Computing in the RAIN: A Reliable Array of Independent Nodes*

Vasken Bohossian, Charles C. Fan, Paul S. LeMahieu, Marc D. Riedel, Lihao Xu, and Jehoshua Bruck

California Institute of Technology, Pasadena, CA 91125, USA
{vincent, fan, lemahieu, riedel, lihao, bruck}@paradise.caltech.edu
http://www.paradise.caltech.edu

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: highly available video and web servers, and a distributed checkpointing system.

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 RAIN platform consists of a heterogeneous cluster of computing and/or storage nodes connected via multiple interfaces to networks configured in fault-tolerant

* Supported in part by an NSF Young Investigator Award (CCR-9457811), by a Sloan Research Fellowship, by an IBM Partnership Award and by DARPA through an agreement with NASA/OSAT.

topologies. The RAIN software components run in conjunction with operating system services and standard network protocols.

Features of the RAIN system include scalability, dynamic reconfigurability, and high availability. Through software-implemented fault tolerance, the system tolerates multiple node, link, and switch failures, with no single point of failure. In addition to reliability, the RAIN architecture permits efficient use of network resources, such as multiple data paths and redundant storage, with graceful degradation in the presence of faults. 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 [18].

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 [21] 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:

- 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 [14]; on the consistent-history protocol in [15]; on the leader election protocol in [11]; and on data storage schemes in [25], [26] and [27].

1.1 Related Work

Cluster computing systems such as the NOW project at the University of California, Berkeley [1] and the Beowulf project [2] have shown that networks of workstations can rival supercomputers in computational power. Packages such as PVM [23] and MPI [21] 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 [4] and Horus [24] systems at Cornell University, the Totem system at the University of California, Santa Barbara [17], and the Transis system at the Hebrew University of Jerusalem [9].

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.

- **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 monitoring*: To correctly use multiple paths between 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.
- **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.
- **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 off 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.

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 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: (i) 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, (ii) a proof that this construction is optimal in the sense that no construction can tolerate more switch failures while avoiding partitioning, and (iii) 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 [14] for further details.

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 [15] we give a modified *ping* protocol that guarantees that each side of the communication channel sees the same 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 will thus take the same error recovery action. Such a guarantee may simplify the writing of applications using this connectivity information.

Our main contributions are: (i) a simple, *stable* protocol for monitoring connectivity that maintains a *consistent history* with *bounded slack*, and (ii) proofs that this protocol exhibits *correctness*, *bounded slack*, and *stability*. See [15] for further details.

2.3 A Port of MPI

A port of MPI [21] (using the MPICH implementation from Argonne Labs [12]) 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 presented in detail in [15]. 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 to return 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 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 non-faulty 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, under the classical asynchronous environment, the group membership problem has been proven impossible to solve in the presence of any failures [7], [10]. 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. Under 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 [3], [8], [19], while practitioners have built a number of real systems that achieve a level of reliability in their particular environment [4], [17].

3.1 Novel Features

The group membership protocol in the RAIN system differs from that of other systems, such as the Totem [17] and Isis [4] projects, in several respects. Firstly, 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. Secondly, 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. Using this protocol, it is possible to build the 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, the 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) [13], [22]. 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 [16]. 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 [16] 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 [6]. 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 [5] 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 [26], [27].

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. Firstly, it provides reliability. The original data can be recovered with up to $n - k$ node failures. Secondly, 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 described in the preceding sections: a video server (RAINVideo), a web server (SNOW), and a distributed checkpointing system (RAINCheck).

5.1 High-Availability Video Server

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.

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 down-time for mission-critical Internet and Intranet applications.

The SNOW project uses several key building blocks of the RAIN technology. Firstly, the reliable communication layer is used to handle all of the message passing between the servers in the SNOW system. Secondly, 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. Thirdly, 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 [20]. 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 [11]. 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 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 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 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 future and current 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.

We are currently benchmarking the system for general assessment of the performance of the algorithms and protocols developed in the project.

References

1. 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.
2. D. J. Becker, T. Sterling, D. Savarese, E. Dorband, U.A. Ranawake , and C.V. Packer, "BEOWULF: A Parallel Workstation for Scientific Computation," *Proceedings of the 1995 International Conference on Parallel Processing*, pp. 11–14, 1995.
3. M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, ACM Press, pp. 27–30, August 1983.
4. K.P. Birman and R. Van Renesse, "Reliable Distributed Computing with the Isis Toolkit," *IEEE Computer Society Press*, 1994.
5. M. Blaum, J. Brady, J. Bruck and J. Menon, "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures," *IEEE Trans. on Computers*, Vol. 44, No. 2, pp. 192–202, 1995.
6. M. Blaum, P.G. Farrell and H.C.A. van Tilborg, "Chapter on Array Codes," *Handbook of Coding Theory*, V.S. Pless and W.C. Huffman eds., to appear.
7. T.D. Chandra, V. Hadzillacos, S. Toueg and B. Charron-Bost, "On the Impossibility of Group Membership," *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, ACM Press, pp. 322–330, 1996.
8. T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, Vol. 43, No. 2, pp. 225–267, 1996.
9. D. Dolev and D. Malki "The Transis Approach to High Availability Cluster Communication," *Communications of the ACM*, Vol. 39, No. 4, pp. 64–70, 1996.
10. M.J. Fischer, N.A. Lynch and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, Vol. 32, No. 2, pp. 374i–382, 1985.
11. 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.

12. 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.
13. T. Krol, "(N,K) Concept Fault Tolerance," *IEEE Trans. on Computers*, Vol. C-35, No. 4, 1986.
14. P.S. LeMahieu, V.Z. Bohossian and J. Bruck, "Fault-Tolerant Switched Local Area Networks," *Proceedings of the International Parallel Processing Symposium*, pp. 747–751, 1998.
15. P.S. LeMahieu and J. Bruck, "Consistent History Link Connectivity Protocol," *Proceedings of the International Parallel Processing Symposium*, pp. 138–142, 1999.
16. F.J. MacWilliams and N.J.A. Sloane, *The Theory of Error Correcting Codes*, North-Holland, 1977.
17. 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," *Communications of the ACM*, Vol. 39, No. 4, pp. 54–63, 1996.
18. "NASA-Funded Software Aids Reliability," *Network World*, Vol. 16, No. 51, Dec. 20, 1999, http://www.nwfusion.com/news/1999/1220infra.html.
19. G. Neiger, "A New Look at Membership Services," *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, ACM Press, pp. 331–340, 1996.
20. J.S. Plank and K. Li, "Faster Checkpointing with N+1 Parity," *IEEE 24th International Symposium on Fault-Tolerant Computing*, pp. 288–297, 1994.
21. 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.
22. H.-M. Sun and S.-P. Shieh, "Optimal Information Dispersal for Increasing the Reliability of a Distributed Service," *IEEE Trans. on Reliability*, Vol. 46, No. 4, pp. 462–472, 1997.
23. 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.
24. R. van Renesse, K.P. Birman and S. Maffeis, "Horus: a Flexible Group Communication System," *Communications of the ACM*, Vol. 39, No. 4, pp. 76–83, 1996.
25. L. Xu and J. Bruck, "Improving the Performance of Data Servers Using Array Codes," *Paradise Electronic Technical Report #027*, http://paradise.caltech.edu/ETR.html, 1998.
26. L. Xu and J. Bruck "X-Code: MDS Array Codes with Optimal Encoding," *IEEE Trans. on Information Theory*, Vol. 45, No. 1, pp. 272–276, January 1999.
27. L. Xu, V. Bohossian, J. Bruck and D. G. Wagner, "Low Density MDS Codes and Factors of Complete Graphs," *IEEE Trans. on Information Theory*, Vol. 45, No. 6, pp. 1817–1826, September 1999.