of encoded data ($n-k$ is called the amount of redundancy or parity). With an m-erasure-correcting code, the original data can be recovered even if m symbols of the encoded data are lost [39]. A code is said to be Maximum Distance Separable (MDS) if $m=n-k$. An MDS code is optimal in terms of the amount of redundancy versus the erasure recovery capability. The Reed-Solomon code [39] is an example of an MDS code.

The complexity of the computations needed to construct the encoded data (a process called encoding) and to recover the original data (a process called decoding) is an important consideration for practical systems. Array codes are ideal in this respect [9], [27]. The only operations needed for encoding and decoding are simple binary exclusive-or (XOR) operations, which can be implemented efficiently in hardware and/or software. Several MDS array codes are known. For example, the EVENODD code [8] is a general (n,k) array code. Recently, we described two classes of $(n,n-2)$ and $(n,2)$ MDS array codes with an optimal number of encoding and decoding operations [56], [57].

Example 4.1. Table 1a shows an example of such a code, called the B-Code [57], for $n=6$ and $k=4$. The original data consists of 12 pieces of equal size, represented as $a,b,\ldots,f,A,B,\ldots,F$. Each piece could be a single bit, or some fixed amount of binary data. The encoded data is shown in Table 1a, with one symbol per column.

For the above (6,4) code, suppose that the original data consists of the following 12 pieces, each one bit in size: 111010101010. Using the code in Table 1a, the data is encoded as in Table 1b. With this code, the amount of data needed for decoding (four columns with three bits

TABLE 1b
A Numerical Example of Table 1a

1	1	1	0	1	0
1	0	1	0	1	0
1	1	0	0	1	1

each) equals the amount of original data (12 bits). Thus, the code is MDS. If any two symbols (i.e., columns in the table) are lost, all the data can be reconstructed. Because of the symmetry of the data placement, we only need to check the following three cases of data loss:

Case 1. Columns 1 and 2 are lost. (Table 2)
The decoding procedure to recover the lost data is as follows:

$$\begin{aligned} A &= C + d + e + (A + C + d + e) \\ b &= A + (E + A + b + c) + c + E \\ a &= b + (D + F + a + b) + D + F \\ B &= a + c + (F + B + c + d) + d. \end{aligned}$$

Notice that here the "+" operation is a simple binary XOR operation, thus $x + x = 0$ and $0 + x = x$ hold for an arbitrary binary data piece x.

In Step 1 above, data pieces C, D, e, and $(A + C + d + e)$ are known from the third, fourth, fifth, and sixth columns, respectively. Thus, the lost data piece A in the first column can be recovered. Once A is known, it is then used to recover another lost data piece, b, in Step 2, and so on. Erasure decoding for array codes is usually done using such decoding *chains*.

Case 2. Columns 1 and 3 are lost.
Now, we need to recover the four data pieces a, A, c, and C. Similar to Case 1, the decoding chain is as follows:

$$\begin{aligned} c &= B + d + (F + B + c + d) + F \\ A &= c + b + (E + A + b + c) + E \\ C &= A + d + e + (A + C + d + e) \\ a &= C + (C + E + f + a) + E + f. \end{aligned}$$

Case 3. Columns 1 and 4 are lost.
Finally, we give the procedure to recover the data pieces a, A, d, and D:

$$\begin{aligned} a &= (C + E + f + a) + C + E + f; \\ A &= b + c + (E + A + b + c) + E \\ D &= \mathbf{a} + b + (D + F + a + b) + F; \\ d &= \mathbf{A} + C + e + (A + C + f + e). \end{aligned}$$

Notably, here we see two *parallel* decoding chains to recover a and A; these are then used to recover D and d, respectively.

General decoding algorithms for the B-Code and the X-Code (another class of MDS array codes we have developed) are described in [57] and [56].

4.2 Distributed Store/Retrieve Operations

Our distributed store and retrieve operations are a straight-forward application of MDS array codes to distributed storage. Suppose that we have n nodes. For a store operation, we encode a block of data of size d into n symbols, each of size $\frac{d}{k}$, using an (n, k) MDS array code. We store one symbol per node. For a retrieve operation, we collect the symbols from any k nodes and decode them to obtain the original data.

This data storage scheme has several attractive features. First, it provides reliability. The original data can be recovered with up to $n - k$ node failures. Second, it permits dynamic reconfigurability and hot-swapping of components. We can dynamically remove and replace up to $n - k$ nodes. In addition, the flexibility to choose any k out of n nodes permits load balancing. We can select the k nodes with the smallest load or, in the case of a wide-area network, the k nodes that are geographically closest.

5 Proof-of-Concept Applications

We present several applications implemented on the RAIN platform based on the fault management, communication, and data storage building blocks: a video server (RAINVideo), a web server (SNOW), and a distributed checkpointing system (RAINCheck).

5.1 High-Availability Video Server

There has been considerable research in the area of fault-tolerant Internet and multimedia servers. Examples are the SunSCALR project at Sun Microsystems [48], as well as papers by Elnozahy [24] and Tobagi et al. [53].

For our RAINVideo application, a collection of videos are encoded and written to all n nodes in the system with distributed store operations. Each node runs a client application that attempts to display a video, as well as a server application that supplies encoded video data. For each block of video data, a client performs a distributed retrieve operation to obtain encoded symbols from k of the servers. It then decodes the block of video data and displays it. If we break network connections or take down nodes, some of the servers may no longer be accessible. However, the videos continue to run without interruption, provided that each client can access at least k servers. Snapshots of the

TABLE 2
Recovery of the First Two Columns of the (6,4) Array Code

?	?	c	d	e	f
?	?	C	D	E	F
?	?	$D + F + a + b$	$E + A + b + c$	$F + B + c + d$	$A + C + d + e$

Fig. 10. The RAINVideo system.

demo are shown in Figs. 10 and 11. There are 10 computers, each with two Myrinet network interfaces and four eight-way Myrinet network switches.

5.2 High-Availability Web Server

SNOW stands for Strong Network Of Web servers. It is a proof-of-concept project that demonstrates the features of the RAIN system. The goal is to develop a highly available fault-tolerant distributed web server cluster that minimizes the risk of downtime for mission-critical Internet and Intranet applications.

The SNOW project uses several key building blocks of the RAIN technology. First, the reliable communication layer is used to handle all of the message passing between the servers. Second, the token-based fault management module is used to establish the set of servers participating in the cluster. In addition, the token protocol is used to guarantee that when a request is received by SNOW, one—and only one—server will reply to the client. The latest information about the HTTP queue is attached to the token. Third, the distributed storage module can be used to store the actual data for the web server.

SNOW also uses the distributed state sharing mechanism enabled by the RAIN system. The state information of the web servers, namely, the queue of HTTP requests, is shared reliably and consistently among the SNOW nodes. High availability and performance are achieved without external load balancing devices, such as the commercially available Cisco LocalDirector. The SNOW system is also readily scalable. In contrast, the commercially available Microsoft Wolfpack is only available for up to two nodes per cluster.

5.3 Distributed Checkpointing Mechanism

The idea of using error-control codes for distributed checkpointing was proposed by Plank [43]. We have implemented a checkpoint and rollback/recovery mechanism on the RAIN platform based on the distributed store and retrieve operations. The scheme runs in conjunction with a leader election protocol, described in [29]. This protocol ensures that there is a unique node designated as leader in every connected set of nodes. The leader node assigns jobs to the other nodes. As each job executes, a checkpoint of its state is taken periodically. The state is encoded and written to all accessible nodes with a distributed store operation. If a node fails or becomes inaccessible, the leader assigns the node's jobs to other nodes. The encoded symbols for the state of each job are

Fig. 11. A client node displaying a video in the RAINVideo system.

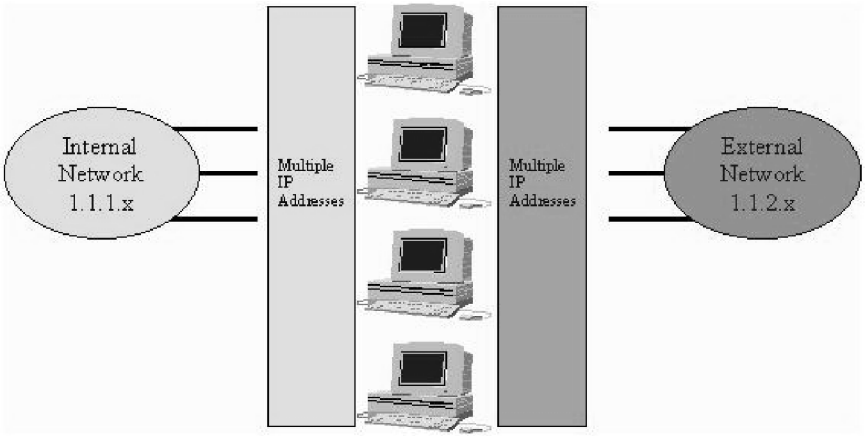

Fig. 12. Simple Rainwall configuration with one external and one internal network. Rainwall presents multiple IP addresses for each subnet and guarentees that those IP addresses are available as long as one gateway machine remains functional.

read from k nodes with a distributed read operation. The state of each job is then decoded and execution is resumed from the last checkpoint. As long as a connected component of k nodes survives, all jobs execute to completion.

6 Rainwall: A Commercial Application

The RAIN technology has been transfered to Rainfinity, a start-up company focusing on creating clustered solutions for improving the performance and availability of Internet data centers [41]. Rainfinity's first commercial product is Rainwall, an application of the RAIN technology to provide a high-availability and load-balancing clustering solution for firewalls. As the Internet continues to grow, firewalls are becoming a requirement as a single security administration point to control access and block intruders. These single points of administration and entry into the network also become single points of failure and bottlenecks for the organization, customers, and partners trying to access the site. Rainwall is a scalable, high-availability software solution that is installed on a cluster of gateways, along with the firewall software, and performs load balancing among the gateways. In the event of the failure of one or more gateways, Rainwall routes traffic through the remaining gateways without interrupting existing connections.

Rainwall is designed to address three issues that affect mission-critical Internet firewalls: availability, performance, and scalability.

- **Availability:** Rainwall detects failures in software and hardware components in real time, shifting traffic from failing gateways to functioning ones without interrupting existing connections.
- **Performance:** Rainwall performs load balancing of traffic among the gateway servers, and because all the servers are actively processing traffic, Rainwall dramatically increases gateway performance.
- **Scalability:** Rainwall is scalable to any number of Internet firewall gateways and allows the addition of new gateways into the cluster without service interruption. Rainwall also allows existing servers in the cluster to be removed for service or maintenance and added back to the cluster, again without any interruption of service.

6.1 Virtual IPs

At its core, Rainwall manages pools of virtual IP addresses. By ensuring that all virtual IPs are always owned by one—and only one—healthy gateway, and by moving these virtual IP addresses between gateway machines, Rainwall is able to balance load and compensate for failed machines. The group membership protocol described in Section 3 is used as the foundation for the virtual IP management.

Rainwall monitors inbound and outbound gateway traffic, ensuring that each gateway is functioning properly and that the traffic is balanced. It guarantees that the pools of virtual IP addresses are always available as long as one machine remains functional in the cluster. There are no leaders, no primary or secondary machines, and no redirectors within the cluster. Rainwall requires no extra network interface cards or subnetworks; all synchronization between gateway machines is allowed to go over existing networks. Fig. 12 shows a simple Rainwall cluster sitting between a single external and a single internal subnet.

The Rainwall software is installed directly onto every firewall machine in the cluster and manages the pools of virtual IP addresses. The virtual IPs are specified to the routers and local clients as default gateways, and are the only advertised IP addresses of the firewall cluster. They are dynamically assigned to different firewall nodes in the Rainwall cluster. Rainwall manages the virtual IPs intelligently and efficiently, so that the firewall availability is guaranteed in the presence of failures and optimal performance is achieved even under heavy load.

Rainwall assures users that while physical machines can go down, the virtual IPs never disappear. In a simple 2-node Rainwall cluster, if Firewall A crashes, all of Firewall A's virtual IPs are moved to Firewall B. In a 3-node Rainwall cluster, an even higher level of availability is achieved. Two out of three firewalls can fail and the healthy one will host all the virtual IPs. With Rainwall's auto-recovery feature, it can reintroduce a recovered server into the cluster on the fly, with no downtime, automatically returning the cluster to its preferred state. This maintains the highest level of availability and performance, all with no downtime and potentially no human intervention.

6.2 Failure Detection

The failure detection mechanism of Rainwall consists of two components, a local failure detector and a cluster failure

Fig. 13. Rainwall Java Graphic User Interface.

detector. A copy of the local failure detector sits on each of the firewall nodes, constantly examining whether required local resources are functioning correctly. Rainwall examines three required components: the network interface cards for link connectivity, the firewall software for proper functionality, and the local machine's ability to reach remote hosts via ping. If any of these required resources fail, Rainwall will, by default, bring down the firewall node, and its virtual IP addresses will be reassigned to other healthy firewall nodes with no interruption of service. Any local monitoring component may be disabled by the administrator, if desired.

The cluster failure detector uses the protocol described in Section 3. This protocol enables healthy firewall nodes to detect crashed firewall nodes. It guarantees that all healthy nodes in the cluster have consensus on the failures. Extensions to this protocol also allow all the firewall nodes to share the knowledge of virtual IP assignment, machine health, machine load, and other global state relevant to Rainwall. The fail-over time of Rainwall is about two seconds. This enables transparent fail-over of all TCP traffic going through the cluster.

6.3 Load Balancing and Performance

Rainwall monitors the total traffic going into each node. When an imbalance is detected, i.e., one firewall node is heavily loaded while another one carries a much lighter load, it moves one or more of the virtual IPs on the more heavily-loaded node to the more lightly-loaded node. The load balancing is based on load request and not load assignment. A heavily-loaded machine does not dump load to other machines; rather, a less-loaded machine requests load from heavily-loaded machines. This avoids the "hot potato" effect, where a particularly busy virtual IP gets bounced from machine to machine.

Rainwall can be configured to manage multiple virtual IP addresses per subnet. The larger the total number of virtual IPs to which traffic is routed, the finer the granularity in load balancing that Rainwall achieves. To utilize multiple virtual IPs, routers can be configured to use multiple routes of equal weight, or individual machines can be configured to use different default routes from within the virtual IP pool on their subnet.

With Rainwall's dynamic load balancing, users experience a dramatic performance boost. According to reports from the Rainfinity Lab, a four-node Rainwall NT cluster running on a Pentium II 450 MHz single processor Dell Dimension system, achieves a benchmark of 251 Mbps. In comparison, the single-node performance is 67 Mbps. In other words, a four-node Rainwall cluster is 3.75 times as powerful as a single-node firewall. (Traffic was generated using Ziff-Davis WebBench.)

6.4 Ease of Use

Rainwall comes with a Java-based management console. From this GUI, the user can monitor the health of firewall nodes, the load, and the virtual IP address assignment in the cluster. The administrator can set virtual IPs to be sticky, so that they stay on particular nodes and do not participate in load balancing. Virtual IPs can be preconfigured with a preference for particular machines, and the GUI allows drag-and-drop of virtual IPs between machines. Thus, virtual IPs can be sticky, have a preference, or be free floating. In all cases, virtual IPs migrate to healthy machines in the event of a machine failure. Fig. 13 shows the graphic user interface of Rainwall.

These sticky, preference, and drag-and-drop options allow for manual adjustment of traffic flow through the network. This flexibility allows endless variations on how a firewall administrator can make the cluster more robust and useful. For example, high priority traffic may be assigned to go through one firewall node, while low priority web-surfing traffic goes through the other,

guaranteeing quality-of-service for the high-priority traffic. As another example, if hacker activity is suspected, the administrator can drag-and-drop a virtual IP into a "trap" firewall, where the behavior can be analyzed, without compromising the network or closing down the firewall.

7 Conclusions

The goal of the RAIN project has been to build a testbed for various building blocks that address fault-management, communication, and storage in a distributed environment. The creation of such building blocks is important for the development of a fully functional distributed computing system. One of the fundamental driving ideas behind this work has been to consolidate the assumptions required to get around the "difficult" parts of distributed computing into several basic building blocks. We feel that the ability to provide basic, provably correct services is essential to building a real fault-tolerant system. In other words, the difficult proofs should be confined to a few basic components of the system. Components of the system built on top of those reliable components should then be easier to develop and easier to establish as correct in their own right. Building blocks that we consider important and that are discussed in this paper are those providing reliable communication, group membership information, and reliable storage. Among the current and future directions of this work are:

- Development of API's for using the various building blocks. We should standardize the packaging of the various components to make them more practical for use by outside groups.
- The implementation of a real distributed file system using the data partitioning schemes developed here. In addition to making this building block more accessible to others, it would help in assessing the performance benefits and penalties from partitioning data in such a manner.

Acknowledgments

This work was supported in part by an US National Sceince Foundation Young Investigator Award (CCR-9457811), by a Sloan Research Fellowship, by an IBM Partnership Award, and by DARPA through an agreement with NASA/OSAT.

References

[1] Y. Amir, L.E. Moser, M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella, "The Totem Single-Ring Ordering and Membership Protocol," *ACM Trans. Computer Systems,* vol. 13, no. 4, pp. 311–342, 1995.

[2] T.E. Anderson, D.E. Culler, and D.A. Patterson, "A Case for NOW (Networks of Workstations)," *IEEE Micro,* vol. 15, no. 1, pp. 54–64, 1995.

[3] D.J. Becker, T. Sterling, D. Savarese, E. Dorband, U.A. Ranawake, and C.V. Packer, "BEOWULF: A Parallel Workstation for Scientific Computation," *Proc. 1995 Int'l Conf. Parallel Processing,* pp. 11–14, 1995.

[4] A. Beguelin, E. Seligman, and P. Stephan, "Application Level Fault Tolerance in Heterogeneous Networks of Workstations," *J. Parallel and Distributed Computing,* vol. 43, no. 2, pp. 147–155, 1997.

[5] M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," *Proc. Second ACM Symp. Principles of Distributed Computing,* pp. 27–30, Aug. 1983.

[6] K.P. Birman and R. Van Renesse, *Reliable Distributed Computing with the Isis Toolkit.* IEEE CS Press, 1994.

[7] K.P. Birman and B.B. Glade, "Reliability Through Consistency," *IEEE Software,* vol. 12, no. 3, pp. 29–41, 1995.

[8] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures," *IEEE Trans. Computers,* vol. 44, no. 2, pp. 192–202, Feb. 1995.

[9] M. Blaum, P.G. Farrell, H.C.A. van Tilborg, "Chapter on Array Codes," *Handbook of Coding Theory,* V.S. Pless and W.C. Huffman, eds., to appear.

[10] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.K. Su, "Myrinet: A Gigabit per Second Local Area Network," *IEEE Micro,* vol. 15, no. 1, pp. 29–36, 1995.

[11] F.T. Boesch and A.P. Felzer, "A General Class of Invulnerable Graphs," *Networks,* vol. 2, pp. 261–283, 1972.

[12] F.T. Boesch and R. Tindell, "Circulants and Their Connectivities," *J. Graph Theory,* vol. 8, pp. 487–499, 1984.

[13] F.T. Boesch and J.F. Wang, "Reliable Circulant Networks with Minimum Transmission Delay," *IEEE Trans. Circuits and Systems,* vol. 12, no. 12, pp. 1286–1291, 1985.

[14] J. Bruck, R. Cypher, and C.T. Ho, "Fault-Tolerant Meshes and Hypercubes with Minimal Numbers of Spares," *IEEE Trans. Computers,* vol. 42, no. 9, pp. 1089–1104, Sept. 1993.

[15] T.D. Chandra, V. Hadzillacos, S. Toueg, and B. Charron-Bost, "On the Impossibility of Group Membership," *Proc. 15th ACM Symp. Principles of Distributed Computing,* pp. 322–330, 1996.

[16] T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM,* vol. 43, no. 2, pp. 225–267, 1996.

[17] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson, "Raid—High-Performance, Reliable Secondary Storage," *ACM Computing Surveys,* vol. 26, no. 2, pp. 145–185, 1994.

[18] J.N. Cotrim Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan, "Dome: Parallel Programming in a Distributed Computing Environment," *Proc. IEEE Symp. Parallel and Distributed Processing,* pp. 218–224, 1996.

[19] D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communication," *Comm. ACM,* vol. 39, no. 4, pp. 64–70, 1996.

[20] D. Dolev and D. Malki, "The Design of the Transis System," *Theory and Practice in Distributed Systems,* pp. 83–98, 1995.

[21] S. Dutt and J.P. Hayes, "On Designing and Reconfiguring K-Fault-Tolerant Tree Architectures," *IEEE Trans. Computers,* vol. 39, no. 4, pp. 490–503, Apr. 1990.

[22] S. Dutt and J.P. Hayes, "Designing Fault-Tolerant Systems Using Auto-morphisms," *J. Parallel and Distributed Computing,* vol. 12, no. 3, pp. 249–268, 1991.

[23] S. Dutt and J.P. Hayes, "Some Practical Issues in the Design of Fault-Tolerant Multiprocessors," *IEEE Trans. Computers,* vol. 41, no. 5, pp. 588–598, May 1992.

[24] E.N. Elnozahy, "Storage Strategies for Fault-Tolerant Video Servers," Technical Report CMU-CS-96-144, Dept. Computer Science, Carnegie Mellon Univ., 1996.

[25] E.N. Elnozahy and W. Zwaenepoel, "Manetho—Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit," *IEEE Trans. Computers,* vol. 41, no. 5, pp. 526–531, May 1992.

[26] G.A. Gibson, D. Stodolsky, F.W. Chang, W.V. Courtright II, C.G. Demetriou, E. Ginting, M. Holland, Q. Ma, L. Neal, R.H. Patterson, J. Su, R. Youssef, and J. Zelenka, "The Scotch Parallel Storage Systems," *Proc. IEEE CompCon Conf.,* 1995.

[27] P.G. Farrell, "A Survey of Array Error Control Codes," *European Trans. Telecommunications,* vol. 3, no. 5, pp. 441–454, 1992.

[28] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374i–382, 1985.

[29] M. Franceschetti and J. Bruck, "A Leader Election Protocol for Fault Recovery in Asynchronous Fully-Connected Networks," Paradise Electronic Technical Report #024, http://paradise.caltech.edu/ETR.html, 1998.

[30] M. Franceschetti and J. Bruck, "On the Possibility of Group Membership Protocols," *Dependable Network Computing,* D.R. Avresky, ed., Kluwer Academic, pp. 77–92, 2000.

[31] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing,* vol. 22, no. 6, pp. 789–828, 1996.
[32] J.H. Hartman and J.K. Ousterhout, "The Zebra Striped Network File System," *ACM Trans. Computer Systems,* vol. 13, no. 3, pp. 274–310, 1995.
[33] J.P. Hayes, "A Graph Model for Fault-Tolerant Computing Systems," *IEEE Trans. Computers,* vol. 25, no. 9, pp. 875–884, Sept. 1976.
[34] T. Krol, "(N,K) Concept Fault Tolerance," *IEEE Trans. Computers,* vol. 35, no. 4, Apr. 1986.
[35] H.K. Ku and J.P. Hayes, "Connective Fault Tolerance in Multiple-Bus Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 6, pp. 574–586, June 1997.
[36] P.S. LeMahieu, V.Z. Bohossian, and J. Bruck, "Fault-Tolerant Switched Local Area Networks," *Proc. Int'l Parallel Processing Symp.,* pp. 747–751, 1998.
[37] P.S. LeMahieu and J. Bruck, "Consistent History Link Connectivity Protocol," *Proc. Int'l Parallel Processing Symp.,* pp. 138–142, 1999.
[38] N. Lynch, *Distributed Algorithms.* New Jersey, Morgan Kaufman, 1996.
[39] F.J. MacWilliams and N.J.A. Sloane, *The Theory of Error Correcting Codes.* North-Holland, 1977.
[40] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos, "Totem: A Fault-Tolerant Multicast Group Communication System," *Comm. ACM,* vol. 39, no. 4, pp. 54–63, 1996.
[41] "NASA-Funded Software Aids Reliability," *Network World,* vol. 16, no. 51, Dec. 1999, http://www. nwfusion. com/news/1999/1220infra. html.
[42] G. Neiger, "A New Look at Membership Services," *Proc. 15th ACM Symp. Principles of Distributed Computing,* pp. 331–340, 1996.
[43] J.S. Plank and K. Li, "Faster Checkpointing with N+1 Parity," *Proc. IEEE 24th Int'l Symp. Fault-Tolerant Computing,* pp. 288–297, 1994.
[44] M.D. Riedel and J. Bruck, "Tolerating Faults in Counting Networks," *Dependable Network Computing,* D.R. Avresky, ed., Kluwer Academic, pp. 267–278, 2000.
[45] T.L. Rodeheffer and M. D. Schroeder, "Automatic Reconfiguration in Autonet," *Proc. 13th ACM Symp. Operating Systems Principles on Operating Systems Review,* vol. 25, no. 5, pp. 183–197, 1991.
[46] T.L. Rodeheffer and M.D. Schroeder, "A Case Study: Automatic Reconfiguration in Autonet," *Distributed Systems,* second ed., ACM Press, 1993.
[47] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere, "CODA—A Highly Available File System for a Distributed Workstation Environment," *IEEE Trans. Computers,* vol. 39, no. 4, pp. 447–459, Apr. 1990.
[48] A. Singhai, S.-B. Lim, and S.R. Radia, "The SunSCALR Framework for Internet Servers," *Proc. IEEE 28th Int'l Symp. Fault-Tolerant Computing,* 1998.
[49] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, "MPI: The Complete Reference," MIT Press, http://www.netlib.org/utk/papers/mpi-book/mpi-book.ps, 1995.
[50] H.-M. Sun and S.-P. Shieh, "Optimal Information Dispersal for Increasing the Reliability of a Distributed Service," *IEEE Trans. Reliability,* vol. 46, no. 4, pp. 462–472, 1997.
[51] V. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience,* vol. 2, no. 4, pp. 315–339, http://www.netlib.org/ncwn/pvmsystem.ps, 1990.
[52] V. Sunderam, J. Dongarra, A. Geist, and R Manchek, "The PVM Concurrent Computing System: Evolution, Experiences, and Trends," *Parallel Computing,* vol. 20, no. 4, pp. 531–547, http://www.netlib.org/ncwn/pvm-pc-evol. ps, 1994.
[53] F.A. Tobagi, J. Pang, R. Baird, and M. Gang, "Streaming RAID: A Disk Array Management System for Video Files," *Proc. ACM Conf. Multimedia,* pp. 393–400, 1993.
[54] R. van Renesse, K.P. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System," *Comm. ACM,* vol. 39, no. 4, pp. 76–83, 1996.
[55] L. Xu, V. Bohossian, J. Bruck, and D. Wagner, "Low Density MDS Codes and Factors of Complete Graphs," *Proc. IEEE Symp. Information Theory,* 1998.
[56] L. Xu and J. Bruck, "X-Code: MDS Array Codes with Optimal Encoding," *IEEE Trans. Information Theory,* vol. 45, no. 1, pp. 272–276, Jan. 1999.
[57] L. Xu, V. Bohossian, J. Bruck, and D.G. Wagner, "Low Density MDS Codes and Factors of Complete Graphs," *IEEE Trans. Information Theory,* vol. 45, no. 6, pp. 1817–1826, Sept. 1999.

Vasken Bohossian received the BSc degree in electrical engineering with honors from McGill University in 1993, the MSc degree in electrical engineering, and the PhD degree in computation and neural systems from the California Institute of Technology in 1994 and 1998, respectively. He is cofounder and senior engineer at Rainfinity, a spin-off company from the California Institute of Technology that focuses on providing software for high performance reliable Internet infrastructure. His research interests include parallel and distributed computing, fault-tolerant computing, error-correcting codes, computation theory and threshold logic.

Chenggong C. Fan received the BE degree from Cooper Union for the Advancement of Science and Art in 1995 and the MS degree from the California Institute of Technology in 1996. He is currently a PhD student advised by Dr. Bruck at the California Institute of Technology. He is a student member of IEEE, a recipient of 1995 US National Science Foundation Graduate Fellowship, and a first place winner in 1995 IEEE Region 1 Student Paper Contest. His research interests include fault-tolerant distributed computing and communication networks.

Paul S. LeMahieu received the BS degree in electrical engineering and computer science from the University of Wisconsin-Madison in 1994 and the MS degree in electrical engineering from the California Institute of Technology in 1995. He is currently finishing his PhD degree at the California Institute of Technology under the instruction of Dr. Bruck. His research interests include fault-tolerant networks, network protocols, and issues in distributed computing.

Marc D. Riedel received the BEng degree in electrical engineering with a minor in mathematics from McGill University in 1995 and the MS degree in electrical engineering from the California Institute of Technology in 1997. He is currently studying for his PhD degree at the California Institute of Technology under the instruction of Dr. Bruck. His research interests include multiprocessor coordination, asynchronous computation, and fault-tolerant distributed systems.

Lihao Xu received the BSc and MSc degrees in electrical engineering from the Shanghai Jiao Tong University, China, in 1988 and 1991 respectively, and the PhD degree in electrical engineering from the California Institute of Technology in 1999. He is currently an assistant professor of computer science at Washington University in St. Louis. From 1991 to 1994, he was a lecturer in the Electrical Engineering Department of the Shanghai Jiao Tong University. His current research interests include distributed data systems, robust computing, error control codes. He is a member of the IEEE.

Jehoshua Bruck received the BSc and MSc degrees in electrical engineering from the Technion, Israel Institute of Technology, in 1982 and 1985, respectively, and the PhD degree in electrical engineering from Stanford University in 1989. He is a professor of computation and neural systems and electrical engineering at the California Institute of Technology. His research interests include parallel and distributed computing, fault-tolerant computing, computation theory, and neural and biological systems. Dr. Bruck has an extensive industrial experience, including working with IBM for 10 years both at the IBM Almaden Research Center and the IBM Haifa Science Center. He is a cofounder and chairman of Rainfinity, a spin-off company from Caltech that is focusing on providing software for high performance reliable Internet infrastructure.

Dr. Bruck is the recipient of a 1997 IBM Partnership Award, a 1995 Sloan Research Fellowship, a 1994 US National Science Foundation Young Investigator Award, six IBM Plateau Invention Achievement Awards, a 1992 IBM Outstanding Innovation Award, and a 1994 IBM Outstanding Technical Achievement Award for his contributions to the design and implementation of the SP-1, the first IBM scalable parallel computer. He published more than 150 journal and conference papers in his areas of interests and he holds 22 US patents. He is a fellow of the IEEE.