FAULT-TOLERANT CLUSTER OF NETWORKING ELEMENTS

Thesis by

Chenggong Charles Fan

In partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

2001

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Professor Shuki Bruck. His vision and guidance made this work happen. I would also like to express my gratitude to the other professors at Caltech, for you not only directly or indirectly helped my graduate studies, but also made Caltech a great place to be. The friends from the Paradise research lab and other labs helped make my Caltech years so enjoyable, that I don’t mind being a graduate student for longer. My colleagues at Rainfinity made valuable contributions to some of the results in this dissertation. To them I say thank you.

I dedicate this dissertation to my family — to my parents and my grandma, to my wife Fang and our precious little Hannah.

ABSTRACT

The explosive growth of the Internet demands higher reliability and performance than what the current networking infrastructure can provide. This dissertation explores novel architectures and protocols that provide a methodology for grouping together multiple networking elements, such as routers, gateways, and switches, to create a more reliable and performant distributed networking system. Clustering of networking elements is a novel concept that requires the invention of distributed computing protocols that facilitate efficient and robust support of networking protocols. We introduce the Raincore protocol architecture that achieves these goals by bridging the fields of computer networks and distributed systems.

In designing Raincore, we paid special attention to the unique requirements from the networking environment. First, networking clusters need to scale up the networking throughput in addition to the scaling up of computing power. Second, task switching between the different services supported by a networking element has a major negative impact on performance. Third, fast fail-over time is critical for maintaining network connections in the event of failures. We discuss in depth the design of Raincore Group Communication Manager that addresses the forgoing requirements and provides group membership management and reliable multicast transport. It is based on a novel token-ring protocol. We prove that this protocol is formally correct, namely, it satisfies the set of formal specifications that defines the Group Membership problem.

The creation of Raincore has already made a substantial impact both at Caltech and the academic community as well as in the industry. The first application is SNOW, a scalable web server cluster that is part of RAIN, a collaborative project between Caltech and JPL / NASA. The second application is RainWall, a commercial solution created by Rainfinity, a Caltech spin-off company, that provides the first fault-tolerant and scalable firewall cluster. These applications exhibit the fast fail-over response, low overhead, and near-linear scalability of the Raincore protocols.

In addition, we studied fault-tolerant networking architectures. In particular, we considered efficient constructions of extra-stage fault-tolerant Multistage Interconnection Networks. Multistage Interconnection Networks provide a way to construct a larger switching network using smaller switching elements. We discovered an optimal family of constructions, in the sense that it requires the least number of extra components to tolerate multiple switching element failures. We prove that this is the only family of constructions that has this optimal fault-tolerance property.

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

We live in an age of information. The science of information has evolved dramatically in the last century. Scientists discovered fundamental truths about the bits and the bytes, and made great progress in the areas of computation, communication, and storage. Computation is the creation and the manipulation of information; communication is the transportation of information; and storage is the placement of the information. Around these three arms of the information science emerged the computer industry and the networking industry.

The advent of the Internet is changing the way that people manage and access information. In the last five years, the amount of traffic on the Internet has been growing at an exponential rate. The World Wide Web has evolved from a hobbyists' toy to become one of the dominating media of our society. E-Commerce has grown past adolescence and multimedia content has come of age. Looking forward, this growth will continue for some time. There are four general trends in the growth of the Internet:

First, Internet clients are becoming more numerous and varied. In addition to the ever-increasing number of PCs in offices and homes, there are new types of clients, such as mobile data units (cell phones, PDAs, etc.) and home internet appliances (set-top boxes, game consoles, etc.). In the future not too far away, these new types of Internet devices will pervade the Internet landscape.

Second, to support these new clients, new types of networks are being designed and implemented. Examples are wireless data networks, broadband networks and voice-over-IP networks. Technologies are being developed to connect these new networks with the existing Internet backbone.

Third, the content delivered over the Internet is evolving, because of the emergence of the new clients and new networks. There will be a growing presence of multimedia content, such as video, voice, music and gaming streams. The growth in content adds not only to the volume of the traffic, but also to the computation complexity in transporting and processing the traffic, thus accelerating the convergence between communication and computation.

Lastly, because of the abovementioned growth, new Internet applications will emerge, both on the server side and the client side. As the Internet penetrates deeper and deeper into everyone's life, the demand for security, reliability, convenience, and performance skyrockets. With the popularity of cars comes the invention of traffic lights and stop signs, the gas station and the drive-thru. As Internet makes its way into daily lives, the demand will grow for firewalls and VPNs, intrusion detection and virus scanning, server load balancing and content management, quality of service and billing/reporting applications. The list goes on and will keep expanding.

This growth of the Internet is bringing together the computer industry and the networking industry. The two industries will converge to build the infrastructure reliable and scalable enough to accommodate the Internet evolution. This poses new challenges, as well as interesting opportunities to the information science. The question is, how can we design the Internet infrastructure to meet the reliability and scalability requirement of the growing Internet?

1.1 Reliability and Performance of Networking Elements

The primary function of the Internet is for information to flow from where it is stored to where it is requested. The Internet is the network that interconnects all clients and servers to allow information to flow in an orderly way. The communication path between a client and a server can be viewed as a chain. Each networking device along the path between the client and the server is a link in the chain. Figure 1 illustrates a n example of a traffic path between a client and a server farm in the Internet.

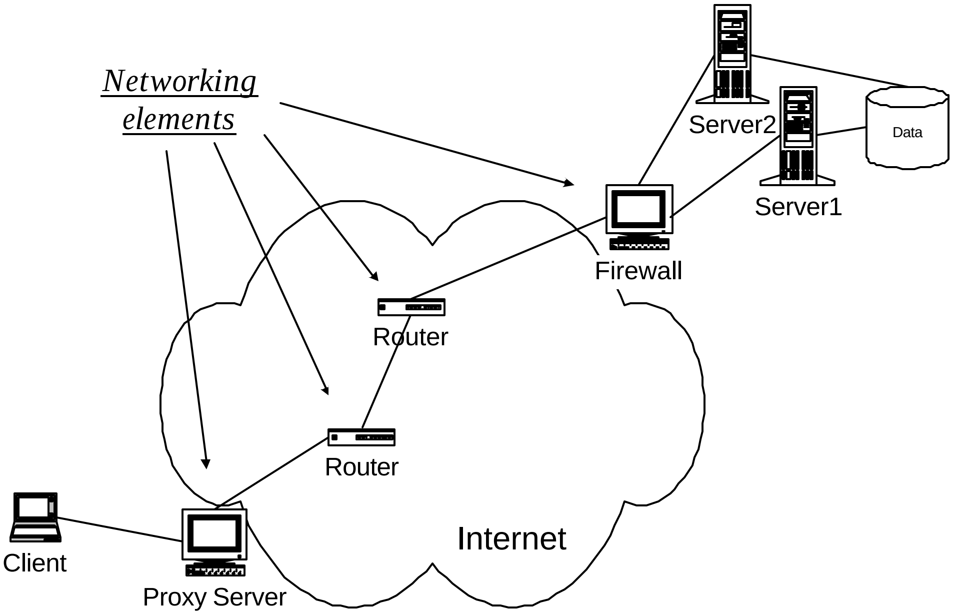


Figure 1 A traffic path between a client and a server farm

For example, for a user to receive an HTML page from yahoo.com, he issues a request which travels from the user's client, through a number of routers and firewalls and other devices to reach the Yahoo web server, before the data returns along the same or a similar chain. The strength of this chain, both in terms of throughput and reliability, will determine the user experience of the Internet. So, how do we make this chain stronger?

The key to reliability is redundancy. If one device fails, there must be a second device ready and able to take its place. If the second device fails, there must be a third one, and so on. The key to performance is processing power. To increase capacity and speed, the customer has a choice of using a bigger, faster processor, or by dividing the task among several processors working in concert. Using a single processor limits scalability to the state of the art in processors, so that performance can only be what Moore's Law will allow. Multiple processors working in a cluster provide a more flexible and scalable architecture. Capacity can be added or subtracted at will; the overall performance to price ratio is higher; and combined with intelligent fail-over protocols, such a cluster enables higher reliability.

A chain is only as strong as its weakest link, and the longer the chain, the weaker it is overall. To increase reliability and performance, one should look for ways to reduce the number of links in the chain and make each remaining link more robust. The weak links in the Internet infrastructure are single points of failure and performance bottlenecks. Single points of failure are devices that have no inherent redundancy or backup. Bottlenecks are devices that do not have enough processing power to handle the amount of traffic they receive.

Currently there are solutions that address both the reliability and the performance problem in the networking world. The most popular solution is typically referred to as the hot standby solution. In this solution, a secondary device is setup to be identical to the primary device. When the primary device fails, the secondary device detects it, and activates itself to perform the function of primary device. This is a very practical solution to address the single point of failure problem. However it does not increase the performance of the primary device, and therefore does not address the performance problem.

Use of load balancers is another solution that aims to address both the reliability problem and the performance problem. When a device is a single point of failure and performance bottleneck, more devices can be setup to perform the same function, and external load balancers can be set up at each of the subnet these devices connect to. The load balancers are capable both of balancing the load to these devices, as well as detecting the failures among these devices, therefore address both the reliability and performance problem. However there are limitations to this approach. Load balancers become additional links to the chain and can become points of failures and performance bottlenecks themselves and load balancers can not share application state among the devices.

We propose to increase the reliability and performance of the Internet by clustering the networking elements. This is using the concept of building a larger system by using multiple smaller units, to improve overall system performance and reliability. In the chain of links analogy, it is equivalent to strengthening one link without adding additional links. In some cases, it may even allow several links to be consolidated into one. By having multiple networking elements work together for the same purpose, this in effect creates a distributed system in a networking environment. For example, in Figure 1, instead of having one firewall front-gate the server farm, a cluster of firewalls can be used. The objective of this distributed system is to enable both load balancing and fail-over among the member nodes. To load balance, there must be a way to distribute network traffic to members of the cluster, so that the overall throughput is multiplied. To fail over, the healthy nodes must discover nodes that have failed, and take over the networking traffic from the failed node without interrupting the traffic flow.

This calls for the fusion between the field of computer networks and the field of distributed systems. Both of these fields have been very active areas of research and enjoyed many important and useful results. The research and development in computer networks fueled the entire networking industry. Important technologies emerged in all seven layers in the OSI seven-layer model. Great strides are being made in optical communication at the bottom layer of the communication stack, the Physical layer. In the last fifteen years, Ethernet has become the backbone at the data link layer (Layer 2) for local area networking. It not only permeates every office, but also enters more and more homes. The TCP/IP protocol suite is widely adopted around the world and has become the de facto standard for Network Layer (Layer-three) and Transport Layer (Layer-4). Meanwhile, trillions of bits of information are being communicated over the Internet using Session Layer (Layer-5), Presentation Layer (Layer-6), and application layer (Layer-7) protocols, such as HTTP and FTP.

At the same time, results from the field of parallel and distributed computing have enabled the making of supercomputers and other powerful computing systems. They push the frontier of scientific discovery by providing more computation power than ever thought possible. Their contributions range from depicting the history of our universe to helping prove open mathematical problems. Parallel and distributed computing technologies are also widely used in the business world for data processing and storage.

1.2 Contributions of this Dissertation

In this dissertation, we will be looking at how to build the bridge connecting these two distinctive fields of research, the field of computer networks and the field of distributed systems. We will try to answer the following questions: How to create the suitable distributed computing protocols for the networking devices, so that multiple network devices can collaborate in performing the same function? How to scale the overall throughput in a near-linear fashion without upper bound in networking throughput? In the presence of faults, how do our protocols discover and mask faults in a quick and transparent manner? How to achieve those goals in a way that complies with the existing networking protocols, and do that in the most efficient and least intrusive way?

The bulk of this thesis is devoted to the presentation of Raincore, a suite of distributed protocols for networking devices that try to answer the questions above [Fan and Bruck, 2001]. Raincore is a collection of protocols that provides reliable unicast, reliable multicast, group membership, state sharing, mutual exclusion, and resource allocation services. The Raincore protocols and services are designed to achieve the best performance in a networking environment, and are applicable to a wide range of networking devices, from layer-three devices, such as routers, to layer-7 devices, such as firewalls and proxy servers. Raincore provides fast failure recovery and near-linear throughput scaling, as demonstrated by a number of networking applications that have been built using it.

A distributed system allows multiple machines to work together as if they were a single system. The key challenge here is that all the machines in the cluster need to have consensus on the exact state of the cluster and make collective decisions without conflicts. To address the issue of reliability, a distributed system must also allow healthy machines within the cluster to automatically and transparently take over for any failed nodes. To address the issue of performance, all healthy nodes in the cluster must be actively processing in parallel, and each additional node must add processing power to the group, not detract from it. Creating such a solution for the Internet infrastructure is an exciting, but difficult task.

Raincore is designed to be such a system that allows Internet applications to run on a reliable, scalable cluster of computing nodes, so that they do not become single points of failures or performance bottlenecks. It manages the load-balancing, transparent fail-over, and helps the application to share the application state among the computing nodes. It scales horizontally without introducing additional hardware layers. Furthermore, multiple Internet applications can coexist with Raincore on the same group of physical computers. This reduces the number of links in this Internet chain, and therefore improves the overall reliability and performance of the Internet.

In this dissertation, after presenting the introductory background on the existing results in computer networks and distributed systems in Chapter 2, we'll introduce the architecture and the key protocols in Raincore Distributed Services in Chapter 3. We will introduce the Raincore protocols and services in the Transport Layer, Session Layer, and Presentation Layer of the networking stack. The focus will be on the Group Communication Manager, which situates in the Session Layer, and provides group membership management, reliable multicast, and mutual exclusion services for the nodes in a cluster. We'll prove that the Raincore group membership protocol meets one specification of the Group Membership Problem.

A number of applications have been implemented using the Raincore Distributed Service. In Chapter 4 we present four of them. The first is the SNOW prototype that provides clustered web service. SNOW evolves to CLOUD, which ensures that a pool of Virtual IPs is always available to the outside world, in the presence of node failures. Both SNOW and CLOUD were developed by me at Caltech, and are the first applications that use Raincore distributed service. Raincore is also used in commercial environment. RainWall is a clustering solution for firewalls using Raincore that allows load balancing and transparent fail-over to happen among a number of firewall nodes. RainWall is a shipping product from Rainfinity, a company that provides performance and reliability software for the Internet. RainFront further takes advantage of the cluster management and state sharing capabilities of Raincore and becomes an open scalable software platform for networking applications. In Chapter 4, we also compare the performance of the Raincore Session Layer protocols with broadcast-based group communication protocols and present the performance benchmarks of some of the Raincore applications.

Also included in this thesis are the interesting results on the optimal constructions for fault-tolerant Multistage Interconnection Networks [Fan and Bruck, 1997]; [Fan and Bruck, 2000]. Multistage Interconnection Networks is a distributed system that serves as a larger Layer-2 switch by grouping a number of smaller switches (typically 2 X 2 switches) together. It has the benefit of overall cost saving and built-in fault-tolerant capabilities. We discovered a family of construction that requires the least number of extra components to tolerate multiple component failures. We also proved that our family of constructions is the only family that has this property. In Chapter 5, we present the results on the fault-tolerant constructions on Multistage Interconnection Networks that use the minimal number of extra stages to tolerate multiple switch faults.

We conclude and point out the directions of future research in Chapter 6.

2 BACKGROUND ON COMPUTER NETWORKS AND DISTRIBUTED SYSTEMS

The work on Raincore suite of protocols and services started as a part of the RAIN research project [Bohossian et al. 2001]. RAIN stands for Reliable Array of Independent Nodes. The goal of the project is to bridge between networking protocols and distributed systems. During this project, we created reliable and high-performance distributed systems by clustering commercial off-the-shelf hardware using fault-tolerant software. Figure 2 shows a 10-node RAIN cluster constructed in the Parallel and Distributed Systems Lab in Caltech.

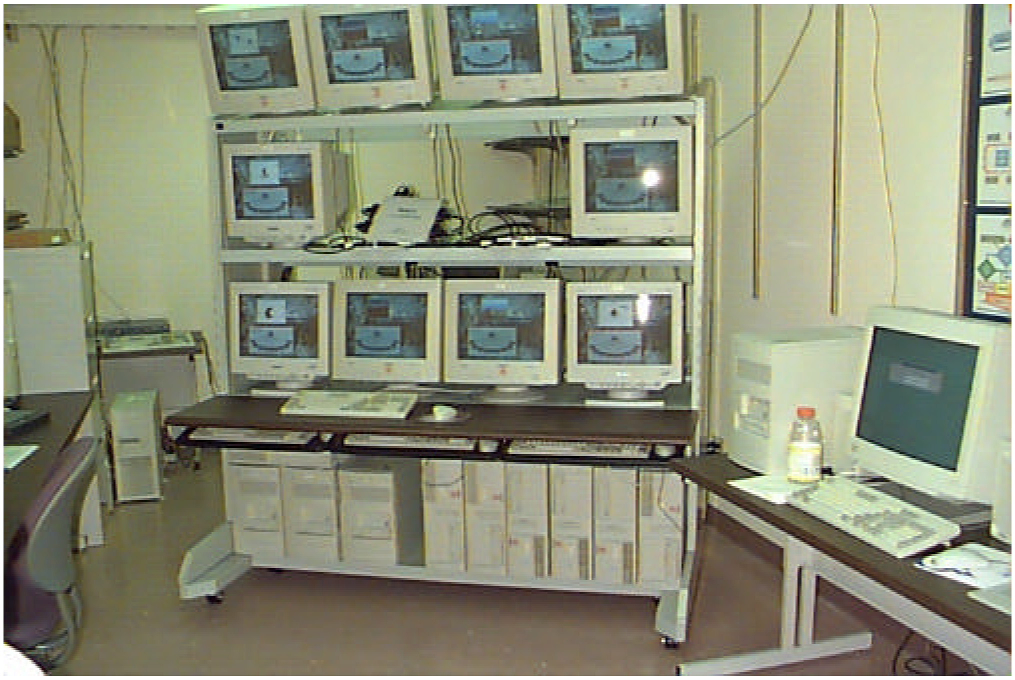


Figure 2 The ten-node RAIN cluster in Paradise Lab, Caltech

Raincore focused on the networking environment, as illustrated in Figure 1. We designed a set of distributed protocols and services that situate in the Layer 4, 5 and 6 of the networking stack. These protocols and services work together to enable the clustering of networking elements in that environment. In particular, the networking elements that we paid special attention to are the routers and their variants. Therefore comes the name “Raincore”. It stands for “RAIN Cluster Of Routing Elements”.

Part of the motivation for the Raincore is the wide gap between the state-of-the-art research in distributed systems in academia and the primitive high-availability solutions deployed in the Internet industry today. Most of the high-availability solutions that can be found in the Computer Networking industry are standby solutions. In a standby solution, a secondary idle node is present to monitor the primary node, and becomes active if the primary node fails. Being a useful high-availability solution, this solution does not take advantage of the processing power of the secondary node and offers no scalability. Raincore is designed to bridge this gap, to create a true distributed system for networking devices, so that multiple networking devices can work together to achieve combined performance and superior reliability.

Before digging into the details of the Raincore Architecture and Raincore protocols, we would like to introduce the background of the field of computer networks and the field of distributed systems.

2.1 What is a Computer Network?

First, what is a computer? By definition, it is a machine that can store and recall information and make calculations at very high speed. The first modern computer was invented in the 1940’s. It occupies a huge room and is capable of performing 5000 calculations per second. Since then, computer industry has made dazzling strides in both reducing the size of a computer and increasing its computation power. According to the famous Moore's Law, the computation power of the state-of-the-art computer chip doubles every 20 months. Indeed, in the last thirty years, the reality closely followed Dr. Moore's prediction.

What is a network? By definition, it is a group or system whose members are connected in some way. Since the beginning of the human history, we have been living in a network. The human society itself is a network that's interconnected by human relationships. We built irrigation networks to increase the farm production; we built transportation networks that linked the towns and the cities of the world; we built the power network that delivered electricity to every home; we built the telephone network so that everyone else in the world is but a few dials away.

What is a computer network? According to Andrew Tannenbaum in his classic namesake book, "a computer network is an interconnected collection of autonomous computers" [Tannenbaum, 1996]. Computer industry is a young industry. During the first twenty years of its existence, there was no need for a computer network. The industry was dominated by the mainframe computers. The basic functions of a computer are the computation and storage of information, and it is obvious why people need to interconnect the computers together. In a medium-size company or university there might have been one or two computers, sitting in a centralized computer center. People who need to use it would come to the computer center, enter the instructions and the data, and out come the results.

As computers continue to become smaller and more powerful, and as more and more business applications found their way into the computer, instead of sharing one computer for the entire company, we find computers on every desk. The big revolution came when the industry decided to merge the computation and communication of information. Voila, a computer network is born. The simple act of connecting the computers together immediately revolutionized the way the business was done.

Client-server becomes a dominating model of operation in most of the modern companies. In a client-server model, the clients are the PCs that sit on each user's desk. The clients serve as an access point to the information, and perform most of the computation on the data. The data, however, is stored on the servers. With this model, the clients perform all user-specific tasks, while the servers handle all the tasks that need to be shared among the users. This allows for both the scalability of computation power and the resource sharing among the users. The first servers in an enterprise world are file servers, which simply stored the bits and bytes of a data file. Gradually, more sophisticated servers are produced and being relied upon. Examples include both the indispensable database servers for an enterprise and ubiquitous web servers on the Internet. Although each of them works in somewhat different ways, the one thing in common is that they hold the data that multiple clients need to share.

In addition to efficient resource sharing, a computer network also provides a powerful communication medium among people. After the postal mail, telegraph, telephone and fax, email is the new mainstream way for interpersonal communication. The Internet and the World Wide Web are great manifestations of the computer networks that linked the world together. Some of the old communication medium is being converted to go over the Internet. We have already seen fax and telephone over the Internet. We will also see more and more video conferencing, as well as meeting and collaboration over the web. In short, the advances in computer networks have been and will continue to reshape the lives of everyone.

There are different types of computer networks, and different ways of categorizing them. One way is by looking at how information flows from the source to the destination. There are generally two types of networks: point-to-point networks and broadcast networks. In a point-to-point network, each message can only have one member of the network as its source, and one member of the network as its destination. The other members of the network, who are neither the source nor the destination, will not be able to see the message. In contrast, in a Broadcast Network, all members of the network will be able to see all messages. They will simply ignore the messages which are not addressed to them.

Another way of categorizing the computer networks is by looking at the distance among the members of the network. There are generally two big classes of networks: LAN (Local Area Networks) and WAN (Wide Area Networks). Local Area Networks usually span within a single campus of up to a few miles in size. Because they situate close to each other, in general we can expect high bandwidth and low latency from any member to any member in a LAN environment. In contrast, Wide Area Networks connect members thousands of miles apart. WAN usually uses point-to-point networking technologies while LAN uses the broadcast ones. The divisions nowadays between WAN and LAN, however, are not clear-cut. There are MANs (Metropolitan Area Networks), which are networks that span larger areas than LANs, but still use a broadcast medium to connect its members. There are also special LANs that are geographically distributed, but connected with private high performance links. This is popular with the bigger enterprises for connecting their major offices in different cities in the world.

2.1.1 The Network Protocol Stack

The basic functionality of a network is to allow communication among its members. To be able to understand each other, the members must speak the same language and follow the same rules. In computer networks, such rules are called protocols. There is not one, two, but many computer networking protocols. Some of them compete with each other, and some of them work with each other. The International Standards Organization (ISO) in 1983 proposed a seven-layer reference model for computer networking protocols [Day and

Zimmermann, 1983]. This model is popularly referred to as the OSI (Open Systems Interconnection) Reference Model (Figure 3). Almost all modern computer network protocols can be categorized with this reference model.

OSI Layer 7: Application Layer

OSI Layer 6: Presentation Layer

OSI Layer 5: Session Layer

OSI Layer 4: Transport Layer

OSI Layer 3: Network Layer

OSI Layer 2: Data Link Layer

OSI Layer 1: Physical Layer

Figure 3 The OSI seven layer reference model

Physical layer is the bottom-most layer that deals with how information is being transmitted over different physical medium. Currently the predominant physical mediums are copper wires and fiber optics. Typically, electrical or optical waveforms are being generated corresponding to the bits and bytes of information that needs to be transmitted. The physical medium by nature is analog, while the information is typically discrete. The physical layer protocols determine how the conversion should happen.

Physical layer only has the knowledge of how to transmit a bit of information from A to B. It has no idea which bit is the beginning of a transmission, which bit is the end of the transmission, nor has it any capability of detecting that the transmission was unsuccessful due to the noise on the wire. In addition, physical layer protocols only specify 1-to-1 transmissions, and are not concerned with how multiple senders and receivers interact with a shared communication channel. Such functionalities belong to the data link layer.

The key concept in the data link layer is the concept of a ***frame***. This is a block of data that is transmitted logically in an atomic way. Most data link layer protocols are capable of resending lost frames by the use of acknowledgement frames. Some protocols even have the capability of dealing with damaged frames and duplicated frames. A special sub-layer of the data link layer, the MAC (Medium Access Layer) layer, also determines how to control access to a shared broadcast channel. This is particularly important for a Local Area Network. The most popular data link layer protocol in a LAN environment is IEEE 802.3, also known as the Ethernet. It uses a version of a MAC protocol called CSMA/CD (Carrier Sense Multiple Access with Collision Detection) [Kleinrock and Tobagi, 1975]. Other important Data Link protocols include IEEE 802.4 (Token Bus), IEEE 802.5 (Token Ring), and IEEE 802.6 (Distributed Queue Dual Bus).

There are many millions of computers in the world that are connected directly or indirectly to the biggest computer network, the Internet. It is impossible for every computer to have a direct connection to all of the other computers. Therefore, we need the concept of a ***subnet*** (sub-network.) A computer network often is consisted of a number of subnets. A member of the subnet can send a network message directly to any other member in the same subnet. However, for a message to reach its destination in a different subnet, it needs to travel through intermediate stations, which are connected to multiple subnets. Such intermediate stations are commonly referred to as ***routers***. The job of finding the right routers to route a message belongs to the network layer. The most popular network layer protocol today is IP, the Internet Protocol.

The fourth layer from the bottom is the transport layer. This is the first “end-to-end” protocol layer. It does not care on which physical medium or via which path should the message be sent. All it knows and cares about is the source and the final destination of the message, and what needs to happen at these endpoints. The actions a transport layer protocol may take include splitting the message into smaller units at the source and reassemble them at the destination.   It may multiplex multiple messages into one communication channel or create multiple channels for faster transport of a single message. It may also incorporate flow control algorithms that the receiver notifies the sender of its available buffer, so that the sender does not overwhelm the receiver. The transport protocols can range from very simple ones, such as UDP (User Datagram Protocol), to more sophisticated ones, such as TCP (Transport Control Protocol).

Layer 5 in the OSI model is the session layer. A session is a long-lasting data connection among members of the network that may also provide enhanced useful services. For example, SSL (Secure Socket Layer) protocol can be considered a session layer protocol. While it is not as well known as some of the other layers, as we will find out in Section 2.3, it is a very important layer for the Raincore architecture.

Similar to the session layer, the presentation layer is less referred to both in the networking industry and in the academic research. One of the main reasons for their anonymity is that the TCP/IP protocol suite largely ignored these two layers [Stevens, 1994]. The duty of this layer is to mask the actual movement of bits and bytes among different members of a heterogeneous network transparent to the applications. Initially, things like the data serialization, character encoding, byte-order aligning of the data are typically functionalities in this layer. In this thesis, some very important protocols belong to this layer.

The top-most layer in the OSI seven-layer model is the application layer. A networking protocol, if it belongs to none of the other six layers, belongs to the application layer. There is a large variety of application layer protocols. They are driven by the diverse needs from the applications. For example, FTP (File Transfer Protocol) is designed to transfer files, while

SMTP (Simple Mail Transfer Protocol) is designed for electronic mail delivery. DNS (Domain Name Service) maps host names onto their network addresses, while HTTP (Hypertext Transfer Protocol) is the backbone of the World Wide Web.

For a computer network to function well, a suite of network protocols is needed. Different protocols in the suite can be categorized into different layers according to the seven-layer reference model. The most popular protocol suite today, the TCP/IP protocol suite, was not exactly designed to conform to the seven-layer model. However, it can be easily referenced by the OSI model. Originally, TCP/IP gained its popularity because it was part of UNIX and what the Internet was built upon. In the 1990's, along with the amazing growth of the Internet and the World Wide Web, TCP/IP reached everywhere, and is expected to be at even more places as the Internet reaches every wireless device and enters every home.

2.1.2 How does it work?

The protocols at different layers work together to generate, transfer, and process messages. Those messages are typically transmitted in the form of ***packets***. As a packet travels down the network stack, each layer adds an additional header to the packet. Contained in the header are the administrative information important for that layer. As a packet travels up the stack, each layer parses the header for that layer, and strips the header away before sending to the upper layers.

Now using the World Wide Web as an example, let's examine how protocols at different layers work together. Suppose I need to find some information about Caltech. From my home, on my web browser I typed in http://www.caltech.edu. Within a second, like magic, the homepage of Caltech appears on my screen. What exactly happened on my computer? How did the information I requested travel all the way from the source to the destination?

The first thing that the web browser application does is invoking the DNS protocol to look up which IP address corresponds to the domain name caltech.edu. DNS tells the browser that the IP address of caltech.edu is 131.215.48.51. The browser therefore creates an HTTP request, asking for the homepage from 131.215.48.51. HTTP is an application layer protocol that uses TCP, the transport layer protocol. The TCP module on my computer (whose IP address, let's say, is 66.27.171.182) tries to create a connection with the TCP module on 131.215.48.51.

Since the HTTP is not the only application module that uses the TCP module, TCP uses a 16-bit ***port number*** to identify the applications. The HTTP server uses a well-known port 80. The TCP module usually randomly picks a port for the HTTP client in the range between 1024 and 5000. In this case, let's say, the port number for the HTTP client is 1372. This five-tuple of information {source IP address, source port number, protocol, destination IP address, destination port number} uniquely identifies a network connection. In this case, the TCP connection for the HTTP request is identified as {66.27.171.182, 1372, TCP, 131.215.48.51, 80}.

The TCP uses a three-way hand-shake protocol to establish a point-to-point connection, of which the first packet to send is a SYN packet. The TCP module on my computer, therefore, asks the IP module on my computer to send out the TCP SYN packet to 131.215.48.51, port 80. The IP module looks into its routing table to decide which router should this packet be delivered to in order to reach that destination. According to the routing table, router 66.27.171.254 is to be used.

When Ethernet is the data link layer protocol, another protocol is invoked to translate the IP address to the Ethernet MAC address. This protocol is called the ARP (Address Resolution Protocol). The ARP request is a broadcast message that is sent to all computers on the same subnetwork. My computer sends out an ARP request, asking who owns the IP address 66.27.171.254. The computer who owns this IP address would send an ARP reply with its Ethernet MAC address. This allows my computer to send out the Ethernet packets to 66.27.171.254, the first router on its way to the source of the information.

When 66.27.171.254 receives the packet, the IP layer sees that it is a packet destined to 131.215.48.51. It looks into its own routing table and forwards it to the next hop router. A message would typically travel through a number of routers before it reaches the final destination 131.215.48.51. Typically, each router is a layer-three device that is only responsible for the routing of the packets. However, there exist more intelligent routers which perform more tasks than simple routing, such as security and traffic management functionalities. These devices are referred to as application gateways.

When the message reaches the destination, 131.215.48.51, it would travel up the TCP/IP protocol stack, with each layer stripping away its headers ensuring the integrity of the message. When it reaches the application, the HTTP web server, the application understands that a remote user at 66.27.171.182 would like to receive some information about Caltech. It creates the message that contains the information that my computer requested, with the destination 66.27.171.182, and sends it back down the stack. There could be more than a single message exchange to complete delivery that I requested. These messages would travel through the Internet, through the hops of routers, back to my computer. And voila, the information showed up on my browser.

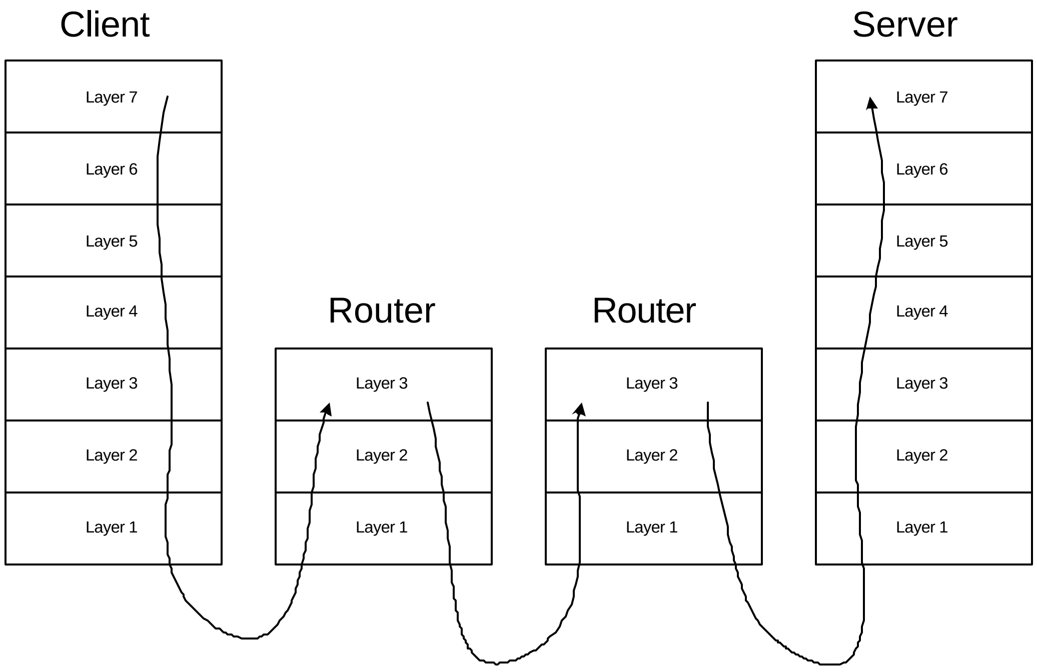


Figure 4 Path of a packet

Figure 4 illustrates the typical path of a typical packet from the source to the destination. As we can see from the figure, the endpoints are seven-layer devices. On the server side, applications include HTTP web servers, TELNET terminal servers, FTP file servers, NFS file servers, and a SQL database server etc. Corresponding clients exist at the client side. Typically in such a client-server environment, the server centralizes the information, and the clients distribute the user interfaces. The CPU processing occurs both on the servers and on the clients.

2.1.3 Life on the Edge

As Internet continues to grow, the boundary between communication and computation becomes increasingly blurred. Instead of what is shown in Figure 1, many routers become more than a pure communication device, and become sophisticated computation devices as well. This is particularly true for a segment of the Internet infrastructure, commonly referred to as the Internet Edge.

Where is the Internet Edge? If we see the Internet as a sphere with many servers and clients hanging from it, Internet edge is the crust of the sphere. If we zoom in, the Internet edge are all the devices that connect the private enterprise network with the public Internet infrastructure. There emerged a large number of useful and powerful network software applications at the Internet edge, in the areas of security, traffic management, content distribution, and statistics collection. Let's take a look at a few examples.

Firewall is possibly the most popular Internet edge software. It serves as a gatekeeper to a private network, allowing the legal traffic to pass, while stopping the illegal traffic. User sets up a policy to specify what traffic is legal, and all other traffic will then be dropped. This aims to prevent unauthorized access to the private network from the Internet. There are commonly two approaches for the implementation of a firewall. The first approach is to implement firewall as a packet filter. A packet filter firewall is a pseudo seven-layer device. It incorporates a special layer-three module that peeks into the application-level information. It however does not terminate TCP or application connections. The second approach is to implement firewall as a proxy server. A proxy server is a seven-layer device that intercepts connections, pretends to be the server to the clients, and pretends to be the clients to the server. There are trade-offs between the two approaches. Packet filters are typically faster, whereas the proxy servers are regarded more secure.

Many enterprises have geographically distributed offices that are connected by the Internet. To prevent sensitive data from being eavesdropped while they travel through the Internet, Virtual Private Network (VPN) software can be implemented at the edge between the offices and the Internet. Before the data leaves the office, the VPN software encrypts it before releasing it onto the Internet. The VPN software on the receipt side decrypts it and sends it to the destination. This way, VPN software turns the very public Internet into a virtual private network between the offices.

Network Address Translation (NAT) is another interesting application at the edge. IP address is composed of 4 bytes of information. It means that it is possible to have about 4 billion unique IP addresses. While this may seem a lot, it is becoming not enough for the growing number of computing devices on the Internet. Fortunately, there are some special segments of IP address, such as 10.x.x.x and 192.168.x.x that are designated to be private IP addresses. NAT has become a common practice that within a private corporate network, the private IP addresses are used. When clients need to access the Internet, the edge devices will perform the "translation". They will map the client IP address to a public IP address and a unique source port number. This allows multiple private clients to share one public IP address. This both solves the IP address scarcity problem, and also turns out to be an improvement of security and manageability as well.

Firewall, VPN, and NAT are just three examples of the myriad of network applications that emerged. As networking devices become more complex and sophisticated, avoiding single point of failures and performance bottlenecks in the networking infrastructure becomes a bigger challenge. This leads to the field of distributed systems.

2.2 What is a Distributed System?

According to Tannenbaum in his book, *Distributed Operating System* [Tannenbaum, 1995], "a distributed system is a collection of independent computers that appear to the users of the system as a single computer." One might ask, why do we need distributed systems? What advantage does it have over a single computer?

Distributed system is a technology in existence for more than 20 years. Its history is shorter than the history of the computer industry. In the early days of computers, distributed systems were not needed. There was Grosch's law that the computing power of a CPU is proportional to the square of its price. This means that the performance to price ratio is higher for faster CPUs. To create a faster computing system, it is always cheaper to buy a computer with a faster CPU. This economics deemed distributed systems to be not necessary.

With the progress made in the microprocessor technology in the 70's and 80's, however, Grosch's law no longer holds. The performance to price ratio has become a concave curve. This means that to perform the same number of computations per second, it is cheaper to use weaker CPUs, several of them, than to use a single but more powerful one. This new economics is one of the most important reasons for the birth of distributed systems.

In addition to the fact that distributed systems provide cheaper performance, it may also be able to achieve the performance level not at all achievable by a single CPU computer. Despite the strides made in the microprocessor technology, there are physical limitations to the speed of a processor. Being able to allow a number of CPUs to work together to perform the same task, simply add an independent dimension on how performance can grow. Combining it with the fastest chips can satisfy the computation needs not satisfiable otherwise.

Furthermore, distributed systems allow incremental growth of computation power without a complete overhaul of the system. While no one can foresee the future of the information technology, one thing we can foresee is that the amount of information, and the amount of information is going to continue to grow. When the computation need outgrows the current system, without a distributed system, the company needs to replace the old computer with a more powerful new one. This operation may incur much trouble and cost. A distributed system can potentially allow the user to upgrade computation, communication, and storage capabilities of the system with complete transparency to all the clients.

This in effect achieves the virtualization of computers. A distributed system allows a user to see the computer as a logical device, rather than a physical device. Scaling up the physical device does not affect the logical identity of the computer, therefore, it will cause less trouble and cost. Another part of enabling the virtualization of computers is that not only addition of physical computers is transparent to the users, so is the removal of physical computers. With the addition of the nodes the system experiences performance scaling. With the removal of the nodes, the system incurs graceful performance degradation.

A distributed system can be designed to be fault-tolerant. In a fault-tolerant distributed system, when part of the system fails, the overall system continues to function. This is an important feature of a distributed system, sometimes even more important than all of the performance and scalability considerations combined for mission-critical applications. Not only is physical failure addressed by such a system, so can the routine maintenance work performed on the system without downtime. User can take offline part of the system to perform maintenance on, while the other part continues to take on the workload. The implementation of a fault-tolerant system can greatly increase the overall reliability and availability of an IT infrastructure.

How to build a fault-tolerant distributed system? It is not easy. There has been much work in both the theory of modeling distributed systems, and the practice of implementing one. The book *Distributed Systems* [Mullender, 1993] is compiled to include chapters authored by leading experts in the field and provides a good introduction to the basic concepts in distributed computing.

2.2.1 Models of Distributed Systems

One of the first theoretical challenges is that a distributed system is not the most intuitive to model. Without a good model, correctness proof becomes very difficult. In the last twenty years a number of models emerged. These include the State Transition Systems [Lam and Shankar 1984], input-output automata [Lynch and Tuttle 1987; Lynch and Tuttle 1989], UNITY [Chandy and Misra 1988], and temporal logic of actions (TLA) [Lamport 1989; Lamport 1990]. These models explore the states of modules, and the events and actions that serve as the input and output of the state machine. They provide ways to formally prove the properties of a distributed system.

One of the first concepts in the distributed systems is the difference between a synchronous system and an asynchronous system. In a synchronous system, the relative speeds of processes are assumed to be bounded, and so are the communication delays between the computing nodes. In an asynchronous system, no such assumptions were made. It is much easier to prove properties of a synchronous system than of an asynchronous one. However, building a synchronous system is much more difficult, and in many cases impractical or impossible. Thus we'll focus our attention on asynchronous systems in this dissertation.

Fault-tolerance in a distributed system is an important consideration. Not only because it is one of the key advantages of a distributed system, but also because a distributed system, by definition, employs multiple elements in a system. The larger the number of independent elements, the higher the probability that some of them will fail. This makes fault-tolerance an imperative requirement for larger distributed systems.

To build a distributed system that tolerates failures, we need, first of all, to model the different types of failures. There have been different models of failure scenarios. In this dissertation, we classify the interesting failures in an asynchronous distributed system into the following five types.

Permanent Processor Failure: A processor fails by halting and it remains in the halting state. Also referred to as "crash" or "failstop".

Transient Processor Failure: A processor fails by halting for a period of time.

Permanent Link Failure: A communication link between two processors fails by omitting all messages that are transmitted on that link.

Transient Link Failure: A communication link between two processors fails by omitting a subset of the messages that is transmitted by that link.

Byzantine Failure: A processor fails by exhibiting arbitrary behavior.

In an asynchronous system, each operation can be arbitrarily fast, or arbitrarily slow. This makes reliable failure detection an impossibility [Fischer et al. 1985]. The reason for the impossibility is that a failed processor can not be distinguished from a very slow one. This impossibility result is a fundamental result that shows the native challenge of creating a reliable distributed system in an asynchronous environment. Researchers have been searching for ways to go around this impossibility result. Possible workarounds include probabilistic approaches [Brancha and Toueg 1983] and specification weakenings [Neiger 1996; Franceschetti and Bruck 1999].

Failure detection is an essential component for reliable group communication. Group communication is a form of one-to-many communication, or multicast. It differentiates from one-to-one communication (unicast) in that it allows one sender to send a message to multiple receivers at the same time. There are two categories of group communication: closed group communication and open group communication. In a closed group communication, the sender itself must be one of the nodes in the group. The purpose of a closed group communication is to allow reliable and efficient communication within the cluster, so that the nodes in the cluster can successful perform one job. Open group communication allows the sender to be out of the group. This allows any computer in the world to communicate to the distributed system like communicating to a single entity.

2.2.2 Fault-tolerant Group Communication

Reliability in a group communication is more complicated than reliability in one-to-one communication, because there are more than one receiver. Reliability needs to be guaranteed even when any of the following three scenarios occur: A new node joins the group; an existing node voluntarily left the group; an existing node left the group due to its failure. This requires that all nodes currently in the group have agreement on the group membership. This problem is commonly referred to as the Group Membership Problem (GMP), and has been extensively studied in the field [Birman and Joseph 1987; Chandra et al. 1996].

Achieving agreement on group membership is a fundamental prerequisite for achieving reliable group communication. There are three aspects to reliable group communication:

Reliable delivery: If a receiver is able to receive messages, it must eventually receive all messages sent to it.

Atomic delivery: When a message is sent to a group, it will either arrive correctly to all members of the group or to none of them.

Consistent message ordering: When a group receives a series of messages, the nodes in the group must receive these messages in the same order.

The reliable delivery property for group communication is no different from the reliable delivery of a unicast message. A mechanism as simple as a receipt acknowledgement system provides a solution. The atomic delivery and consistent message ordering properties are unique to group communication. They make it easier to program a distributed system. Consistent ordering of multicast messages does not mean correct ordering of multicast messages. While message A may be sent before message B, as long as every node in the group agrees that message B happens before message A, consistent ordering of those two messages is achieved, and that ordering is BA. Logical clocks are often used to achieve the consistent ordering of multicast messages [Lamport 1978].

A number of group communication systems have been built. ISIS built at Cornell [Birman 1993; Birman and Van Renesse 1994; Birman and Joseph 1987] is a classic example. ISIS introduces the ideas of a loosely synchronous system and virtually synchronous system. As we mentioned previously, a synchronous distributed system is a beautiful thing, except that it is impossible to build. A loosely synchronous system weakens the requirements. In a loosely synchronous system, atomic broadcast messages arrive to all nodes in the group in the same order. A virtually synchronous system relaxes the requirement even further. It only guarantees the correct ordering of messages that are causally related to each other.

ISIS made both the loosely synchronous system and the virtually synchronous system possible, by defining and implementing three broadcast primitives: ABCAST, CBCAST, and GBCAST. ABCAST and GBCAST provide loosely synchronous communication, while CBCAST provides virtually synchronous communication. Besides ISIS, there are other examples of distributed systems that have been implemented. This includes HORUS projects [van Renesse et al. 1994], the TRANSIS project [Amir et al. 1992], the TOTEM project [Amir et al. 1995], and the MPI project [Gropp et al. 1999]. These projects assume a broadcast communication medium, and designed group communication systems of different scalability and overhead requirements.

2.2.3 Reliable Distributed Computing

On top of the reliable group communication substrate, reliable distributed computation can be designed. In a distributed system, the synchronization primitives that are easy to implement in a centralized system become much more complicated. In a centralized system, a critical region is used to control access to a shared resource, and guarantees the mutual exclusion that only one thread can access that resource at any one time. In a distributed system, a critical region becomes more difficult, but not impossible to implement. A lot of good research has taken place to study different ways to implement the critical regions and mutual exclusions [Raynal 1991; Chandy et al. 1983]. These methods differ in performance. Specifically, they differ in the amount of delay between the request for the mutex and the receipt of the mutex, and they differ in how many communication messages are needed in obtaining a mutex.

With the availability of mutual exclusion primitives, it is possible to implement distributed shared memory (DSM) [Li 1986; Li and Hudak 1989] across multiple computers in the same group. DSM provides a single virtual memory space to the users without building a physical shared memory, which is costly and not always feasible. In principle, a distributed shared memory can allow programmers to program just like on a single-processor computer. In reality, however, programming in this model will usually result in a very poor performance of the program. To improve the performance of a distributed shared memory, the consistency model of a memory should be relaxed.

The most stringent consistency model is strict consistency. In this model, any read to a memory location x returns the value stored by the most recent write operation to x. To implement strict consistency in DSM, we must count the time of the read and write operations not at the moment when the requests are issued, but when the requests have been performed. Even under this assumption, a mutex needs to be acquired each time any memory location is to be accessed for write or read operation. This means that a significant delay is associated with every read or write operation. It is very costly, both in delays and in the amount of mutexes needed.

To improve performance, other weaker models have been proposed. These include sequential consistency [Lamport 1979], causal consistency [Hutto and Ahamad 1990], PRAM consistency [Lipton and Sandberg 1988], processor consistency [Goodman 1989], weak consistency [Dubois et al. 1986], release consistency [Gharachorloo et al. 1990], and entry consistency [Bershad et al. 1993].

Based on different consistency models, a number of DSM systems have been implemented. These include page-based DSM systems, such as IVY [Li 1986; Li and Hudak, 1989], shared-variable DSM systems, such as Munin [Bennett et al. 1990; Carter et al. 1991; Carter et al. 1994], and Midway [Bershad and Zekauskas 1991; Bershad et al. 1993], and object-based DSM systems, such as Linda [Gelernter 1985; Carriero and Gelernter 1986; Carriero and Gelernter 1989] and Orca [Bal 1991; Bal et al. 1990; Bal et al. 1992].

2.2.4 Distributed Applications

As we can see, the field of distributed systems has been a very active and fruitful area of research. The results have been widely used in a range of applications. First, powerful multi-computers have been constructed to achieve performance not achievable by a single computer for scientific computation. Most of such systems were designed and constructed in the academia and research labs. Prominent examples include the NOW (Network Of Workstations) project at Berkeley [Anderson et al. 1995], the SHRIMP project at Princeton [Damianakis et al. 1997], and the Beowulf project at NASA [Becker et al. 1995]. These systems can bundle together, tens, even hundreds of cheaper computers to create a powerful computing system that demonstrates superior performance to price ratio.

Meanwhile, in the computer industry, there also emerged a class of server clustering technologies that took advantage of the results in distributed systems. These systems are motivated by the need to scale up the performance of the servers in a client-server model. In the client-server world, one server usually needs to serve a large number of clients simultaneously. This asymmetry leads to a major challenge: the scalability of the server. The microelectronics industry has been building ever-increasingly powerful CPU chips, so that they have enough capacity to power the server to handle a large number of less powerful clients. However, the economics of microelectronics tells us that there are two limitations to this approach. First, to increase the speed of a chip by 100%, the cost is usually much higher than 100%. Second, there is a physical upper bound on how fast a chip can be. These limitations lead to server solutions that use multiple computers. SunCluster and Microsoft Cluster Server are examples of such systems.

Recently there emerged another approach to address the server bottleneck problem in the client-server model. The solution is to use distributed systems to decentralize the server functionality onto the clients themselves. It uses the spare computation cycles and disk storage on the clients to function as a virtual server. Thus there are no physical boundaries between the servers and clients. This model of computation is sometimes referred to as peer-to-peer computing. The challenge this technology faces in getting adopted by the enterprise world is how to reduce the management cost once the servers reside on the thousands of desktops.

From a certain perspective, distributed system is a subfield of computer networks. However, the distributed protocols and algorithms that can be found in networking devices, such as routers, are very limited. There are hot standby technologies, such as HSRP (Hot Standby Router Protocol) and VRRP (Virtual Redundant Router Protocol), but very little beyond that. One reason for the lack of sophistication is that there has not been too much need, until recently, when there are more and more computationally intensive applications found on the path of data, not endpoints of data. This motivates the creation of Raincore, a suite of distributed protocols and services designed to help networking devices become more fault-tolerant and performant.

3 RAINCORE ARCHITECTURE AND PROTOCOLS

3.1 Architecture

The advance of Internet is bringing together communication and computation. Raincore reflects this convergence by building distributed computing protocols into the communication stack. Raincore is first and foremost a distributed network computing architecture. The overall architecture of the Raincore Distributed Protocols and Services is described in Figure 5. As Figure 5 illustrates, the Raincore Distributed Services are mapped into Layer 4 through Layer 7 in the OSI seven-layer networking model.

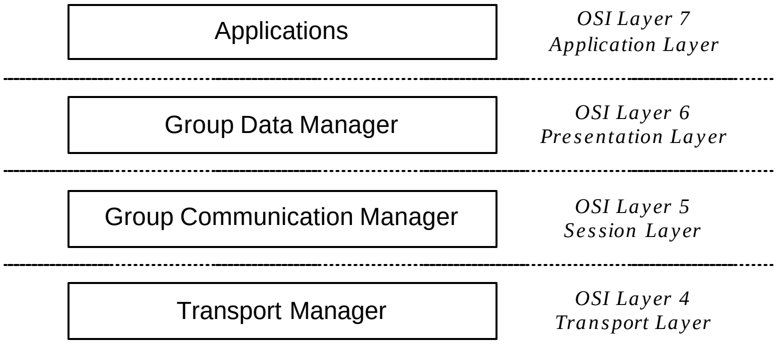


Figure 5 Raincore distributed services architecture

Given the Raincore architecture, we designed and implemented protocols and services that fit into this architecture. It is our goal that Raincore protocols and services will make it easy for the application developers' to port their application to run on top of a cluster of networking elements. Raincore will help the applications to share the traffic load among the nodes in the cluster; Raincore will also help applications to mask failures so that it will not affect the availability of the overall network service.

The bottom-most layer in Raincore is the transport layer protocols and services. This layer provides reliable atomic message unicast services. Reliable delivery and flow control algorithms belong to this layer. We implemented a Raincore module called Transport Manager at this layer. Raincore Transport Manager can use redundant path to communicate to the destination, and will notify the upper layers if all delivery attempts for a message fail. It can transport a message of any size and is responsible of breaking it into smaller units in sending and reassembling the units in receiving. The atomicity of the transport is ensured that either the whole message arrives, or none of it arrives. We have also built in the sliding-window flow control mechanism that is useful when transporting a large packet of data.

The Raincore session layer protocols and services situate on top of the Raincore transport layer protocols and services, and provide group communication. The module we designed and implemented for this layer is called Group Communication Manager. It performs group membership management and reliable multicast with total ordering among the group members. In addition, it offers a mutual exclusion service that is useful for the presentation layer modules, as well as the applications.

In the presentation layer, there are two Raincore modules. The Group Data Manager provides a shared-variable distributed shared memory. It maintains the shared data items and allows nodes in the group to read from and write to these data items. Being a presentation layer service, it hides all the communications from the users and provides an easy-to-use C programming environment for developers who need to create a distributed application. It also has available the global locks that provide critical regions for the shared data items, and transaction primitives for the atomic access of multiple data items.

Distributed applications can be developed on top of these three layers of services. Each layer provides an Application Programming Interface, making its features available to the applications. In this chapter, we'll go into more details into the protocols and algorithms at the transport layer and the session layer. In the next chapter, we'll introduce some application examples that use the Raincore services.

3.2 Transport Manager

The Raincore Transport Manager is a module placed in the transport layer within the Raincore architecture. It is a module situated at the bottom of the Raincore stack and requires the availability of an unreliable unicast interface to send and receive packets. In typical implementations, it uses either an UDP socket or raw socket as the packet sending and receiving interface. However, it is not limited to that and can use any packet-sending interface. This service can be used to communicate between nodes over a Local Area Network or a Wide Area Network. And there is no limitation to the Layer-2 protocols.

Raincore Transport Manager provides an atomic reliable unicast transport and failure-on-delivery notification to the upper layer. One might ask, why don't we use the industry-standard TCP. Both TCP and TM provide reliable unicast and flow control. However, Raincore Transport Manager provides the following functionalities that are not available in TCP:

1. Raincore Transport Manager is an atomic connectionless message delivery mechanism. A packet is either completely delivered, or not delivered at all. It does not employ the concepts of connections or streams. Therefore, there is no connection state information to track as nodes go up and down.

2. The Transport Manager generates notification to the upper layer both when it receives the acknowledgement from the destination, as well as when all sending efforts have failed. The second notification, the failure-on-delivery notification, is particularly significant, as it serves as a local-view failure detector for the Raincore Group Communication Manager.

3. The Transport Manager allows one node to communicate to another node using multiple physical addresses. It also allows a node to automatically resend a packet if earlier attempts failed. It will exhaust all of the configured resends on all physical addresses, before sending the failure notification to the upper layer. This allows redundant links between the nodes in the group, therefore making the group more resilient to link failures and less likely being partitioned. Packet-sending strategy using multiple physical addresses can be specified, where the physical addresses can be targeted in sequential or parallel order.

When the upper layer calls the Raincore Transport Manager to send a message to another node, the TM is responsible for partitioning the message into packets. The size of the packet depends on the packet-sending interface and the Layer-2 protocol that the TM is using. For sending UDP packet over Ethernet, it is a good idea to keep the packet size under 1.5 KB to avoid UDP packet fragmentation. TM employs a sliding-window protocol to provide reliable delivery and flow control. The receiving side won't pass the message to the upper layer until all packets related to that message have been received.

A Transport Manager allows multiple upper layer users and distinguishes them by assigning different channel IDs to different users. Multiple Transport Managers can also coexist on the same physical node by using different ports.

3.3 Group Communication Manager

Group communication is a key component for a distributed system in a network environment, similar to its importance in other distributed environments. Its goal is to provide a reliable and efficient many-to-many communication transport, and to continue to function as nodes leave and join the group, or the cluster. We characterize this functionality to be a session layer functionality and create a Group Communication Manager module to carry it out.

Group Communication Manager maintains the consistent group membership of the cluster and serves as reliable multicast transport to share state information among the member nodes. This module is used as a group communication transport to share arbitrary application state among the cluster group and to facilitate transparent fail-over of traffic from a failed node to a healthy node, without the clients or the servers being aware of the failures.

Group Communication sits on top of the Transport Manager. Multiple Group Communication Managers can coexist on the same node and use different channels of the same Transport Manager.

3.4 Group Membership Specifications

In an asynchronous distributed system, a Group Membership Protocol provides nodes in the cluster with a consistent view of the membership of that cluster. When a change in the membership occurs, either by nodes failing or leaving the cluster, or by new nodes joining the group, all current members of the cluster need to achieve agreement on the current membership.

It has been shown that it is impossible to find solution to the consensus problem in asynchronous environment [Fischer et al. 1985] and this result has been extended to group membership agreement [Chandra et al. 1996]. The root of the impossibility lies in the lack of a reliable failure detector: one cannot distinguish between a failed node and a slow node. However, as we mentioned, different distributed systems have been built despite of this impossibility. This motivated a series of recent research that explored nontrivial ways to weaken the specification of the Group Membership problem [Neiger 1996; Franceschetti and Bruck 1999].

It is a challenge to define a specification that is achievable, and yet no trivial protocol can achieve it. Franceschetti and Bruck proposed a specification based on the idea of Quiescent State. They accepted that when local failure monitor continues to send input, it is difficult for the protocol to make progress. But once local failure detector stops sending input, each process will enter Quiescent State, and progress can be made during the Quiescent State. In this section, we will use the their specification and prove that the Raincore Group Communication Manager protocols meet that specification.

The model assumed in Franceschetti and Bruck's Group Membership specification is an asynchronous distributed system, where processes (a.k.a. nodes) are identified by unique IDs. There is a communication channel between every pair of processes, and the communication channels are considered to be reliable, FIFO, and to have an infinite buffer capacity. Message transmission and processing times are finite but without upper bound. The failure model is permanent process failures, allowing processes to crash, silently halting their execution. Each process has a local failure detector which is not always accurate. The local failure detector sends input to the system of either process removal or process addition.

Each process p maintains two fundamental data structures: a set v_p that contains the current local view of the membership, and a sequence $S_p = [V_{p1}, V_{p2}, \ldots, V_{pk}, \ldots]$ of global views. A process's local view, v_p, contains a set of processes. These processes are currently live from this process' perspective. The process modifies the local view, v_p, based on the inputs from the local failure detector. The global view vector S_p contains an ordered and indexed list of global views. Each global view contains a set of processes. A Group Membership Protocol specifies when and how to extend the global view sequence S_p with new global views, based on changes in the local view v_p, and doing so consistently at different processes.

To formally present their group membership specification, we need to first define ***Consistent History*** and ***Quiescent State***.

Definition 3.1 Consistent History. A set of processes Q has a consistent history of views if the sequences S are the same at all processes in the set, unless sets V_{pj} and V_{qj} are disjoint. Namely:

$$\forall p, q \in Q \; \forall j: (V_{pj} = V_{qj}) \vee (V_{pj} \cap V_{qj} = \emptyset),$$

where p and q are processes IDs and V_{pj} and V_{qj} are the j-th elements in p's and q's global sequence of views respectively.

Definition 3.2 Quiescent State. A process p is in a quiescent state if it does not change its sequence of global views anymore. Namely:

ÿ($S_p = [S_p]$),

where $[S_p]$ is a constant sequence of sets.

They then define a specification consisting of four properties. If an algorithm meets all four of these properties, it has solved the Group Membership Problem with this specification.

Property 1 Agreement. At any point in time all processes have a consistent history.

$$\text{True} \Rightarrow \ddot{y}(\text{Consistent History})$$

Property 2 Termination. If there are no more changes in their local views, all processes eventually reach a quiescent state.

$$\forall p\text{: } \ddot{y}(v_p = [v_p]) \Rightarrow \Diamond(\text{Quiescent State}),$$

where each $[v_p]$ is a constant set.

Property 3 Validity. If all processes in a view v* perceive view v* as their local view and have reached a quiescent state, they must have view v* as the last element of their sequence of global views.

$$\forall p \in v^*\text{: } [\text{Quiescent State} \wedge \ddot{y}\ (v_p = v^*)] \Rightarrow \forall p \in v^*\ (V_{pmax} = v^*),$$

where V_{pmax} is the last nonempty set in the sequence S_p.

Property 4 Safety. Once a view is committed in the sequence of global views, it cannot be changed.

$$(V_{pj} = v^*) \Rightarrow \ddot{y}(V_{pj} = v^*)$$

By the introduction of Quiescent State, this specification weakens that of [Fischer et al. 1985] and [Chandra et al. 1996], and yet excludes the trivial solutions. The first property ensures that consistent history of global view is an invariant for the system. The second property guarantees that all processes in the system enter quiescent state once there is no more input from local failure detectors. The third property expresses progress that if all processes have agreement on their local views at quiescent state, they will commit it to the global view. The fourth property rules out some trivial solutions by preventing processes changing old views in their global view sequence.

In the next subsection, we'll present the group membership protocol used in GCM. We'll also prove that this protocol meets the above specification.

3.5 Group Membership Protocol

At the heart of the Raincore Group Communication Manager is a token-ring protocol. This protocol is the basic component in providing the consistent group membership, reliable multicast and mutual exclusion services. Token ring is one of the best-known protocols used in the distributed computing world. Started with the work by Chang and Maxemchuk [Chang and Maxemchuk 1984], and continued both in the ISIS/HORUS projects [Birman and van Renesse 1994; van Renesse et al. 1994] and the TOTEM project [Amir et al. 1995], token ring have been used to sequence the broadcast message ordering and manage the group membership.

The nodes in the group are ordered in a logical ring. A *token* is a message that is being passed at a regular time interval from one node to the next node in the ring. The Raincore Transport Manager provides the reliable unicast for the transmission of the token. In addition, the Raincore Transport Manager is also used as the local failure detector. When the Transport Manager fails to deliver a message to a node, it will notify the Group Communication Manager.

Group Communication Manager will take that notification as an input and take appropriate action depending on the internal state.

Token Signature	Sequence Number	Nodelist	Pointer to current node	Pointer to dest. node	Global View Number

Figure 6 Token message format

The fields on a token are shown in Figure 6. The signature field indicates that this is a token message. After the signature field is the sequence number field. Every time a token is being passed from one node to another, the sequence number on the token is increased by one. Also contained in the token is a field called *nodelist*. It is a cyclic ordered list of all nodees that the token traversed in the last pass around the ring. There is a pointer that points to the current in the nodelist, and a pointer that points to the destination node in the nodelist. Last but not least, the token contains the *global view number* that indicates the largest index in the committed global view sequence $S = [V_1, V_2, \ldots, V_k]$.

911 Signature	Sequence Number	Nodelist	Pointer to current node	Pointer to dest. node	Pointer to Orig. node	Status

Figure 7 The 911 message format

In addition to the token, there is another important message in the protocol, the 911 message (Figure 7). The 911 message is used for token regeneration and new node joins. It is also transmitted using the Transport Manager. Similar to the token message, the 911 message contains the signature and a nodelist fields. In addition to a pointer to the current node, and a pointer to the destination node, it also has a pointer to the originating node, which initiated this 911 message. The sequence number field indicates the sequence number of the last received token on the originating node. At the end is a status field, which can hold one of three values: YES, NO, or REJECT.

globalView	globalView Number	lastToken	lastToken Seq.	localView	newNodes

Figure 8 Local data structures on each node

Each node keeps a number of local data structures as well (Figure 8). First of all, each node keeps a vector of consistent history of group memberships, called *globalView*. The most recent group membership is referred to as *lastGlobalView*, and its index number in the globalView vector is called *globalViewNumber*. In addition, each local node keeps a copy of the last received token, stores it in *lastToken*. The sequence number of the lastToken is *lastTokenSeq*, and the nodelist on lastToken is *localView*. This shows that the nodelist on the token is in effect the localView of the node that it last resides. Furthermore, each node keeps a queue of *newNodes*, which lists all the nodes that are applying to join the cluster and not in the cluster yet.

As the token is being passed around in the ring, each node operates in a state machine. There are two sets of states that are important to the Raincore group membership protocol. The first is the View State, and the second is the Token State. The View State indicates each node's situation with regard to the agreement of local group membership views. The Token State indicates each node's situation with regard to the ownership of the token. Each node operates in any of the three View State, and any of the three Token State. Overall, each node operates in a nine-state state machine. Two of the nine states are not reachable, so it is actually a seven-state state machine. A node transits between these states, triggered by events.

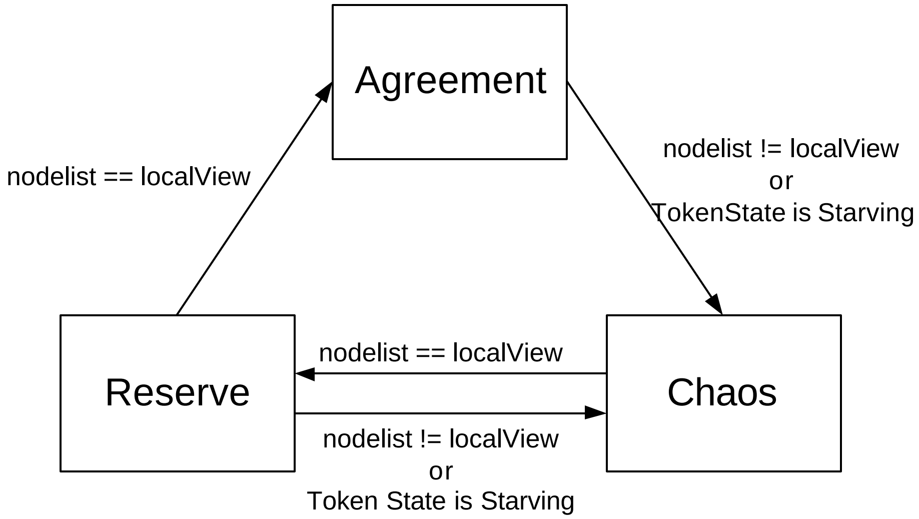


Figure 9 View State transition diagram

There are three View States (*Agreement*, *Chaos*, and *Reserve*). When a node is in Agree state, it indicates that this node's local view is in agreement with the local view of the other nodes in the cluster. A node enters Chaos state from Agreement state when it notices difference between its localView and the nodelist on the token. This indicates that there is an inconsistency among the local views of the nodes that needs to be resolved. A node transits from Chaos state to Reserve state when its localView is the same as the nodelist on the token. This indicates that all nodes in the current group have agreed on this new view, and it should be committed to the globalView on this node. A node will reserve the next free location on its globalView vector when it enters the Reserve State to get ready to Commit. When A node in Reserve State receives a token, it will compare whether the nodelist on the token is the same as its localView. If it is the same, it would commit to the globalView vector and enter the Agreement state. If it is different, it would commit NULL to the reserved location on the globalView vector and enter Chaos State. Figure 9 shows the state transition diagram for the View States. We can see that a node never enters Agreement state directly from Chaos state, and never enters Reserve State directly from Agreement State.

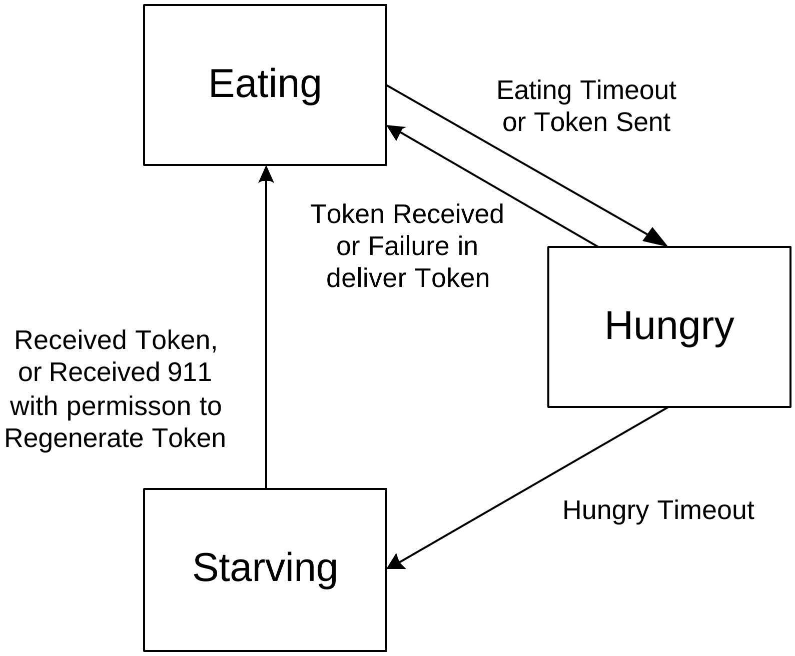


Figure 10 Token State transition diagram

There are three Token States (*Eating*, *Hungry*, and *Starving*). A node enters the Eating state when it receives a valid token. A token is valid when its sequence number is bigger than the lastToken Sequence number. When a node is in the eating state, it will keep the token for a preconfigured eating time, before trying to send the token to the next node. The transition from Eating state to Hungry state is triggered by the eating timeout. The node will update the token and pass the token to the Transport Manager to send it out. It enters the Hungry state after the token is passed to the Transport Manager. The normal view state transition can only occur in Eating token state.

When a node is in Hungry state, it expects the arrival of the token. There is a timeout associated with the Hungry state. If a node remains in the hungry state for a certain period of time, it enters the Starving state. The node suspects that the token has been lost, and a 911 message is sent to the next node in the ring. At the same time, whenever a node enters

Starving token state, it would enter Chaos View State. The node put the sequence number of its lastToken on the 911 message and initiated the status field with a YES, assuming that it will have the right to regenerate the token. If the token has not been lost, or if there is another node that has received a token with a higher sequence number, or the same sequence number and a higher node ID, it will veto the 911 message by filling the status field with a NO.

This 911 mechanism ensures that one and only one node in the cluster will regenerate the token. In the case when the token is lost, every live node sends out a 911 request after its hungry timeout expires. Only the node with the latest copy of token (i.e., the token with the highest sequence number) will be granted the right to regenerate the token. That node would receive its own 911 message with the status field YES. It will enter Eating state and create a new token making a copy of its lastToken.

The 911 message is not only used as a token regeneration request, but also as a request to join the membership. When a new node wishes to participate in the membership, it sends a 911 message to a ny node in the group. The receiving node notices that the originating node of this 911 is not a member of the group, and therefore treats it as a join request. The next time before the receiving node sends out the token, if the new node has not yet been added to the nodelist on the token, it will add it to the nodelist on the token, and will enter Chaos View State. It then sends the token to the new node. The new node thereafter enters Chaos View State as well. When all nodes commit the nodelist to their globalView vector, the new node becomes part of the group.

As we explained above, a node enters Hungry state when the Group Communication Manager passes the token to the Transport Manager to send it out. If the Transport Manager fails to send the token to the destination, it invokes a failure-on-delivery notification. The GCM would then enter the Eating token state. It would modify the destination node on the token, as well as the nodelist on the token, and enter Chaos view state. It will then pass the modified token to the Transport Manager to resend it and enter Hungry token state.

The Transport Manager also sends failure-on-delivery notification when it fails to send a 911 message. In this case, the GCM will only modify the destination node on the 911 message and send it via TM again. There is also a timeout associated with the Starving token state. When a node remains in the starving state for a while without receiving the token or permission to regenerate the token, it will send out a 911 message again. Figure 10 shows the state transition diagram for the Token States triggered by events.

Summarizing these two state machines, there are 7 possible states: Eating Agreement, Hungry Agreement, Eating Reservation, Hungry Reservation, Eating Chaos, Hungry Chaos and Starving Chaos. There are eight possible events in the system: the initialization, the receipt of a token message, the receipt of a 911 message, notifications from the Transport Manager on token delivery failure, notification from the Transport Manager on 911 delivery failure, Eating state timeout, Hungry state timeout, and Starving state timeout. The complete Raincore token-ring protocol can be described by the actions and state transitions triggered by these eight events. We show here the overall state transition diagram (Figure 11) and the pseudo code.

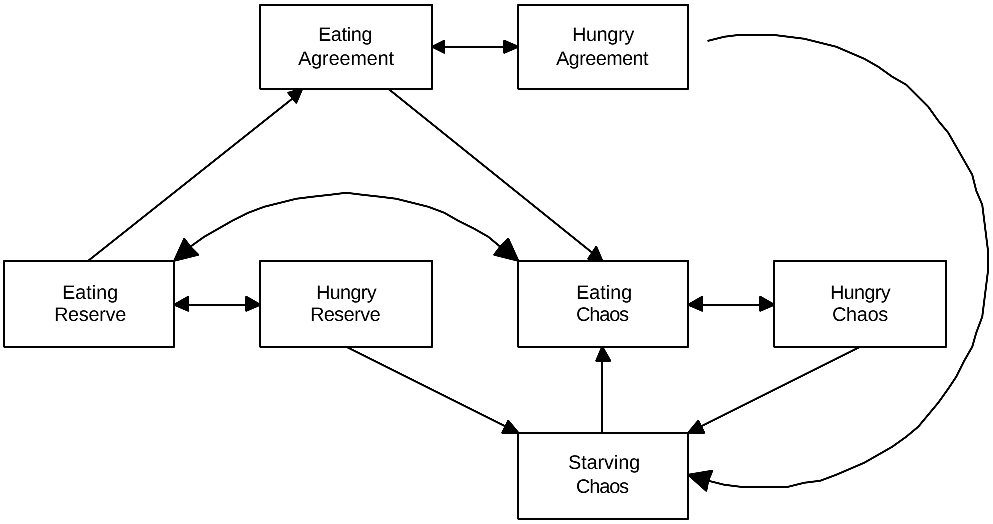


Figure 11 Overall state transition diagram

1. Initialization

```
ViewState=CHAOS;
tokenState=EATING;
Initialize local data structures;
Initialize the token with a unique intial sequence number;
Send Token;
TokenState=HUNGRY;
Start HUNGRY timeout;
```

2. Token Received.

```
If (tokenState == EATING)
{
    Drop the token;
}
else if (sequence number mis-match)
{
    Drop the token;
}
else if (tokenState == HUNGRY or tokenState == STARVING)
{
    tokenState=EATING;
    Start the clock for EATING timeout;
    Update token.currentNode;
    If (viewState == AGREEMENT)
    {
        if (token.nodelist != localView)
        {
            viewState=CHAOS;
        }
    }
    else if (viewState==CHAOS)
    {
        if (token.nodelist == localView)
        {
            viewState=RESERVE;
            increment globalViewNumber;
            Update token.globalViewNumber;
        }
    }
    else if (viewState==RESERVE)
    {
        if (token.nodelist == localView)
        {
            commit to globalViewVector;
            viewState=AGREEMENT;
        }
        else
        {
            viewState=CHAOS;
        }
    }
    if (token.globalViewNumber > globalViewNumber)
    {
        Update globalViewNumber;
    }
}
```

3. 911 Received.

```
Update 911.currentNode;
if (911 is from self)
{
    if (911.status == YES and tokenState == STARVING) {
        Recreate token from local copy of token;
        tokenState=EATING;
        Start EATING timeout;
    }
}
else if (911 is from one of the nodes in lastGlobalView)
{
    if (911.seq < lastTokenSeq)
    {
        911.status=NO;
        Send 911 to token.origNode;
    }
    else
    {
        Send 911 to the next node on the 911.nodelist;
    }
}
else if (911 is not from one of the nodes in the lastGlobalView)
{
    Queue it up onto the newNode queue, to be joined into the cluster.
}
```

4. Token delivery failure.

```
ringState=CHAOS;
Update token with new nodelist;
Update token pointer to destination node;
Increment token Sequence Number;
Update local copy of token;
if (nodelist contains only one node, self)
{
    viewState=RESERVE;
    Increment globalViewNumber;
    Commit to globalViewVector;
    Update token.globalViewNumber;
    Update local copy of token;
    viewState=AGREEMENT;
    tokenState=EATING;
    Start EATING timeout;
}
else
{
    Update local copy of token;
    Send token;
}
```

5. 911 delivery failure.

```
Update 911 destination pointer;
Send 911;
```

6. Eating timeout

```
if (the new node queue has node not in token.nodelist already)
{
    Add newNodes to the token nodelist;
    viewState=CHAOS;
}
Clear newNodes queue;
if (there is more than one node on token.nodelist) {
    Update the token destination;
    Increment token sequence number;
    Copy token to lastToken;
    Send token;
    tokenState=HUNGRY;
    Start HUNGRY timeout;
}
else {
    Restart EATING timeout;
}
```

7. Hungry timeout

```
tokenState=STARVING;
Start STARVING timeout;
viewState=CHAOS;
Initiate 911 with lastTokenSeq;
911.status = YES;
Send out 911 to the next node on lastGlobalView;
```

8. Starving timeout

```
Restart STARVING timeout;
Initiate 911 with local Token sequence number;
911.status = YES;
Send out 911 to the next node on lastGlobalView;
```

Here we show an example of the protocol at work. Suppose that there are four nodes in the system, A B C and D. Figure 23 shows the token-ring protocol at work with no failures. In the figure, on top of each node is the globalViewVector. To the side of each node is the history of the localView, the viewState, and the lastTokenSeq at each node. As the token circulates around the ring, the sequence number increases, and every node is in the Agreement State, and the globalViewVector has one entry ABCD.

In Figure 13, a node crash failure occurs on node B. Node A detects the failure, and it takes three rounds of token circulation before A, C and D all commit the view ACD to the globalViewVector. In Figure 14, node B recovers and keeps all its existing local state before the failure. It would enter STARVING state and sends out a 911 message to node C. Node C adds node B to the nodelist after it receives the token, and then it takes three rounds of token circulation before every node commits ACBD as the 3rd entry in their globalViewVector. The consistent history in the globalViewVector is preserved.

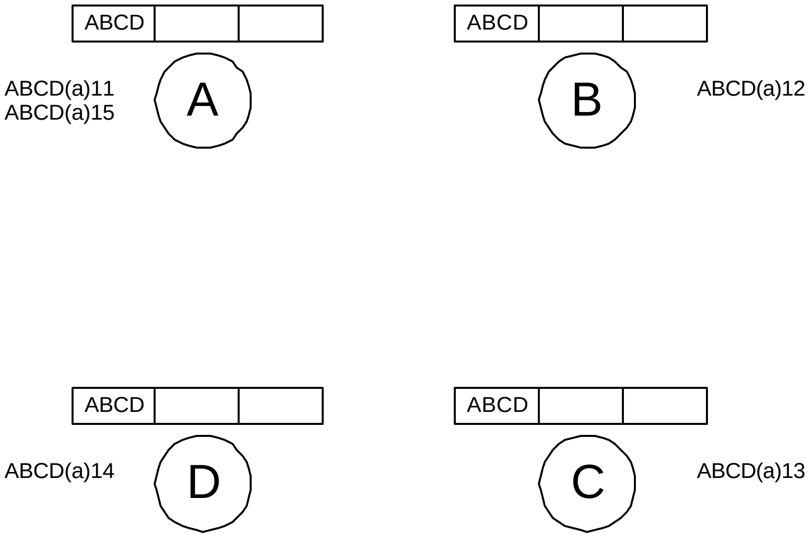


Figure 12 Example of token-ring protocol without failures

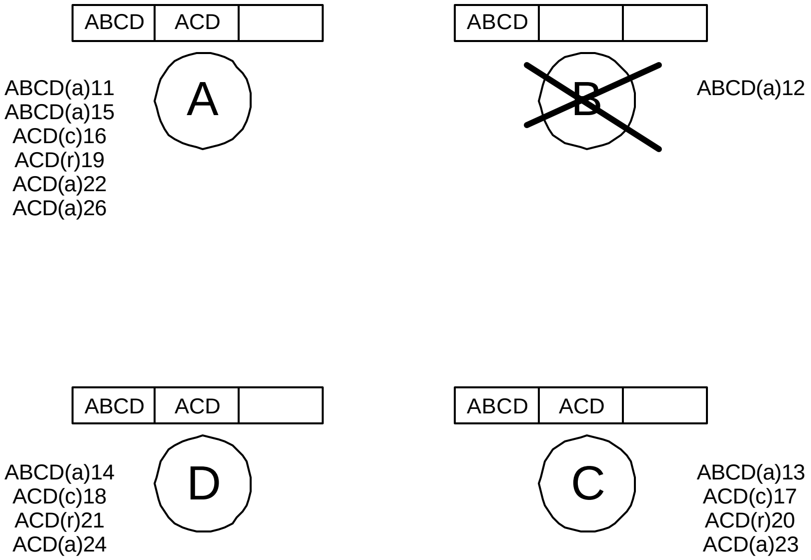


Figure 13 Example of token-ring protocol with a node failure

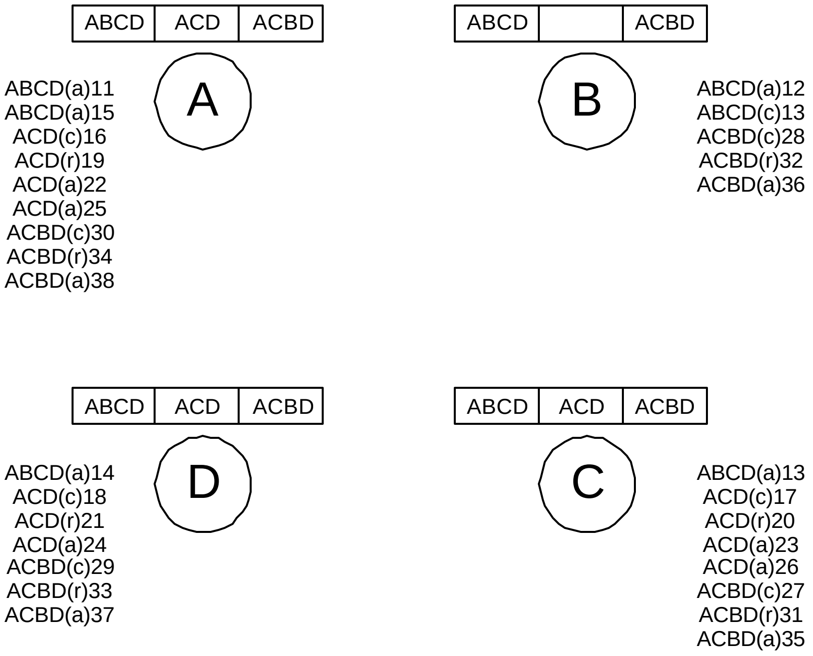


Figure 14 Example of token-ring protocol with a node recovery

3.6 Correctness Proof

We will now prove that the Raincore token-ring Group Membership Protocol meets the group membership specification from the work by Franceschetti and Bruck [Franceschetti and Bruck 1999]. Before we prove that the protocol meets the four properties: Agreement, Termination, Validity, and Safety, we'll first prove three lemmas.

Lemma 3.1: After a token arrives at node B from node A, A will not be followed by another node on the token nodelist until node A receives the token again.

Proof: A node is only able to remove or add a node that is adjacent to itself in the nodelist. The only node that can remove node B from the position after node A is node A. Node A can only remove nodes when it has the token. Therefore, if node A is not followed by node B, node A must have received the token. ÿ

Lemma 3.2: When a node transits from Chaos view state to Reserve view state, all the nodes on this node’s localView have the same localView.

Proof: Without loss of generality, suppose node A transits from Chaos state to Reserve state when it receives the token. We know that by definition of the protocol that the transition from the Chaos state to the Reserve state can only happen when localView on node A is the same as the nodelist on the token. Let’s call this view V1. We also know that by definition the localView is a copy of the nodelist on the token last time token left node A. So the nodelist on the token when the token last left node A is also V1.

We know from lemma 3.1 that when a node makes a change to the nodelist on the token, that change can not be undone until the next time the same node receives the token again. Therefore when the token last left any of the node on V1, the nodelist on the token must also be V1. (Otherwise, the current nodelist cannot be V1.) Therefore all nodes on V1 have the same localView V1. ÿ

Lemma 3.3: When a node transits from Reserve view state to Agreement view state, all the nodes on this node’s localView have the same localView, and the same location on the globalViewVector has been reserved by all node on the localView.

Proof: The proof is similar to the proof of Lemma 3.2. Without loss of generality suppose node A transits from Reserve state to Agreement state when it receives the token. We know by definition the transition from the Reserve state to Agreement state can only happen when localView on node A is the same as the nodelist on the token. Let’s call this view V1. We also know that by definition the localView is a copy of the nodelist on the token last time token left node A. So the nodelist on the token when the token last left node A is also V1.

We know from lemma 3.1 that when a node makes a change to the nodelist on the token, that change can not be undone until the next time the same node receives the token again. Therefore when the token last left any of the node on V1, the nodelist on the token must also be V1. (Otherwise, the current nodelist cannot be V1.) Therefore all nodes on V1 have the same localView V1.

Because the nodelist on the token records the list of nodes that the token last traveled, all nodes on the localView must all have incremented to the same globalViewNumber. Therefore the same location on the globalViewVectors has been reserved by all nodes on the localView.ÿ

Theorem 1 Agreement. At any point in time, all nodes in the group have consistent history in the globalView vector.

Proof: At any point in time, only modification to the globalView vector may disrupt the consistent history property. It only occurs when that node enters Agreement view state from the Reserve view state. Without loss of generality, suppose node A is the one that is making modification to the globalView vector.

By Lemma 3.3, all nodes on node A's localView must have the same localView when node A enters the Commit state, and the same location on the globalViewVector has been reserved. The nodes on the localView either have just committed this same localView to the same location on the globalView vector, or they have not done so, and the corresponding location on the globalView vector is empty. In either case, consistent history property is maintained. ÿ

Theorem 2 Termination. If there are no more changes in their local views, all nodes eventually reach Agreement view state.

Proof: When there are no more changes in the local views, it means that the Transport Managers on all the nodes are able to deliver the token to its destination, and there will be no changes to the nodelist on the token. Therefore there will be no differences between the nodelist on the token and the localView on each of the nodes. According to the definition of the protocol, all nodes eventually reach Agree view state. ÿ

Theorem 3.3 Validity. If all nodes in a view v* perceive view v* as their local view and are all in Agreement view state, they must have view v* as the last element of their globalView vector.

Proof: By the definition of the protocol, when a node's localView is different from the last element in the globalView vector, it means that at some point the nodelist on the token is different from the localView, and the node would have exited the Agreement state already. We also know from the definition of the protocol that when all of the nodes are in the Agreement view state, they have the same localView. Therefore when all of the nodes are in the Agreement state, they must all have their localView as the last element of their globalView vector.ÿ

Theorem 3.4 Safety. Once a view is committed in globalView vector, it cannot be changed.

Proof: The value of globalViewNumber is monotonically increasing on all nodes. Existing global Views in the globalView vector is not modified once it is commited. ÿ

3.7 Extensions

3.7.1 Fast-convergence Token-Ring Protocol

In implementing the Raincore Group Communication Manager, different variations of the token-ring group membership protocol described above were used. In particular, for the networking environment that Raincore is targeting, by weakening the consistent history requirement, we can achieve faster convergence time in the case of group membership changes. This results in faster fail-over time that is important in the networking environment.

For example, one variation of the token-ring protocol uses two View States, instead of three. There is no globalView vector, there is only one globalView set. The two View States are Agreement and Chaos. In this variation, consistent history is not relevant, as the globalView only has one value. If a node is in Agreement state, when it receives a token, if the nodelist on the token is different from the localView, the node enters Chaos state. If a node is in Chaos state, when the nodelist on the token is the same as the localView, the node commits localView to globalView and enters the Agreement state.

With the three-phase token-ring protocol introduced in the previous subsections, it generally takes three token round trips to commit a change in the membership. It guarantees consistent history of group memberships. With this two-phase protocol, it generally takes two token round trips to achieve agreement on a change of the membership. Whether to use the three-phase or the two-phase protocol is a choice depending on the need from the application. For some networking applications, fail-over action can even take place before agreement on group membership is achieved, by commiting to globalView every time the token is received.

3.7.2 Link Failures and Transient Failures

In the token protocol presented, one unique feature is the unification of the token regeneration request and the join request. This facilitates the treatment of the link failures. For example, in the group ABCD, the link between A and B fails, as illustrated in Figure 15. Node B is removed from the membership and the ring becomes ACD. Node B stays in Hungry token state for a while and then enters the Starving token state. It sends out a 911 message to node C according to the protocol. Node C notices that node B is not part of the current membership and therefore treats the 911 as a join request. The ring is then changed to ACBD. Not only does Node B rejoin the membership, but also the broken link between A and B is naturally bypassed in the new ring.

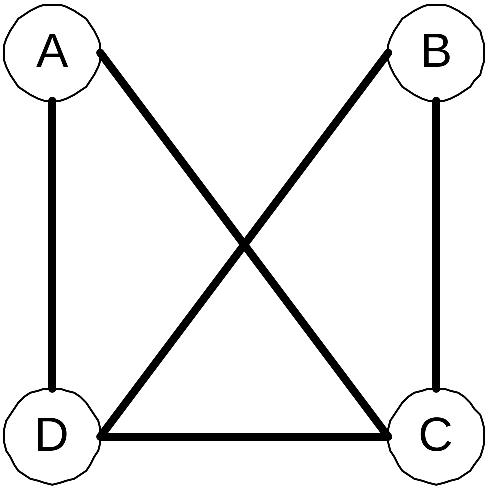


Figure 15 Link failure scenario

With the same mechanism, failure detector false alarms can be corrected. When a false alarm occurs, a node is removed from the membership by mistake. It sends out an 911 message as it enters the Starving state. The 911 message is recognized as a join request, and the node is being added back into the group. This shows that while the failure detector can never be

100% correct, after a wrong decision is made, the wrongfully excluded node will be able to automatically rejoin the group given the 911 protocol.

3.7.3 Group Partitioning

The token-ring protocol can also be extended to treat group partitioning. The partitioning of a group is a well-known problem in this field for which there exists no perfect solution. This problem arises when the group is broken into more than one subgroup. Each subgroup by itself is functional, but it cannot communicate with other subgroups. This problem is also referred to as the "split-brain" problem. There are two common strategies to this problem. The first strategy is a quorum decider. If N is the maximum size of the group, when the size of the group is N/2 or less, every node in the group shuts down itself. This is a safe strategy that prevents the split brain from happening. However, it sets severe limitations on the scalability and fault-tolerance capabilities. The second strategy is to allow each partitioned subgroup continuing to be functional, and design a protocol to discover when the communication between subgroups resumes and to merge the subgroups into one group.

In the Raincore design, while a pure quorum decider is not used, attention has been paid to first prevent the brain from being split. The Raincore Transport Service supports redundant communication links between nodes, which makes the isolation of subgroups less likely to occur. Another feature that Raincore offers is the ability to define critical resources for each of the member nodes. A node will shut down itself when any of its critical resources becomes unavailable. Sometimes it makes sense in a network environment to use this mechanism to set up a common critical resource, such as an Internet connection, for a group of nodes. When the group partitions, only one subgroup will continue to have the connectivity to that common resource, and all other subgroups will shut down themselves, thereby preventing the split brain.

This strategy is not always attractive, since it does introduce that common resource as a single point of failure.

When all preventions fail, Raincore will allow all subgroups to continue to function on their own. When the communication between subgroups resumes, it is desirable to merge the subgroups. In order for the merge to happen, the Raincore session layer protocol suite contains two more protocols, the Raincore discovery protocol and the Raincore merge Protocol. The discovery protocol is for the subgroups to discover the existence of each other. The merge protocol is designed to perform the merging between the subgroups.

To explain the Raincore discovery protocol, we first introduce the concept of eligible membership. The eligible membership contains the IDs of all eligible members of the group. It is configured on each member node, and the configuration can be changed and updated online. Each healthy member node sends a *hand-shake* message periodically to the other nodes that are in the eligible membership, but not the current group membership. Hand-shake message is a small message sent with a regular, but low frequency, so that it does not impose a major overhead onto the system. Contained in the hand-shake message is the node ID of the sender, and the group ID of sender's current group. It is common to use the lowest node ID in the current group membership as the group ID. When a node receives a hand-shake message, and if the sender is not in its own membership, but is in its eligible membership, the receiver has discovered a new eligible node. The goal of the discovery protocol is achieved.

Now the challenge is to merge the tokens. In order to avoid deadlocks in the merge process, the group IDs are used as tiebreakers. The hand-shake message is considered as a join request if and only if the sender's group ID is lower then the receiver's group ID. When it is regarded as a join message, the receiving node will wait for its token, check that the hand-shake sender is not on the membership on the token, add the hand-shake sender to the membership, mark the special TBM (To Be Merged) flag on the token, and send the TBM token to the hand-shake sender.

When the hand-shake sender receives a TBM token, it will wait for its original token, merge the memberships on the two tokens, and concatenate the multicast messages attached to the two tokens, and merge the two tokens into one. Thereby the merge between the two subgroups is completed. When there are more than two subgroups, by using the group ID ordering, the eventual merge among all of them can be completed without deadlocks.

3.8 Reliable Multicast

Reliable Multicast allows one node to communicate to a group of nodes in a reliable manner. It is the primary service that a group communication module should provide. In addition to the reliable point-to-point delivery that can be accomplished by an acknowledgement scheme, there are two additional properties that are desired from a Reliable Multicast service: atomicity and consistent ordering [Tanenbaum, 1995]. Atomicity refers to the all-or-nothing property, that when a message is sent to a group of nodes, it will correctly arrive at either all members of the group, or none of them. Consistent Ordering refers to the desire that all nodes in the group should receive all messages in the same order.

There are two types of group communication, the open group communication, where a node outside the group communicates to the group, and the closed group communication, where a node within the group communicates to all the nodes in the group. The Raincore Group Communication Manager directly provides closed group communication. A member of the group can use GCM to reliably multicast to all other members of the group. Raincore achieves both atomicity and consistent ordering for closed group communication. In addition, open group communication can also be achieved. A node outside the group can send a message to any member of the Raincore group, and that member then forwards the message to the entire group using GCM.

In the Raincore Group Communication Manager, the token-ring protocol plays multiple important roles. We know that it is a central mechanism for the group membership protocol. The transmission of the token message also serves as a smart failure detection mechanism for creating the local view without using omnidirectional multicast heartbeats. In this subsection, we'll show that the token also serves as a "locomotive" for the reliable multicast transport. In other words, the reliable multicast messages are packed and attached to the token, and circulated around the ring, until it comes back to the originating node. The token together with the messages are transported using the Raincore Transport Manager.

When the token returns to the originating nodes, all the nodes that token traversed have received all attached message from this originating nodes. This serves as an overall multicast acknowledgement, where the originating node can find out which nodes have received the messages by examining the nodelist field on the token. The originating node can then forward this multicast acknowledge message around the ring to inform the nodes that have received the original message. The receiving nodes, upon receipt of the original multicast messages, depending on atomicity requirements, have the option of immediately pass the message to the upper layer, or wait till receive a copy of the multicast acknowledgement to decide whether to pass the message to the upper layer.

Reliable multicast can be achieved with three different levels of consistent ordering [Amir et al., 1992]. Causal ordering guarantees all receiving nodes agree on the order of messages that are causally related to all nodes; agreed ordering guarantees that all receiving nodes agree on the ordering of all messages; safe ordering guarantees that each node delivers a message to upper layer only after all nodes in the membership have acknowledged the reception.

The token-ring protocol in the Raincore Group Communication Manager naturally achieves causal ordering and agreed ordering multicast. Because the token serves as a unique multicast transport, the ordering of the messages on the token determines the global message ordering of messages from all nodes in the cluster. Agreement of message ordering is achieved rather easily, and it requires no extra cost to achieve agreed ordering than no ordering.

To achieve safe ordering multicast, GCM needs to pass the message to the upper layer only when it receives the multicast acknowledgement for that message. This requires one more round of travel by the token. This guarantees the receipt of the message by all members before the upper-layer applications receive the message.

Raincore GCM is not the first one to use a token-ring protocol to enable reliable multicast. However, most of the other reliable multicast technologies use token ring for group membership and multicast sequencer. The actual messages are not attached to the token, but instead are physically multicasted to the group. Raincore GCM does not assume a broadcast medium and aggregate the messages with the token, and achieves reliable multicast purely using a unicast medium. This is particularly relevant in the networking environment, and we will go into more details in Chapter 5.

3.9 Mutual Exclusion

In addition to group membership and reliable multicast, the Raincore Group Communication Manager provides a mutual exclusion service as well. Because of the uniqueness of the token, it guarantees that at most one node can be in the Eating state at any time. When a node has the token, it can be assured that no other node has the token and the token can be used as critical region to guarantee mutual exclusivity. Because the token is circulating around in the ring, each node has a fair chance of getting the token. The 911 protocol makes this token-based mutual exclusion fault-tolerant.

This is a powerful service for the upper layers. In particular, a Raincore distributed lock manager is implemented as part of the Raincore Group Data Manager, using the mutual exclusion service to acquire and release data locks. The data locks provided by the distributed lock manager, comparing to this master-lock, can be associated with one or more shared data items, and can be owned by a node without requiring the node to hold on to the token.

3.10 Group Data Manager

The Raincore Group Data Manager is implemented on top of the Group Communication Manager. It is a presentation layer service to the applications. It manages *data items* for the applications, which the application can *read* from and *write* to. These data items are mirrored on all nodes in the cluster. The GDM synchronizes automatically any modification to the data items by any nodes. It uses GCM to multicast the changes to the other nodes. The underlying communication complexity is hidden from the GDM users. The detailed description of the GDM is outside the scope of this thesis. We'll give a brief overview of this module here.

The GDM implementation includes a global clock, implemented under the guideline of [Lamport, 1978]. It is not a physical clock, but it provides consistent ordering of events in a cluster. (Although the ordering may be incorrect, at least all nodes agree to the order.) The clock is used to resolve the conflict when two nodes try to write to the same data item. GDM may maintain a large amount of data items of various sizes. GDM provides interfaces to allow user either to treat the data item as an atomic unit or to only modify part of the data item.

Included in the Group Data Manager is a distributed lock manager. User can create locks to associate with one or more data items. These locks themselves are, in fact, data items too. GDM uses the mutual exclusion service provided by the Group Communication Manager to manage these locks. It waits for the arrival of the token to perform acquisition and release of the locks so that there is no conflict.

The Group Data Manager also allows user to use the concept of a *transaction* to atomically access multiple data items. A user may start a transaction and then write to a number of data items. The GDM will not synchronize these changes until the transaction concludes.

4 RAINCORE APPLICATIONS AND PERFORMANCE

4.1 SNOW

A number of applications were built using Raincore suite of protocols and services. The first application built using Raincore is a prototype called SNOW, which stands for Strong Network Of Web-servers. This is a proof-of-concept prototype built in Caltech that was part of the RAIN project. The Raincore protocols introduced in Chapter 3 were conceived when we were building this prototype. Figure 16 shows the six-node SNOW cluster in the Paradise (Parallel and Distributed Systems) Lab. They are constructed using six dual Pentium Pro 200MHz servers, redundantly interconnected by four Fast Ethernet Switches.

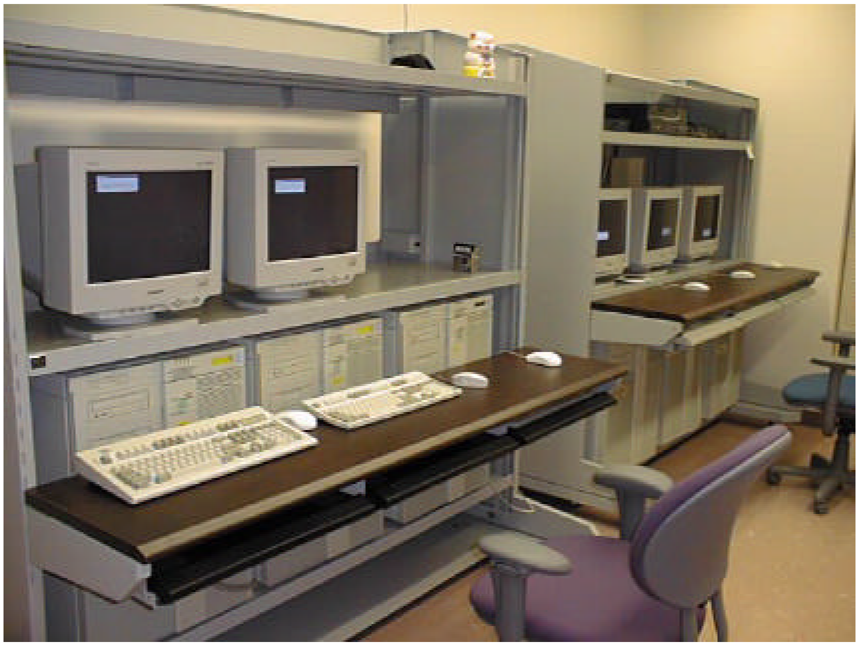


Figure 16 Six-node SNOW cluster

The design goal of SNOW is to develop a highly available fault-tolerant web server cluster that will distribute load among the nodes in the cluster. It needs to continue to function when parts of the system fail. The SNOW system uses the Raincore Transport Manager to handle all messages passing between the servers in the SNOW system. It takes advantage of the redundant interconnections, so that the partitioning of the cluster becomes less likely. The Raincore Group Communication Manager is used to manage the group membership of the cluster. Node crashes will be detected, and fail-over will take place.

A multicast model is assumed in the implementation of SNOW. The HTTP requests are received by all servers in the SNOW cluster. The servers use the mutual exclusion service that Raincore provides to decide which server node will own and reply to that request. Servers other than the designated one will simply drop the HTTP request. The assignment of who will own the connection is a function of the load distribution on all the server nodes. The server health and load information, as well as the HTTP request assignment table are being shared among the cluster by an early version of the Group Data Manager. In SNOW, we first used the token-ring protocols to detect web server failures, to provide mutual exclusion service, and to transport load information and HTTP request assignment table among the nodes in the cluster.

4.2 CLOUD

SNOW uses the multicast model to distribute the traffic among the nodes in the cluster. This method can effectively scale up the performance of the cluster, when CPU is the bottleneck. It however is not able to scale up the networking throughput. In other words, running in a Fast Ethernet environment, no matter how many servers are there in the cluster, SNOW can not process more than 100 Mbps of incoming traffic, since every packet arrives at every node.

Also, in the implementation of SNOW, the Raincore protocols were tightly coupled with the web server logic, and do not have a modular form that can be useful to other applications. These are the motivations of the making of CLOUD.

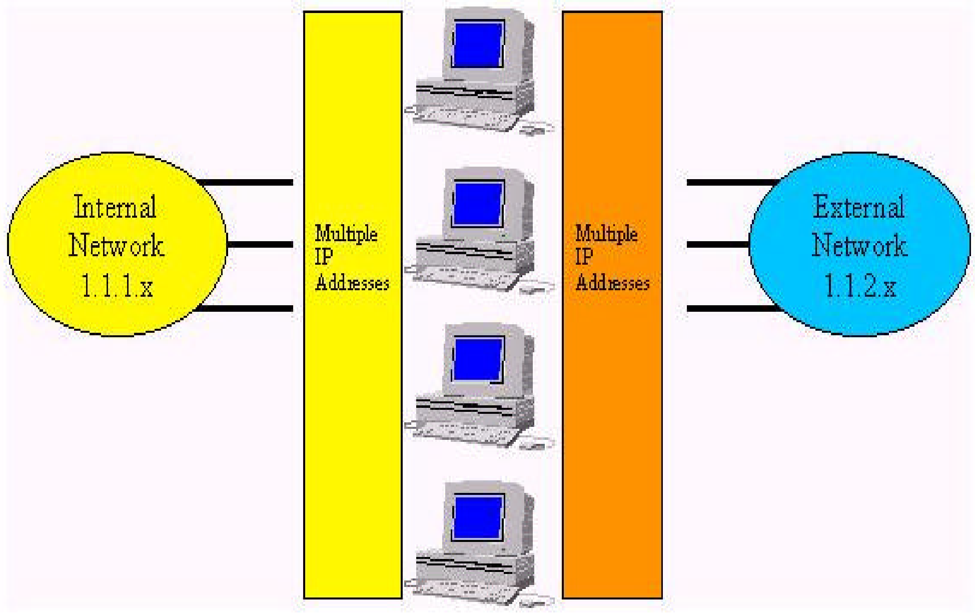


Figure 17 A cluster of CLOUD nodes with two pools of Virtual IPs.

CLOUD stands for Cluster Of URL Directors. What is significant in CLOUD is that we designed a way of distributing traffic to a group of networking elements by maintaining a pool of highly available virtual IPs among the group members (Figure 17). These Virtual IPs are the actual publicly advertised IP addresses for this network cluster. All traffic that goes through the cluster is being directed to one of the virtual IPs. The virtual IPs are mutually exclusively assigned to different nodes in the cluster by the Virtual IP manager module in CLOUD. In the presence of failures, Raincore Group Membership Protocol allows the healthy nodes discover and agree on the failure. This allows the Virtual IP manager to promptly move all the virtual IPs that were owned by the failed node to healthy nodes. The virtual IP manager assures users that while physical machines can go down, the virtual IPs never disappear as long as at least one physical node is functional.

In addition to the virtual IP movements during fail-over situations, they can also be moved for load balancing reasons. When a node is more heavily loaded than the other nodes and it has more than one virtual IPs, the load balancing algorithm in CLOUD may decide to move one of the virtual IPs to a less heavily loaded node. Moreover, CLOUD also allows user to manually move the virtual IPs from one node to another.

While the virtual IPs are being moved around, MAC addresses are never moved and remain unique to each node. When a virtual IP is being moved from one node to another, a gratuitous ARP message is sent to refresh the ARP cache so that the virtual IP corresponds to the new MAC address on all computers and routers on the same subnet. Therefore during Virtual IP movement, the traffic that designated for the Virtual IP will be redirected to the new node.

Using this approach, we can effectively scale up the network throughput by having more than one Virtual IP for each subnet. For example, a 4-node cluster using 4 virtual IPs, when connected by a fast Ethernet switch, can handle up to 400 Mbps of incoming traffic. The VIP Manager uses the Raincore Group Data Manager to share the assignment of the virtual IPs, as well as uses the mutual exclusion protocol to make sure that there is no conflict in the virtual IP address assignments. The VIP Manager module in CLOUD becomes an important building block in using distributed computing technology to provide load balancing and fail-over for networking devices. This technology was transferred to Rainfinity in 1998. A number of Rainfinity commercial applications have been built incorporating the VIP manager. One example is RainWall.

4.3 RainWall

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 commercial application using Raincore to deliver a high-availability and load-balancing clustering solution for firewalls.

RainWall is installed on a cluster of gateways, along with the firewall software, and performs load balancing among the gateways. The software installs on the same machines as the firewall itself and so doesn't introduce any new layers of components in the network. In the event of failure of one or more gateways, RainWall routes traffic through the remaining gateways without interrupting existing connections.

RainWall uses the same VIP manager that CLOUD uses to manage the pools of virtual IP addresses for the firewall cluster. The virtual IPs are the only advertised IP addresses of the firewall cluster. The VIPs are specified in the routers and local clients as default gateways. All traffic that goes through the firewall is being directed to one of the virtual IPs. By managing the virtual IPs intelligently and efficiently, RainWall guarantees the firewall availability in the presence of failures and achieves optimal performance even under heavy load.

In addition to the Virtual IP Manager that provides coarse load balancing and traffic fail-over among the firewalls, RainWall also includes a kernel-level software packet engine that load-balances traffic connection by connection to all firewall nodes in the cluster. This module also provides a way to synchronize connection state information without race conditions. The load and connection assignment information are shared among the cluster using the Raincore.

The local failure detectors are important parts of RainWall, 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 function, and the local machine's ability to reach remote hosts via ping. If any of these required resources go down, RainWall will, by default, bring down that firewall node, and its virtual IP addresses will be reassigned to other healthy firewall nodes with no interruption of service.

The local failure detectors, working together with Raincore, go a long way in handling faults in today's networking environment. Local fault monitor also serves to prevent partitioned clusters, usually a challenging problem for a distributed system to handle. The fail-over time of RainWall is on average 2-3 seconds. For example, suppose a client is downloading a file from a server through a firewall. If a network cable connecting one of the RainWall firewalls is accidentally unplugged, the client, instead of losing the connection, will only see a 2-to-3-seconds hiccup in the traffic flow, before it fully resumes. This is well within the TCP timeout, therefore all TCP-connections can fail-over transparently.

In fact RainWall not only works with firewalls, but also works in general with any routers or gateways. It creates a cluster of virtual routers that never fails, as long as at least one physical router is operational. It also allows a variety of applications to be present on the routers and is able to monitor the health of critical resources, such as the applications, the network interfaces, as well as the remote Internet links. When any of the critical resource fails, RainWall will shift traffic away from the failed node.

The core RainWall code is written in C/C++ and is compiled to native code for efficiency. It has been implemented for Solaris, Windows NT and Linux. RainWall runs as a user-space process and a kernel module. Raincore portion was implemented in the user-process level. Figure 18 shows the graphic user interface of RainWall. From this GUI, the user can monitor the health of each firewall node, the load going through each node, and the virtual IP address assignment in the cluster. The administrator can set VIPs to be sticky, so they stay on particular nodes and don't participate in load balancing. VIPs can be preconfigured with a preference for particular machines, and the GUI allows drag-and-drop of VIPs between machines. These state information are all shared using Raincore protocols. RainWall has been well received by the customers and is operational at more than 200 major customer sites worldwide to provide high availability and load balancing to their firewalls.

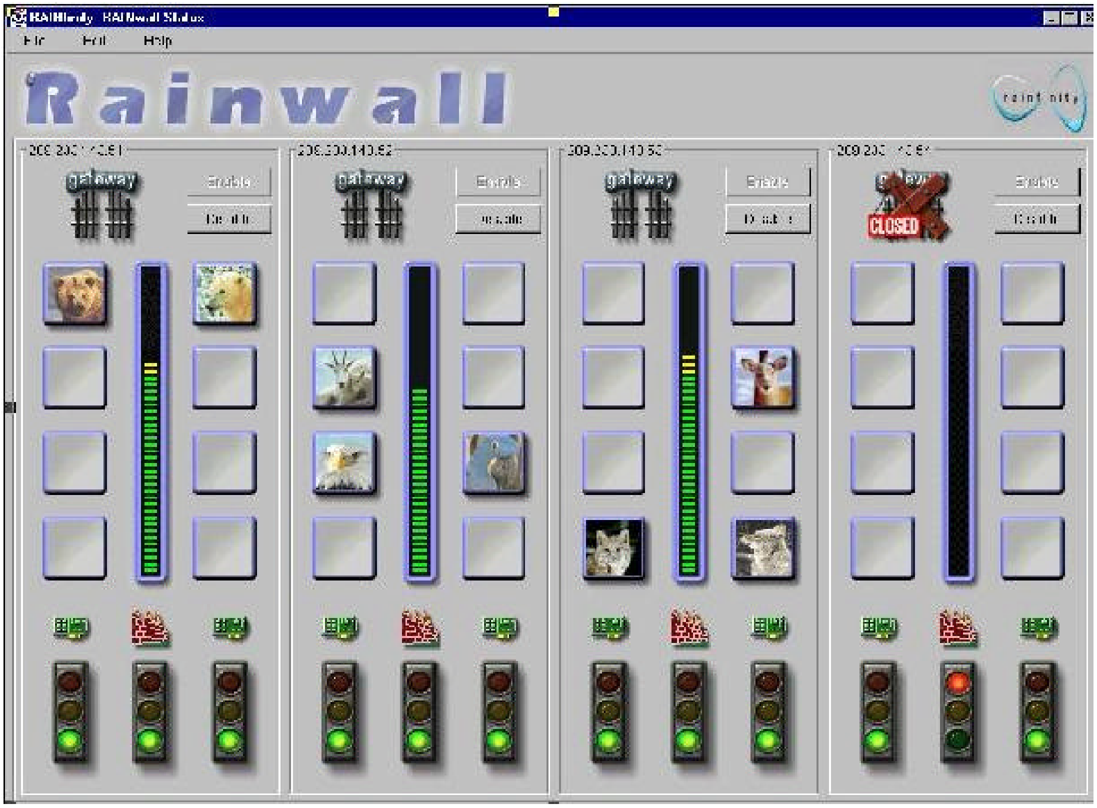


Figure 18 RainWall Graphical User Interface

4.4 RainFront Platform

RainFront is the next generation software platform using the Raincore protocols. Multiple modules can be plugged into the RainFront platform at the same time, all take advantage of the functionalities that Raincore provides. This allows the customer to use this one scalable layer to perform multiple critical functions at the edge of the Internet, therefore reduces the number of physical links in the end-to-end chain of communication. RainFront is still under development. The development plan includes the kernal and hardware level implementation of the Raincore protocols.

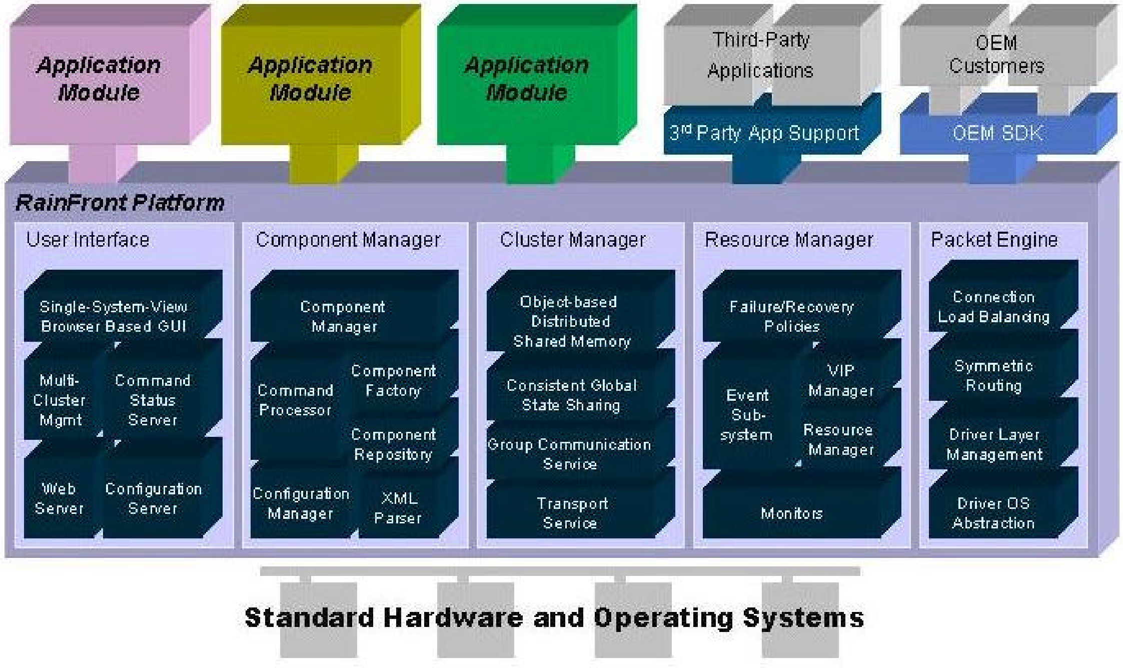


Figure 19 RainFront platform architecture

RainFront consists of the RainFront platform and RainFront modules as illustrated in Figure 19. The RainFront platform contains five base components:

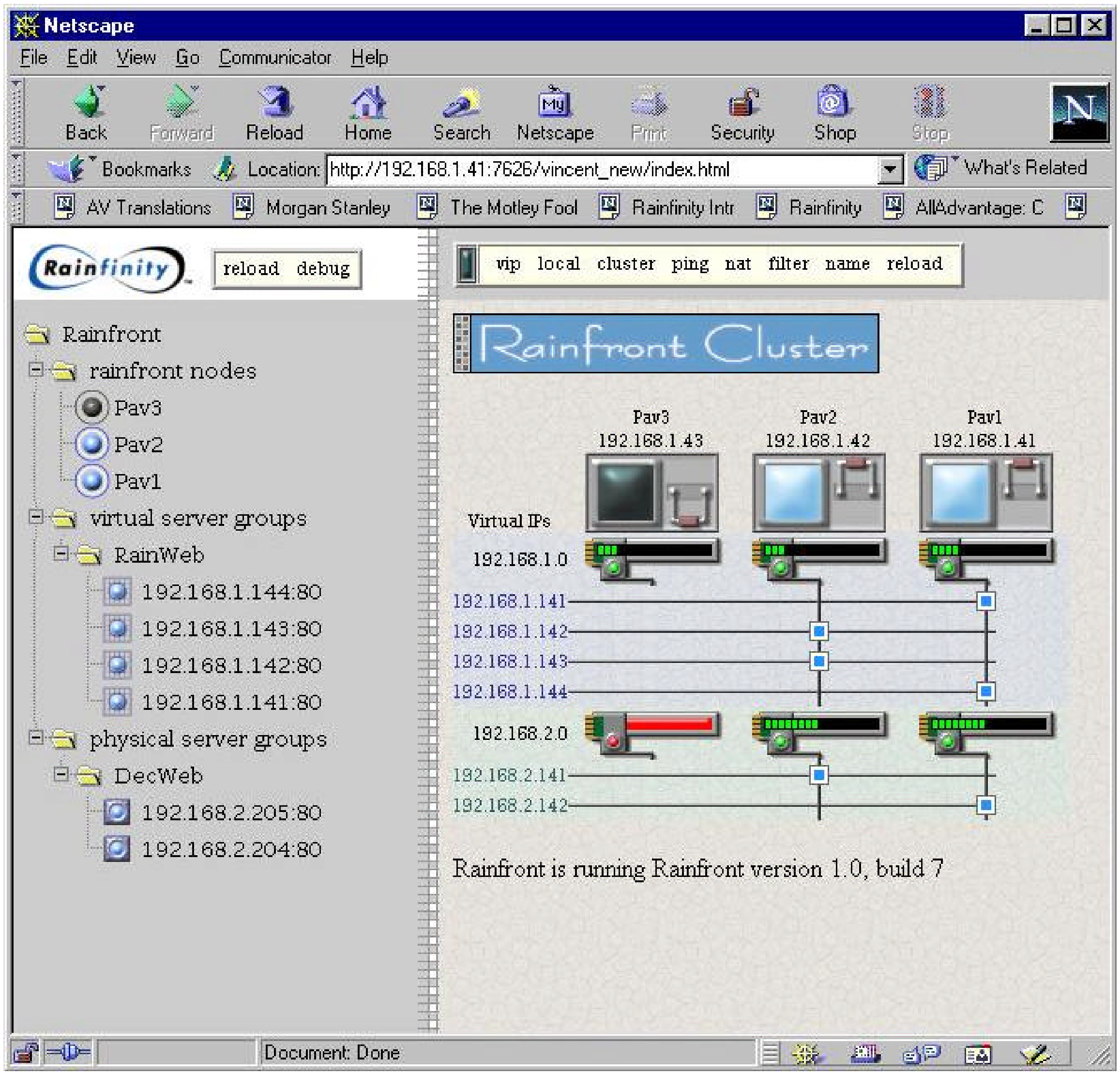
Cluster Management: RainFront platform uses the Raincore suite of protocols and services to provide distributed services to the application modules. This includes cluster membership change notification, reliable multicast, shared-variable distributed shared memory, relative global clock, and mutual exclusion services.

Resource Management: RainFront platform provides resource management framework that can help RainFront modules to share a set of "resources" among the nodes in the cluster. It can guarantee the mutual exclusiveness of the resource ownerships. It also employs a monitoring framework that detects failures and is able to make the resources always available when at least one node survives. User can also specify different policies on what to do with the resources when node leaves the cluster, new node joins the cluster, or when the load is unbalanced.

Packet Engine: Packet Engine is implemented at low-level (below the TCP/IP stack) and is capable to perform connection-level and packet-level load balancing. It can also provide symmetric routing to gateways or routers, that guarantees that the traffic in different directions of the same connection would be going through the same node in the cluster. This is in essence immediate state sharing without encountering race conditions.

Component Management: RainFront allows multiple application components to reside on top of it. Module Management provides the Application Programming Interface that these modules can co-exist peacefully with each other. It also provides configuration input/output and synchronization, as well as the status information exchange with the user interface.

User Interface: RainFront provides both a browser based graphical user interface, as well as a command line user interface. User can access, monitor, configure, and control both the platform features and the application modules' features, from a single interface. User can also access multiple clusters from the same GUI. Figure 20 shows a typical RainFront GUI.



Rainfront is running Rainfront version 1.0, build 7

Document: Done

Figure 20 RainFront Graphical User Interface

4.5 Overhead Analysis

Raincore Group Communication Manager provides reliable multicast by packing the multicast messages and attaching them to the token. The token message is essentially the locomotive of the multicast transport. Questions arise whether this design has satisfactory performance to validate it as a practical design. In particular, in the other distributed systems projects, such as ISIS [Birman and van Renesse 1994, van Renesse et al. 1994] and TOTEM [Amir et al. 1995], a broadcast medium is assumed. Important concepts and novel algorithms have been invented and implemented for a broadcast environment to provide reliable multicast with multiple levels of consistency. We would like to answer how the performance of Raincore compares to the broadcast-based protocols, in terms of CPU overhead, network overhead, and message delivery latency.

To address this fully, we need to evaluate the environment very closely. Raincore is designed for the networking elements on the Internet. They typically operate in a high-throughput, high-speed networking environment. It is realistic to assume that the network latency is very low. This fact alleviates the latency concerns over the token-based protocols. In this networking environment, huge amount of network traffic is passing through the cluster at the same time Raincore is used for intra-cluster state sharing. As opposed to the traditional distributed scientific computing where communication only plays a supporting role to computation, processing of the regular traffic is the main task of the cluster.

The group communication is happening at the same time as the regular network traffic being processed by the members of the distributed system. The CPUs in the networking elements are frequently optimized for high-throughput single-task processing of network traffic, where task-switching can be costly. Our design goal is then to have robust and consistent group communication among the member nodes, with minimal extra overhead on the CPU and the network. The main purpose of the group communication is to share among the networking devices the cluster state and the state information associated with the regular network traffic, so that load balancing and fail-over can occur smoothly. This will enable a cluster of networking elements to have the maximum throughput and the highest reliability. There are two implications from this unique property:

1. **Unicast-based Communication:** Unlike existing studies in distributed systems, broadcast-based medium cannot be assumed. It is true that many of the currently popular local area networks are broadcast medium, example being an Ethernet environment using a hub. However, to be able to multicast messages to all members of the group, either each member sets its network interface card to the promiscuous mode, or all the cards on all of the nodes must use a single multicast MAC address. This poses a limitation on the overall network traffic throughput of the networking cluster. Setting up a broadcast medium among the members of the cluster would mean that the total throughput of the entire cluster on that network is limited by the throughput of one network interface card, no matter how many nodes are in the cluster. In contrast, assuming a unicast model would allow the throughput of N network interface cards on a network, for a cluster of N nodes interconnected by a switch. In this study, we make design decisions on a unicast-based network medium.

2. **CPU Task-switching Performance Metric:** In a networking environment, the primary job of the networking elements is to process the network traffic traveling through the device. Group communication plays an assisting role to organize the devices and share state for the devices in the cluster. The design goal of the group communication service is that it poses minimal additional CPU overhead and network overhead onto the cluster. Both the network overhead and the CPU overhead can be

> measured by the number of packets sent and received by each node in the cluster. A more subtle measurement of the CPU overhead is the number of task-switching actions that the CPU makes between the processing of regular network traffic, and the processing of the group communication. This measurement is important since the network elements are usually optimized to handle extremely high throughput, and switching between the traffic processing and group communication has significant latency cost.

Raincore Group Communication Manager is designed with these two properties in mind. It chooses to use a token-ring based protocol that exclusively uses unicast messages. In addition to using the token message to manage the group membership, synchronization, and message ordering, it actually attaches the group communication messages to the token to increase efficiency of the system. It can be shown that with this way, the message complexity is optimized. In addition, the task-switches between the processing of regular traffic and group communication messages are also minimized.

For example, suppose a cluster contains N networking devices, and each device needs to send M messages to the group every seconds. And suppose that the token travel around the cluster at a frequency of L round trips per second, $L<M$. Using Raincore, only L task-switching actions are needed every second to achieve reliable atomic consistent multicast. Using a broadcast-based protocol, at least $M \times N$ task-switching actions are needed. If a two-phase commit protocol is used to guarantee consistent ordering, up to $6 \times M \times N$ task-switching actions are needed at every node. This will impact both the CPU overhead and the latency on the group communication and the processing of the network traffic.

Now let us look at the network overhead. While it is true network mediums, such as Ethernet, is in its nature a broadcast medium, configuring it to be one has practical penalties. Either all nodes need to share the same MAC address, or they need to be configured to work in promiscuous mode. Both methods impose undesirable performance bottleneck for the processing of regular network traffic. For example, in a Fast Ethernet environment, to configure the cluster to share a broadcast medium means that no more than 100 Mbps can travel through the cluster of N nodes in any direction. In contrast, in a switched unicast Fast Ethernet environment, the aggregate throughput of the cluster can reach N × 100 Mbps.

In a unicast environment, broadcast messages are achieved by sending multiple unicast messages, which could introduce more overhead than a token-ring based protocol. For example, in a cluster of N nodes, when each node needs to multicast one message of M bytes, there will be $(N-1)^2$ packets of M bytes on the network when a broadcast-based protocol is used. Number of packets will be doubled if acknowledgements are implemented for reliable delivery. The packet count will be further increased when consistent ordering is required. In contrast, using the token-based protocol, there are N packets of N × M bytes on the network, and the delivery is reliable and the order of delivery is consistent. This is especially efficient when M is small.

4.6 Performance Benchmark

The most important performance measurement is to measure the scaling in network throughput of an application that uses Raincore as the cluster size increases. We have performed this measurement on RainWall.

RainWall operates as a firewall cluster. We measure the throughput of the cluster as more nodes are added to the cluster. This is both to understand the scaling capability of the RainWall application and to validate the approach of the Raincore services. The Benchmark presented in Figure 21 is obtained from the Rainfinity Lab. The results were obtained by running RainWall on a cluster of one, two, and four gateways. The throughput number on the chart indicates the total traffic that travels through the RainWall cluster. We can see that the throughput scaling from one node to two nodes is 1.97, and that the scaling from one node to four nodes is 3.76.

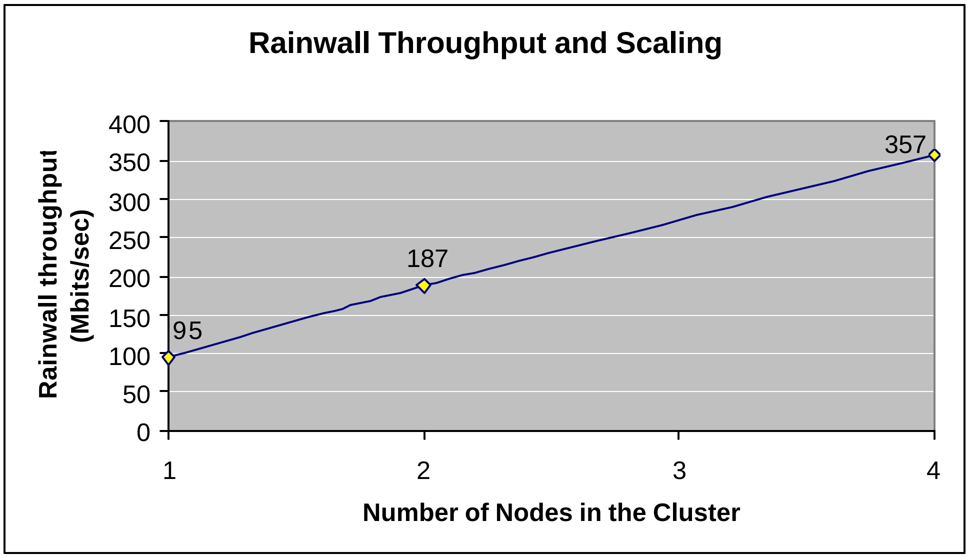


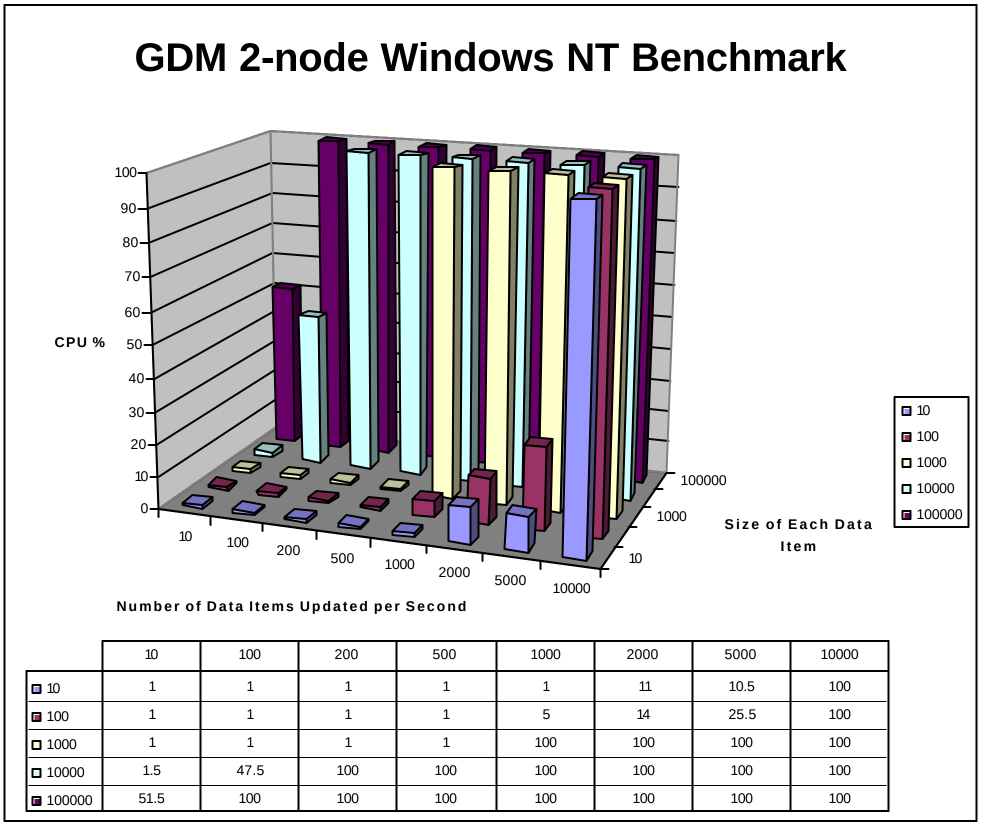
Figure 21 RainWall performance benchmark

The benchmark test was performed using Sun Ultra-5 single-CPU 360MHz workstations as the RainWall gateways in a Fast Ethernet switched environment. Each workstation has 256 MB of RAM and runs Solaris 2.6. HTTP clients are placed at one side to request data from Apache web servers on the other side of the RainWall cluster. The low overhead of the Raincore Distributed Services ensures that most of the CPU cycles on the gateways being used by the firewall traffic processing. This contributes to this impressive near-linear scaling of the performance.

To understand exactly how much CPU usage does Raincore use, we created a benchmarking program that uses Raincore, and varies the size of the state being synchronized, and measures the CPU usage. In particular, this benchmark program uses the API of the Raincore Group Data Manager. It maintains 10000 data items among the nodes in the cluster. We vary the size of the data items, as well as how many data items we update each second, and then measure the CPU overhead on each of the nodes in the cluster and calculate the average. We present here the result we have obtained.

We performed experiments on a two-node Windows NT cluster, each node is a dual-CPU Pentium III 933 MHz computer, with 128 MB of RAM. We set the number of data items updated per second per node at 10, 100, 200, 500, 1000, 2000, 5000, 10000, and set the size of the data item at 10 Bytes, 100 Bytes, 1 Kbytes, 10 Kbytes, 100 Kbytes. The average CPU usage is presented in Figure 22.

Typically in a networking application, the state being shared is the state associated with each networking sessions, and the size of data for each session ranges from 100 bytes to 1 Kbytes. Please notice that at 1 Kbytes per data item, 500 data items updated per second per node, the CPU overhead is still below 1%. This also shows that for a networking application using Raincore, if 1 Kbytes of data is used to record the information about each session, a two-node Raincore cluster can accept 1000 new sessions per second, and still encounters insignificant CPU overhead for the state synchronization between nodes in the cluster. This translates to 3.6 million new sessions per hour, or 86.4 million new sessions per day.



	10	100	200	500	1000	2000	5000	10000
■ 10	1	1	1	1	1	11	10.5	100
■ 100	1	1	1	1	5	14	25.5	100
□ 1000	1	1	1	1	100	100	100	100
□ 10000	1.5	47.5	100	100	100	100	100	100
■ 100000	51.5	100	100	100	100	100	100	100

Figure 22 Two-node Raincore Windows NT benchmark

More performance benchmarking is currently underway to further understand the performance of Raincore, both on larger size of clusters, and for different applications.

5 FAULT-TOLERANT MULTISTAGE INTERCONNECTION NETWORKS

In Chapter 3 and Chapter 4 we introduced Raincore, a collection of distributed protocols designed for the transport layer, session layer and presentation layer in the network stack. The goal of Raincore is to create a more powerful and reliable networking element by clustering a number of smaller units together. Raincore can be applied to routers, application gateways, and other high-level networking elements. It, however, does not apply to Layer-2 networking elements, such as switches. In this chapter, we focus our attention on Multistage Interconnection Networks (MIN). It is a class of switch constructions that use a number of smaller switches to make a bigger one, so that the bigger one is more capable and fault-tolerant. Our main result from this study is the discovery of a class of constructions that tolerates multiple switch faults with a minimal number of redundant switching stages.

The common building block for a MIN is a 2 X 2 switch. Two connectivity models are considered in our study, namely, the point-to-point connectivity between any two nodes and the broadcast connectivity. For point-to-point connections, the 2 X 2 switch operates in either the “straight mode” or the “exchange mode”, as illustrated by Figure 23. Two additional broadcast modes of operation exist to enable one node to send a message simultaneously to all other nodes, also illustrated in Figure 23. In this paper, we will first prove the results for the point-to-point connections. The results will then be extended to the broadcast case.

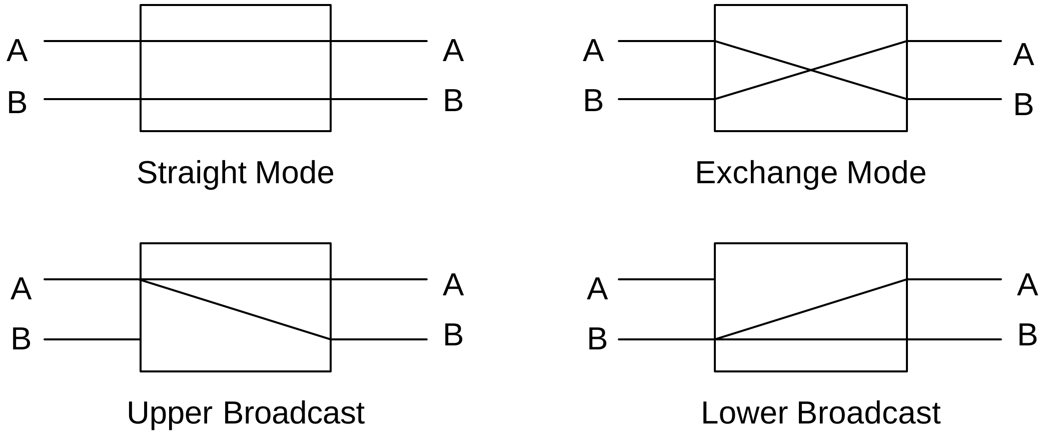


Figure 23 Operation models of a 2 X 2 switching element

Many types of cube-type Multistage Interconnection Networks have been proposed, such as the baseline, delta, generalized cube, indirect binary n-cube, omega, and shuffle-exchange [Leighton 1992]. It has been proved that these cube-type MINs are topologically equivalent [Agrawal 1983]. They share the basic concept that by using multiple 2 X 2 switches, a much bigger switch can be constructed to allow both point-to-point connections between any pair of input and output nodes, and broadcast from any input node. The attractiveness in using MIN over using a crossbar switch is that it saves complexity and provides fault-tolerance.

In this study, we will focus our attention to one particular construction, shown in Figure 24. This MIN allows point-to-point connection from an input node to any output nodes. In the example shown in the figure, there are three stages of 2 X 2 switches that interconnect the eight nodes. Each node is labeled with a binary vector. The length of this vector, n, is the ***dimension*** of the MIN. Clearly, n=$\log_2 N$, where N is the number of nodes in the MIN.

Each switch is characterized with an n-bit vector, called a ***mask***. The mask indicates the difference between the two input nodes, B-A. This difference is obtained by modular-2 vector subtraction. Notice that the switches in the same stage have the same masks, and therefore we can associate the entire stage with a single stage mask.

In this example, the MIN has the singleton mask set: $\{m_1=001, m_2=010, m_3=100\}$. This mask set forms a basis of the three-dimensional space. All vectors in this space can be represented as a linear combination of the masks. In other words, this mask set spans the three-dimensional vector space. Consequently, we can find a path between any pair of nodes by using the following routing algorithm: to route a connection between node A and node B, we decompose the difference between A and B into a linear combination of the masks.

$$B - A = \Sigma^{n}{}_{i=1}(c_i \times m_i)$$

The switches in stage i operate in the ***straight*** mode if $c_i=0$ and in the ***exchange*** mode if $c_i \neq 0$. We can see that there is only one path between a pair of nodes. When there is a fault on that path, the communication between the pair of nodes will fail.

Multistage Interconnection Network (MIN) has enjoyed important applications in fields such as telecommunications and parallel computing in the past decades [Benes 1964; Pease 1977; Adams et al. 1987; Linial and Tarsi 1989]. The fault-tolerant capabilities of MIN have been widely studied. In 1982, Adams and Siegel introduced the Extra Stage Cube [Adams and Siegel 1982], a construction that tolerates one switch failure with one additional switching stage. In this chapter, we study how to construct Multistage Interconnection Networks to tolerate multiple switch faults with extra stages.

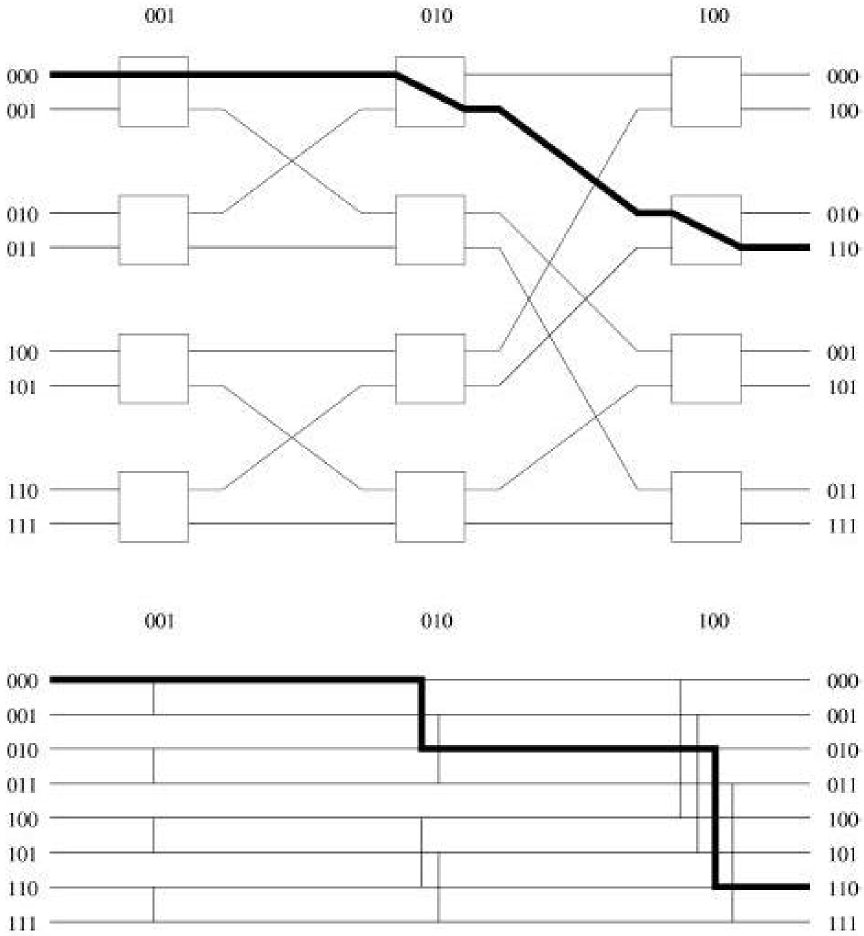


Figure 24 A three-dimensional MIN using 2X2 switches

We assume the same fault model as the Adams and Siegel paper [Adams and Siegel, 1982]. When a switch is at fault, it is stuck in the straight mode. This fault model can be implemented by using two demultiplexor/multiplexor pairs for each 2 X 2 switch.   The multiplexors and the demultiplexors are assumed to be fault-free. In the solution proposed by Adams and Siegel, not all switches need to be accompanied by these demultiplexor/multiplexor pairs. Only the switches in the first and last stages need them. Similarly, for the solutions proposed in this paper, they may not be needed for all switches. Only the switches in the first f and last f stages need them to tolerate f switch faults.

Shown in the bottom half of Figure 24 is the bar diagram [Knuth 1973] representation of the same MIN. In the bar diagram, each node in the MIN is represented by a horizontal bar and each switch is represented by a vertical bar. A broken vertical bar in the diagram indicates a faulty switch in the MIN. Connectivity exists between two nodes if and only if a path can be found between these two nodes. Such a path must use at most one switch at each stage and must not change direction inside the MIN, as shown in the figure. Tolerating f switch faults in the MIN is equivalent to tolerating f broken vertical bars in the bar diagram.

To tolerate broken vertical bars in the bar diagram, we need to find disjoint paths between any pair of nodes. Two paths are disjoint in a bar diagram if they share no vertical bars. To tolerate f broken vertical bars, it is sufficient and necessary to find f+1 mutually disjoint paths between all pairs of nodes. It is sufficient because f broken vertical bars can at most break f disjoint paths, and there is at least one paths left between each pair of nodes. It is necessary because if only f disjoint paths can be found between some pair, f broken vertical bars can break all of them and destroy the connectivity between that pair.

In the MIN shown in Figure 24, one and only one path can be found between any pair of nodes. Therefore, it cannot tolerate any switch faults. To make this MIN single-fault-tolerant, redundant stages need to be added. This problem of tolerating a single switch fault with extra stages has been investigated extensively in the past. Adams and Siegel first proposed a solution, called Extra Stage Cube (ESC) [Adams and Siegel, 1982]. Xu and Shen proposed a solution, which can be viewed as adding an extra stage with an all-1 mask [Xu and Shen, 1992]. An example of single-fault-tolerant MIN with eight nodes is shown in Figure 25.

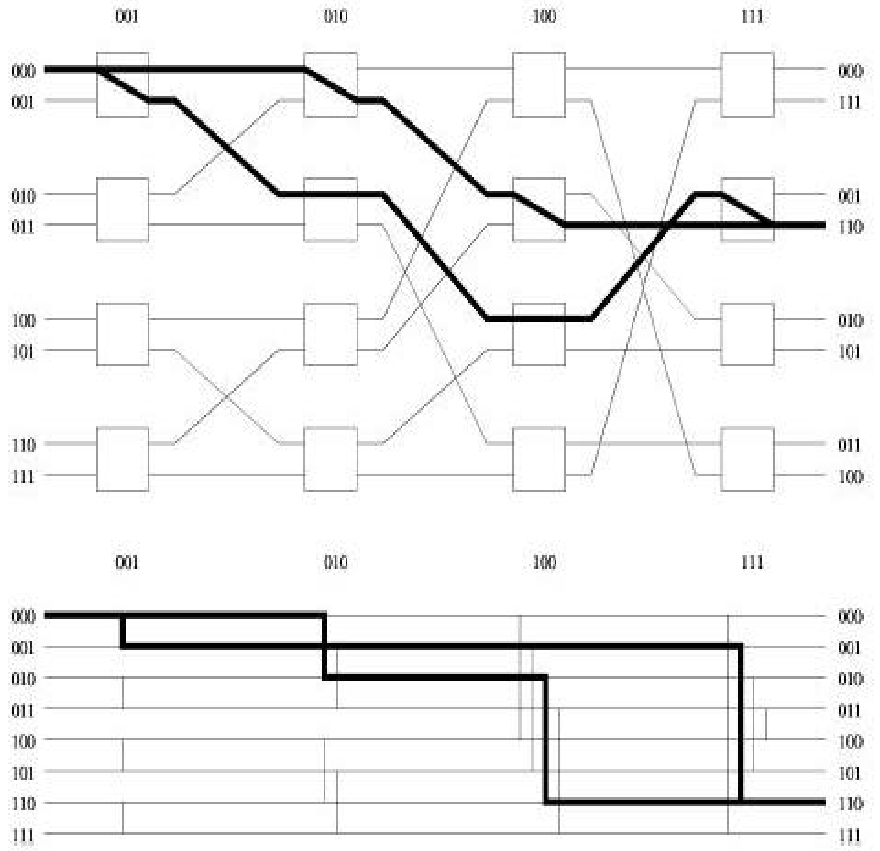


Figure 25 Three-dimensional one-extra-stage Extra Stage Cube MIN

Adams and Siegel showed that the ESC tolerates one switch fault by finding two disjoint paths between all pairs of nodes. As an example, two disjoint paths between 000 and 110 are outlined in the figure. There is another simple way to see that ESC is one-fault-tolerant. After a switch fault occurs, even if we discard the entire stage that contains the faulty switch, the masks of the remaining three stages still span the space. Therefore the difference between any pair of nodes can still be decomposed into a linear combination of the remaining three masks and a correct routing is therefore available by using the three surviving stages. In essence, this scheme tolerates more than a single switch fault. It tolerates any number of switch faults if all of them occur in the same stage. The ESC solution does not, however, tolerate two switch faults when they occur in different stages.

ESC is not a unique solution to the single-fault-tolerant problem. There exist other solutions that tolerate a single switch fault. We present one of these solutions in Figure 26. This is also an 1-extra-stage construction and the extra stage is masked 001. This MIN does not tolerate a stage fault, since erasing stages 010 or 100, the masks of the three remaining stages do not span the space. But this MIN can indeed tolerate a single switch fault. The two disjoint paths between 000 and 110 are outlined in the figure as an example.

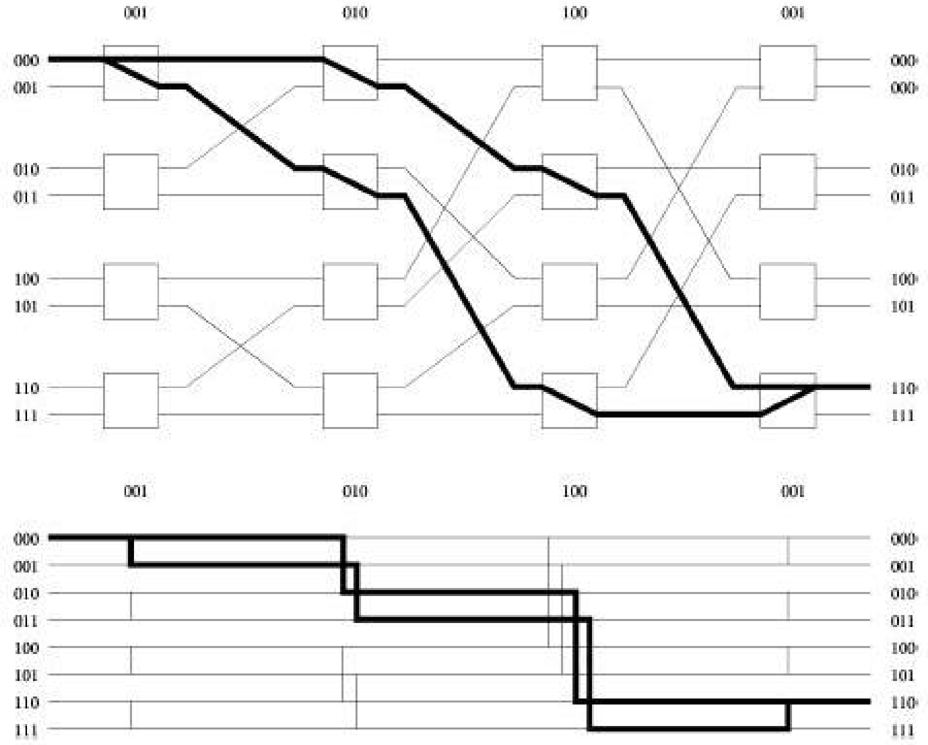


Figure 26 Three-dimensional one-extra-stage Cyclic MIN

The problem of tolerating multiple switch faults has been investigated in previous research works [Shih and Batcher 1994; Bruck and Ho 1996]. Bruck and Ho correlated the problem of fault-tolerant MIN to the results in the field of Error-Correcting Codes, and proved that a MIN constructed according to a (n, k, d) code can tolerate d-1 switch faults, as well as stage faults [Bruck]. It showed that a fault-tolerant MIN constructed according to a MDS code uses optimum number of extra stages to tolerate f stage faults. Despite extensive research in the field [Kumar and Reddy 1987; Varma and Raghavendra 1989; Cam and Fortes, 1990; Chuang 1996; Liao et al. 1996], constructions that use a minimal number of extra stages to tolerate f switch faults, f > 1, had not been discovered until this study.

The two examples we showed led us to consider the following questions: Are the existing solutions the best we can do in tolerating multiple switch faults? If not, what is the optimal extra-stage solution? Furthermore, if we are able to find optimal constructions, are those constructions the only solutions? We answered all of these questions in this study.

In Section 6.1, we will propose a construction of fault-tolerant Multistage Interconnection Networks that uses an optimal number of extra stages to tolerate f switch faults. In that section, we first prove that to tolerate f switch faults, at least f extra stages must be added. None of the previously proposed constructions meets this lower bound when f is greater than 1. We then propose a new construction that meets this lower bound. The routing algorithm is also given in that section. We will also show that this construction is easy to implement in practice. In Section 6.2, we generalize the construction proposed in Section 6.1 and prove a necessary and sufficient condition for MINs to tolerate any given number of switch faults with an optimal number of extra stages. While we focus on the MINs that use 2 X 2 switches under the point-to-point connection model in Section 6.1 and Section 6.2, we extend the results to the Multistage Interconnection Networks that use $t \times t$ switches and MINs under the broadcast model in Section6.3.

5.1 An Optimal Construction

We first present the following theorem which states the lower bound on the number of extra stages required to tolerate f switch faults for MINs with t × t switches.

Theorem 6.1: *To tolerate f switch faults in an n-dimensional Multistage Interconnection Network with t ´ t switches, at least f extra stages must be added.*

Proof: (by contradiction) Suppose only f-1 extra stages were added. When f failures occur at the switches that are connected to node 0 in the first f stages, the first f stages are completely paralyzed in connecting node 0 to other nodes. Only n-1 stages can be used to connect node 0 to the other t^{n-1} nodes. It is clearly not possible, since with n-1 stages, at most t^{n-1} nodes can be reached. ð

None of the previously proposed MINs meets this lower bound for any given f. For example, the ESC solution only works for f=1 [Adam and Siegel, 1982]; The number of switch faults that the Bruck and Ho solution tolerates is in general less than the redundant stages required [Bruck and Ho, 1996].

Now we present a new construction of MINs with 2 X 2 switches that meets this lower bound.

Definition 6.1: *Cyclic Multistage Interconnection Networks*

An (n, f) Cyclic Multistage Interconnection Network is an n-dimensional f-extra-stage MIN which has the singleton basis of the n-dimensional binary vector space as the masks of its first n stages and $m_{n+i} = m_i$ for 1£ i £ f, where m_i is the mask vector of stage i.

A (3, 4) cyclic Multistage Interconnection Network is illustrated in Figure 27. The following theorem implies that this MIN tolerates four faults, therefore meets the lower bound stated in Theorem 6.1. The five mutually disjoint paths can be found between any pair of nodes. In the figure, the paths between node 000 and node 110 are illustrated.

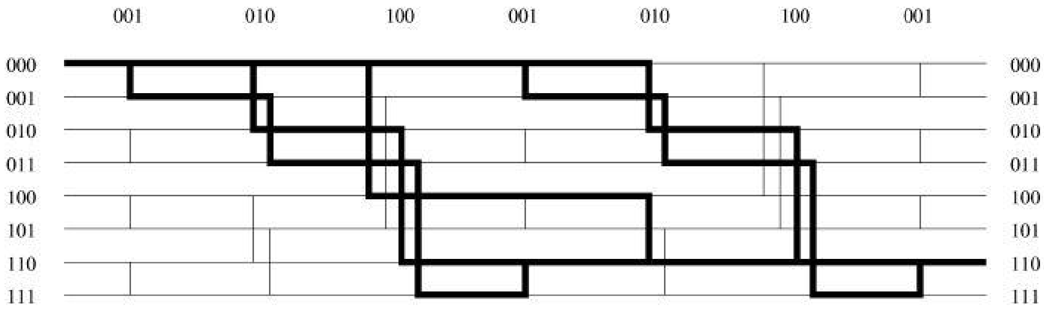


Figure 27 Three-dimensional four-extra-stage Cyclic MIN

Theorem 6.2: *A (n, f) Cyclic Multistage Interconnection Network with 2 X 2 switches tolerates f switch faults.*

Proof: We will prove the theorem by explicitly showing that between any two nodes, A and B, A ≠ B, there are f+1 mutually disjoint paths in the Bar Diagram. Two paths are disjoint if they share no vertical bars in the Bar diagram. In other words, there is a conflict if and only if one switch operates in the *exchange* mode in two different paths. Please note that in this proof, the nodes A and B and the masks {m_1, m_2, ..., m_{n+f}} are n-bit binary vectors and all arithmetic operations between them are bit-wise modular-2 additions.

We construct the f+1 paths as follows:

In path i, 1 ≤ i ≤ f, the switch in stage i operates in the *exchange* mode. Stages i+1 through i+n are used to route the path to destination B. This is possible, because every n consecutive masks in the Cyclic MIN span the n-dimensional binary vector space by definition. The switches in all other stages operate in the *straight* mode.

In the last path, path f+1, the switches in stages 1 through f operate in the *straight* mode and the switches in stages f+1 through n+f route the path from node A to node B. Figure 28 illustrates this construction.

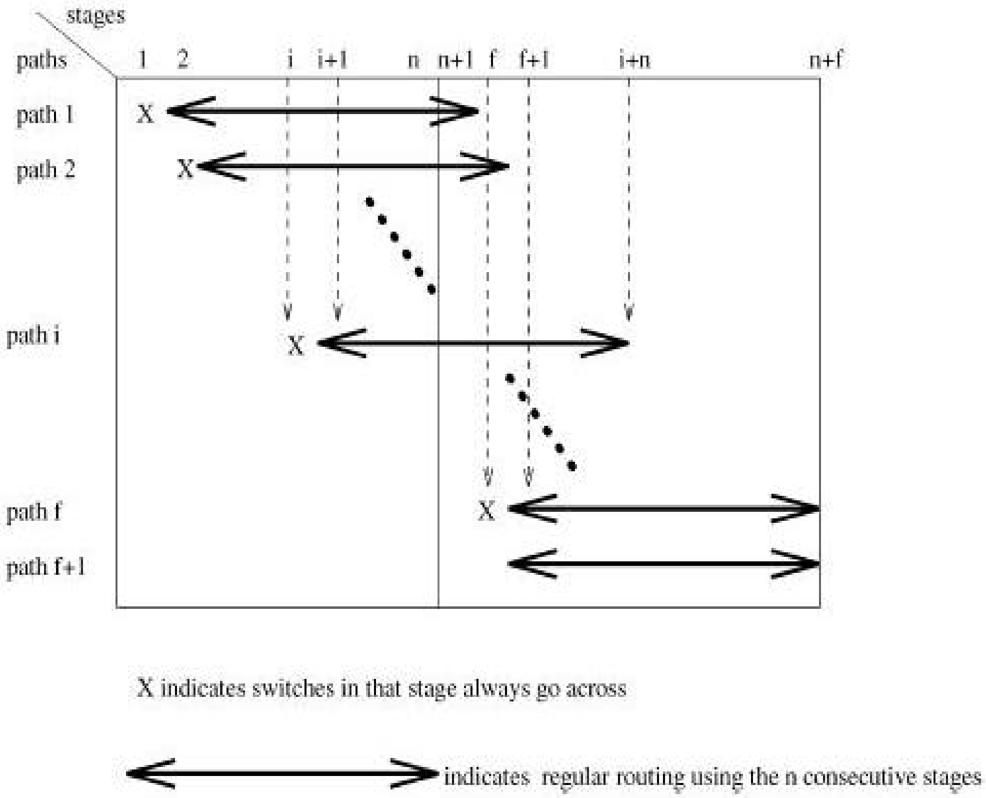

stages
paths
1
2
i
i+1
n
n+1
f
f+1
i+n
n+f
path 1
path 2
path i
path f
path f+1
X
X indicates switches in that stage always go across
indicates regular routing using the n consecutive stages

Figure 28 Construction of f+1 disjoint paths

Now we need to show that these paths are mutually disjoint from each other. To prove that, we will prove that path i, 1 ≤ i ≤ f+1, is disjoint from path j, 1 ≤ j ≤ i. Three cases are considered:

Case 1: $j < i-n$, there are no common stages in which both paths operate in the *exchange* mode. Therefore path i and path j are disjoint.

Case 2: $j = i-n$, the only stage of possible conflict is in stage i, if both path j and path i perform the *exchange* operation on the same switch in this stage. We know that in stage i, by construction, path i is switching from node A to some node, and path j is switching from some node to node B. Given $A \neq B$, the only possibility of conflict is that both paths are switching from A to B. It is not possible. Path j switch from A to some other node at stage j, and because the fact that n consecutive masks in a Cyclic MIN are linearly independent, path j will not reach A until after stage i. Therefore path i and path j are disjoint.

For $i-n < j < i$, the two paths are disjoint till stage $j+n$ since path j *exchanges* at stage j while path i goes *straight* and they differ at bit (j mod n). After stage $j+n$ they must agree on bit (j mod n), since they must reach the same destination B. Therefore only one of the two paths will use the switch at stage $j+n$. Consequently path i and path j are disjoint.

Hence the $f+1$ paths from A to B are mutually disjoint and a (n, f) Cyclic Multistage Interconnection Networks tolerates f switch faults. ÿ

Theorem 6.2 shows that the performance of Cyclic Multistage Interconnection Networks meets the lower bound stated in Theorem 6.1. In other words, this construction is optimal in the number of extra stages used to tolerate any given number of switch faults.

Notice in the construction of the Cyclic Multistage Interconnection Networks, the $f+1$ paths share horizontal lines only in the first f and last f stages. This indicates that the demultiplexor/multiplexor pairs that enable the *stuck in straight* fault model are only needed for those stages.

Routings in a Cyclic Multistage Interconnection Network are performed using routing tags [Adams et al. 1987]. The routing tag is a binary vector composed of the coefficients obtained by decomposing the difference between the source and destination into the linear combinations of the stage masks. The tag is computed at the source node only once and is attached to each message. Whether the switch in the ith stage operates in the *exchange* or the *straight* mode is determined by the ith bit in the routing tag. The proof for Theorem 6.2 explicitly gives the construction of f+1 disjoint paths between any two nodes. Therefore for a source node in a failure-free situation, there are f+1 sets of routing tags to each destination. A message can be sent by any one of the f+1 tags. When a switch failure occurs, it should be detected by the source node, and the route that uses the faulty switch is eliminated from the tag selections. This tag elimination process only occurs once for each failure, thus is not computationally intensive for the source node.

Cyclic Multistage Interconnection Networks presents considerable practical merit. For example, attaching two MIN switches together head-to-tail, we naturally constructed an n-extra stage Cyclic MIN, and it can tolerate n switch failures.

5.2 Necessary and Sufficient Condition

In the previous section, we introduced the Cyclic Multistage Interconnection Network that demonstrates optimal performance in tolerating any number of switch faults. The construction, however, is not unique. In this section we extend the results to a more general class of fault-tolerant Multistage Interconnection Networks, named Generalized Cyclic Multistage Interconnection Networks.

Definition 6.2: ***The Generalized Cyclic Multistage Interconnection Network***

A (n, f) Generalized Cyclic Multistage Interconnection Network is an n-dimensional f-extra-stage Multistage Interconnection Network with the property that the masks of every n consecutive stages span the n-dimensional vector space.

Figure 29 illustrates a (3, 4) generalized cyclic MIN using a non-singleton and non-repetitive mask set. The five disjoint paths between the nodes 000 and 110 are shown in the illustration.

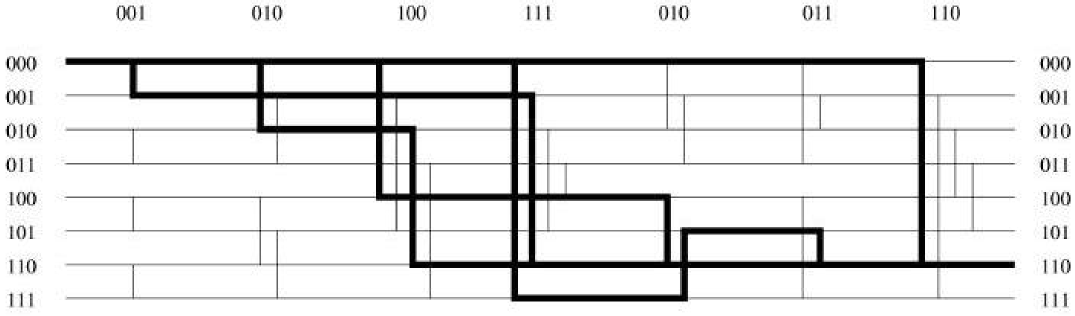


Figure 29 Three-dimensional 4-extra-stage Generalized Cyclic MIN

Clearly, the Cyclic MIN is a subclass of the Generalized Cyclic MIN. We will prove that the Generalized Cyclic MIN has the same fault-tolerance capabilities as the Cyclic MIN, namely, they tolerate f faults with f extra stages. In addition, Generalized Cyclic MINs are the only extra-stage Multistage Interconnection Networks that demonstrate this optimal fault-tolerance capability.

Theorem 6.3: *An n-dimensional f-extra-stage Multistage Interconnection Network with 2 X 2 switches tolerates f switch faults if and only if the masks of every n consecutive stages span the n-dimensional vector space.*

Proof: We prove the forward direction of the theorem by contradiction. Suppose an n-dimensional f-extra-stage Multistage Interconnection Network does not have the property that the masks of any n consecutive stages span the space. There exist n consecutive stages in the

MIN whose masks do not span the n-dimensional space. We can find node A and B, between which a path cannot be found in the non-spanning n stages. Suppose the faults happen on the switches of the remaining f stages at both sides of these n non-spanning stages in such a way that all the faults before the n stages happen at switches connected to node A, and all the faults after the n stages happen at switches connected to node B. The communication between A and B fails. Therefore an n-dimensional f-extra-stage MIN tolerates f switch faults only if the masks of every n consecutive stages span the n-dimensional vector space.

The proof of the backward direction is similar to the proof of Theorem 6.2. The constructions of the f+1 paths from node A to node B are the same. We need to show that these paths are all mutually disjoint from each other. Again we prove that path i, $i \leq f+1$, is disjoint from path j, $j < i$ by considering three cases:

Case 1: $j < i-n$, there are no common stages that the switch in the stage operates in *exchange* mode for both paths. Therefore path i and path j are disjoint.

Case 2: $j = i-n$, the two paths share stage i. We know that path j *exchanges* at stage j, while path i goes *straight*. Since any n consecutive masks are linearly independent, m_j cannot be represented by a linear combination of m_{j+1} through m_{i-1}. Therefore the two paths are disjoint till stage i, and the only way that the two paths conflict is that at stage i, path j *exchanges* from $A - m_i$ to A while path i *exchanges* from A to $A - m_i$. But it is not possible since path j must reach B after ith stage and $A \neq B$. Therefore path i and path j are disjoint.

For $i-n < j < i$, since path j *exchanges* at stage j while path i goes *straight*, the two paths are disjoint till stage j+n with the same reasoning as the previous case. At stage j+n, only one of the two paths *exchanges* since they must reach the same destination. Therefore path i and path j are disjoint. ÿ

5.3 Extensions

In this section, we will make two extensions to the results presented in the previous sections. First, instead of looking at Multistage Interconnection Networks with 2 X 2 Switching Elements, we will show that the theorems presented in the previous sections also apply to the MINs consisting of $t \times t$ switching elements. Following that, we will show that the results are also valid if we are to guarantee the broadcast capabilities of the network.

Let us look at a nine-node three-extra stage (2, 3) generalized cyclic Multistage Interconnection Network consisting of 3×3 switches. Figure 30 shows the four mutually disjoint paths from node 00 to node 20.

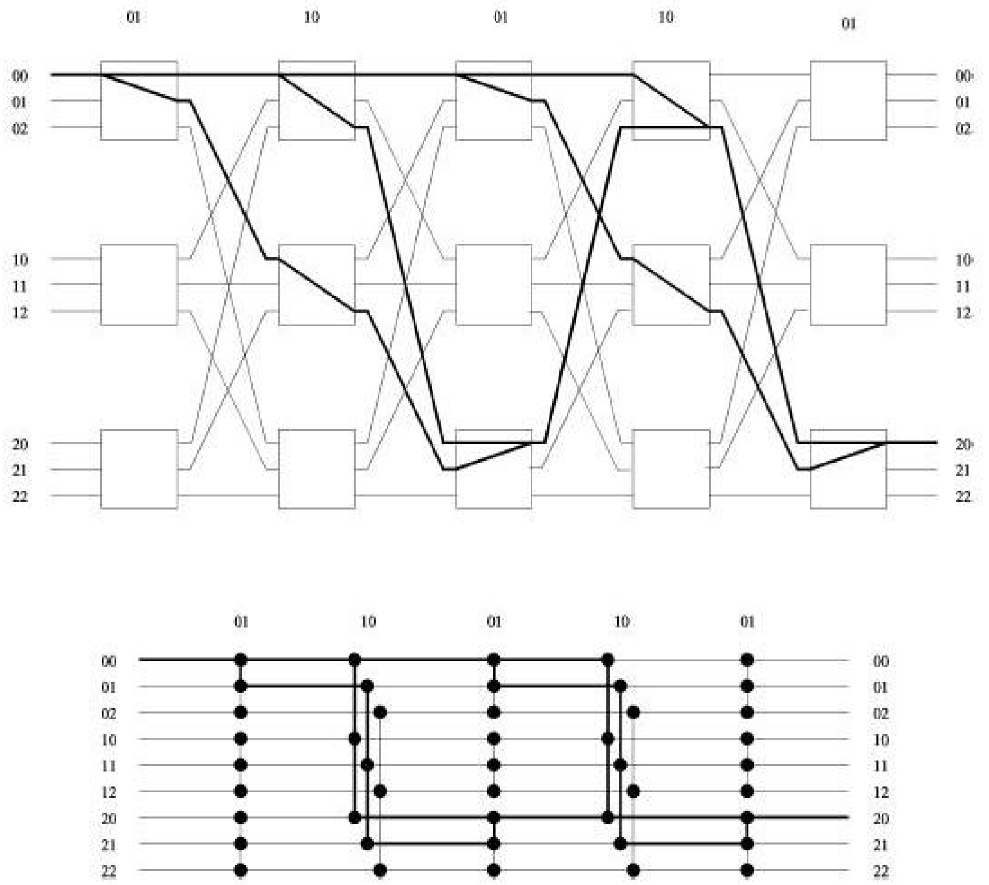


Figure 30 Extension to the MINs with 3 x 3 switching elements

Theorem 6.4: *An n-dimensional f-extra-stage Multistage Interconnection Network with t ´ t switches tolerates f switch faults if and only if the masks of every n consecutive stages span the n-dimensional vector space.*

Proof: The proof of the forward direction is the same as the proof for the 2 X 2 case. To prove the backward direction, we similarly construct f+1 paths from node A to node B and prove that they are disjoint. The difference lies in the construction of the first f paths. The reason for the modification is that a 2 X 2 switch can only go *straight* or *exchange*, while a t × t switch has t ways of switching. We say a switch is in mode s if for that switch, output = input + s × mask , $0 \le s \le t-1$

In this proof all vector operations are modular-t. In the construction of path i, $i \le f$, we decompose the n-dimensional vector B-A into a linear combinations of the masks $\{m_i, m_{i+1}, ..., m_{i+n-1}\}$:

$$B - A = \Sigma_{j=i}^{i+n-1} (c_j \times m_j)$$

Since $\{m_i, m_{i+1}, ..., m_{i+n-1}\}$ span the space, such a decomposition is always possible. If the coefficient $c_i \neq 0$, the switches in stage i are forced to be in mode c_i, i.e., output = input + $c_i \times m_i$. If $c_i=0$, we only need to make sure that the switches in stage i *exchange* to some output, as long as they do not go *straight*. The path i will reach the destination B by a regular routing in the next n stages, i.e., stages i+1 through i+n. The construction of path f+1 and the proof that these f+1 paths are disjoint to each other are the same as the proof for the 2 X 2 case. ÿ

In the previous sections, we have shown that in an f-extra-stage Cyclic MIN, there exist f+1 disjoint paths between any pair of nodes. It follows that in the presence of f faults, at least one path remains intact between any pair of nodes. The broadcast from any node A is achieved by picking a survived path between node A and every other node. If a switch is used by more than one path and the two paths enter the switch from two different inputs, we collapse the part of the two paths before the switch, so that the switch will operate in one of the legal modes. It is obvious that after such collapses the connections between the node pairs remain.

Therefore an f-extra-stage Cyclic Multistage Interconnection Network guarantees broadcast connectivity in the presence of f switch faults. As an example, Figure 31 shows the survived broadcast switch faults.

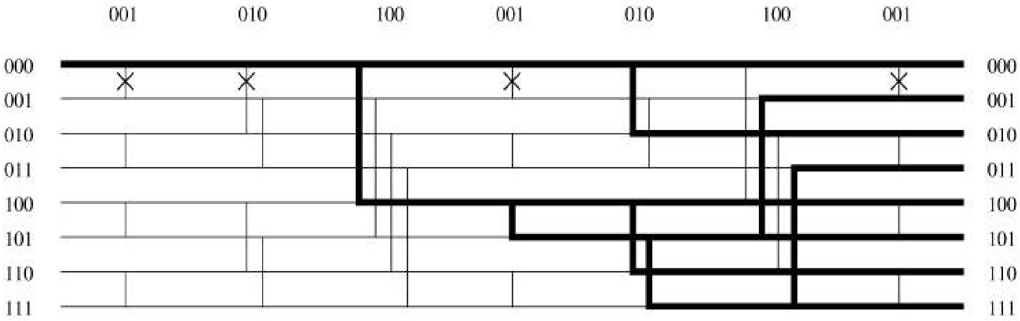


Figure 31 Survived broadcast tree with four faults in a (3, 4) cyclic MIN

6 CONCLUSION AND FUTURE DIRECTIONS

Internet is bringing communication and computation together like never before. This is an opportunity to bring the distributed computing protocols and algorithms into the field of computer networks. This dissertation is the beginning of an effort in that direction. The result is a set of practical and efficient distributed protocols and architectures for networking devices. With these discoveries, a number of networking devices can work together as a single system, which is more fault-tolerant and more scalable in performance.

What we have studied and presented in this thesis is only a small step in this research direction. We are continuing working on the following areas to march further in this direction:

First, we are continuing the performance analysis and measurement of the Raincore protocols. At the same time we are also exploring other topologies and mechanisms for group communication in addition to a simple ring, for example, ring of rings. We'll compare the performance of different topologies for different applications and size of clusters.

Second, the current Raincore protocol is implemented on top of the UDP protocol in an Ipv4 environment. We can take advantage of the features offered in emerging standards in the networking world, for example, VI architecture and Ipv6, for performance optimization.

Third, there are outstanding possibilities of building the Raincore protocols lower into the system. Without a doubt it has the potential to be implemented as part of an Operating System kernel, to become a standard part of the networking stack. Moreover there can be hardware implementation of the Raincore protocols (Raincore on a chip). It can potentially become a part of the next-generation network-computing platform.

Last but not least, the Raincore Protocols will be extended and enriched to address more specific and challenging networking applications. For example, we are looking at the application in the areas of wireless networking and networked storage.

7 BIBLIOGRAPHY

Adams, G., D. Agarwal, and H. Siefel. A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks, *IEEE Computer*, pp. 14-27, June 1987.

Adams, G., and H. Siegel. The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems, *IEEE Transactions on Computers*, pp. 443-454, May 1982.

Agrawal, D. P.. Graph Theoretical Analysis and Design of Multistage Interconnection Networks, *IEEE Transactions on Computers*, pp. 637-648, July 1983.

Amir, Y., L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem Single-Ring Ordering and Membership Protocol. *ACM Transactions On Computer Systems*, 13(4), pages 311-342, November 1995.

Amir, Y., D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-system for High Availability. *Proceedings of the IEEE 22nd Annual International Symposium on Fault-Tolerant Computing* (Boston, MA). IEEE, New York, 76-84.

Anderson, T. E., D. E. Culler, D. A. Patterson, et al.. A Case for Networks of Workstations: NOW. *IEEE Micro*, Feb 1995.

Bal, H. E.. *Programming Distributed Systems*. Hemel Hempstead, England: Prentice Hall International, 1991.

Bal, H. E., M. F. Kaashoek, and A. S. Tanenbaum. Experience with Distributed Programming in Orca. *Proceedings of International Conference On Computer Languages*, IEEE, pp. 79-89, 1990.

Bal, H. E., M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transaction on Software Engineering*. Vol. 18, pp. 190-205, March, 1992.

Becker, D. J., T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, C. V. Packer. Beowulf: A Parallel Workstation for Scientific Computation. *Proceedings, International Conference on Parallel Processing*, 1995.

Benes, V.E.. Optimal Rearrangeable Multistage Connecting Networks, *Bell System Technical Journal*, No. 4, II, July 1964.

Bennett, J. K., J. K. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. *Proceedings of Second ACM Symposium on Principles and Practice of Parallel Programming*, ACM, pp. 168-176, 1990.

Bershad, B. N., and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. *CMU Report CMU-CS-91-170*, Sept., 1991.

Bershad, B. N., M. J. Z ekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. *Proceedings of IEEE COMPCON Conference*. IEEE, pp. 528-537, 1993.

Birman, K. P.. The Process Group Approach to Reliable distributed Computing, *Communications of the ACM*. Vol. 36, pp. 36-53, Dec. 1993.

Birman, K. P. and R. V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press.

Birman, K. P. and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems,* 5(1), pages 47-76, February 1987.

Bohossian V., C. C. Fan, P. LeMahieu, M. Riedel, L. Xu, and J. Bruck. Computing in the RAIN: A Reliable Array of Independent Nodes, *IEEE Transactions on Parallel and Distributed Systems*, February 2001.

Bruck, J. and C. Ho. Fault-Tolerant Cube Graphs and Coding Theory, *IEEE Transactions on Information Theory*, November 1996.

Cam, H. and J. Fortes. Rearrangeability of Shuffle-Exchange Networks, *Proceedings of 3rd Symposium on the Frontiers of Massively Parallel Computation*, pp. 303-314, October 1990.

Carriero, N., and D. Gelernter. The S/Net's Linda Kernel. *ACM Transactions on Computer Systems*, vol. 4, pp. 110-129, May 1986.

Carriero, N., and D. Gelernter. Linda in Contect. *Communications of the ACM*, vol. 32, pp. 444-458, April, 1989.

Carter, J. B,, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. *Proceedings of 13th Symposium on Operating Systems Principles*, ACM, pp. 152-164, 1991.

Carter, J. B., Bennett, J. K., and Zwaenepoel, W., 1994. Techniques for Reducing Consistency-Related Communication in Disributed Shared Memory Systems. *ACM Transactions on Computer Systems*, vol. 12, 1994.

Chuang, P. J.. CGIN: A Fault Tolerant Modified Gamma Interconnection Network, *IEEE Transactions on Parallel and Distributed Systems*, pp. 1303-1308, December 1996.

Chandra, T.D., V. Hadzillacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the fifteenth ACM symposium on principles of Distributed Computing*, ACM Press, 322-330, 1996.

Chandy, K. M., J. Misra, and L. M. Haas. Distributed Deadlock Detection. *ACM Transactions on Computer Systems*, vol. 1, pp. 144-156, May 1983.

Chandy, K. M. and J. Misra. *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

Chang, J. M. and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computers Systems 2*, 3, 251-273, August, 1984.

Damianakis, S., A. Bilas, C. Dubnicki, and E. W. Felten. Client Server Computing on the SHRIMP Multicomputer. *IEEE Micro*, February 1997.

Day, J.D., and H. Zimmerman. The OSI Reference Model, *Proceedings of IEEE*, vol. 71, pp. 1334-1340, Dec. 1983.

Dubois, M., C. Scheurich, and F. A. Briggs. Memory Access Buffering in Multiprocessors. *Proceedings of 13th Annual International Symposium on Computer Architecture*, ACM, p. 434-442, 1986.

Fan, C. C. and J. Bruck. Optimal Construction of Fault-Tolerant Multistage Interconnection Networks. *Proceedings of 10th International Conference on Parallel and Distributed Computing Systems.* October, 1997.

Fan, C. C. and J. Bruck. Tolerating Multiple Faults in Fault-Tolerant Multistage Interconnection Networks with Minimal Extra Stages. *IEEE Transactions on Computers,* September 2000.

Fan, C. C., and J. Bruck. The Raincore Distributed Session Service for Networking Elements, *International Parallel and Distributed Processing Symposium, Workshop on Communication Architecture for Clusters*, April 2001.

Fischer, M. J., N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32*, 2, 374-382, April, 1985.

Franceschetti, M., and J. Bruck. On the Possibility of Group Membership. *Dependable Network Computing,* Kluwer, Nov. 1999.

Gelernter, D.. Generative Communication in Linda. *ACM Transaction on Programming Languages and Systems.* Vol. 7, pp. 80-112, Jan., 1985.

Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *Proceedings of 17th Annual International Symposium on Computer Architecture*, ACM, pp. 15-26, 1990.

Goodman, J. R.. Cache Consistency and Sequential Consistency. *Tech Report 61, IEEE Scalable Coherent Interface Working Group*, IEEE, 1989.

Gropp, W., E. Lusk and J. Skjellum. *Using MPI : Portable Parallel Programming With the Message-Passing Interface*. MIT Press, 1998.

Hutto, P. W., and M. Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. *Proceedings of 10th International Conference on Distributed Computing Systems*, IEEE, pp. 302-311, 1990.

Kleinrock, L. and F. A. Tobagi. Packet Switching in Radio Channels: Part I — Carrier Sense Multiple-Access Modes and Their Throughput-Delay Characteristics, *IEEE Transactions on Communications*, Vol. COM-23, No. 12, pp. 1400-1416, Dec. 1975.

Knuth, D.. *The Art of Computer Programming: Sorting and Search*, Reading, MA: Addison-Wesley, 1973.

Kumar, V.P. and S. M. Reddy. Augmented Shuffle-Exchange Multistage Interconnection Networks, *IEEE Computer*, pp. 30-40, June 1987.

Lam, S. S. and A. U. Shankar. Protocol Verification via Projections, *IEEE Transactions on Software Engineering* 10, pp. 325-342, 1984.

Lamport, L.. Time Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, vol. 21, pp. 558-564, July, 1978.

Lamport, L.. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transaction on Computers*, vol. C-28, pp. 690-691, Sept. 1979.

Lamport, L.. A simple Approach to Specifying Concurrent Systems, *Communicaions of the ACM* 32, pp. 32-45, 1989.

Lamport, L.. A Temporal Logic of Actions, *DEC Systems Research Center SRC Report 57*, 1990.

Leighton, F.. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, San Mateo, CA: Morgan Kaufmann, 1992.

Li, K.. Shared Virtual Memory on Loosely Coupled Multiprocessors, *Ph.D. Thesis*, Yale University, 1986.

Li, K., and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, vol. 7, pp. 321-359, Nov, 1989.

Liao, Y., M. Lu, and N. F. Tzeng, The Palindrome Network for Fault-Tolerant Interconnection, *Proceedings of 8th IEEE Symposium on Parallel and Distributed Processing*, pp. 556-561, October 1996.

Linial, N. and M. Tarsi. Interpolation Between Bases and the Shuffle Exchange Network, *European Journal of Combinatorics*, pp. 29-39, October 1989.

Lipton, R. J., and J. S. Sandberg. PRAM: A Scalable Shared Memory. Tech Report CS-TR-180-88, Princeton University, Sept, 1988.

Lynch, N. A. and M. R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms, *Proceedings of the Sixth ACM Annual Symposium on Principles of Distributed Computing*, pp 137-151, April 1987.

Lynch, N. A. and M. R. Tuttle. An Introduction to Input/Output Automata, *CWI-Quarterly* 2(3), pp. 219-246, 1989.

Moser, L. E., P. M. Melliar-Smith, and V. Agrawala. Processor membership in asynchronous distributed systems. *IEEE Transactions on Parallel and Distributed Systems 5*, 5, 459-473, May 1994.

Mullender, S.. *Distributed Systems*. 2nd Edition. ACM Press, New York, 1993.

Neiger, G.. A New Look at Membership Services. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, ACM press, 331-340, 1996.

Pease, M.. The Indirect Binary n-Cube Microprocessor Array, *IEEE Transactions on Computers*, vol. C-26, No. 5, pp. 458-473, May 1977.

Raynal, M.. A Simple Taxonomy for Distributed Mutual Exclusion Algorithms. *Operating Systems Review*, vol. 25, pp. 47-50, April 1991.

Ricciardi, A. M., and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, ACM Press, 341-352, May 1991.

Shih, C. and K. Batcher. Adding Multiple-Fault Tolerance to Generalized Cube Networks, *IEEE Transactions on Parallel and Distributed Systems*, pp. 785-792, August 1994.

Smith, C.. *Theory and the Art of Communications Design.* State of the University Press, 1997.

Stevens, W.R.. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, Massachusetts, 1994.

Tanenbaum, A. S.. *Distributed Operating Systems*. Prentice-Hall, New Jersey, 1995.

Tanenbaum, A. S.. *Computer Networks, third edition*. Prentice-Hall, New Jersey, 1996.

Van Renesse, R., T. M. Hickey, and K. P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report 94-1441, Cornell University, Department of Computer Science, August 1994.

Varma, A. and C. S. Raghavendra. Fault-Tolerant Routing in Multistage Interconnection Networks, *IEEE Transactions on Computers*, pp. 185-393, March 1989.

Xu, M. and X. Shen. A New Fault-Tolerant Generalized Cube with an Extra Stage, *Proceedings of International Computer Science Conference*, p99-105, December, 1992.