This paper appears as Chapter 1 in the book Distributed Systems,
edited by Sape Mullender and published by Addison-Wesley and ACM Press
in 1993, 1994, and 1995.

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

Michael D. Schroeder
DEC Systems Research Center
130 Lytton Ave
Palo Alto, CA 94301

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. 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 three 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.

o Insecure Communication -- The interconnections among the computers may be exposed to unauthorized eavesdropping and message modification.

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 demon 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 never need to confront unreliable and insecure communications. In a distributed system all three 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.

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.

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 ther 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.

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.

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.

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 (tens or hundreds of 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.

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.

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.

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.

4.0 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.

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:

o a set of servers for the named object
o rules for choosing a server name
o the service interface ID, e.g. "BOB File Service 2.3"
o 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:

o a machine name
o 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.

The following figure 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.

```
/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, etc)
 |   |   |             / ... 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
 |   |   |
 |   |   / ... other parts of DEC
 |   |
 |   / ... other commercial organizations
 |
 / ... other sectors and countries
```

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, 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.

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:

o on the user's own workstation;
o on public workstations or compute servers in the user's domain;
o on public workstations in another domain on the user's LAN;
o 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.

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.

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 domains. 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.

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, 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.

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.

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.