Chapter 1

A State-of-the-Art Distributed System: Computing with BOB

Michael D. Schroeder

Distributed systems are a popular and powerful computing paradigm. Yet existing examples have serious short-comings as a base for general-purpose computing. This chapter explores the characteristics, strengths, and weaknesses of distributed systems. It describes a model for a state-of-the-art distributed system that can do a better job of supporting general-purpose computing than existing systems. This model system combines the best features of centralized and networked systems, with improved availability and security. The chapter concludes by outlining the technical approaches that appear most promising for structuring such a system.

This chapter does not specifically discuss application-specific distributed systems, such as automated banking systems, control systems for roaming and tracking cellular telephones, and retail point-of-sales systems, although there are many economically important examples. The issues raised in the chapter do apply to such systems and the model system described here should be a suitable base for many of them. Some systems that are designed to support a narrow set of uses, however, may need to be structured in application-specific ways that can be simpler and more efficient for those uses. The extent to which special-purpose systems should be built on a general-purpose distributed base needs to be investigated further. This chapter also does not address real-time distributed systems, such as control systems for factories, aircraft, or automobiles, which face distinctive scheduling and resource utilization requirements.

1.1 Characteristics of Distributed Systems

A distributed system is several computers doing something together. Thus, a distributed system has three primary characteristics.

- **Multiple Computers** — A distributed system contains more than one physical computer, each consisting of cpu's, some local memory, possibly some stable storage like disks, and I/O paths to connect it with the environment.
- **Interconnections** — Some of the I/O paths will interconnect the computers. If they cannot talk to each other, then it is not going to be a very interesting distributed system.
- **Shared State** — The computers cooperate to maintain some shared state. Put another way, if the correct operation of the system is described in terms of some global invariants, then maintaining those invariants requires the correct and coordinated operation of multiple computers.

Building a system out of interconnected computers requires that four major issues be addressed.

- **Independent Failure** — Because there are several distinct computers involved, when one breaks others may keep going. Often we want the 'system' to keep working after one or more have failed.
- **Unreliable Communication** — Because in most cases the interconnections among computers are not confined to a carefully controlled environment, they will not work correctly all the time. Connections may be unavailable. Messages may be lost or garbled. One computer cannot count on being able to communicate clearly with another, even if both are working.
- **Insecure Communication** — The interconnections among the computers may be exposed to unauthorized eavesdropping and message modification.
- **Costly Communication** — The interconnections among the computers usually provide lower bandwidth, higher latency, and higher cost communication than that available between the independent processes within a single machine.

A centralized system that supports multiple processes and provides some form of interprocess communication, e.g. a UNIX timesharing system, can exhibit in virtual form the three primary characteristics of a distributed system. A centralized system may even manifest independent failure, e.g. the dæmon process for mail transport may crash without stopping interactive user processes on the same system. Thus, design and programming techniques associated with communicating sequential processes in centralized systems form part of the basic techniques in the distributed systems arena. However, centralized systems are usually successful without dealing with independent failure and usually do not confront unreliable, insecure, and costly communications. In a distributed system these issues must be addressed.

The canonical example of a general-purpose distributed system today is a networked system — a set of workstations/PCs and servers interconnected with a network.

1.2 Networked vs Centralized Systems

It is easy to understand why networked systems are popular. Such systems allow the sharing of information and resources over a wide geographic and organizational spread. They allow the use of small, cost-effective computers and get the computing cycles close to the data. They can grow in small increments over a large range of sizes. They allow a great deal of autonomy through separate component purchasing decisions, selection of multiple vendors, use of multiple software versions, and adoption of multiple management policies. Finally, they do not necessarily all crash at once. Thus, in the areas of sharing, cost, growth, and autonomy, networked systems are better than traditional centralized systems as exemplified, say, by timesharing.

On the other hand, centralized systems do some things better than today's networked systems. All information and resources in a centralized system are equally accessible. Functions work the same way and objects are named the same way everywhere in a centralized system. And a centralized system is easier to manage. So despite the advantages of networked systems, centralized systems are often easier to use because they are more accessible, coherent, and manageable.

In the areas of security and availability, the comparison between networked systems and centralized systems produces no clear-cut advantage for either.

1.2.1 Security

In usual practice to date, neither centralized nor networked systems offer real security, but for different reasons.

A centralized system has a single security domain under the control of a single authority. The trusted computing base is contained in a single operating system. That operating system executes on a single computer that can have a physically secure environment. With all the security eggs in one basket, so to speak, users understand the level of trust to assign to the system and who to speak to when problems arise. On the other hand, it is notoriously difficult to eliminate all the security flaws from an operating system or from the operating environment, and with a single security domain one such flaw can be exploited to break the security of the entire system.

Networked systems have multiple security domains and thus exhibit the inverse set of security properties. The trusted computing base is scattered among many components that operate in environments with varying degrees of physical security, differing security policies, and possibly under different authorities. The interconnections among the computers are physically insecure. It is hard to know what is being trusted and what can be trusted. But, because the system contains many computers, exploiting a security flaw in the software or environment of one computer does not automatically compromise the entire system.

1.2.2 Availability

A similar two-sided analysis applies to availability. A centralized system can have a controlled physical and operational environment. Since a high proportion of system failures are the result of operational and environmental factors, careful management of this single environment can produce good availability. But when something does go wrong the whole system goes down at once, stopping all users from getting work done.

In a networked system the various computers fail independently. However, it is often the case that several computers must be in operation simultaneously before a user can get work done, so the probability of the system failing is greater than the probability of one component failing. This increased probability of not working, compared to a centralized system, is the result of ignoring independent failure. The consequence is Leslie Lamport's definition of a distributed system: 'You know you have one when the crash of a computer you've never heard of stops you from getting any work done.'

On the other hand, independent failure in a distributed system can be exploited to increase availability and reliability. When independent failure is properly harnessed by replicating functions on independent components, multiple component failures are required before system availability and reliability suffer. The probability of the system failing thus can be less than the probability in a centralized system. Dealing with independent failure to avoid making availability worse, or even to make it better, is a major task for designers of distributed systems.

A distributed system also must cope with communication failures. Unreliable communication not only contributes to unavailability, it can lead to incorrect functioning. A computer cannot reliably distinguish a down neighbor from a disconnected neighbor and therefore can never be sure an unresponsive neighbor has actually stopped. Maintaining global state invariants in such circumstances is tricky. Careful design is required to actually achieve correct operation and high availability using replicated components.

1.2.3 A State-of-the-Art Distributed System

It seems feasible to develop a distributed system that combines the accessibility, coherence, and manageability advantages of centralized systems with the sharing, growth, cost, and autonomy advantages of networked systems. If real security and high availability were added to the mix, then we would have a state-of-the-art computing base for many purposes. Achieving this combination of features in a single system is the central challenge of supporting general-purpose computing well with a distributed system. No existing system fulfills this ideal.

1.3 The Properties and Services Model

We can describe this best-of-both-worlds (BOB) distributed computing base in terms of a model of its properties and services. This more technical description of the goals provides a structure that will help us to understand the mechanisms needed to achieve them.

The properties and services model defines BOB as:

- a heterogeneous set of hardware, software, and data components;
- whose size and geographic extent can vary over a large range;
- connected by a network;
- providing a uniform set of services (naming, remote invocation, user registration, time, files, etc);
- with certain global properties (names, access, security, management and availability).

Because we are talking about a base for general-purpose computing, the model is defined in terms most appropriate to understanding by the programmers who develop components that are part of the base and who develop the many different applications that are to be built on top of it. But the fundamental coherence provided by the model will show through such components and applications (when they are correctly implemented) to provide a coherent system as viewed by its human users too.

The coherence that makes BOB a system rather than a collection of computers is a result of its uniform services and global properties. The services are available in the same way to every part of the system and the properties allow every part of the system to be viewed in the same way, regardless of system size. Designers are well aware of the care that must be taken to produce implementations that can support growth to very large sizes (thousands of nodes). A similar challenge exists in making such expandable systems light-weight and simple enough to be suitable for very small configurations too.

BOB's coherence does not mean that all the components of the system must be the same. The model applies to a heterogeneous collection of computers running operating systems such as UNIX, VMS, MS-DOS, Windows NT, and others. In short, all platforms can operate in this framework, even computers and systems from multiple vendors. The underlying network can be a collection of local area network segments, bridges, routers, gateways, and various types of long distance services with connectivity provided by various transport protocols.

1.3.1 Properties

What do BOB's pervasive properties mean in more detail?

- **Global names** — the same names work everywhere. Machines, users, files, distribution lists, access control groups, and services have full names that mean the same thing regardless of where in the system the names are used. For instance, Butler Lampson's user name might be something like `/com/dec/src/bwl` throughout the system. He will operate under that name when using any computer. Global naming underlies the ability to share things.
- **Global access** — the same functions are usable everywhere with reasonable

performance. If Butler sits down at a machine when visiting in California, he can do everything there that he can do when in his usual office in Massachusetts, with perhaps some performance degradations. For instance, from Palo Alto Butler could command the local printing facilities to print a file stored on his computer in Cambridge. Global access also includes the idea of data coherence. Suppose Butler is in Cambridge on the phone to Mike Schroeder in Palo Alto and Butler makes a change to a file and writes it. Mike should be able to read the new version as soon as Butler thinks he has saved it. Neither Mike nor Butler should have to take any special action to make this possible.

- **Global security** — the same user authentication and access control work everywhere. For instance, Butler can authenticate himself to any computer in the system; he can arrange for data transfer secure from eavesdropping and modification between any two computers; and assuming that the access control policy permits it, Butler can use exactly the same mechanism to let the person next door and someone from another site read his files. All the facilities that require controlled access (logins, files, printers, management functions, etc.) use the same machinery to provide access control.
- **Global management** — the same person can manage components anywhere. Obviously one person will not manage all of a large system. But the system should not impose a priori constraints on which set of components a single person can manage. All of the components of the system provide a common interface to management tools. The tools allow a manager to perform the same action on large numbers of components at once. For instance, a single system manager can configure all the workstations in an organization without leaving his office.
- **Global availability** — the same services work even after some failures. System managers get to decide (and pay for) the level of replication for each service. As long as the failures do not exceed the redundancy provided, each service will go on working. For instance, a group might decide to duplicate its file servers but get by with one printer per floor. System-wide policy might dictate a higher level of replication for the underlying communication network. Mail does not have to fail between Palo Alto and Cambridge just because some machine goes down in Lafayette, Indiana.

1.3.2 Services

The standard services defined by BOB include the following fundamental facilities:

- **Names** — provides access to a replicated, distributed database of global names and associated values for machines, users, files, distribution lists, access control groups, and services. A name service is the key BOB component for providing global names, although most of the work involved in implementing global names is making all the other components of the distributed system, e.g. existing operating systems, use the name service in a consistent way.

- **Remote procedure call** — provides a standard way to define and securely invoke service interfaces. This allows service instances to be local or remote. The RPC mechanism can be organized to operate by dynamically choosing one of a variety of transport protocols. Choosing RPC for the standard service invocation mechanism does not force blocking call semantics on all programs. RPC defines a way to match response messages with request messages. It does not require that the calling program block to await a response. Methods for dealing with asynchrony inside a single program are a local option. Blocking on RPC calls is a good choice when the local environment provides multiple threads per address space.
- **User registrations** — allows users to be registered and authenticated and issues certificates permitting access to system resources and information.
- **Time** — distributes consistent and accurate time globally.
- **Files** — provides access to a replicated, distributed, global file system. Each component machine of BOB can make available the files it stores locally through this standard interface. The multiple file name spaces are connected by the name service. The file service specification should include standard presentations for the different VMS, UNIX, etc. file types. For example, all implementations should support a standard view of any file as an array of bytes.
- **Management** — provides access to the management data and operations of each component.

In addition to these base level facilities, BOB can provide other services appropriate to the intended applications, such as:

- **Records** — provides access to records, either sequentially or via indexes, with record locking to allow concurrent reading and writing, and journaling to preserve integrity after a failure.
- **Printers** — allows printing throughout the network of documents in standard formats such as Postscript and ANSI, including job control and scheduling.
- **Execution** — allows programs to be run on any machine (or set of machines) in the network, subject to access and resource controls, and efficiently schedules both interactive and batch jobs on available machines, taking account of priorities, quotas, deadlines, and failures. The exact configuration and utilization of cycle servers (as well as idle workstations that can be used for computing) fluctuates constantly, so users and applications need automatic help in picking the machines on which to run.
- **Mailboxes** — provides a transport service for electronic mail.
- **Terminals** — provides access to a windowing graphics terminal from a computation anywhere in the network.
- **Accounting** — provides access to a system-wide collection of data on resource usage which can be used for billing and monitoring.

In many cases adequate, widely accepted standards already exist for the definition of the base and additional services. Each service must be defined and implemented to provide the five pervasive properties.

1.3.3 Interfaces

Interfaces are the key to making BOB be a coherent and open system. Each of the services is defined by an interface specification that serves as a contract between the service and its clients. The interface defines the operations to be provided, the parameters of each, and the detailed semantics of each relative to a model of the state maintained by the service. The specification is normally represented as an RPC interface definition. Some characterizations of the performance of the operations must be provided (although it is not well understood how to provide precise performance characterizations for operations whose performance varies widely depending on the parameters and/or use history). A precisely defined interface enables interworking across models, versions, and vendors. Several implementations of each interface can exist and this variety allows the system to be heterogeneous in its components. In its interfaces, however, the system is homogeneous.

It is this homogeneity that makes it a system with predictable behavior rather than a collection of components that can communicate. If more than one interface exists for the same function, it is unavoidable that the function will work differently through the different interfaces. The system will consequently be more complicated and less reliable. Perhaps some components will not be able to use others at all because they have no interface in common. Certainly customers and salesmen will find it much more difficult to configure workable collections of components, and programmers will not know what services they can depend upon being present.

1.4 Achieving the Global Properties

Experience and research have suggested a set of models for achieving global naming, access, security, management, and availability. For each of these pervasive properties, we will consider the general approach that seems most promising.

1.4.1 Naming Model

Every user and every client program sees the entire system as the same tree of named objects. A global name is interpreted by following the named branches in this tree starting from the global root. Every node has a way to find a copy of the root of the global name tree.

A hierarchic name space is used because it is the only naming structure we know of that scales well, allows autonomy in the selection of names, and is sufficiently malleable to allow for a long lifetime. A global root for the name space is required to provide each object with a single name that will work from everywhere. Non-hierarchic links can be added where a richer naming structure is required.

For each object type there is some service, whose interface is defined by BOB, that provides operations to create and delete objects of that type and to read and change their state.

The top part of the naming tree is provided by the BOB name service and the objects near the root of the tree are implemented by the BOB name service. A node in the naming tree, however, can be a *junction* between the name service and some other service, e.g. a file service. A junction object contains:

- a set of servers for the named object
- rules for choosing a server name
- the service interface ID, e.g. 'BOB File Service 2.3'
- an object parameter, e.g. a volume identifier

To look up a name through a junction, choose a server and call the service interface there with the name and the object parameter. The server looks up the rest of the name.

The servers listed in a junction object are designated by global names. To call a service at a server the client must convert the server name to something more useful, like the network address of the server machine and information on which protocols to use in making the call. Looking up a server name in the global name tree produces a server object that contains:

- a machine name
- a set of communication protocol names

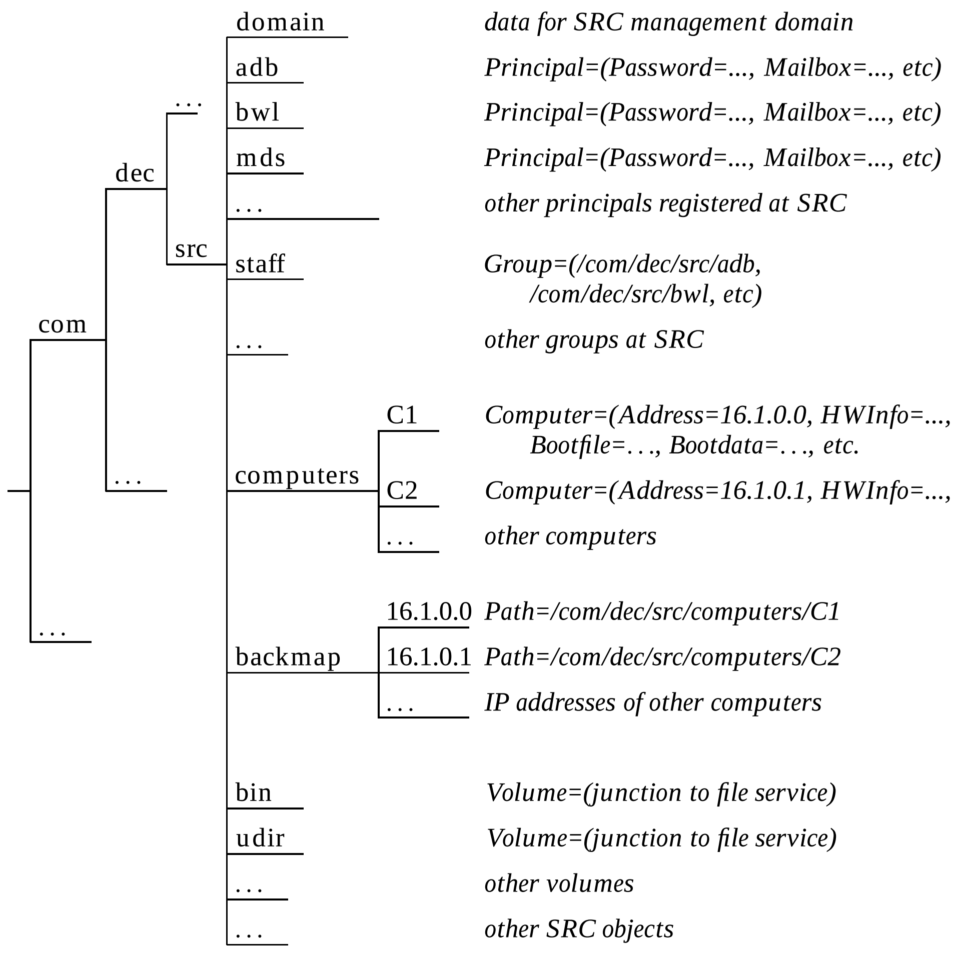
A final name lookup maps the (global) machine name into the network address that will be the destination for the actual RPC to the service.

The junction machinery can be used at several levels, as appropriate. The junction is a powerful technique for unifying multiple implementations of naming mechanisms within the same hierarchic name space.

Figure 1.1 gives an example of what the top parts of the global name space might look like, based on the naming scheme of the Internet Domain Name Service. An X.500 service could also provide this part of the name space (or both the Internet DNS and X.500 could be accommodated). An important part of implementing BOB is defining the actual name hierarchy to be used in top levels of the global name space.

Consider some of the objects named here. `/com`, `/com/dec`, and `/com/dec/src` are directories implemented by the BOB name service. `/com/dec/src/adb` is a registered user, also an object implemented by the name service. The object `/com/dec/src/adb` contains a suitably encrypted password, a set of mailbox sites, and other information that is associated with this system user.

`/com/dec/src/staff` is a group of global names. Group objects are provided by BOB's name service to implement things like distribution lists, access control lists, and sets of servers. `/com/dec/src/bin` is a file system volume. Note that this object is a junction to the BOB file service. The figure does not show the content of this junction object, but it contains a group naming the set of servers implementing this file volume and rules for choosing which one to use, e.g., the first that responds. To look up the name `/com/dec/src/bin/ls`, for example, the operating system on a client machine traverses the path `/com/dec/src/bin` using the name service. The result at that point is the content of the junction object,



com
dec
src
. . .
domain
data for SRC management domain
adb
Principal=(Password=..., Mailbox=..., etc)
bwl
Principal=(Password=..., Mailbox=..., etc)
mds
Principal=(Password=..., Mailbox=..., etc)
. . .
other principals registered at SRC
staff
Group=(/com/dec/src/adb, /com/dec/src/bwl, etc)
. . .
other groups at SRC
computers
C1
Computer=(Address=16.1.0.0, HWInfo=..., Bootfile=. . ., Bootdata=. . ., etc.
C2
Computer=(Address=16.1.0.1, HWInfo=...,
. . .
other computers
backmap
16.1.0.0
Path=/com/dec/src/computers/C1
16.1.0.1
Path=/com/dec/src/computers/C2
. . .
IP addresses of other computers
bin
Volume=(junction to file service)
udir
Volume=(junction to file service)
. . .
other volumes
. . .
other SRC objects

Figure 1.1. Global name-space hierarchy in BOB

which then allows the client to contact a suitable file server to complete the lookup.

The local management autonomy provided by hierarchic names allows system implementors and administrators to build and use their systems without waiting for a planet-wide agreement to be reached about the structure of the first few levels of the global hierarchy. A local hierarchic name space can be constructed that is sufficient to the local need, treating the local root as a global root. Later, this local name space can be incorporated as a subtree in a larger name space. Using a variable to provide the name of the operational root (set to NIL at first) will ease the transition to the larger name space (and shorten the names that people actually use). Another technique is to initially name the local root with an identifier that is unlikely to appear in any future global root; then symbolic links in the global root, or special case code in the local name resolution machinery, can ease the transition.

1.4.2 Access model

Global access means that a program can run anywhere in BOB (on a computer and operating system compatible with the program binary) and get the same result, although the performance may vary depending on the machine chosen. Thus, a program can be executed either on the user's workstation or on a fast cycle server in the machine room, while still accessing the same user files through the same names.

Achieving global access requires allowing all elements of the computing environment of a program to be remote from the computer where the program is executing. All services and objects required for a program to run need to be available to a program executing anywhere in the distributed system. For a particular user, 'anywhere' includes at least:

- on the user's own workstation;
- on public workstations or compute servers in the user's domain;
- on public workstations in another domain on the user's LAN;
- on public workstations across a low-bandwidth WAN.

Performance in the first three cases should be similar. Performance in the fourth case is fundamentally limited by the WAN characteristics, although use of caching can make the difference small in many cases.

In BOB, global naming and standard services exported via a uniform RPC mechanism provide the keys to achieving global access. All BOB services accept global names for the objects on which they operate. All BOB services are available to remote clients. Thus, any object whose global name is known can be accessed remotely.

In addition, programs must access their environments only by using the global names of objects. This last step will require a thorough examination of the computing environment provided by each existing operating system to identify all the ways in which programs can access their environment. For each way identified a mechanism must be designed to provide the global name of the desired object. For example, in UNIX systems that operate as part of BOB, the identities of the file system root directory, working directory, and the `/tmp` directory of a process must be specified by global names. Altering VMS, UNIX, and other operating systems to accept global names everywhere will be a major undertaking.

Another aspect of global access is making sure that BOB services have operation semantics that are location transparent. Without location transparency in the services it uses, a program will not get the same result when it runs on different computers. A service that allows read / write sharing of object state must provide a data coherence model. The model allows client programs to maintain correct object state and to behave in a manner that does not surprise users, no matter where clients and servers execute. Depending on the nature of the service, it is possible to trade off performance, availability, scale, and coherence.

In the name service, for example, it is appropriate to increase performance, availability, and scalability at the expense of coherence. A client update to the name service database can be made by contacting any server. After the client operation has completed, the server propagates the update to object replicas at other servers. Until propagation completes, different clients can read different values for the object. This lack of coherence produces several advantages: it increases performance by limiting the client's wait for update completion; it increases availability by allowing a client to perform an update when just one server is accessible; and it increases scale by making propagation latency separate from the visible latency of the client update operation. For the objects that the name service will store, this lack of coherence is deemed acceptable given the benefits it produces. The data coherence model for the name service defines the loose coherence invariants that programmers can depend upon, thereby meeting the requirement of a coherence model that is insensitive to client and server location.

On the other hand, the BOB file service needs to provide consistent write sharing, even at some cost in performance, scale, and availability. Many programs and users are accustomed to using the file system as a communication channel between programs. For example, a programmer may save a source file for a module from an editor and then trigger a recompilation and relinking on a remote cycle server. He will be annoyed if the program is rebuilt with an old version of the module because the cycle server retrieved an old, cached version of the file. File read / write coherence is also important among elements of distributed computations running, say, on multiple computers on the same LAN. The file system coherence model must cover file names and attributes as well as file data.

There is diversity of opinion among researchers about the best consistency model for a general-purpose distributed file system. Some feel that an open / close consistency model provides the best tradeoff. With this model changes made by one client are not propagated until that client closes the file and others (re)open it. Others feel that byte-level write sharing is feasible and desirable. With this model clients share the file as though all were accessing it in the same local memory. Successful systems have been built using variants of both models. BOB can be successful based on either model.

1.4.3 Security model

Security is based on three notions:

- **Authentication** — for every request to do an operation, the name of the user or computer system making the request is known reliably. The source of a request is called a *principal*.
- **Access control** — for every resource (computer, printer, file, database, etc.) and every operation on that resource (read, write, delete, etc.), it is possible to specify the names of the principals allowed to do that operation on that resource. Every request for an operation is checked to ensure that its principal is allowed to do that operation.
- **Auditing** — every access to a resource can be logged if desired, as can the evidence used to authenticate every request. If trouble comes up, there is a record of exactly what happened.

To authenticate a request as coming from a particular principal, the system must determine that the principal originated the request, and that it was not modified on the way to its destination. We do the latter by establishing a *secure channel* between the system that originates the request and the one that carries it out. Practical security in a distributed system requires encryption to secure the communication channels. The encryption must not slow down communication, since in general it is too hard to be sure that a particular message does not need to be encrypted. So the security architecture should include methods of doing encryption and decryption on the fly, as data flows from computers into the network and back.

To determine who originated a request, it is necessary to know who is on the other end of the secure channel. Usually this is done by having the principal at the other end demonstrate that it knows some secret (such as a password), and then finding out in a reliable way the name of the principal that knows that secret. BOB's security architecture needs to specify how to do both these things. It is best if a principal can show that it knows the secret without giving it away, since otherwise the system can later impersonate the principal. Password-based schemes reveal the secret, but schemes based on encryption do not.

It is desirable to authenticate a user by his possession of a device which knows his secret and can demonstrate this by encryption. Such a device is called a *smart card*. An inferior alternative is for the user to type his password to a trusted agent. To authenticate a computer system, we need to be sure that it has been properly loaded with a good operating system image; BOB must specify methods to ensure this.

Security depends on naming, since access control identifies the principals that are allowed access by name. Practical security also depends on being able to have groups of principals e.g., the Executive Committee, or the system administrators for the cluster named 'Star'. Both these facilities must be provided by the name service. To ensure that the names and groups are defined reliably, digital signatures are used to certify information in the name service; the signatures are generated by a special *certification authority* which is engineered for high reliability and kept off-line, perhaps in a safe, when its services are not needed. Authentication depends only on the smallest sub-tree of the full naming tree that includes both the requesting principal and the resource; certification authorities that are more remote are assumed to be less trusted.

Security also depends on time. Authentication, access control, and secure channels require correct timestamps and clocks. The time service must distribute time securely.

1.4.4 Management model

System management is the adjustment of system state by a human manager. Management is needed when satisfactory algorithmic adjustments cannot be provided — when human judgement is required. The problem in a large-scale distributed system is to provide each system manager with the means to monitor and adjust a fairly large collection of different types of geographically distributed components. Of the five persuasive properties of BOB, global management is the one we understand least well how to achieve. The facilities, however, are likely to be structured along the following lines.

The BOB management model is based on the concept of a *domain*. Every component in a distributed system is assigned to a domain. (A component is a piece of equipment or a piece of management-relevant object state.) Each domain has a responsible system manager. In the simple version of the model (that is probably adequate for most, perhaps all, systems) domains are disjoint and managers are disjoint, although more complex arrangements are possible, e.g. overlapping domains, a hierarchy of managers. Ideally a domain would not depend on any other domains for its correct operation.

There needs to be quite a bit of flexibility available for defining domains, as different arrangements will be effective in different installations. Example domains include:

- components used by a group of people with common goals;
- components that a group of users expects to find working;
- largest pile of components under one system manager;
- arbitrary pile of components that is not too big.

As a practical matter, customers will require guidelines for defining effective management domains.

BOB requires that each component define and export a management interface, using RPC if possible. Each component is managed via calls to this interface from interactive tools run by human managers. Some requirements for the management interface of a component are:

- **Remote access** — The management interface provides remote access to all management functions. Local error logs are maintained that can be read from the management interface. A secure channel is provided from management tools to the interface and the operations are subject to authentication and access control. No running around by the manager is required.
- **Program interface** — Each component's management interface is designed to be driven by a program, not a person. Actual invocation of management functions is by RPC calls from management tools. This allows a manager to do a lot with a little typing. A good management interface provides end-to-end checks to verify successful completion of a series of complex actions and provides operations that are independent of initial component state to make it easier to achieve the desired final state.
- **Relevance** — The management interface operates only on management-relevant state. In places where the flexibility is useful rather than just confusing, the management interface permits decentralized management by individual users.
- **Uniformity** — Different kinds of components should strive for uniformity in their management interfaces. This allows a single manager to retain intellectual control of a larger number of kinds of components.

The management interfaces and tools make it practical for one person to manage large domains. An interactive management tool can invoke the management interfaces of all components in a domain. It provides suitable ways to display and correlate the data, and to change the management-relevant state of components. Management tools are capable of making the same state change in a large set of similar components in a domain via iterative calls. To provide the flexibility to invent new management operations, some management tools support the construction of programs that call the management interfaces of domain components.

1.4.5 Availability Model

To achieve high availability of a service there must be multiple servers for that service. If these servers are structured to fail independently, then any desired degree of availability can be achieved by adjusting the degree of replication.

The most practical scheme for replication of the services in BOB is *primary/backup replication*, in which a client uses one server at a time and servers are arranged (as far as possible) to fail in a benign way, say by stopping. The alternative method, called *active replication*, has the client perform each operation at several servers. Active replication uses more resources than primary / backup but has no failover delays and can tolerate arbitrary failure behavior by servers.

To see how primary / backup works, recall from the naming model discussion that the object that represents a service includes a set of servers and some rules for choosing one. If the chosen server (the primary) fails, then the client can failover to another server (the backup) and repeat the operation. The client assumes failure if a response to an operation does not occur within a timeout period. The timeout should be as short as possible so that the latency of an operation that fails over is comparable to the usual latency. In practice, setting good timeouts is hard and fixed timeouts may not be adequate. If clients can do operations that change long-term state then the primary server must keep the backup servers up-to-date.

To achieve transparent failover from the point of view of client programs, knowledge of the multiple servers should be encapsulated in an agent on the client computer. In this chapter we refer to the agent as a *clerk*. The clerk can export a logically centralized, logically local service to the client program, even when the underlying service implementation is distributed, replicated, and remote. The clerk software can have many different structural relationships to its client. In simple cases it can be runtime libraries loaded into the client address space and be invoked with local procedure calls. Or it may operate in a separate address space on the same machine as the client and be invoked by same-machine IPC, same-machine RPC, or callbacks from the operating system. Or it can be in the operating system itself.

The clerk interface need not be the same as the server interface. Indeed, the server interface usually will be significantly more complex. In addition to implementing server selection and failover, a clerk may provide caching and write behind to improve performance, and can implement aggregate operations that read-modify-write server state. As simple examples of caching, a name service clerk might remember the results of recently looked up names and maintain open connections to frequently used name servers, or a file service clerk might cache the results of recent file directory and file data reads. Write-behind allows a clerk to batch several updates as a single server operation which can be done asynchronously, thus reducing the latency of operations at the clerk interface. Implementing a client's read-modify-write operation might require the clerk to use complex retry strategies involving several server operations when a failover occurs at some intermediate step.

As an example of how a clerk masks the existence of multiple servers, consider the actions involved in listing the contents of `/com/dec/src/udir/bwl/-Mail/inbox`, a BOB file system directory. The client program presents the entire path name to the file service clerk. The clerk locates a name server that stores the root directory and presents the complete name. That server may store the directories down to, say, `/com/dec`. The directory entry for `/com/dec/src` will indicate another set of servers. So the first lookup operation will return the new server set and the remaining unresolved path name. The clerk will then contact a server in the new set and present the unresolved path `src/udir/bwl/Mail/inbox`. This server discovers that `src/udir` is a junction to a file system volume, so returns the junction information and the unresolved path name `udir/bwl/Mail/inbox`. Finally, the clerk uses the junction information to contact a file server, which in this example actually stores the target directory and responds with the directory contents. What looks like a single operation to the client program actually involves RPCs to three different servers by the clerk.

This example of a name resolution would be completed with fewer or no operations at remote servers if the clerk has a cache that already contains the necessary information. In practice, caches are part of most clerk implementations and most operations are substantially speeded by the presence of cached data.

Other issues arise when implementing a high-availability service with long term state. The servers must cooperate to maintain consistent state among themselves, so a backup can take over reasonably quickly. Problems to be solved include arranging that no writes get lost during failover from one server to another and that a server that has been down can recover the current state when it is restarted. Combining these requirements with caching and write-behind to obtain good performance, without sacrificing consistent sharing, can make implementing a highly available service quite challenging.

1.5 Conclusion

This chapter covers the inherent strengths of centralized and networked computing systems. It outlines the structure and properties of BOB, a state-of-the-art distributed computing base for supporting general-purpose computing. This system combines the best features of centralized and networked systems with recent advances in security and availability to produce a powerful, cost-effective, easy-to-use computing environment.

Getting systems like BOB into widespread use will be hard. Given the state-of-the-art in distributed systems technology, building a prototype for proof of concept is certainly feasible. But the only practical method for getting widespread use of systems with these properties is to figure out ways to approach the goal by making incremental improvements to existing networked systems. This requires producing a sequence of useful, palatable changes that lead all the way to the goal.

1.6 Acknowledgements

The material in this chapter was jointly developed by Michael Schroeder, Butler Lampson, and Andrew Birrell. The ideas explored here aggregate many years of experience of the designers, builders, and users of distributed systems. Colleagues at SRC and fellow authors of this book have provided useful suggestions on the presentation.

Chapter 11

A Case Study: Automatic Reconfiguration in Autonet

Michael D. Schroeder

This chapter reprints the paper 'Automatic Reconfiguration in Autonet' from the *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Operating Systems Review **25**(5), October 1991, pp 183–197, as a case study in distributed computing. Local area networks (LANs) are one of the reasons that distributed systems are so popular. Inexpensive networking with Ethernet and PC LANs has enabled workstations and PC's to communicate easily and quickly, allowing networked systems to become a prevalent way to compute. In this case study, however, we are not interested in a distributed system that uses the network, but a distributed system that is inside it.

Autonet is a switch-based LAN. In an Autonet installation a set of multiported switches is interconnected in an arbitrary pattern using point-to-point links, as illustrated in Figure 11.1. Additional links connect each host computer to two switches. Packets from a source host contain the address of the destination host and are forwarded through the network from switch to switch along a predetermined route. The distributed system in Autonet is the automatic reconfiguration mechanism, implemented by the switches, that monitors the configuration and adjusts the packet routes to provide maximum connectivity and to make effective use of the available switches and links.

The purpose of Autonet's reconfiguration mechanism is to provide high availability by exploiting redundancy in the physical configuration. When any switch detects a change in the set of neighboring links or switches that are working it triggers a distributed algorithm with which the switches discover the new topology and recalculate the best routes. This reconfiguration is done fast enough that it does not disrupt high-level communication protocols. Thus, Autonet continues to provide communication service to the hosts that are connected by the working components. The paper gives a glimpse into the surprisingly subtle mechanisms required to exploit redundancy and independent failure so that the resulting sys-

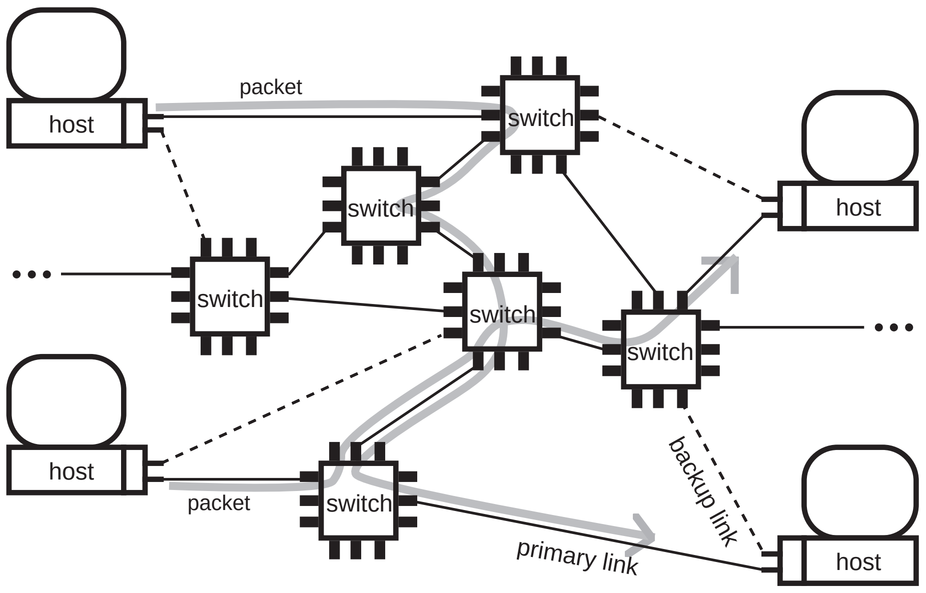


Figure 11.1. Portion of an Autonet showing two packets in transit.

tem surpasses the reliability of its components.

The monitoring, topology acquisition and topology distribution mechanisms described in the paper are a complex example of primary / backup replication. The service that is being replicated is communication between each pair of switches in the network (and thus communication between the hosts connected to those switches). Autonet also contains a much simpler example of primary / backup replication that the paper does not describe. As shown in Figure 11.1, a host attached to an Autonet can have links to two different switches. Low level communication software in the host functions as a clerk module for the communication service provided by the network. This clerk sends and arranges to receive all packets through one of the links (the primary) until there is evidence that it is not working. Evidence comes from hardware link status indicators as well as periodic packet exchanges with the nearest switch. If the active link stops working the clerk fails-over to the other link (the backup). Thus, a host loses network service only if both of its connections fail. The combination of replicated host connections and automatic reconfiguration of the switch-to-switch forwarding routes produces a high-availability LAN.

Given the close attention paid to availability in Autonet it is reasonable to ask whether the network is viewed by its customers as being continuously available. To answer this question we should first distinguish between reliability and availability. Reliability in the case of a LAN refers to the probability that a packet is delivered correctly to its destination. In Autonet, as in any LAN, each packet is delivered correctly with high probability, but not with probability one. Network reconfigurations and environmentally triggered transmission errors on links occasionally corrupt or lose packets. So customers must (and do) take these reliability characteristics into account when they design communication protocols. The typical method for dealing with occasional packet loss is to use acknowledgments, timeouts and retransmissions. Availability, on the other hand, concerns outages that are long enough to disrupt protocols or to be noticed by users.

In service use our Autonet is more available than the other (product) LANs in the building, but it has not been continuously available. Autonet outages have come from two sources: environment and software. The major environmental cause of outages has been power failures, usually building-wide, that affect enough switches to overcome the redundancy in the installation. A second order environmental cause has been disruptive experimentation with the service network. The power failures, however, usually take out all the hosts along with the network, so are not perceived as network outages. And in a service system serious about availability managers would carefully control experimentation.

Of more concern are the software-caused outages – bugs and design flaws in the switch software that disrupt service for noticeable periods of time. Software flaws that occasionally crash a single switch (an independent failure) do not cause network outages: the other switches remove the crashed one from the network and continue to provide service; the crashed switch later reboots itself and rejoins the network. The flaws of concern are distributed bugs that take out most or all switches (a coordinated failure). Will such distributed bugs disappear as the network matures or will they always be a source of outages? My opinion is that a mature system would have a very low probability of an outage due to a software failure. The Autonet approach to high availability can produce a network that will run for years without such an outage. It is imprudent, however, to ever assume it cannot happen, even in a mature system. Something may change about the way the network is managed or used that will discover a new bug.

A third potential cause for apparent outages is unnoticed increases in load. As use grows some links may become overloaded. The result can be performance so bad that it appears to be an outage to some customers. The usual scenario is for the increasing load slowly to use up the capacity of the redundant links so that eventually a failure produces a diminished configuration that is paralyzed by the load. The only remedy for this problem is tools for monitoring network load, periodic management attention to analyzing the load measurements, and addition of capacity as necessary. Whenever spare capacity is provided for availability reasons, it is essential to make sure that it stays spare and does not get used up as load grows.

Automatic Reconfiguration in Autonet

Thomas L. Rodeheffer and Michael D. Schroeder

Abstract

Autonet is a switch-based local area network using 100 Mbit/s full-duplex point-to-point links. Crossbar switches are interconnected to other switches and to host controllers in an arbitrary pattern. Switch hardware uses the destination address in each packet to determine the proper outgoing link for the next step in the path from source to destination. Autonet automatically recalculates these forwarding paths in response to failures and additions of network components. This automatic reconfiguration allows the network to continue normal operation without need of human intervention. Reconfiguration occurs quickly enough that higher-level protocols are not disrupted. This paper describes the fault monitoring and topology acquisition mechanisms that are central to automatic reconfiguration in Autonet.

11.1 Introduction

Autonet is a switch-based local area network. In an Autonet, 12-by-12 crossbar switches are interconnected to other switches and to host controllers with 100 Mbit/s full-duplex links in an arbitrary pattern. In normal operation each packet follows a precomputed link-to-link path from source to destination. At each switch, hardware uses the destination address in each packet as the lookup index in a forwarding table to determine the proper outgoing link for the next step in the path. An earlier paper (Schroeder *et al.* [1990]) provides an overview of the Autonet design. In the present paper we concentrate on automatic reconfiguration in Autonet.

Automatic operation and high availability are important objectives for Autonet. Our goal was to make Autonet look to host communications software like a fast, high-capacity Ethernet segment that never failed permanently. To provide automatic operation and high availability an Autonet automatically reconfigures itself to use the available topology of switches and links. A processor in each switch monitors the directly connected links and neighboring switches. Whenever this monitor notices a change in what is working (either additions or removals), it triggers a distributed algorithm on all switch processors that determines and distributes the new network topology to all switches. Once each switch knows the new topology, it recalculates routing information and reloads its forwarding table to permit operation with the new topology. This automatic reconfiguration is fast enough that high level communication protocols are not permanently disrupted, even though client packets may be lost while reconfiguration is in process.

When an Autonet is installed with a redundant topology, automatic reconfiguration allows it to continue to provide full interconnection of all hosts as components fail or are removed from service. If there are so many failures that connectivity is lost, the Autonet will partition, but service will continue within each connected portion. When components are repaired, or the topology is extended with new switches or links, automatic reconfiguration incorporates the added components in the operational network.

Autonet has been the service LAN for our research center since February of 1990, with 31 switches providing service to over 100 hosts. Operational experience has allowed (forced) us to improve the sensitivity, stability, and performance of the automatic reconfiguration mechanisms. So this paper, in addition to giving a more detailed description of reconfiguration than previously published, also highlights the important changes that were dictated by our experience.

The paper is organized as follows. Section 11.2 compares Autonet with other networks with automatic reconfiguration. Section 11.3 gives the overall structure of reconfiguration in Autonet. Section 11.4 discusses monitoring and Section 11.5 topology acquisition. Section 11.6 presents conclusions.

11.2 Reconfiguration in Other Networks

The standard example of automatic reconfiguration in a computer network is the ARPANET (McQuillan, Richer and Rosen [1980]; McQuillan and Walden [1977]). The principle differences between ARPANET and Autonet relate to the fact that ARPANET is designed as a wide-area, moderate-speed network while Autonet is designed as a local-area, high-speed network. The ARPANET performs store-and-forward routing based on topology descriptions maintained at each switch (IMP), and tolerates temporary forwarding loops by discarding packets if necessary. Each switch regularly broadcasts updates of the status of its local links.

The Autonet switch hardware processes packets first-come-first-served from each link and uses cut-through in order to decrease the expected delay through the switch. This design was chosen because it provided the best light-load performance for the simplest hardware. However as a consequence, transient forwarding loops might result in deadlock and thus cannot be tolerated. (We have efficient means neither to detect a deadlock nor to clear one.) We took the simplest approach of rapidly recalculating the entire topology whenever it changes and expunging all old forwarding tables before installing any new ones. This global-recalculation design is simpler than incremental approaches and represents an appropriate engineering tradeoff for a moderate-sized network of several dozen switches.

Another network that provides automatic topology maintenance is PARIS (Awerbuch *et al.* [1990]). Like ARPANET, PARIS uses the strategy of maintaining a topology description at each switch via regular broadcasts of local link status updates. PARIS is designed more as a fast connection network than a packet switching network. Packets travel on explicit source routes that are determined at connection setup by examining a description of the current topology. Topology changes have no effect on existing connections, except that a link failure kills all of the connections using that link. Reliable and very high bandwidth link update broadcasts are provided by hardware flooding over a software-managed spanning tree. The software tree management is very careful not to introduce inconsistencies into the tree. In contrast to PARIS, Autonet routes each new packet independently and thus automatically maintains ongoing conversations by routing around link failures and exploiting link recoveries.

Bridged Ethernet (Perlman [1985]) is another network that provides automatic reconfiguration. The principle difference from Autonet is that a bridged Ethernet supports multiple-access links with no way to distinguish forwarded packets from originals. A bridged Ethernet carefully maintains a loop-free forwarding tree so that each bridge can deduce what to do with each packet. Although the time constants required to maintain consistency in the forwarding tree are on the order of several seconds, a bridged Ethernet does eventually adapt to any topology change. In contrast, Autonet has an implicit addressing structure induced by its point-to-point links. An arriving packet is always known to be intended for the recipient, at least as an intermediate hop. We also designed a packet encapsulation using network-assigned destination addresses, in order to make forwarding easier (much like Cypress (Comer and Narten [1988])). As a consequence, Autonet uses more forwarding paths and reconfigures much faster than a bridged Ethernet.

11.3 Overall Structure of Automatic Reconfiguration Mechanism

Automatic reconfiguration in Autonet involves three main tasks: monitoring, topology acquisition and routing. Monitoring involves watching the neighborhood of each switch to determine when the network topology changes. Topology acquisition involves collecting and distributing the description of the network topology. Routing involves recalculating the forwarding table at each switch.

Monitoring determines which links are useful for carrying client packets from one switch to another. From the point of view of reconfiguration, a link is useful if and only if it has an acceptable error rate in both directions, the nodes at each end are distinct, operational switches and each switch knows the identity of the other. (A switch is identified by a 48-bit unique identifier stored in a ROM.) Topology acquisition and route recalculation is triggered whenever the set of useful links changes. Of course, host-to-switch links also carry client packets, but changes in the state of such links never trigger topology acquisition and route recalculations. At most, changes in the host links to a particular switch cause locally calculated changes in that switch's forwarding table. So, from the point of view of network reconfiguration, we largely ignore such links.

Monitoring guarantees that topology acquisition (and client) packets will not travel over a link unless both switches agree the link is useful. Because there are two switches involved there will always be transient disagreement whenever the link is changing state, but the monitoring task makes the period of disagreement as brief as possible. A monitor runs independently for each link in each switch and is always active. It will trigger topology acquisition whenever its link changes from useful to not useful or vice versa. The monitors guarantee that, eventually, links remain stable in one state or the other long enough that topology acquisition and routing can finish.

Topology acquisition is responsible for discovering the network topology and delivering a description of it to every switch that is currently part of the network. This task runs in an artificial environment in which changes in link state do not occur. When two switches disagree about the state of a link, the task does not complete. The artificial environment is implemented on top of the monitoring layer by means of an epoch mechanism: any change or inconsistency triggers a new epoch corresponding to a new stable environment. Topology acquisition is a distributed computation that spreads to all switches from the one where a link monitor triggered it.

Routing, the final task of reconfiguration, uses the topology description to compute the forwarding tables for each switch. Because each switch knows the entire topology, each can calculate its own forwarding table. In this paper we are not concerned with the algorithm for constructing the forwarding tables. During topology acquisition and routing, the switch discards client packets. Once the forwarding table has been recalculated, a switch is able to forward any client packets it receives. The reason for discarding client packets during reconfiguration is to prevent deadlock.

The remainder of the paper concentrates on monitoring and topology acquisition. In considering these topics in more detail we can model the Autonet as a collection of nodes (switches) with numbered ports. Nodes may be interconnected in an arbitrary pattern by full-duplex, port-to-port links. Each node is a computer that can send and receive packets on each attached link that works. Each node has a predetermined unique identifier. From now on we will largely ignore links to hosts, the hosts themselves, forwarding of host packets and even the forwarding tables in the switches.

The neighborhood of a node N is the set of all useful switch-to-switch links that have N as one endpoint. The neighborhood of a set of nodes is the union of the neighborhoods of the members. Autonet reconfiguration can be characterized in terms of these neighborhoods. The monitoring task on node N is responsible for knowing the current neighborhood of N and for initiating a topology task whenever the neighborhood changes. Topology acquisition works by building a spanning tree, merging neighborhoods of larger and larger subtrees until the root has the neighborhood of the entire graph and then flooding the topology down the spanning tree to all nodes.

We are now ready to describe monitoring and topology acquisition in more detail. In this discussion, *packet* and *signal* refer to information passing between separate nodes over a connecting link. Packets are regular data packets whereas signals are transported in link protocols below the level of packet traffic. Message refers to information passing between software components in a single node.

11.4 Monitoring

The monitoring task imposes a model that allows only two types of changes in a node's neighborhood: link failure, which removes a connection from the network topology and link recovery, which adds a connection. All changes in network interconnection result in some combination of these two types of neighborhood change. For example, if a technician powers off a switch, all of the adjacent nodes see link failures on their links to the dead node.

The monitoring task responds rapidly to link failures and less rapidly to link recoveries. Link failure, especially abrupt failure, must be detected and reported quickly, because failure can disrupt ongoing client communication. It is not so urgent to rush back into service a link that recently gave problems. Although it is true that link recovery might heal a network partition or increase network capacity, repairing or adding a link usually takes quite a bit of time, so clients usually will not notice a small additional delay. Many networks, for example, the ARPANET, have a delay before placing links back in service (Heart *et al.* [1970]). By delaying longer as a link proves its reliability, we achieve network stability despite intermittent failures.

A useful link is one that allows bi-directional packet transfer with acceptably low error rates between two distinct nodes. The only way this condition can be verified, of course, is for the two nodes to periodically exchange packets and this is what the monitoring task does. This strategy has the strength that it is an end-to-end check (Saltzer, Reed and Clark [1984]). It has the disadvantage that failure detection may not be very prompt, because it depends on a time-out whose minimum value is bounded by processing overhead. In order to give prompt detection of expected modes of link failure, the monitoring task also treats certain hardware error status conditions as indicating failure. For example, if more than three link coding violations are detected in a ten millisecond interval, the monitoring task immediately assumes that the link has failed.

The monitoring task is organized as two layers: a transmission layer and a connectivity layer. The transmission layer deals with proper transmission and reception of data on the link as seen by the hardware. It makes sure that problems on this link do not interfere with other links and it responds promptly to expected modes of link failure. The connectivity layer, which rests on top of the transmission layer, deals with exchanging packets with the remote node and agreeing on the state of the link. Both of these layers make use of a method for defending against intermittent operation called the skeptic. We describe the skeptic and then the two layers of the monitoring task in detail.

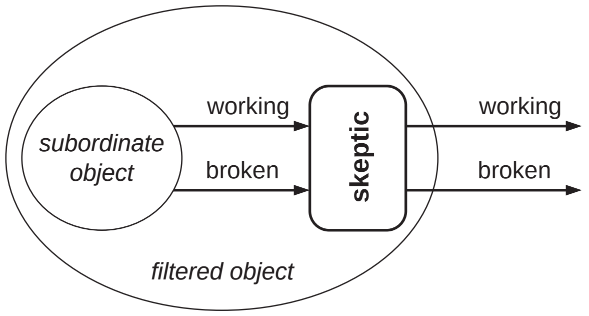


Figure 11.2. Concept of the skeptic.

11.4.1 The Skeptic

The skeptic limits the failure rate of a link by delaying its recovery if it has a bad history. Without the skeptic, an intermittent link could cause an unlimited amount of disruption: we are especially concerned with limiting the frequency of reconfigurations. There are four requirements in the design of the skeptic: 1) A link with a good history must be allowed to fail and recover several times without significant penalty. 2) In the worst case, a link's average long-term failure rate must not be allowed to exceed some low rate. 3) Common behaviors shown by bad links should result in exceedingly low average long-term failure rates. 4) A link that stops being bad must eventually be forgiven its bad history. (Note: failure rate here means the number of transitions a link undergoes from working to broken per unit of time.)

Requirement 3 distinguishes the skeptic from typical fault isolation and forgiveness methods such as the auto restart mechanism in Hydra (Wulf, Levin and Harbison [1981]). The typical method to meet requirements 1, 2 and 4 sets a quota of say, ten failures per hour and refuses to recover any link that is over quota. We have observed a common pattern of intermittent behavior in which a link fails again soon after being recovered, in spite of its passing all diagnostics performed in the interim. With the quota method, this pattern would produce a long-term average failure rate of ten failures per hour. This kind of error pattern may not be uncommon, for example Lin and Siewiorek observed a clustering pattern of transient errors in the VICE file system (Lin and Siewiorek [1990]).

The skeptic can be used with any object whose status may change intermittently: it provides a 'filtered object' whose rate of status change is limited. As seen by the skeptic, an object is an abstraction that emits a series of messages, each of which says either 'working' or 'broken'. The skeptic in turn sends out a filtered version of these messages to the next higher level of abstraction. See Figure 11.2.

The skeptic is a state machine with auxiliary variables, timers, and policy parameters. See Figure 11.3. Dead state means that the subordinate object is broken, wait state means that the object is working but the skeptic is delaying for a while before passing on that information, and good state means that the object is working and the skeptic has concurred.

Three of the state transitions are caused by messages from the subordinate object.

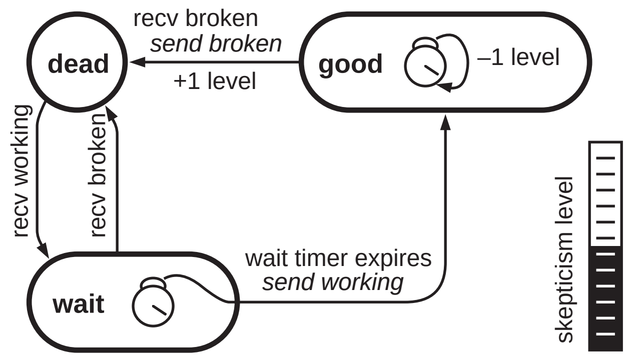


Figure 11.3. Internals of the skeptic.

When the skeptic is in wait state or good state and it receives a 'broken' message, it moves to dead state. When the skeptic is in dead state and it receives a 'working' message, it moves to wait state. Otherwise the messages have no effect. The only other transition in the state machine happens when the wait timer expires; in this case the skeptic moves from wait state to good state.

When the skeptic moves from wait to good, it sends a 'working' message to the next higher level of abstraction. When the skeptic moves from good to dead, it sends a 'broken' message. Hence, in the filtered view provided by the skeptic, the object appears to be working only when the skeptic is in good. If the subordinate object fails intermittently, the skeptic alternates between dead and wait without ever reaching good.

When the skeptic enters wait state, it sets and starts the wait timer. The duration set on this timer is calculated by a formula described below. If the skeptic returns to dead before the timer expires, the timer is stopped. Otherwise, when the timer expires, the skeptic moves to good. This is the only way the skeptic can get to good state.

The skeptic responds to intermittent failures by maintaining a level of skepticism about the subordinate object. The skepticism level is kept in an auxiliary variable. Every time the skeptic leaves good state it increments the level. The skepticism level is used in computing wtime, the duration set on the wait timer, according to the formula

$$wtime = wbase + wmult * 2^{level}$$

where wbase and wmult are policy parameters and level is the skepticism level. A policy parameter maxlevel establishes an upper limit on skepticism.

The skeptic forgives old failures by decrementing the skepticism level occasionally. When the skeptic enters good state, it sets and starts the good timer. When the good timer expires, the skeptic decrements the skepticism level and sets and starts the good timer again. The good timer is always running as long as the skeptic is in good state. When the skeptic leaves good state, the good timer is stopped. The formula used to compute gtime, the duration set on the good timer, is identical to the formula used for the wait timer, except that it uses different policy parameters, gbase and gmult. The skepticism level never decrements below zero.

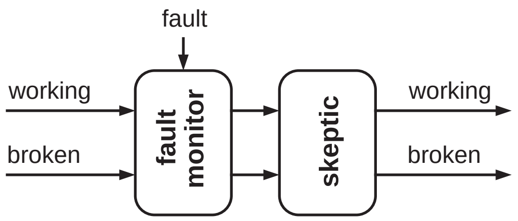


Figure 11.4. Skeptic with fault monitor.

Skeptics are used in both the transmission layer and the connectivity layer. Each of these layers has mechanisms to decide when a significant error has occurred. A significant error is called a fault. Faults feed into the skeptic though a mechanism called the fault monitor. See Figure 11.4. The fault monitor relays the state of the subordinate object to the skeptic. Whenever the fault monitor receives a 'fault' message and the subordinate object is working, the fault monitor presents an interruption to the skeptic by sending it 'broken' immediately followed by 'working'. This causes the skeptic to notice the fault and enter wait state. If the subordinate object was already broken, the fault monitor takes no action on a 'fault' message.

In the actual implementation, the fault monitor and the skeptic are combined as one unit. Procedure calls are used for the messages.

Figure 11.5 shows an example of how the skeptic timers work, using the transmission layer policy parameters. Observe that for low skepticism levels, say those below 10, the wait time is approximately constant at 5 seconds. For high levels, say those above 12, the wait time doubles with each additional level. The interplay between wbase and wmult establishes the crossover point between low and high levels of skepticism. Policy parameters for all skeptics are given in Table 1.

A few examples of how the transmission layer skeptic responds to common problems will illustrate its utility. One common link failure mode we have observed, especially in newly installed hardware, is that the link transceiver hardware continuously detects coding violations. In this case the transmission layer will declare a fault about once every 170 milliseconds. Because this is much less than the five

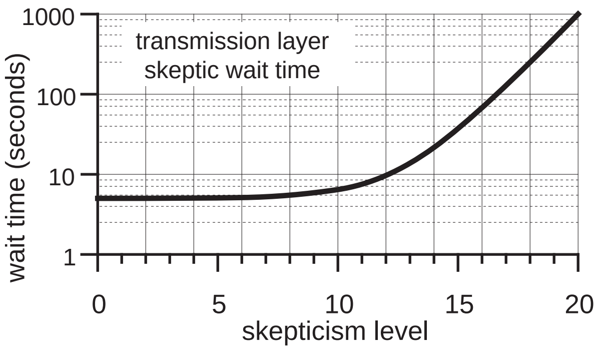


Figure 11.5. Skepticism level vs. wait time.

	transmission skeptic		connectivity skeptic	
wbase	5	sec	1	sec
wmult	0.001	sec	0.1	sec
gbase	600	sec	600	sec
gmult	0.01	sec	0.1	sec
maxlevel	20		20	

Table 11.1. Skeptic policy parameters.

second minimum wait time, the skeptic never lets the link recover. To higher levels of abstraction, the link appears permanently broken.

Another common link failure mode occurs when a technician screws in a link cable. As the metal components scrape past each other, the link transceiver hardware detects bursts of coding violations that the transmission layer quite reliably evaluates as faults. Unfortunately, the cable connectors we use are quite difficult to thread correctly, and often several tries and some wiggling are needed before the cable allows itself to be properly screwed in. Each additional wiggle tends to generate more faults. The five second minimum wait time in the skeptic causes all of these faults to be reflected as only one failure.

A third common failure mode occurs on marginal links. In our experience, the error rate on a marginal link is very data dependent: it is much higher when the link is carrying packets than when it is idle. This results in such a link failing soon after it recovers, but then having no further faults until it recovers again. The skepticism level on such a link increases over time. Eventually the transmission layer skepticism level reaches its maximum value of 20, at which point the wait time is about 17 minutes. If the link is part of the switch-to-switch topology, so that failures and recoveries cause network-wide reconfigurations, the connectivity layer skeptic gets involved and its policy parameters produce a maximum wait time of about 28 hours.

Now let us consider how the skeptic fulfills the design requirements. 1) A good history is represented by a low skepticism level. In this case, the skeptic delays a minimum time in wait state and consequently the filtered object recovers soon after the subordinate object recovers. 2) The worst case long-term average failure rate of the filtered object results when the skeptic spends the minimum time in good state required to forgive the lowest level of skepticism. The bound can be proved using a counting argument on the number of failures and observing that at sufficiently high skepticism levels the wait time exceeds the lowest level time to forgive. 3) A subordinate object that tends to fail again soon after the filtered object recovers will tend to increase the skepticism level. 4) If the subordinate object remains working, eventually all skepticism will be forgiven.

There is one more feature in the skeptic, which is that the duration set on the wait timer actually varies as a random fraction between one and two times the value calculated for wtime. This random variation causes different skeptics to disperse their wait timer expirations. If the network is running with several intermittent links, this randomness reduces the possibility of getting caught in some systematic pattern.

We chose the skeptic parameters as follows. The transmission layer skeptic deals with physical phenomena, so several of its parameters derive from maintenance needs. Five seconds is the shortest minimum wait time that will cover the process of screwing in a link cable. A technician often recables a host controller several times during testing, so we allow about eight levels before the increase in wait time becomes perceptible. Twenty minutes is the longest maximum wait time that a technician will bear to see if an attempted hardware repair has any effect on the system. Ten minutes seems like a reasonable interval for the minimum good time.

We expect problems in the connectivity layer to be unusual except when induced by the transmission layer, so we set its minimum wait time smaller, at one second and its crossover point lower, at level four. Because connectivity layer failures cause network-wide reconfigurations, the maximum wait time should be as long as possible. We chose 28 hours because we did not want the system to hold off much more than a day on its own authority.

Many networks contain a mechanism for discriminating against unreliable links. For example, PARIS (Awerbuch *et al.* [1990]) increments an (un)reliability counter with each link failure. The current value of a link's unreliability counter forms the most significant component of a link's weight, which is broadcast in regular link status updates. Connection setup and the tree manager shy away from links with high weight and thus unreliable links will tend not to get used unless necessary. The value in a link's unreliability counter decays over time so that information about old failures expires eventually (Awerbuch Private communication]). However, PARIS does not have a backoff strategy, so if the unreliable link is the only connection between two parts of the network, PARIS will suffer repeated topology changes rather than permit the network to remain partitioned.

Jacobson observes that a network closely approximates a linear system and speculates that consequently its stability may be ensured by adding exponential damping to its primary excitation (Jacobsen [1988]). This speculation supports the exponential increase in wait time at high skepticism levels.

11.4.2 Transmission Layer

The transmission layer contains a skeptic with fault monitor, three error detectors and a round-trip verifier. See Figure 11.6. The transmission layer watches the error indicators in the link hardware and determines if the link appears to be successful at sending and receiving data. It passes its conclusion up to the connectivity layer. The transmission layer does not care where the data might be going to or coming from – it is the responsibility of the connectivity layer to determine that. If the transmission layer determines that the link is broken, it sets the switch hardware to discard all incoming and outgoing packets, isolating the link. No packets can be sent or received over an isolated link. The reason for isolating a broken link is to prevent it from interfering with the rest of the network.

The fault monitor here at the lowest level of the system really has no subordinate object, so it is connected to a dummy object that is always working. 'Working'

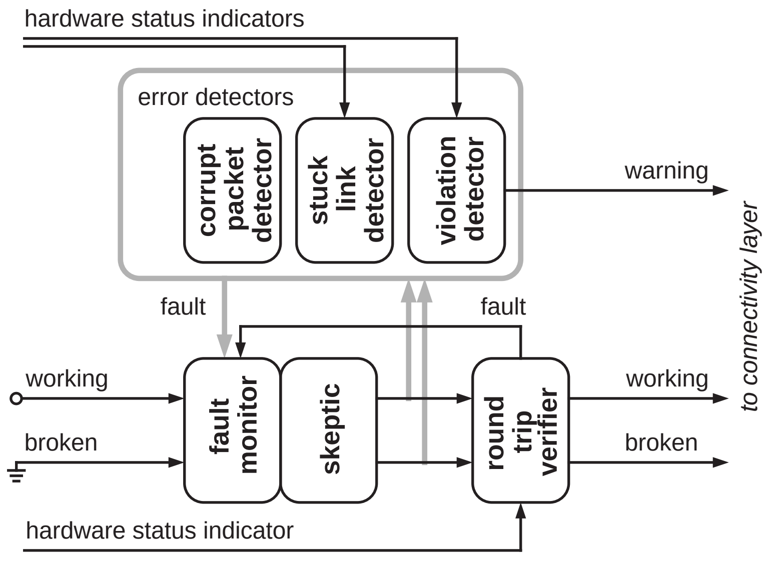


Figure 11.6. Diagram of the transmission layer.

and ‘broken’ are abstractions that are synthesized based on the hardware status indicators, as interpreted by the error detectors and round-trip verifier. The round-trip verifier filters the output of the skeptic by delaying ‘working’ messages until it believes that the transmission layer skeptic on the other end of the link also believes the link is working.

Each error detector analyzes and responds to a different type of error indication available in the switch hardware.

Corrupt packet detector

The corrupt packet detector examines all packets received by the switch control processor and declares a fault when CRC errors or impermissible packet lengths are seen too frequently. It is possible for packets to be corrupted without any detectable coding violations, when a data error happens inside the crossbar at some switch. Such an error is eventually detected as a CRC error at the packet’s ultimate destination. It would be better if each link verified the CRC of all incoming packets, but this feature was omitted from the hardware. We achieve some protection against corruption by checking all of the packets destined for the local switch control processor.

An isolated corrupt packet might be the result of a random glitch, so it should be forgiven. Corrupt packets become a significant error if they happen too frequently. The corrupt packet detector imposes a quota on how often it forgives corrupt packets by using a leaky bucket mechanism (Turner [1986]). Every time it encounters a corrupt packet, it puts a token in the bucket. One token leaks out of the bucket every ten minutes. Whenever adding a token to the bucket causes the bucket to hold more than five tokens, the corrupt packet detector declares a fault.

Because the transmission layer isolates a broken link so that it can neither send nor receive packets, no further corrupt packets will arrive from the link until the skeptic recovers it.

Stuck link detector

In Autonet, certain problems cause a link to become stuck in a state that prevents any data transmission. Typically this is due to some corruption of flow control commands. If a link has been trying to transmit a packet, but has made no progress for several milliseconds, something is wrong on the link. At this point it is necessary to dump the stuck packet and free up its resources.

This recovery is bound to be disruptive, although in our switches it is perhaps more disruptive than absolutely necessary, since our only mechanism for dumping a packet is to reinitialize the entire switch. This destroys all packets in the switch. Fortunately, reinitializing only takes about ten microseconds. Isolating a broken link removes the opportunity for it to cause further switch reinitializations.

Although a stuck link should not happen in normal operation, links can appear to be stuck as a result of mistransmission of a single flow-control or packet framing command code. Thus the stuck link detector must be willing to forgive an isolated occurrence. The detector samples its hardware indicators every 100 milliseconds and, if the link is stuck, responds by reinitializing the switch. The stuck link detector imposes a quota on how often it forgives by using a leaky bucket mechanism to declare faults, exactly like the corrupt packet detector.

Violation detector

The violation detector analyzes and responds to coding and format violations received on the link. A coding violation usually means that the link receiver heard a piece of static on the line. For example, coding violations result from connecting or disconnecting the link cable, from a cable that is too long for good transmission, or from a nearby heavy-duty electric motor. Although connecting or disconnecting a link generates a burst of coding violations tens of milliseconds long, even the best links in our system pick up one or two isolated coding violations per week. A format violation means that the link receiver did not hear proper packet framing or flow-control where it expected. Static can cause isolated format violations, as can occasional activity such as reinitializing the switch. Hence a burst of violations is a significant error, but isolated violations should be ignored.

The violation detector samples link hardware status registers once every 1.3 milliseconds and accumulates the results for a block of 128 samples. At the end of each block, the violation detector checks the number of violations and, if there are too many it declares a fault. The permitted number of violations depends on whether the skeptic says the link is working or broken, which is why Figure 11.6 shows the error detectors receiving the ‘working’ and ‘broken’ messages from the skeptic. If the link is working, three errors are permitted in a block, but if the link is broken no errors are permitted. The more strict rule for broken links insures that no link will recover unless it can pass the entire skeptic recovery time without a single violation, while occasional violations on working links are ignored.

If a broken link continues to have violations, the violation detector continues to declare a fault at the end of each block. The transmission layer skeptic always spends at least five seconds in wait state, so it will keep believing that the link is broken.

In order promptly to detect the bursts of violations that result from the anticipated activity of plugging and unplugging link cables, the violation detector examines subblocks of eight samples. If a subblock contains more than three violations, the violation detector immediately declares a fault. In order to eliminate the processing overhead of declaring faults every subblock, subblock checking applies only to working links.

Our method of ignoring occasional problems by declaring a fault only when more than three violations occur in 128 samples is known as the k out of n method. This method is used in testing neighbor reachability in EGP (Mills [1984]) and in Cypress (Comer and Narten [1988]). For simplicity, we test k out of n only at the end of n samples, rather than continuously. EGP also shares our idea of using different threshold parameters depending on the current state of the link.

Round-trip verifier

Now let us consider the problem of getting the transmission layers in two adjacent nodes to agree about the state of a connecting link. The solution is a protocol in which each node indicates to its peer whether it thinks the link should be working or not. When a node knows both from itself and from its peer that the link should be working, then it declares the link to be working. Otherwise, the link is broken. This function is implemented by the round-trip verifier.

The round-trip verifier contains a state machine with three states: dead, test and good. See Figure 11.7. Because it supports the same structure of interactions between subordinate object and filtered object as the skeptic, the state machine resembles that of the skeptic (see Figure 11.3). Dead state means that the underlying skeptic declares that the link is broken, test state that the skeptic declares the link is working but the remote node does not yet concur and good state that both agree the link is working.

The round-trip verifier uses the flow-control channel on a link to send a signal to the other node. The flow-control channel is a dedicated time slot in which the link's transmitter normally sends flow-control command codes. Under software control, the transmitter fills this slot instead with a distinguished command code called *idhy*, which stands for 'I Don't Hear You'. The round-trip verifier uses *idhy* to send a 'bad' signal and uses the absence of *idhy* to send an 'okay' signal. The verifier receives these signals by decoding hardware status indicators using the same sampling system of blocks and subblocks as the violation detector.

The round-trip verifier continually sends 'bad' to its peer in dead state and 'okay' in test state and good state. Because the link transports these signals below the level of packet traffic, the nodes can exchange this information even when the link is isolated. The verifier remains in test state until it detects an 'okay' signal, at which point it moves to good. When the verifier is in good state, it immediately

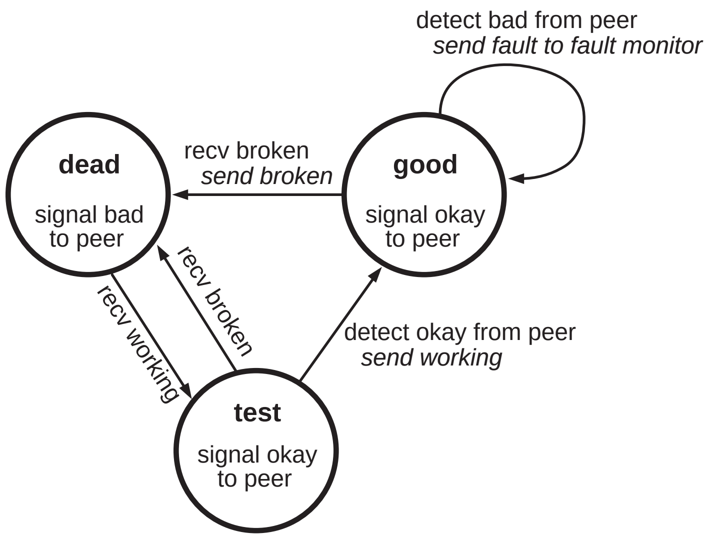


Figure 11.7. Round-trip verifier state machine.

declares a fault if it ever detects a 'bad' signal. The fault causes the underlying skeptic to declare the link broken which results in the verifier moving back to dead state. The verifier sends and receives 'working' and 'broken' messages from adjacent software layers just like the skeptic does.

Notice the effect of the round-trip verifier on a link that works well in only one direction, say from node A to node B. The error detector in node B has no cause for complaint and its skeptic declares that the link is working. The error detector in node A is upset and its skeptic declares that the link is broken. The round-trip verifier in node A is signaling 'bad', which tells the round-trip verifier in node B that A is upset. Consequently, the transmission layers in both nodes agree that the link is broken. The verifier in A is in dead state and the verifier in B is in test state.

Now suppose that the link is repaired and the error detector in node A is now happy. After the skeptic recovery delay, the round-trip verifier in node A moves to test state and begins signaling 'okay'. It detects 'okay' from B, which is in test state and moves to good state. The round-trip verifier in node B soon detects the 'okay' from A and moves to good state. Both nodes now believe that the link is working. The transmission layer always brings a link up with this handshake.

One other task of the round-trip verifier is to filter out links that connect to host controllers. A host controller uses different command codes than a switch and this difference is reflected in the link's hardware status. The round-trip verifier classifies a link as a host link or a switch link based on an examination of this status and it declares a fault whenever the classification changes. A link that connects to a host has no effect on switch-to-switch connectivity and therefore is considered as broken for the purpose of reconfiguration, but assuming that the skeptic and verifier are otherwise happy, the link is taken out of isolation so that packets can pass across it to the host.

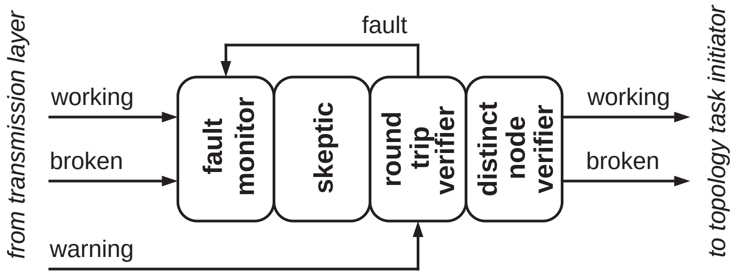


Figure 11.8. Diagram of the connectivity layer.

As we have seen, the transmission layer filters out links with error conditions and links that connect to host controllers. It is the job of the connectivity layer to filter out links that do not connect anywhere or that connect back to the same switch.

11.4.3 Connectivity Layer

The connectivity layer comes into action once the transmission layer has declared that a link is working. The connectivity layer sends packets back and forth across the link to determine if link is a useful node-to-node connection. When the connectivity layer declares that a link is useful, it also provides the identity of the remote node. The union of the results of the connectivity layers for the links at a node comprises the current state of the neighborhood monitoring task at that node.

The connectivity layer contains a skeptic with fault monitor, a round trip verifier and a distinct node verifier. See Figure 11.8. The fault monitor receives the messages from the transmission layer about whether the link is working or broken. The round-trip verifier filters the output of the skeptic by delaying ‘working’ messages until it has exchanged packets over the link and has determined the identity of the remote node. The distinct node verifier filters the output of the round-trip verifier by checking that the remote node is indeed different from the local node. Whenever the connectivity layer generates either a ‘working’ or ‘broken’ message, topology acquisition is initiated.

The design of the round-trip verifier satisfies several goals. 1) A link is tested vigorously whenever there is reason to believe that the testing might change the link’s state. 2) A link is always regularly retested. 3) Retesting a stable link occurs at a rate low enough to impose little overhead. By adjusting the testing effort according to a hint of how interesting the test result might be, we achieve prompt detection of common changes while incurring little overhead on average. The testing effort never falls below a certain minimum to guarantee that any change is detected eventually. A more detailed description follows.

The round-trip verifier exchanges connectivity packets with its peer on the remote end of the link, with the purpose of determining the identity of its peer. Peer identity (id) has two components: a 48-bit unique node identifier and a 4-bit port number within the node. To insure that an identity is current, we create a sequenced identity (*s*-id) by attaching a 32-bit sequence number. The round-trip verifier knows its own local *s*-id and it maintains a current estimate of the *s*-id of its remote peer. A connectivity packet carries the local and remote *s*-ids from the transmitter as source and destination *s*-ids.

The connectivity round-trip verifier has a state machine similar to that of the transmission round-trip verifier. In place of detecting 'okay' and 'bad' signals, the connectivity round-trip verifier checks the result of a round-trip exchange of connectivity packets.

Some connectivity packets are requests and others are replies. The only difference is that the receiver must send back a connectivity packet in response to a request, whereas a reply does not need a response. Requests and replies are distinguished by a flag in the packet header.

When the round-trip verifier enters test state, it begins sending request packets, waiting for a matching packet to be received. A matching packet contains a destination *s*-id equal to the local *s*-id. When it receives a matching packet, the verifier moves to good state and declares that the link is working. In any case, the receiver saves the source *s*-id of any received packet as its estimate of the remote *s*-id. This causes any subsequent packet sent back to be seen as a matching packet at the other end. The verifier retransmits very frequently at first but backs off exponentially if no matching packet is forthcoming.

The round-trip verifier continues sending request packets in good state, waiting for confirming packets to be received. A confirming packet contains a destination *s*-id equal to the local *s*-id and a source id equal to the remote id. (It is not necessary to inspect the remote sequence number to insure that the packet exchange is current.) We save the source *s*-id as the estimate of the remote *s*-id and increment the local sequence number. A packet that would be confirming except that the destination and local sequence numbers fail to match is ignored. Any other received connectivity packet causes an immediate fault (causing the skeptic to declare the link broken causing the verifier to move to dead state). The verifier transmits requests very frequently at first but backs off exponentially as confirming packets are received. When confirming packets fail to be received within five times the transmission interval, the interval is decreased by half and another five transmissions are attempted. A fault is declared if the interval had already attained its minimum value.

Whenever the skeptic declares that the link is broken, the round trip verifier passes on the declaration and enters dead state. In dead state, the round-trip verifier sends no requests. If any requests are received during dead state, they are answered with replies that good state treats as contradictory and test state ignores.

The effect of the round-trip verifier is as follows. Suppose that the link appears healthy to the transmission layer but connects to a node that does not answer. The round-trip verifier will be in test state and exponentially back off until the transmission rate is one request packet every ten seconds. Now suppose that the remote node begins responding. With the first matching reply, the round-trip verifier moves to good state with the minimum transmission interval. As confirmations arrive, the transmission interval increases to its maximum, at which point the round-trip verifier will be sending one request packet every ten seconds to continue verifying that the link is still working.

The exponential backoff in the round-trip verifier transmission interval allows a gradual transition between the vigorous testing regime and the background testing regime. Occasional losses or delays in responding to the round-trip protocol do not significantly alter the regime, while a reasonable response time is still provided in the case of complete autism.

The distinct node verifier filters the output of the round-trip verifier by comparing the remote node identity with the local node identity. If they are equal, the distinct node verifier does not pass on the declaration that the link is working. Links that connect back to the same node are not useful node-to-node connections even though they may carry packets perfectly. Whenever the distinct node verifier changes a link from working to broken or back, the neighborhood monitoring task declares a change in the neighborhood, which causes the topology task to recompute the network topology. The topology task is discussed in detail later.

In addition to the actions described above, the round-trip verifier also responds to warnings issued by the transmission layer error detector. The violation error detector issues an immediate warning whenever it detects one or more violations in a subblock, assuming the condition is not serious enough to warrant declaring a fault. The connectivity layer round-trip verifier responds to a warning in good state by resetting its transmission interval to the minimum. This results in rapid transmission of round-trip request packets and a fault if no confirmation is soon received. Warnings are ignored in dead state and test state.

All packets sent by the topology task, to be described in Section 5, have a header that contains the source id. These values are checked against the remote id and any mismatch results in a connectivity fault just like receiving a contradictory connectivity packet. If the distinct node verifier says that the link is broken, any topology packets received will be discarded.

11.4.4 Development History and Experience

In our original implementation we had a much simpler filter in place of the skeptic: the simple filter just refrained from recovering the link more than once every ten seconds. We had not realized how perverse malfunctioning hardware could be. As more and more hardware was deployed we soon had several marginal links that each cycled through failure and recovery once a minute or so. The problem was that the rate of errors on a marginal link depended on the data pattern being sent on the link and links that were in service tended to provoke a lot more errors than links that were being tested. At that time the topology task took about five seconds to complete, so the network was completely unusable. The skeptic fixed the problem by effectively removing marginal links from the network.

We encountered another difficulty due to a malfunctioning host controller that sent incorrect flow control commands. This caused the adjacent node to get stuck when it attempted to send the host a packet. We had not realized how important it would be to detect stuck links. The stuck node could not respond to connectivity requests from its neighbors, so after ten seconds they gave up and reconfigured. The stuck node also reconfigured into a partition by itself. Part of reconfiguration involves reinitializing the crossbar switch, which has the side effect of unsticking a stuck node. The node would then join back with its neighbors and everyone would reconfigure again. Soon the node would attempt to send the host another packet, and the process would repeat. Once we implemented the skeptic it defended against this problem by effectively partitioning the stuck node out of the network, which at least saved the network from collapse. Implementing stuck link detection allowed specific problem links to be identified and isolated, without partitioning the network.

We added the warnings when we observed an unexpectedly disruptive link failure mode that was not being detected promptly. The failure produced one or two error samples in the transmission layer error detector (not enough to declare a fault), but no other indications until the next time the connectivity layer round-trip verifier performed its end-to-end check, which would occur on average five seconds later. The failure mode was provoked by turning off the power of the remote node, which was quite frequent in the early days. The way the hardware is implemented, a powered-off node reflects data perfectly. A link watching its remote node power-off sees the line change from node-to-node to reflecting. In some cases, this change would be so clean that it dependably provoked only a single error sample. We needed to detect this failure promptly. If the link to the powered off node was on the broadcast distribution tree, outgoing broadcast packets would reflect back into the network and continuously regenerate, causing network collapse. This was unexpected. Users were unhappy. Implementing the warnings caused the end-to-end check to detect the failure promptly and fixed the problem.

One excellent success of the connectivity layer occurred when a node's crossbar switch started acting erratically. The crossbar began switching occasional packets to the wrong output link. The neighboring nodes would detect a connectivity violation and break their links, but the links would then seem to work fine in test state so they would recover for a while before breaking again. Eventually the skeptics partitioned the malfunctioning node out of the network.

One scenario that exercises most mechanisms in the monitoring task is the powering-on of a switch. Let *A* be a switch that is about to be powered-on. As mentioned above, a powered-off switch reflects data perfectly. Therefore, while *A* is powered-off, all of its neighbors *B*, *C* and *D* see links that work fine at the transmission layer and at the connectivity layer, except that the distinct node verifier refuses to pass on the working declaration. The connectivity layer round-trip verifiers are in good state at maximum back-off level. Now a technician powers on *A*. The software initializes all the state machines in dead state, which causes *A*'s transmission layer round trip verifiers to send *idhy*. *B*, *C* and *D* probably hear enough static during the power-on to declare faults, but if not then their transmission layer round-trip verifiers will do so when the *idhy* from *A* starts arriving. At this point everything pauses at least five seconds for transmission layer skeptics to finish their wait time-outs. Whenever both transmission layer skeptics on a link concur, the transmission layers pass on working declarations to the connectivity layers, which then start the connectivity layer skeptic time-outs. Finally, both connectivity layer skeptics concur, a round-trip packet exchange verifies the connectivity of the link and the topology task performs a reconfiguration. Because of the random wait timer adjustment, this happens at different times on different links. In this example, we would probably have three separate reconfigurations as each of *B*, *C* and *D* established its connection to *A*.

11.5 Topology Acquisition

The monitoring task provides each node *N* with a description of its neighborhood. In this description the links that do not work or that connect from *N* back to *N* have been eliminated. The responsibility of the topology acquisition task is to provide each node with a description of the current topology of the entire network.

We first describe the basic method of the topology task assuming a very simple scenario: some single node initiates the task (for an unknown reason) and the task runs to completion without any confusion from topology changes (which are assumed never to happen). Then we describe how to extend the basic method to deal with multiple initiators and with topology changes.

11.5.1 Basic Method

Let us consider a single instance of topology acquisition as if it were the only thing that ever happened in the network. The basic method presumes that the network is quiet, some single node spontaneously initiates the topology task, it runs for a while and then the network is quiet forever after. Note that, even in this simple case, it is possible that two nodes may disagree about the state of their connecting link. In this circumstance it is required that the topology task never claim to produce a complete topology description.

The topology acquisition task consists of three phases: 1) propagation, which constructs a rooted spanning tree over the set of all reachable nodes; 2) collection, which merges descriptions of larger and larger subtree neighborhoods; and 3) distribution, which sends the complete description from the root back down the spanning tree to all nodes.

The propagation phase consists of a wave of packets that spreads across all links through the network starting from the initiating node. The initiating node becomes the root of the spanning tree, and each other node joins the spanning tree by designating as its parent link the link on which it is first contacted. This is called a propagation order spanning tree. Generally one would expect the depth of a propagation order spanning tree not to exceed by more than a small factor the minimum possible depth of a spanning tree rooted at the initiating node and experience in our network supports this intuition.

During the propagation phase, each node *N* in the spanning tree contacts each of its neighbors, *M*, to offer *M* the opportunity to join the spanning tree as a child of *N*. If *M* has not yet joined the spanning tree, it accepts the offer and joins. *M* then contacts each of its neighbors, in turn. Otherwise, *M* is already in the tree and it refuses the offer. *M* sends back a reply to *N* so that *N* knows whether *M* accepted or refused. In this way each node comes to know its parent and its children in the spanning tree.

During the propagation phase each node conducts a query-reply exchange with each of its neighbors. The neighborhood monitoring task guarantees that topology-task packets will pass only if both end nodes agree that the link is useful. Suppose there is disagreement about the state of a link between nodes *P* and *Q*: *P* considers the link to be useful but *Q* does not. Then the propagation phase will get stuck when it arrives at *P*, because *P* will not be able to get a reply from *Q*. Therefore the propagation phase will manage to finish building a spanning tree only if all nodes agree about the state of their connecting links.

The propagation phase dies out when all nodes have been contacted and a spanning tree has been formed. This is a global condition, however and no individual node knows when it has been attained. Instead, there is a rolling transition from the propagation phase to the collection phase that begins at the leaves of the spanning tree. A node knows that it is a leaf in the spanning tree when it has contacted all of its neighbors and they have all refused to be children.

The collection phase begins at the leaves of the spanning tree and rises up to the root. When a node *M* accepts a propagation-phase offer to be a child of *N*, it also commits to sending up to *N* a description of the neighborhood of *M*'s subtree. If *M* is a leaf in the spanning tree, this is easy, because the link monitoring task provides each node with a description of its neighborhood. Otherwise, *M* has children. In this case, *M* waits for subtree neighborhood descriptions from all of its children, merges them with the description of its own neighborhood and then sends the result on up to *N*.

Eventually the collection phase reaches the point at which the root — just like any other node – has merged a description of its neighborhood with descriptions from all of its children to produce a description of its subtree neighborhood. But, in the case of the root, this is a description of the entire network. At this point the collection phase ends and the distribution phase begins. The root sends to each of its children the full network description. The children in turn send it to their children and so on, until every node in the network possesses the full network description.

11.5.2 Multiple Initiators

The basic method assumes that exactly one node initiates the topology task. Now we extend the method to deal with multiple initiators. Multiple initiators cause confusion because more than one node is claiming to be the root of the spanning tree. The confusion is solved by separating the activity into distinct instances of the topology task based on the initiator. Each initiator creates a new, distinct instance of the topology task, which runs independently of all other instances. All state records and packets are labeled with the unique identifier of the initiator in order to keep things straight.

For the purposes of both time and space efficiency, we do not want to run multiple instances of the topology task to completion. Also, the topology task is supposed to come to a single definite conclusion in each node. The solution is to conduct a competition during the propagation phase, so that exactly one instance wins and completes the phase, while all the others die out. Observe that the propagation phase spreads over the entire network, so if multiple instances do get started, they will come into competition with each other. Because the propagation phase has no definite end but instead rolls into the collection phase, no node knows which instance wins the competition until the end of the collection phase.

We conduct the competition as follows. Each node is allowed to belong to at most one instance of the topology task at a time. When the propagation phase of instance *I* first arrives at a node, the node can be in one of two states. If the node does not yet belong to any instance, it responds by joining instance *I* and then within that instance it joins the spanning tree in the normal way. Otherwise the node already belongs to some other instance *J*. In this case the node must decide whether to ignore *I* and remain in *J*, or discard *J* and join *I*. The node makes this decision by comparing the unique identifier labels of the instances. The instance with the lower unique identifier wins.

With the further proviso that only those nodes that do not yet belong to an instance may initiate instances, this competition assures us that exactly one instance will manage to complete the propagation phase — it will be the instance whose initiator has the lowest unique identifier over all initiators. All other instances will die out, as their nodes get taken over by the winning instance.

11.5.3 Topology Changes

We have extended the basic method to deal with multiple initiators. Now we further extend the method to deal with topology changes. Topology changes cause confusion because the method depends on running in a network whose topology is stable. So we simulate a stable topology by using an epoch mechanism. Each node maintains an epoch number that identifies the epoch in which its topology task is running, and this epoch number is included in all topology task packets. When a node's neighborhood monitoring task reports a change in the neighborhood, the node forgets all of its old topology task state, increments its epoch number and initiates a topology task in the new epoch. Whenever a node receives a topology task packet, it compares the epoch number in the packet to its own epoch number. If the packet has an old epoch number it is ignored, if it has the current epoch number it is processed, and if it has a new epoch number, the node forgets all its old topology task state, adopts the new epoch number and then processes the packet. In this latter case, the only possible packet is an offer to join the spanning tree.

One way to think of epochs is as competing instances of the topology task. (Of course, these are the multiple-initiator method tasks that have their own sub-instances based on initiators.) Each node is always trying to promulgate the newest epoch it has heard about. We optimize the competition by having a node keep track of the state of the topology task only for the newest epoch.

If any instance of the topology task runs to completion, it must have appeared that the network topology was consistent and stable. This is because a node effectively locks its current neighborhood into the epoch at the moment it adopts the epoch number. If any changes occur in the node's neighborhood, the neighborhood monitoring task reports it and the node advances to the next epoch. If there are nodes with inconsistent neighborhoods, which is a possible transient state of the network, the neighborhood monitoring task rapidly eliminates these inconsistencies and reports neighborhood changes in at least one of the affected nodes, which causes new epochs to be created.

11.5.4 Development History

An initial version of our terminating distributed spanning tree algorithm was invented in 1987 by Leslie Lamport and K. Mani Chandy. The current topology task method differs principally in constructing a propagation-order spanning tree with the initiator as the root. Lamport and Chandy's version constructs a unique minimum-depth spanning tree with the node of globally lowest unique identification as the root. In fact, we still use the Lamport-Chandy tree as our broadcast distribution tree, but each node computes it from the topology description rather than during reconfiguration. The current method also provides for retransmission and acknowledgment to deal with lost packets.

Although experience has not revealed problems with the basic algorithm, considerable work has gone into tuning the implementation. Initially, we had 27 switches in our network and the goal was reconfiguration in less than 200 milliseconds. The original implementation took about five seconds to run the topology task. We soon discovered that a major obstacle was that each node generated a voluminous debugging log. Code for logging events was optimized for flexibility, not speed, since its purpose was to help locate correctness bugs, which it did. Removing most of the debugging log events reduced the run time to about 1360 milliseconds. All of these times are for the topology task. Additionally, it takes about 20 milliseconds for the monitoring task to notice a link failure.

In order to speed up the topology task, we had to find performance bugs. We added code to each node to keep a trace log of interesting events such as packet arrival and departure and we added fields to topology packets to carry clock exchange information. This code was optimized for speed so that it would reveal rather than create performance bugs. Then we wrote a diagnostic program to extract trace logs from all the nodes, correlate them by computing clock synchronization, and then print out the result as a single, global event trace of all of the activity during a reconfiguration. This tool was essential in locating and fixing performance bugs. We ran many experiments and looked long and hard at the resulting traces.

We saved about 470 milliseconds by cranking down retransmission timers: in the propagation phase from 500 to 20 milliseconds and in the collection phase from 100 to 10 milliseconds. The event trace revealed that one or two retransmissions were always showing up in the critical path. After studying the situation, we concluded that these critical path retransmissions were unavoidable. During the propagation phase, each contacted node clears its forwarding table in order to purge old client traffic, during which the node is deaf for about 15 milliseconds. If the nodes do not purge old client traffic, forwarding cycles may arise during reconfiguration and cause deadlock or regenerative broadcasts. We tried doing without the purge and sure enough, we occasionally got regenerative broadcasts, which, due to a bug in the host controller microcode, tended to crash the hosts. Because the original switch software for retransmission had too much overhead to permit the desired small timer values, we had to reimplement it as part of this improvement.

We saved about 420 milliseconds when we discovered that each node copied its forwarding table into its debugging log. The debugging log is flexible, but very slow. We thought these printings were not on the critical path, but it turned out that some were. The simplest solution was just to delete the printings.

We saved about 100 milliseconds when we replaced our software CRC algorithm with a software checksum for topology packets. There is no hardware support in the switches for CRC, so it had to be performed in software.

At this point we saw that the majority of the run time went into the topology distribution phase. We completely reimplemented this phase and saved about 170 milliseconds. The original implementation unmarshaled the topology description packets into an internal data structure and then for each child remarshaled the data structure back into packets to send. The redesigned implementation marshals the data structure once at the root, distributes the packets from generation to generation as quickly as possible and then unmarshals the data structure in all nodes in parallel.

At various times we made changes guided by intuition about what would speed things up, rather than by study of the trace log. The resulting improvements were uniformly disappointing. Changing from the Lamport-Chandy tree to an early version of the propagation order tree made the code much simpler but saved only 26 milliseconds. A later simplification saved an additional 14 milliseconds. Polishing all the code in the collection and distribution phases saved a total of 2 milliseconds. Changes guided by the performance trace were much more effective at speeding things up.

The result of performance tuning was a system that ran the topology task in 154 milliseconds on the 27-switch network. Our network has since expanded to 31 switches on which the topology task runs in 179 milliseconds. Based on trials using different subsets of our network, the formula

$$time = 58 + 3.34 * d + 1.36 * n + 0.315 * d * n$$

where d is the diameter of the network and n is the number of switches, gives a fairly reasonable approximation to the reconfiguration time in milliseconds. Extrapolating using this formula, a 100-switch network arranged in a diameter 10 torus would have a reconfiguration time of 542 milliseconds. Such a reconfiguration time would perhaps just barely be acceptable. The time would of course be less given a faster switch control processor.

11.5.5 Related Work

Topology acquisition based on computing a spanning tree was described by Perlman [1985] for Ethernet bridges. Her version of the spanning tree algorithm reaches steady state without any explicit termination.

The propagation and collection phases of topology acquisition in Autonet are an example of termination detection in a diffusing computation as described by

Dijkstra and Scholten [1980].

The propagation-order tree used for coordinating the topology task represents an engineering tradeoff in favor of average-case performance for our network. Although in the worst case the propagation-order tree might be linear, the trees constructed in our network are almost always no more than one level deeper than a minimum depth tree. Distributed algorithms for constructing minimum depth trees with good asymptotic worst-case performance are known (Awerbuch and Gallager [1987]; Gallager [1982]), but they are considerably more complicated than the propagation-order method and are not nearly as efficient for moderate-sized networks.

11.6 Conclusions

The automatic reconfiguration of Autonet has succeeded in eliminating human management from day-to-day operation of the network. Our experience over the past year has been that the network runs itself. Reconfiguration runs quickly enough that the occasional network outages indeed are covered by normal retransmission in higher-level protocols.

Currently our Autonet installation experiences about ten reconfigurations per week, usually due to one of the 'usual suspects' – a small number of occasionally flaky switch-to-switch links that have not been worth trying to fix. A recent spell of hot weather provoked hundreds of isolated reconfigurations due to intermittent malfunctions in a few overheated switch crossbars, but in spite of the many brief disruptions the Autonet was almost always available and no user noticed an outage. Experimental hardware has the advantage of providing tests like this. Occasional reconfigurations also result from demonstrations to visiting dignitaries.

Redundancy is exploited to work around failed components as well as to facilitate repair. For most of the past year, several percent of our installed hardware link interfaces did not work at any given time. Because our Autonet had redundancy and automatically reconfigured itself around these failures, there was no urgency about getting them fixed. When a technician did finally get around to repairing a failure, it could be accomplished by powering-off the problem switch, replacing the faulty components and then powering the switch back on. Normal network operation continued during the repair because network redundancy and automatic reconfiguration covered for the missing switch. A similar scenario applies to downloading (compatible) new versions of the switch control program.

The skeptic mechanism has succeeded in defending the network against unreliable hardware, while still allowing quick recovery from isolated failures. An advantage of the skeptic structure is that none of the error detectors ever have to figure out what to do to recover from an error: all they have to do is declare a fault. We have found it much easier to figure out how to detect possible errors than to worry about how to respond to each one individually.

Our Autonet would be completely unusable without these automatic mechanisms.

It might be noted that the description of the monitoring task in this paper is much more lengthy than the description of the topology task. The reason is that the monitoring task bridges an enormous gulf between idiosyncratic hardware functionality and the abstraction of node neighborhoods, whereas the topology task builds upon a simple abstraction using well-understood concepts. The monitoring task has to get many details right in order to work acceptably. The topology task only has to work fast. In our implementation, the monitoring task contains about twice as many lines of code as the topology task.

As is usual in robust systems of this sort, it is important to report component status through a management interface so that timely repairs may be effected. Several times we have been surprised to discover that some section had only a single connection to the rest of the network, usually due to the combination of poor interconnection and multiple link failures. We are still working on management functions.

Two examples of needed improvements in reconfiguration are suppression of multiple reconfigurations during switch booting and provision to command a timely response to an attempted link repair.

Booting a switch causes separate reconfigurations as each of its switch-to-switch links is discovered. It would be less disruptive if all the links could be brought up using only one reconfiguration. Our idea to accomplish this would be to add a delay between declaring a link as working in the monitoring task and initiating the topology task. During this delay, any attempt to perform an action that depends on the state of the link would cancel the remainder of the delay. Such actions are receiving a topology message and running the topology task (either an old one or a new one started by some other link).

Because the skeptics may have attained a high level of skepticism and refuse to respond to a bad link, a technician may have difficulty determining if the link has indeed been repaired. The maximum wait time of the transmission skeptic is 17 to 34 minutes and of the connectivity skeptic is 28 to 56 hours. Our technicians have adopted two strategies: one is to come back the next day and the other is to reboot a switch. Unfortunately the technician may have to go to another floor to get to the necessary switch. There should be a more convenient mechanism that a technician could use to instruct the network that a link repair has been attempted.

Acknowledgments

Autonet was designed and built by Andrew Birrell, Michael Burrows, Butler Lampson, Hal Murray, Roger Needham, Tom Rodeheffer, Ed Satterthwaite, Michael Schroeder and Charles Thacker.

11.7 References

Awerbuch, B., Cidon, I., Gopal, I., Kaplan, M. and Kutten, S. (1990), Distributed control for PARIS, *Proceedings of the ninth ACM Annual Symposium on Principles of Distributed Computing*, Québec City, Québec.

Awerbuch, B. and Gallager, R.G. (1987), A New Distributed Algorithm to Find Breadth-First Search Trees, *IEEE Transactions on Information Theory* **33**, 315–322.

Comer, D. and Narten, T. (1988), UNIX Systems as Cypress Implets, *USENIX 88 Winter Conference*.

Dijkstra, E.W. and Scholten, C.S. (1980), Termination Detection for Diffusing Computations, *Information Processing Letters* **11**, 1–4.

Gallager, R.G. (1982), Distributed Minimum Hop Algorithms, Massachusetts Institute of Technology, Technical Report LIDS-P-1175, Cambridge, MA.

Heart, F.E., Kahn, R.E., Ornstein, S.M., Crowther, W.R. and Walden, D.C. (1970), The interface message processor for the ARPA computer network, *AFIPS 1970 Spring Joint Computer Conference*.

Jacobsen, V. (1988), Congestion Avoidance and Control, *ACM SIGCOMM Communications Architectures and Protocols*.

Lin, T.Y. and Siewiorek, D.P. (1990), Error log analysis: Statistical modeling and heuristic trend analysis, *IEEE Transactions on Reliability* **39**, 419–432.

McQuillan, J.M., Richer, I. and Rosen, E.C. (1980), The New Routing Algorithm for the ARPANET, *IEEE Transactions on Communication* **COM-28**, 711–719.

McQuillan, J.M. and Walden, D.C. (1977), The ARPA Network Design Decisions, *Computer Networks* **1**, 243–289.

Mills, D.L. (1984), Exterior gateway protocol formal specification, Network Information Center, Request for Comments 904, Menlo Park, CA.

Perlman, R. (1985), An Algorithm for Distributed Computation of a Spanning Tree, *Proceedings of the Ninth Data Communications Symposium*.

Saltzer, J.H., Reed, D.P. and Clark, D.D. (1984), End-to-End Arguments in System Design, *ACM Transactions on Computer Systems* **2**, 277–278.

Schroeder, M.D., Birrell, A.D., Burrows, M., Murray, H., Needham, R.M., Rodeheffer, T.L., Satterthwaite, E.H. and Thacker, C.P. (1990), Autonet: A High-speed, Self-configuring Local Area Network Using, Digital Systems Research Center, Research Report 59, Palo Alto, CA, Also available in *IEEE Journal on Selected Areas in Communications,*.

Turner, J.S. (1986), New directions in communications (or which way to the information age?), *IEEE Communications Magazine* **24**, 8–15.

Wulf, W.A., Levin, R. and Harbison, S.P. (1981), *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, NYC.