Scale in Distributed Systems

B. Clifford Neuman

Information Sciences Institute
University of Southern California

Abstract

In recent years, scale has become a factor of increasing importance in the design of distributed systems. The scale of a system has three dimensions: numerical, geographical, and administrative. The numerical dimension consists of the number of users of the system, and the number of objects and services encompassed. The geographical dimension consists of the distance over which the system is scattered. The administrative dimension consists of the number of organizations that exert control over pieces of the system.

The three dimensions of scale affect distributed systems in many ways. Among the affected components are naming, authentication, authorization, accounting, communication, the use of remote resources, and the mechanisms by which users view the system. Scale affects reliability: as a system scales numerically, the likelihood that some host will be down increases; as it scales geographically, the likelihood that all hosts can communicate will decrease. Scale also affects performance: its numerical component affects the load on the servers and the amount of communication; its geographic component affects communication latency. Administrative complexity is also affected by scale: administration becomes more difficult as changes become more frequent and as they require the interaction of different administrative entities, possibly with conflicting policies. Finally, scale affects heterogeneity: as the size of a system grows it becomes less likely that all pieces will be identical.

This paper looks at scale and how it affects distributed systems. Approaches taken by existing systems are examined and their common aspects highlighted. The limits of scalability in these systems are discussed. A set of principles for scalable systems is presented along with a list of questions to be asked when considering how far a system scales.

1 What is Scale?

In recent years scale has become an increasingly important factor in the design of distributed systems. Large computer networks such as the Internet have broadened the pool of resources from which distributed systems can be constructed. Building a system to fully use such resources requires an understanding of the problems of scale.

A system is said to be *scalable* if it can handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity. Scale has three components: the number of users and ob-

This work was performed while the author was completing a PhD at the University of Washington and was supported in part by the National Science Foundation (Grant Number CCR-8619663), the Washington Technology Centers, and Digital Equipment Corporation.

Author's address: University of Southern California, Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, California 90292 USA. (bcn@isi.edu)

B. Clifford Neuman. Scale in Distributed Systems. In *Readings in Distributed Computing Systems*. IEEE Computer Society Press, 1994.

jects that are part of the system, the distance between the farthest nodes in the system, and the number of organizations that exert administrative control over pieces of the system.

If a system is expected to grow, its ability to scale must be considered when the system is designed. Naming, authentication, authorization, accounting, communication, and the use of remote resources are all affected by scale. Scale also affects the user's ability to easily interact with the system.

Grapevine was one of the earliest distributed computer systems consciously designed to scale. More recent projects such as the Internet Domain Naming System (IDNS), Kerberos, Sprite, and DEC's Global Naming and Authentication Services have concentrated on particular subsystems. Other projects have attempted to provide complete scalable systems. Among them are are Locus, Andrew, Project Athena, Dash, and Amoeba. Scale affects the way users perceive a system: as the number of objects that are accessible grows, it becomes increasingly difficult to locate the objects of interest. Plan 9, Profile, Prospero, QuickSilver, and Tilde are a few systems that address this aspect of scale.

This paper examines the methods used to handle scale in these and other systems. Section 3 discusses the problems of scale and presents general solutions to the problems. Sections 4 through 6 look at the problems specific to individual subsystems and discuss the particular solutions used by several systems. These solutions generally fall into three categories: replication, distribution, and caching, defined in Section 2 and discussed further in Sections 7 through 9. While earlier sections describe the affect of scale on the systems themselves, Section 10 examines some of the problems that confront the users of large systems. The techniques covered in this paper are summarized in Section 11 as a list of suggestions to be followed and questions to be asked when building scalable systems. Section 12 summarizes the scope, limitations, and conclusions drawn in this paper. Short descriptions of the systems mentioned in this paper may be found in an appendix.

2 Definitions

There are several terms used repeatedly throughout this paper. They are defined here for quick reference.

When used in this paper the term *system* refers to a distributed system. A *distributed system* is a collection of computers, connected by a computer network, working together to collectively implement some minimal set of services. A *node* is an individual computer within the system. A *site* is a collection of related nodes, a subset of the nodes in the system.

A service or resource is *replicated* when it has multiple logically identical instances appearing on different nodes in a system. A request for access to the resource can be directed to any of its instances.

A service is *distributed* when it is provided by multiple nodes each capable of handling a subset of the requests for service. Each request can be handled by one of the nodes implementing the service (or a subset of the nodes if the service is also replicated). A *distribution function* maps requests to the subset of the nodes that can handle it.

The results of a query are *cached* by saving them at the requesting node so that they may be reused instead of repeating the query. Caching improves the performance of a local node by reducing the time spent waiting for a response. Caching improves scalability by reducing the number of repeated queries sent to a server. Caches employ *validation tech-*

niques to make sure that data from the cache are not used if the results of the query might have changed. Caching is a temporary form of replication.

3 The Effects of Scale

Scale affects systems in numerous ways. This section examines the effects of scale on reliability, load, administration, and heterogeneity. These effects are felt by all parts of the system.

3.1 Reliability

As the number of components in a distributed system increases, the likelihood decreases that they will all be working simultaneously. As the system scales geographically, it becomes less likely that all components will be able to communicate. A system should not cease to operate just because certain nodes are unavailable.

Reliability can often be improved by increasing the autonomy of the nodes in a system. A collection of nodes is autonomous if it runs independently from the other nodes in the system. A failure in an autonomous system only affects access to resources in the neighborhood of the failure. For example, failure of a name server in one part of a network would not prevent access to local resources in another.

Replication can also improve the reliability of a system. Replication allows a resource to be used even if some of the instances are not running or are inaccessible. Replicas can be scattered across the network so that a network failure is less likely to isolate any part of the system from all of the replicas. It might also be possible to dynamically reconfigure the set of servers used by a client so that if a server goes down, clients can continue with as little disruption as possible.

3.2 System Load

Scale affects system load in a number of ways. As a system gets bigger the amount of data that must be managed by network services grows, as does the total number of requests for service. Replication, distribution, and caching are all used to reduce the number of requests that must be handled by each server. Replication and distribution allow requests to be spread across multiple servers, while caching reduces repeated requests. The use of multiple file servers, each providing storage for different files, is an example of distribution. The existence of the same system binaries on more than one server is an example of replication. With replication, the choice of server can be based on factors such as load and proximity.

3.3 Administration

The administrative dimension of scale adds its own problems. As the number of nodes in a system grows, it becomes impractical to maintain information about the system and its users on each node; there are too many copies to keep up-to-date. Administration of a collection of nodes is made easier when common information is maintained centrally; for example, through a name server, authentication server, or through a file server that provides a central repository for system binaries.

As a system continues to grow, information about the system changes more frequently. This makes it less practical for a single individual to keep it up-to-date. Additionally, as a system crosses administrative boundaries, organizations want control over their own part of the system. They are less willing to delegate that control to individuals outside their organization. These problems can be addressed by distribution. Responsibility for maintaining pieces of the database are assigned to each organization, and each organization maintains that part of the databases concerning its own systems. Section 8 describes the methods that can be used to distribute the database.

3.4 Heterogeneity

The administrative dimension of scale compounds the problem of heterogeneity. It is likely that systems which cross administrative boundaries will not only include hardware of different types, but they may also be running different operating systems or different versions of the same operating system. It is not practical to guarantee that everyone runs exactly the same software.

Coherence is one approach to dealing with heterogeneity. In a coherent system, all computers that are part of the system support a common interface. This requirement takes several forms. All nodes might be required to support the same instruction set, but this is not often practical. A looser requirement is that all nodes support a common execution abstraction. Two computers share the same execution abstraction if software that runs on one computer can be easily recompiled to run on the other. Still looser is coherence at the protocol level: all nodes are required to support a common set of protocols, and these protocols define the interfaces to the subsystems which tie the system together. MIT's Project Athena [6] is an example of a system that uses coherence (of the execution abstraction) to deal with heterogeneity.

The Heterogeneous Computer Systems Project [21] provides explicit support for heterogeneity. A mechanism is provided that allows the use of a single interface when communicating with nodes that use different underlying protocols. The HCS approach shows that it is possible to support multiple mechanisms in heterogeneous systems. This ability is important when different mechanisms have different strengths and weaknesses.

We have just seen some of the issues that affect the scalability of a system as a whole. In the next few sections we will examine the effects of scale on particular subsystems.

4 Naming and Directory Services

A name refers to an object. An address tells where that object can be found. The binding of a name is the object to which it refers. A name server (or directory server) maps a name to information about the name's binding. The information might be the address of an object, or it might be more general, e.g., personal information about a user. An attribute-based name service maps information about an object to the object(s) matching that information. Attribute-based naming is discussed in Section 10.

4.1 Granularity of Naming

Name servers differ in the size of the objects they name. Some name servers name only hosts. The names of finer grained objects such as services and files must then include the name of the host so that the object can be found. A problem with this approach is that it is difficult to move objects. Other name servers name individual users and services. The names of such entities do not change frequently, so the ratio of updates to references is usually fairly low. This simplifies the job of a name server considerably. A few name servers name individual files. There are a huge number of files and they are often transient in nature. Supporting naming at this level requires support for frequent updates and a massive number of queries.

An intermediate approach is used by Sprite [22] and a number of other file systems. Groups of objects sharing a common prefix are assigned to servers. The name service maps the prefix to the server, and the remainder of the name is resolved locally by the server on which the object is stored. An advantage of this approach is that the name service handles fewer names, and the prefixes change less frequently than the full names of the objects. This also allows clients to easily cache the mappings they have learned. Another advantage is that names need not include the name of the server on which the object resides, allowing groups of objects (sharing a common prefix) to be moved. The main disadvantage is that objects sharing common prefixes must be stored together[1].

The size of the naming database, the frequency of queries, and the read-to-write ratio are all affected by the granularity of the objects named. These factors affect the techniques that can be used to support naming in large systems.

4.2 Reducing Load

Three techniques are used to reduce the number of requests that must be handled by a name server. The simplest is replication. By allowing multiple name servers to handle the same queries, different clients are able to send their requests to different servers. This choice can be based on physical location, the relative loads of the different servers, or made at random. The difficulty with replication lies in keeping the replicas consistent. Consistency mechanisms are discussed in Section 7.

Distribution is a second technique for spreading the load across servers. In distribution, different parts of the name space are assigned to different servers. Advantages to distribution are that only part of the naming database is stored on each server, thus reducing the number of queries and updates to be processed. Further, because the size of each database is smaller, each request can usually be handled faster. With distribution, the client must be able to determine which server contains the requested information. Techniques for doing so are described in Section 8.

Caching is a third technique that reduces the load on name servers. If a name is resolved once, it will often need to be resolved again. If the results are remembered, additional requests for the same information can be avoided. As will be seen later, caching is of particular importance in domain-based distribution of names. Not only is the same name likely to be used again, but so are names with common prefixes. By caching the mapping from a prefix to the name server handling it, future names sharing the same prefix can be resolved with fewer messages. This is extremely important because, as prefixes become shorter, the number of names that share them grows. Without the caching of prefixes, high-level name servers would be overloaded, and would become a bottleneck for name resolution. Caching is a form of replication, and like replication, the need to keep things consistent is its biggest difficulty. Caching is described in greater detail in Section 9.

4.3 UID-Based Naming

Not all distributed systems use a hierarchical name service like those that have been described. Some systems use unique identifiers to name objects. Capability-based systems such as Amoeba [30] fall into this category. A capability is a unique identifier that both names and grants access rights for an object. Unique IDs may be thought of as addresses. They usually contain information identifying the server

[1] Actually, a prefix can be an entire file name, but this can only be done with a very limited number of objects and does not scale.

that maintains the object, and an identifier to be interpreted by the server. The information identifying the server might be an address or it might be a unique identifier to be included in requests broadcast by the client. A client needing to access an object or service is expected to already possess its unique identifier.

A problem with uid-based naming is that objects move, but the UIDs often identify the server on which an object resides. Since the UIDs are scattered about without any way to find them all, they might continue to exist with incorrect addresses for the objects they reference. A technique often used to solve this problem is forwarding pointers [8]. With forwarding pointers, a user attempting to use an old address to access an object is given a new UID containing the new address. A drawback to forwarding pointers is that the chain of links to be followed can become lengthy. This reduces performance, and if one of the nodes in the chain is down, it prevents access to the object. This drawback is solved in Emerald by requiring that each object have a home site and that the forwarding pointer at that site is kept up to date. Another solution is for the client to update the forwarding pointers traversed if subsequent forwarding pointers are encountered.

Prospero [20] supports UIDs with expiration dates. Its directory service guarantees that the UIDs it maintains are kept up-to-date. By using expiration dates, it becomes possible to get rid of forwarding pointers once all possible UIDs with the old address have expired.

4.4 Directory Services

Even in uid-based systems, it is often desirable to translate from symbolic names that humans use into the UIDs for the named objects. Directory service do this. Given a UID for a directory it is possible to read the contents of that directory, to map from a symbolic name in the directory to another UID, and to add a symbolic name/UID pair to the directory. A directory can contain UIDs for files, other directories, or in fact, any object for which a UID exists.

The load on directory servers is easily distributed. There is no requirement that a subdirectory be on the same server as its parent. Different parts of a name space can reside on different machines. Replication can be supported by associating multiple UIDs with the same symbolic name, or through the use of UIDs that identify multiple replicas of the same object or directory.

The primary differences between a name server and a directory server is that the directory server usually possess little information about the full name of an object. A directory server can support pieces of independent name spaces, and it is possible for those name spaces to overlap, or even to contain cycles. Both Prospero and Amoeba use directory servers to translate names to UIDs.

4.5 Growth and Reorganization

For a system to be scalable, it must be able to grow gracefully. If two organizations with separate global name spaces merge, reorganize, or otherwise combine their name spaces, a problem arises if the name spaces are not disjoint. The problem arises because one or both name spaces suddenly change. The new names correspond to the old names, but with a new prefix corresponding to the point in the new name space at which the original name space was attached. This causes problems for any names which were hardcoded in programs or otherwise specified before the change.

DEC's Global Name Service [14] addresses this problem by associating a unique number with the root of every independent name space.

When a file name is stored, the number for the root of the name space can be stored along with the name. When name spaces are merged, an entry is made in the new root pairing the unique ID of each previous root with the prefix required to find it. When a name with an associated root ID is resolved, the ID is checked, and if it doesn't match that for the current root, the corresponding prefix is prepended, allowing the hardcoded name to work.

5 The Security Subsystem

As the size of a system grows, security becomes increasingly important and increasingly difficult to implement. The bigger the system, the more vulnerable it is to attack: there are more points from which an intruder can enter the network; the system has a greater number of legitimate users; and it is more likely that the users will have conflicting goals. This is particularly troublesome when a distributed system spans administrative boundaries. The security mechanisms employed in different parts of a system will have different strengths. It is important that the effects of a security breach can be contained to the part of the system that was broken.

Security has three aspects: authentication, how the system verifies a user's identity; authorization, how it decides whether a user is allowed to perform the requested operation; and accounting, how it records what the user has done, and how it makes sure that a user does not use excessive resources. Accounting can include mechanisms to bill the user for the resources used. Many systems implement a distributed mechanism for authentication, but leave authorization to the individual server. Few systems provide for accounting in a distributed manner.

5.1 Authentication

Several techniques are used to authenticate users in distributed systems. The simplest, the use of passwords on each host, requires maintenance of a password database on multiple nodes. To make it easier to administer, Grapevine [3] supported a central service to verify passwords. Password-based authentication can be cumbersome if the user is required to present a password each time a new service is requested. Unfortunately, letting the workstation remember the users password is risky. Password based authentication is also vulnerable to the theft of passwords by attackers that can eavesdrop on the network.

Host-based authentication, as used for `rlogin` and `rsh` in Berkeley Unix, has problems too. In host-based authentication, the client is authenticated by the local host. Remote servers trust the host to properly identify the client. As one loses control of the nodes in a system, one is less willing to trust the claims made by other systems about the identity of its users.

Encryption-based authentication does not suffer from these problems. Passwords are never sent across the network. Instead, each user is assigned an encryption key, and that key is used to prove the user's identity. Encryption-based authentication is not without its own problems. Principals (users and servers) must maintain a key for use with every other principal with which they might possibly communicate. This is impractical in large systems. Altogether, `(n x m)` keys are required where n is the number of users, and m the number of servers.

In [17] Needham and Schroeder show how the number of keys to be maintained can be reduced through the use of an authentication server (AS). An AS securely generates keys as they are needed and distributes them to the parties wishing to communicate. Each party

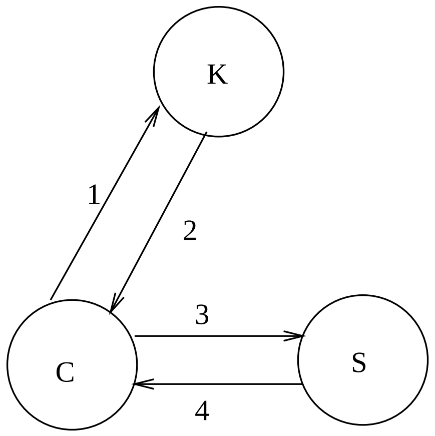

Figure 1: Kerberos Authentication Protocol

shares a key (or key pair) with the AS.

Authentication in Kerberos [29] is based on a modified version of the Needham and Schroeder protocol (Figure 1). When a client wishes to communicate with a server it contacts the AS, sending its own name and the name of the server to be contacted (1). The AS randomly generates a session key and returns it to the client encrypted in the key that the client registered with the AS (2). Accompanying the encrypted session key is a *ticket* that contains the name of the client and the session key, all encrypted in the key that the server registered with the AS.

In Kerberos the session key and ticket received from the AS are valid for a limited time and are cached by the client, reducing the number of requests to the AS. Additionally, the user's secret key is only needed when initially logging in. Subsequent requests during the same login session use a session key returned by the AS in response to the initial request.

To prove its identity to the server, the client forwards the ticket together with a timestamp encrypted in the session key from the ticket (3). The server decrypts the ticket and uses the session key contained therein to decrypt the timestamp. If recent, the server knows that the message was sent by a principal who knew the session key, and that the session key was only issued to the principal named in the ticket. This authenticates the client. If the client requires authentication from the server, the server adds one to the timestamp, re-encrypts it using the session key and returns it to the client (4).

As a system scales, it becomes less practical for an authentication server to share keys with every client and server. Additionally, it becomes less likely that everyone will trust a single entity. Kerberos allows the registration of principals to be distributed across multiple *realms*. The distribution mechanism is described in Section 8.

The Kerberos authentication protocol is based on conventional cryptography, but authentication can also be accomplished using public-key cryptography. In public-key cryptography, separate keys are used for encryption and decryption, and the key distribution step of authentication can be accomplished by publishing each principal's public key. When issues such as revocation are considered, authentication protocols based on public key cryptography make different tradeoffs, but provide little reduction in complexity. Authentication based on public key cryptography does, however, make a significant difference when authenticating a single message to multiple recipients.

5.2 Authorization

There are a number of ways distributed systems approach authorization. In one, a request is sent to an authorization service whenever a server needs to make an access control decision. The authorization service makes the decision and sends its answer back to the server. This approach allows the access control decision to take into account factors such as recent use of other servers, global quotas, etc. The disadvantage is that it can be cumbersome and the access control service becomes a bottleneck.

In a second approach the client is first authenticated, then the server makes its own decision about whether the client is authorized to perform an operation. The server knows the most about the request and is in the best position to decide whether it should be allowed. For example, in the Andrew file system [12] each directory has an associated list, known as an access control list (ACL), identifying the users authorized to access the files within the directory. When access to a file is requested, the client's name is compared with those in the ACL.

ACL entries in Andrew can contain the names of groups. The use of groups allow rights to be granted to named collections of individuals without the need to update multiple ACLs each time membership in the group changes. Each Andrew file server maintains the list of the groups to which each user belongs and that list is consulted before checking the ACL.

The server making an authorization decision should be provided with as much information as possible. For example, if authentication required the participation of more than one AS, the names of the AS's that took part should be available. It should also be possible for the server to use external sources to obtain information such as group membership. This approach, used in Grapevine, is similar to using an authorization service. It differs in that not all requests require information from the group server, and the final decision is left to the end server.

Like Andrew, authorization in Grapevine is based on membership in ACLs. ACLs contain individuals or groups that themselves contain individuals or groups. Group membership is determined by sending to a name server a query containing the name of the individual and the name of the group. The name server recursively checks the group for membership by the individual. If necessary, recursive queries can be sent to other name servers. One of the most noticeable bottlenecks in Grapevine was the time required to check membership in large groups, especially when other name servers were involved. [27]

External information can be made available to a server without the need for it to contact another service. The client can request cryptographically sealed credentials either authorizing its access to a particular object or verifying its membership in a particular group. These credentials can be passed to the server in a manner similar to that for the capability-based approach described next. The difference from capabilities is that these credentials might only be usable by a particular user, or they might require further proof that they were really issued to the user presenting them. Version 5 of Kerberos supports such credentials. Their use is described separately in [19].

5.2.1 Capabilities

The approaches discussed so far have been based on an access control list model for authorization. A disadvantage of this model is that the client must first be authenticated, then looked up in a potentially long list, the lookup may involve the recursive expansion of multiple groups, and interaction may be required with other servers. The advantages of the access control list model are that it leaves the final decision with the server itself, and that it is straightforward to revoke access should that be required.

Amoeba [30] uses the capability model for authorization. In the capability model, the user maintains the list of the objects for which access is authorized. Each object is represented by a capability which, when presented to a server, grants the bearer access to the object. To prevent users from forging capabilities, Amoeba includes a random bit pattern. By choosing the bit pattern from a sparse enough address space, it is sufficiently difficult for a user to create its own capability. A client presents its capability when it wishes to access an object. The server then compares the bit pattern of the capability with that stored along with the object, and if they match, the access is allowed.

The advantage of the capability model is that, once contacted by the client, the server can make its access control decision without contacting other servers. Yet, the server does not need to maintain a large authorization database that would be difficult to keep up-to-date in a large system. A disadvantage is that capabilities can only be revoked en masse. Capabilities are revoked by changing the bit pattern, but this causes *all* outstanding capabilities for that object to be immediately invalidated. The new capability must then be reissued to all legitimate users. In a large system, this might be a significant task.

Authorization in capability-based distributed systems is still dependent on authentication and related mechanisms. Authentication is required when a user logs in to the system before the user is granted an initial capability that can be used to obtain other capabilities from a directory service. Additionally, as was the case with passwords, capabilities can be easily intercepted when they are presented to a server over the network. Thus, they can not simply be sent in the clear. Instead, they must be sent encrypted, together with sufficient information to prevent replay. This mechanism is quite similar to that used for encryption-based authentication.

5.3 Accounting

Most distributed systems handle accounting on a host-by-host basis. There is a need for distributed, secure, and scalable accounting mechanism, especially in large systems that cross administrative boundaries. To date, few systems have even considered the problem. The difficulty lies in the inability to trust servers run by unknown individuals or organizations. The bank server [16] and accounting based on proxies [19] are among the few approaches that have been described.

In Amoeba, accounting is handled by bank servers which maintain accounts on behalf of users and servers. Users transfer money to servers, which then draw upon the balance as resources are used. Proxy-based accounting is tied much closer to authentication and authorization. The client grants the server a proxy allowing the server to transfer funds from the client's account.

Both approaches require support for multiple *currencies*. This is important as systems span international boundaries, or as the accounting service is called on to maintain information about different types of resources. The currencies can represent the actual funds for which clients can be billed, or they can represent limits on the use of resources such as printer pages or CPU cycles. Quotas for reusable resources (such as disk pages) can be represented as a deposit which is refunded when the resource is released.

Authorization and accounting depend on one another. In one direction, the transfer of funds requires the authorization of the owner of the account from which funds will be taken. In the other, a server might verify that the client has sufficient funds (or quota) to pay for an operation before it will be performed.

5.4 On Replication, Distribution and Caching

This section has described the problems specific to scaling the security subsystems of large systems and has discussed the mechanisms used to solve them. Many of problems that we saw with naming also arise with security. As with naming, replication, distribution, and caching are often used. When applying these techniques in the security area, a few considerations must be kept in mind.

When replicating a server that maintains secret keys, the compromise of any replica can result in the compromise of important keys. The security of the service is that of the weakest of all replicas. When distribution is used, multiple servers may be involved in a particular exchange. It is important that both principals know which servers were involved so that they can correctly decide how much trust to place in the results. Finally, the longer one allows credentials to be cached, the longer it will take to recover when a key is compromised.

As a system grows, less trust can be placed in its component pieces. For this reason, encryption-based security mechanisms are the appropriate choice for large distributed systems. Even encryption-based mechanisms rely on trust of certain pieces of a system. By making it clear which pieces need to be trusted, end services are better able to decide when a request is authentic.

6 Remote Resources

Naming and security are not the only parts of the system affected by scale. Scale affects the sharing of many kinds of resources. Among them are processors, memory, storage, programs, and physical devices. The services that provide access to these resources often inherit scalability problems from the naming and security mechanisms they use. For example, one can't access a resource without first finding it. This involves both identifying the resource that is needed and determining its location given its name. Once a resource has been found, authentication and authorization might be required for its use.

These services sometimes have scalability problems of their own, and similar techniques are employed to solve them. Problems of load and reliability are often addressed through replication, distribution, and caching. Some services further reduce load by by shifting as much computation to the client as possible; however, this should only be done when all the information needed for the computation is readily accessible to the client.

The services used to access remote resources are very dependent on the underlying communications mechanisms they employ. This section will look at the scaling issues related to network communication in such services. To provide an example of the problems that arise when supporting access to remote resources, it will then look at the effect of scale on a heavily used resource, the network file system.

6.1 Communication

As a system grows geographically, the medium of communications places limits on the system's performance. These limits must be considered when deciding how best to access a remote resource. Approaches which might seem reasonable given a low latency connection might not be reasonable across a satellite link.

Because they can greatly affect the usability of a system, the underlying communications parameters must not be completely hidden from the application. The Dash system [1] does a good job at exposing the communication parameters in an appropriate manner. When a connection is established, it is possible for the application to require that the connection meet certain requirements. If the requirements are not met, an error is returned. When one set of required communication parameters can not be met, it might still be possible for the application to access the resource via an alternate mechanism; e.g., whole file caching instead of remote reads and writes.

Communication typically takes one of two forms: point-to-point or broadcast. In point-to-point communication the client sends messages to the particular server that can satisfy the request. If the contacted server can not satisfy the request, it might respond with the identity of a server that can. With broadcast, the client sends the message to everyone, and only those servers that can satisfy the request respond.

The advantage of broadcast is that it is easy to find a server that can handle a request; just send the request and the correct server responds. Unfortunately, broadcast does not scale well. Preliminary processing is required by all servers whether or not they can handle a request. As the total number of requests grows, the load due to preliminary processing on *each* server will also grow.

The use of global broadcast also limits the scalability of computer networks. Computer networks improve their aggregate throughput by distributing network traffic across multiple subnets. Only those messages that need to pass through a subnet to reach their destination are transmitted on a particular subnet. Local communications in one part of a network is not seen by users in another. When messages are broadcast globally, they are transmitted on all subnets, consuming available bandwidth on each.

Although global broadcast should be avoided in scalable systems, broadcast need not be ruled out entirely. Amoeba [30] uses broadcast on its subnets to improve the performance of local operations. Communications beyond the local subnet uses point-to-point communication.

Multicast, a broadcast-like mechanism, can also be used. In multicast, a single message can be sent to a group of servers. This reduces the number of messages required to transmit the same message to multiple recipients. For multicast to scale, the groups to which messages are sent should be kept small (only those recipients that need to receive a message). Additionally, the network should only transmit multicast message across those subnets necessary to reach the intended recipients.

6.2 File Systems

The file system provides an excellent example of a service affected by scale. It is heavily used, and it requires the transfer of large amounts of data.

In a global file system, distribution is the first line of defense against overloading file servers. Files are spread across many servers, and each server only processes requests for the files that it stores. Mechanisms used to find the server storing a file given the file's name are described in Section 8. In most distributed systems, files are assigned to servers based on a prefix of the file name. For example, on a system where the names of binaries start with "/bin", it is likely that such files will be assigned to a common server. Unfortunately, since binaries are more frequently referenced than files in other parts of the file system, such an assignment might not evenly distribute requests across file servers.

Requests can also be spread across file servers through the use of replication. Files are assigned to multiple servers, and clients contact a subset of the servers when making requests. The difficulty with replication lies in keeping the replicas consistent. Techniques for doing so are described in Section 7. Since binaries rarely change, manual techniques are often sufficient for keeping their replicas consistent.

Caching is extremely important in network file systems. A local cache of file blocks can be used to make network delays less noticeable. A file can be read over the network a block at a time, and access to data within that block can be made locally. Caching significantly reduces the number of requests sent to a file server, especially for applications that read a file several bytes at a time. The primary difficulty with caching lies in making sure the cached data is correct. In a file system, a problem arises if a file is modified while other systems have the file, or parts of the file, in their cache. Mechanisms to maintain the consistency of caches are described in Section 9.

An issue of importance when caching files is the size of the chunks to be cached. Most systems cache pieces of files. This is appropriate when only parts of a file are read. Coda [26] and early versions of the Andrew File System [12] support whole file caching, in which the entire file is transferred to the client's workstation when opened. Files that are modified are copied back when closed. Files remain cached on the workstation between opens so that a subsequent open does not require the file to be fetched again. Approaches such as whole file caching work well on networks with high latency, and this is important in a geographically large system. But, whole file caching can be expensive if an application only wants to access a small part of a large file. Another problem is that it is difficult for diskless workstations to support whole file caching for large files. Because of the range in capabilities of the computers and communication channels that make up a distributed system, multiple file access mechanisms should be supported.

7 Replication

Replication is an important tool for building scalable distributed systems. Its use in naming, authentication, and file services reduces the load on individual servers and improves the reliability and availability of the services as a whole. The issues of importance for replication are the placement of the replicas and the mechanisms by which they are kept consistent.

7.1 Placement of Replicas

The placement of replicas in a distributed system depends on the purpose for replicating the resource. If a service is being replicated to improve the availability of the service in the face of network partitions, or if it is being replicated to reduce the network delays when the service is accessed, then the replicas should be scattered across the system. Replicas should be located so that a network partition will not make the service unavailable to a significant number of users.

If the majority of users are local, and if the service is being replicated to improve the reliability of the service, to improve its availability in the face of server failure, or to spread the load across multiple servers, then replicas may be placed near one another. The placement of replicas affects the choice of the mechanism that maintains the consistency of replicas.

7.2 Consistency

A replicated object can logically be thought of as a single object. If a change is made to the object, the change should be visible to everyone. At a particular point in time, a set of replicas is said to be *consistent* if the value of the object is the same for all readers. The following are some of the approaches that have been used to maintain the consistency of replicas in distributed systems.

Some systems only support replication of read-only information. Andrew and Athena take this approach for replicating system binaries. Because such files change infrequently, and because they can't be changed by normal users, external mechanisms are used to keep the replicas consistent.

Closely related to the read-only approach is replication of immutable information. This approach is used by the Amoeba file server. Files in Amoeba are immutable, and as a result, they can be replicated at will. Changes to files are made by creating new files, then changing the directory so that the new version of the file will be found.

A less restrictive alternative is to allow updates, but to require that updates are sent to all replicas. The limitations of this approach are that updates can only take place when all of the replicas are available, thus reducing the availability of the system for write operations. This mechanism also requires an absolute ordering on updates so that inconsistencies do not result if updates are received by replicas in different orders. A final difficulty is that a client might fail during an update, resulting in its receipt by only some of the replicas.

In primary-site replication, all updates are directed to a primary replica which then forwards the updates to the others. Updates may be forwarded individually, as in Echo [11], or the whole database might be periodically downloaded by the replicas as in Kerberos [29] and the Berkeley Internet Domain Naming system (BIND) [31], an implementation of IDNS [15]. The advantage of the primary-site approach is that the ordering of updates is determined by the order in which they are received at the primary site, and updates only require the availability of the primary site. A disadvantage of the primary-site approach is that the availability of updates still depends on a single server, though some systems select a new primary site if the existing primary goes down. An additional disadvantage applies if changes are distributed periodically: the updates are delayed until the next update cycle.

For some applications, absolute consistency is often not an overriding concern. Some delay in propagating a change is often acceptable, especially if one can tell when a response is incorrect. This observation was exploited by Grapevine, allowing it to guarantee only loose consistency. With loose consistency, it is guaranteed that replicas will *eventually* contain identical data. Updates are allowed even when the network is partitioned or servers are down. Updates are sent to any replica, and that replica forwards the update to the others as they become available. If conflicting updates are received by different replicas in different orders, timestamps indicate the order in which they are to be applied. The disadvantage of loose consistency is that there is no guarantee that a query returns the most recent data. With name servers, however, it is often possible to check whether the response was correct at the time it is used.

Maintaining a consistent view of replicated data does not require that all replicas are up-to-date. It only requires that the up-to-date information is always visible to the users of the data. In the mechanisms described so far, updates eventually make it to every replica. In quorum-consensus, or voting [9], updates may be sent to a subset replicas. A consistent view is maintained by requiring that all reads are directed to at least one replica that is up-to-date. This is accomplished by assigning votes to each replica, by selecting two numbers, a read-quorum and write-quorum, such that the read-quorum plus the write-quorum exceeds the total number of votes, and by requiring that reads and writes are directed to a sufficient number of replicas to collect enough votes to satisfy the quorum. This guarantees that the set of replicas read will intersect with the set written during the most recent update. Timestamps or version numbers stored with each replica allow the client to determine which data is most recent.

8 Distribution

Distribution allows the information maintained by a distributed service to be spread across multiple servers. This is important for several reasons: there may be too much information to fit on a single server; it reduces the number of requests to be handed by each server; it allows administration of parts of a service to be assigned to different individuals; and it allows information that is used more frequently in one part of a network to be maintained nearby.

This section will describe the use of distribution in naming, authentication, and file services. Some of the issues of importance for distribution are the placement of the servers and the mechanisms by which the client finds the server with the desired information.

8.1 Placement of Servers

Distributed systems exhibit locality. Certain pieces of information are more likely to be accessed by users in one part of a network than by users in another. Information should be distributed to servers that are near the users that will most frequently access the information. For example, a user's files could be assigned to a file server on same subnet as the workstation usually used by that user. Similarly, the names maintained by name servers can be assigned so that names for nearby objects can be obtained from local name servers. In addition to reducing network traffic, such assignments improve reliability, since it is less likely that a network partition will make a local server inaccessible. In any case, it is desirable to avoid the need to contact a name server across the country in order to find a resource in the next room.

By assigning information is to servers along administrative lines, an organization can avoid dependence on others. When distributed along organizational lines, objects maintained by an organization are often said to be within a particular *domain* (IDNS), or a *cell* (Andrew). Kerberos uses the term *realm* to describe the unit of distribution when there exists an explicit trust relationship between the server and the principals assigned to it.

8.2 Finding the Right Server

The difficulty with distribution lies in the distribution function: the client must determine which server contains the requested information. Hierarchical name spaces make the task easier since names with common prefixes are often stored together[2], but it is still necessary to identify the server maintaining that part of the name space. The methods most frequently used are mounts, broadcast and domain-based queries.

Sun's Network File System [25], Locus [32], and Plan 9 [24] use a mount table to identify the server on which a a named object resides. The system maintains a table mapping name prefixes to servers. When an object is referenced, the name is looked up in the mount table, and the request is forwarded to the appropriate server. In NFS, the table can be different on different systems meaning that the same name might refer to different objects on differ-

[2]In this discussion, prefix means the most significant part of the name. For file names, or for names in DEC's Global Naming System, it is the prefix. For domain names it is really the suffix.

ent systems. Locus supports a uniform name space by keeping the mount table the same on all systems. In Plan 9, the table is maintained on a per-process basis.

Broadcast is used by Sprite [22] to identify the server on which a particular file can be found. The client broadcasts a request, and the server with the file replies. The reply includes the prefix for the files maintained by the server. This prefix is cached so that subsequent requests for files with the same prefix can be sent directly to that server. As discussed in Section 6.1, this approach does not scale beyond a local network. In fact, most of the systems that use this approach provide a secondary name resolution mechanism to be used when a broadcast goes unanswered.

Distribution in Grapevine, IDNS, and X.500 [5] is domain-based. Like the other techniques described, the distribution function in domain-based naming is based on a prefix of the name to be resolved. Names are divided into multiple components. One component specifies the name to be resolved by a particular name server and the others specify the server that is to resolve the name. For example, names in Grapevine consist of a registry and a name within the registry. A name of the form NEUMAN.UW would be stored in the UW registry under the name NEUMAN. IDNS and DEC's Global Naming System both support variable depth names. In these systems, the point at which the name and the domain are separated can vary. In IDNS, the last components of the name specify the domain, and the first components specify the name within that domain. For example, VENERA.ISI.EDU is registered in the name server for the ISI.EDU domain.

To find a name server containing information for a given domain or registry, a client sends a request to the local name server. The local name server sends back an answer, or information redirecting the query to another name server. With the two level name space supported by Grapevine, only two queries are required: one to find the server for a given registry, and one to resolve the name. The server for a given registry is found by looking up the name in the GV registry which is replicated on every Grapevine server.

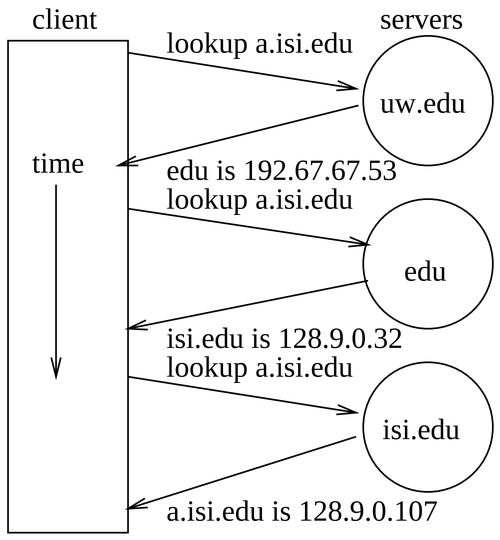

Figure 2: Resolving a domain-based name

The resolution of a name with a variable number of components is shown in figure 2. The client sends a request to its local server requesting resolution of the host name A.ISI.EDU. That server returns the name and address of the EDU server. The client repeats its request to the EDU server which responds with the name and address for the ISI.EDU server. The process repeats, with successively longer prefixes, until a server (in this case ISI.EDU) returns the address for the requested host. The client caches intermediate responses mapping prefixes to servers so that subsequent requests can be handled with fewer messages.

Domain-based distribution of names scales well. As the system grows and queries become more frequent, additional replicas of frequently queried registries or domains can be added. Grapevine's two level name space, though, places limits on scalability. Since every name server must maintain the GV registry, and because the size of this registry grows linearly with the total number of name servers, the total number of name servers that can be supported is limited. Clearinghouse, a production version of Grapevine, addressed this problem by supporting a three level name space. This allows the name service to scale to a larger number of names, but it still eventually reaches a limit due to the size of the root or second-level registries.

The primary disadvantage of domain-based distribution of names is that it can take many queries to resolve a single name. Fortunately, with the appropriate use of caching, many of these additional queries can be eliminated.

Domain-based distribution can also be used for authentication. In Kerberos, principals may be registered in multiple *realms*. This allows an organization to set up its own Kerberos server, eliminating the need for global trust. The server's realm is used to determine the sequence of authentication servers to be contacted. If a client in one Kerberos realm wishes to use a server in another, it must first authenticate itself to the authentication server in the server's realm using the AS in its own realm.

Figure 3 shows multiple-hop cross realm authentication in Kerberos. In this figure, the numbers on messages loosely correspond to those in figure 1. **3+1** is a message authenticating the client to the next Kerberos server, accompanied with a request for credentials. Message **2T** is the same as message **2** in figure 1, except that instead of being encrypted in the client's key, the response is encrypted in the session key from the ticket sent to the AS in the previous message.

The initial version of Kerberos only supported single-hop cross-realm authentication. This required that each realm had to know about every other realm with which it was to communicate. This limitation does not exist in Version 5 of Kerberos, or in DEC's global authentication system [2]. With multiple-hop cross-realm authentication, what is known after a client has been authenticated may be as weak as "the local AS claims that a remote AS claims that another AS has authenticated the client as A". To allow the end server to make an informed decision, it is necessary that it knows the complete list of realms that took part in authenticating the client. In global authentication this information is part of the name of the authenticated principal. The principal's name is constructed by concatenating the names of the links that were traversed at each step in the authentication process. In Version 5 of Kerberos a list of the transited realms is included in the credentials. Both protocols allow intermediaries to be skipped. This not only speeds up the authentication process, but it can make it more secure.

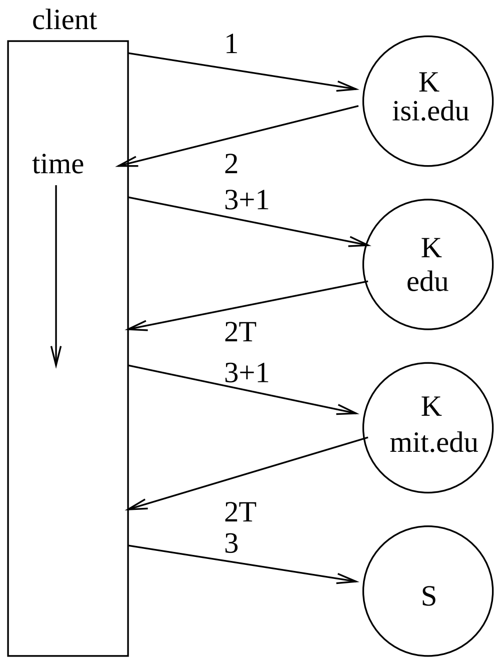

Figure 3: Cross-Realm Authentication

9 Caching

Caching is another important tool for building scalable systems. It is a form of replication, and like replication, two issues of importance are the placement of caches, and how they are kept consistent. The difference between replication and caching is that cached data is a temporary replica. Updates need not be propagated to caches. Instead, consistency is maintained by invalidating cached data when consistency can not be guaranteed.

9.1 Placement of Caches

Caching can occur in multiple places. Caching is usually performed by the client, eliminating repeated requests to network services. Caching can also take place on the servers implementing those services. For example, in addition to caching on the workstation, Sprite [22] caches blocks in the memory of the file server. Reading a file from the memory cached copy on the file server is often faster than reading it from the client's local disk. The additional caching on the file server can improve performance because the file server might have more memory than the client and because many of the blocks cached on the file server might be read by multiple clients.

Caching in multiple places is also useful for name servers. Most name servers unable to answer to a query will return the address of the name server sharing the longest prefix in common with the name being resolved. In many cases, that might be the name server for the root. BIND [31] may be configured so that a local name server makes queries on behalf of the client and caches the response (and any intermediate responses) for use by other local clients. This additional level of caching allows higher levels of the naming hierarchy to be bypassed, even if the client does not know the server for the desired prefix. For example, if a client in the CS.WASHINGTON.EDU domain wishes to resolve VENERA.ISI.EDU, the local name server could make the query to the root name server on behalf of the client, return the address for that name server, and cache it. If a second host in the CS.WASHINGTON.EDU domain wanted to resolve A.ISI.EDU, the local name server would then be able to return the address of the correct name server without an additional query to root. For this to work, clients must be willing to first look for information locally instead of initially asking the root name server.

9.2 Cache Consistency

As was the case with replication, there are many techniques that are used to maintain the consistency of caches. The four most common approaches used to keep caches consistent in distributed systems are timeouts, check-on-use (hints), callbacks, and leases.

Timeouts are used to maintain cache consistency in IDNS, DEC's name service, Prospero, and a number of other systems. In these systems, responses from servers always include the time for which they may be considered valid (time-to-live or TTL). The TTL can vary from item to item. It will usually be long for infrequently changing information, and shorter for information expected to change. Clients can cache information until the TTL expires. When information changes, the TTL sets an upper bound on the time required before everyone will be using the new information. If a change is expected in advance, the TTL can be reduced so that the change will take effect quickly.

If it is possible to tell when incorrect information has been obtained, cached entries can be treated as hints. Hints don't have to be kept consistent; if out of date, that fact will be detected when the data is used and the entry can be flushed. Grapevine and QuickSilver [4] both use hints. Hints are useful in naming systems if an objects identifier can be stored along with the object itself. The cached data tells where the object can be found, but if the object has moved, that fact will be apparent when the client attempts to retrieve it. By treating cached data as hints, it may be used until a change is detected, avoiding the need for more complicated mechanisms to keep the cache consistent.

In some systems, the only way to check the validity of cached data is to go back to the server that provided the information originally. For small amounts of data, the cost of doing so is the same as if the information were not cached at all. For larger amounts of data, the check takes significantly less time than transferring the data itself. The Andrew File System [12] originally used this form of check-on-use to decide if a locally cached copy of a file could be used. Experience showed that the checks were the primary bottleneck, and that, in the common case, the files were unchanged. For this reason, the next implementation (and subsequently Coda) used callbacks. When a file is cached, the file server adds the caching site to a list stored along with the file. If the file changes, a message is sent to all sites with copies telling them that the cached copy is no longer valid. By requiring that clients check the validity of files when they reboot (or if contact with the file server has been lost), problems due to lost callbacks can be minimized.

Leases [10] are similar to callbacks, but there are several important differences. A lease eventually expires, and a server granting a lease guarantees that it will not make a change during the period the lease is valid unless it first gets approval from the lease holder. A client holding a lease can cache the data to which the lease applies for the term of the lease, or until it authorizes the server to break the lease. Tradeoffs similar to those for choosing a TTL apply to the selection of the term of a lease.

10 The User's View

Many mechanisms are used to help the system deal with scale. Unfortunately, the effect of scale on the user has received relatively little attention. The user has finite mental capacity. As the number of computers in a system grows, as the system expands geographically, and as it begins to cross administrative boundaries, the potential exists for the size of the system to overwhelm the user.

Mechanisms are needed to allow objects and resources that are of interest to be organized in a manner that allows them to be easily found again. One doesn't want them quickly lost in a sea of objects and resources in which there is little interest. It is also important that the user be able to identify additional objects and resources of potential interest.

Traditional systems such as Andrew, Locus and Sprite support a uniform global name space which uniquely names all objects. This approach allows simple sharing of names, and it has the advantage that it is no harder for users to understand than the systems they previously used. Unfortunately, as systems cross administrative boundaries it becomes difficult to obtain agreement on what should appear where in the name space. The solution is that the names of sites appear at the top level and each site names its own objects. Unfortunately, this results in related information being scattered across the global name space and users don't know where to look.

Even with a familiar system model, the number of objects and resources that are available can overwhelm the user. For this reason, mechanisms are needed to help the user organize information and to reduce the amount of information that has to be dealt with. The solution is to allow individual users to customize their name space so that they see only the objects that are of interest. This approach is taken in Plan 9 [24], Prospero [20], Tilde [7], and QuickSilver [4]. Naming in these systems is often described as user-centered though, with the exception of Prospero, it might better be described as user-exclusive; an object must be added to the user's name space before it can be named. In Prospero, it is expected that most objects start out in a user's name space, but with lengthy names. When a user expresses an interest in an object, a link is added to the name space, bringing the object closer to the center (root).

An objection to user-centered naming is that the same name can refer to different objects when used by different users. To address this objection, Prospero supports closure: every object in the system has an associated name space. Names are normally resolved within the name space associated with the object in which the name is found.

A few systems have looked at mechanisms for identifying objects that are needed when the object's full name is not known. Profile [23] supports attribute-based naming. In attribute-based naming, the user specifies known attributes of an object instead of its name. To be used in place of a name, enough attributes must specified to uniquely identify the object. In order to scale, information must be distributed across multiple name servers. In Profile, each user has a working set of name servers, and each is contacted. Responses from a name server may require further resolution (perhaps by a different server). This successive resolution of links is similar to the mechanism used to resolve names in traditional distributed systems. The key difference is that the links do not necessarily form a hierarchy.

Alternative approaches are being examined by the Resource Discovery Project at the University of Colorado. These approaches use information already available over the network. In one approach, resource discovery agents [28] collect and share information with other agents scattered across the system. A user wishing to find a resource asks one of these agents, and the agents route queries among themselves, exploiting the semantics of the query to limit the activity that must take place. If the resource is found, a response is returned to the client.

Prospero takes a slightly different approach to identifying objects of interest. Tools are provided to allow users to customize and organize their views of the system. Prospero supports a user-centered name space and closure. Naming of all objects and resources is handled through a uid-based directory service. Name spaces may overlap, and cycles are allowed. The Prospero directory service supports filters and union links. A filter is a program that can modify the results of a directory query when the path for the queried directory passes through the filtered link. A union link allows the results of a (possibly filtered) query to be merged with the contents of the directory containing the link.

The nature of the directory service, and its support for filters and union links allows users to organize their own objects and those of others in many ways. The ability for objects to have multiple names makes it much easier to find things; one can look for an object using the organization that best fits the information that is known. Users and organizations set up directories within which they organize objects and they make these directories available to others. It is through this sharing that users find new objects.

11 Building Scalable Systems

This section presents suggestions for building scalable systems. These suggestions are discussed in greater detail in the paper and are presented here in a form that can be used as a guide. The hints are broken into groups corresponding to the primary techniques of replication, distribution and caching.

When building systems it is important to consider factors other than scalability. An excellent collection of hints on the general design of computer systems is presented by Lampson in [13].

11.1 Replication

Replicate important resources. Replication increases availability and allows requests to be spread across multiple servers, thus reducing the load on each.

Distribute the replicas. Placing replicas in different parts of the network improves availability during network partitions. By placing at least one replica in any area with frequent requests, those requests can be directed to a local replica reducing the load on the network and minimizing response time.

Use loose consistency. Absolute consistency doesn't scale well. By using loose consistency the cost of updates can be reduced, while changes are guaranteed to *eventually* make it to each replica. In systems that use loose consistency it is desirable to be able to detect out-of-date information at the time it is used.

11.2 Distribution

Distribute across multiple servers. Distributing data across multiple servers decreases the size of the database that must be maintained by each server, reducing the time needed to search the database. Distribution also spreads the load across the servers reducing the number of requests that are handled by each.

Distribute evenly. The greatest impact on scalability will be felt if requests can be distributed to servers in proportion to their power. With an uneven distribution, one server may be idle while others are overloaded.

Exploit locality. Network traffic and latency can be reduced if data are assigned to servers close to the location from which they are most frequently used. The Internet Domain Naming System does this. Each site maintains the information for its own hosts in its own servers. Most queries to a name server are for local hosts. As a result, most queries never leave the local network.

Bypass upper levels of hierarchies. In hierarchically organized systems, just about everyone needs information from the root. If cached copies are available from subordinate servers, the upper levels can be bypassed. In some cases, it might be desirable for a server to answer queries only from its immediate subordinates, and to let the subordinates make the responses available to their subordinates.

11.3 Caching

Cache frequently accessed data. Caching decreases the load on servers and the network. Cached information can be accessed more quickly than if a new request is made.

Consider access patterns when caching. The amount of data normally referenced together, the ratio of reads to writes, the likelihood of conflicts, the number of simultaneous users, and other factors will affect the choice of caching mechanisms. For example, if files are normally read from start to finish, caching the entire file might be more efficient than caching blocks. If conflicts between readers and writers are rare, using callbacks to maintain consistency might reduce requests. The ability to detect invalid data on use allows cached data to be used until such a condition is detected.

Cache timeout. By associating a time-to-live (TTL) with cached data an upper bound can be placed on the time required for changes to be observed. This is useful when only eventual consistency is required, or as a backup to other cache consistency mechanisms. The TTL should be chosen by the server holding the authoritative copy. If a change is expected, the TTL can be decreased accordingly.

Cache at multiple levels. Additional levels of caching often reduce the number of requests to the next level. For example, if a name server handling requests for a local network caches information from the root name servers, it can request it once, then answer local requests for that information instead of requiring each client to request it separately. Similarly, caching on file servers allows a block to be read (and cached) by multiple clients, but only requires one disk access.

Look first locally. By looking first for nearby copies of data before contacting central servers, the load on central servers can be reduced. For example, if a name is not available from a cache in the local system, contact a name server on the local network before contacting a distant name server. Even if it is not the authority for the name to be resolved, the local name server may possess information allowing the root name server to be bypassed.

The more extensively something is shared, the less frequently it should be changed. When an extensively shared object is changed, a large number of cached copies become invalid, and each must be refreshed. A system should be organized so that extensively shared data is relatively stable. A hierarchical name space exhibits this property. Most changes occur at the leaves of the hierarchy. Upper levels rarely change.

11.4 General

Shed load, but not too much. When computation can be done as easily by the client as the server, it is often best to leave it to the client. However, if allowing the client to perform the computation requires the return of a significantly greater amount of information (as might be the case for a database query), it is more appropriate for the server to do the computation. Additionally, if the result can be cached by the server, and later provided to others, it is appropriate to do the computation on the server, especially if the computation requires contacting additional servers.

Avoid global broadcast. Broadcast does not scale well. It requires all systems to process a message whether or not they need to. Multicast is acceptable, but groups should include only those servers that need to receive the message.

Support multiple access mechanisms. Applications place varying requirements on access mechanisms. What is best for one application might not be so for another. Changing communication parameters can also affect the choice of mechanism. Multiple mechanisms should be supported when accessing objects and resources. The client should choose the method based on the prevailing conditions.

Keep the user in mind. Many mechanisms are used to help the system deal with scale. The mechanisms that are used should not make the system more difficult to understand. Even with a familiar system model, the number of available objects and resources can overwhelm the user. Large systems require mechanisms that reduce the amount of information to be processed and remembered by the user. These mechanisms should not hide information that might be of interest.

11.5 Evaluating Scalable Systems

There are many questions to be asked when evaluating the scalability of a distributed system. This subsection lists some of the questions that are important. It does not provide a formula that yields a number. In fact, different systems scale in different ways. One system may scale better administratively, while another scales better numerically. There are so many unknowns that affect scaling that experience is often the only true test of a system's ability to scale.

The first set of questions concerns the use of the system. How will the frequency of queries grow as the system grows? What percentage of those queries must be handled by central servers? How many replicas of the central servers are there, is this enough, can more be added, what problems are introduced by doing so, and are there any bottlenecks?

The next set of questions concerns the data that must be maintained. How does the size of the databases handled by the individual servers grow? How does this affect query time? How often will information change? What update mechanism is used, and how does it scale? How will an update affect the frequency of queries? Will caches be invalidated, and will this result in a sudden increase in requests as caches are refreshed?

The final question concerns the administrative component of scale. Many systems require a single authority that makes final decisions concerning the system. Is this required, and is it practical in the environment for which the system will be used?

Asking these questions will point out some of the problem areas in a system. This is not a complete list. It is entirely possible that important factors not addressed will cause a system to stop scaling even earlier.

12 Conclusions

This paper examined the problems that arise as systems scale. It has used examples from many systems to demonstrate the problems and their solutions. The systems mentioned are not the only systems for which scale was a factor in their design; they simply provided the most readily available examples for the mechanisms that were discussed. The discussion has necessarily taken a narrow view of the systems that were discussed, examining individual subsystems instead of the systems as a whole. The effects of scale, however, are felt throughout the system.

This paper has shown how scale affects large systems. Scale can be broken into its numerical, geographical, and administrative components. Each component introduces its own problems, and the solutions employed by a number of systems were discussed. The three techniques used repeatedly to handle scale are replication, distribution, and caching.

A collection of suggestions for designing scalable systems was presented in Section 11. These suggestions expand upon the three primary techniques and suggest additional ways in which they can be applied. It is hoped that these hints will help system designers address scale in the design of future distributed systems.

Acknowledgments

I would like to thank Brian Bershad, Robert Cooper, Barbara Gordon, Bruce Gordon, Terry Gray, Andrew Herbert, Richard Ladner, Ed Lazowska, Hank Levy, Mary Ann G. Neuman, David Notkin, John Zahorjan, and the anonymous referees who commented on earlier drafts of this paper.

Appendix: Systems Designed with Scale in Mind

Scalability is included among the design criteria of a number of recent systems. The degree to which these systems scale ranges from a collection of computers on a local area network, to computers distributed across the entire Internet. This appendix describes some of these systems, states the degree to which each system is intended to scale, and lists some of the ways in which the system addresses the problems of scale. Table 1 summarizes this information in tabular form.

Amoeba, developed at Vrije Universiteit and CWI in Amsterdam, is a capability-based distributed operating system which has been used across long haul networks spanning multiple organizations. Objects are referenced by capabilities which include identifiers for the server and object, and access rights for the object. The capabilities provide both a distributed naming and authorization mechanism. [16, 30]

The **Andrew** system, developed at Carnegie-Mellon University, runs on thousands of computers distributed across the university campus. Its most notable component is the **Andrew File System** which now ties together file systems at sites distributed across the United States. **Coda** is a follow-on to Andrew, improving availability, especially in the face of network partitions. [12, 26]

MIT's **Project Athena** is a system built from thousands of computers distributed across campus. Distributed services provide authentication, naming, filing, printing, mail and administrative functions. Kerberos was developed as part of Project Athena. [6]

Dash, under development at Berkeley, is a distributed operating system designed for use across large networks exhibiting a range of transmission characteristics. Dash is notable for exposing these characteristics by allowing the application to require that the connection meet certain requirements and returning an error if those requirements cannot be met. [1]

DEC's Global Naming System, developed at at DEC's Systems Research Center, was designed to support naming in large networks spanning multiple organizations. It is notable for the attention paid to reorganization of the name space as independent name spaces are merged, or as the external relationship between organizations change (e.g. mergers or acquisitions). **Echo** is a distributed file system supporting consistent replication of local partitions, but with partitions tied together using the loosely consistent Global Naming System. DEC's **Global Authentication System** is notable for the fact that a principal's name is not absolute, but is instead determined by the sequence of authentication servers used to authenticate the principal. [2, 11, 14]

Grapevine was one of the earliest distributed systems designed to scale to a large network. It was developed at Xerox PARC to support electronic mail, to provide a name service for the location of network services, and to support simple password-based authentication on a world-wide network connecting Xerox sites. [3, 27]

The **Heterogeneous Computer Systems Project** at the University of Washington demonstrated that a single interface could be used to communicate with systems using different underlying protocols and data representations. This is important for large systems when it is not practical to dictate the choice of hardware and software across multiple sites, or when the underlying mechanisms have different strengths and weaknesses. [21]

The **Internet Domain Naming System** (IDNS) is a distributed name service, running on the Internet, supporting the translation of host names to Internet addresses and mail forwarders. Each organization maintains replicated servers supporting the translation of names for its own part of the name space. [15, 31]

Kerberos is an encryption-based network authentication system, developed by MIT's Project Athena, which supports authentication of users both locally, and across organizational boundaries. [29]

Locus, developed at the UCLA, was designed to run on systems distributed across a local-area network. Locus is notable as one of the earliest distributed systems to support a uniform view of the file system across all nodes in the system. [32]

SUN's **Network File System** supports transparent access to files stored on remote hosts. Files are named independently on each host. Before a remote file can be accessed, the remote file system containing the file must be mounted on the local system, establishing a mapping of part of the local file name space to files on the remote system. The NFS server maintains very little information (state) about the clients that use it. [25]

Plan 9 from Bell Labs, intended for use by a large corporation, supports a process-centered[3] name space, allowing users to incorporate into their name space those parts of the global system that are useful. [24]

Profile, developed at the University of Arizona, is an attribute-based name service that maps possibly incomplete information about coarse-grained objects on a large network to the object(s) matching that information. [23]

Prospero, developed at the University of Washington, runs on systems distributed across the Internet. It supports an object-centered view of the entire system, allowing users to define their own virtual system by specifying the pieces of the global system that are of interest. Prospero's support for closure resolves the problems caused by the use of multiple name spaces. [20]

QuickSilver, developed at IBM's Almaden Research Center, is notable for its proposed use of a user-centered[3] name space. In a system spanning a large, multi-national corporation, such a name space allows users to see only those parts of the system that concern them. [4]

Sprite, a network operating system developed at Berkeley, was designed for use across a local area network. Its file system is notable for its use of caching on both the client and the server to improve performance, and for its use of prefix tables to distribute requests to the correct file server. [22]

The **Tilde** naming system, developed at Purdue, supports process-centered[3] naming. This allows one to specify, on a per-process basis, how names will map to pieces of the global system. This ability provides applications with the advantages of a global name space for those file names that should be resolved globally, while allowing parts of the name space to be specified locally for file names which would be better resolved to local files. [7]

X.500 is an ISO standard describing a distributed directory service that is designed to store information about users, organizations, resources, and similar entities worldwide. Scalability is addressed in largely the same manner as in the Internet Domain Name Service. [5]

[3] Perhaps better described as process- or user-exclusive since objects must first be added to the user's name space before they can be named.

		Intended Environment			The Methods Used		
System	Service	# nodes	geographic	administrative	replication	distribution	caching
Amoeba	general	∞	wide-area	multiple organizations	immutable	capabilities	yes
Andrew	file system	10,000	wide-area	multiple organizations	read-only	cell/volume	blocks
Athena	general	10,000	campus	university	service	clusters	yes
Coda	file system	10,000	global	multiple organizations	optimistic	volume	whole file
Dash	general	∞	wide-area	multiple organizations	yes	yes	yes
DEC's Global naming		∞	global	multiple organizations	loose	directories	time-to-live
DEC's Global authentication		∞	global	multiple organizations	loose	directories	-
Echo	file system	∞	wide-area	multiple organizations	loose/primary	volume	yes
Grapevine	general	2,000	company	multiple departments	loose	registry	yes
HCS	general	-	wide-area	multiple organizations	-	yes	-
IDNS	naming	∞	global	multiple organizations	primary	domain	yes
Kerberos	authentication	∞	global	multiple organizations	primary	realm	tickets
Locus	general	100	local	department	primary	mount	yes
NFS	file system	-	local	single organization	no	mount	blocks
Plan 9	general	10,000	company	multiple depatments	no	mount	no
Profile	naming	∞	wide-area	multiple organizations	information	principal	client-managed
Prospero	naming	∞	global	multiple organizations	yes	uid	yes
Quicksilver	file system	10,000	company	multiple departments	no	prefix	immutable
Sprite	file system	100	local	department	read-only	prefix	client&server
Tilde	naming	100	local	single organization	no	trees	yes
X.500	naming	∞	global	multiple organizations	yes	yes	yes

Table 1: Important distributed systems and the methods they use to handle scale

References

[1] David P. Anderson and Domenico Ferrari. The Dash project: An overview. Technical Report 88/405, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley, August 1988.

[2] Andrew D. Birrell, Butler W. Lampson, Roger M. Needham, and Michael D. Schroeder. A global authentication service without global trust. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 223–230, April 1986.

[3] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[4] Luis-Felipe Cabrera and Jim Wyllie. QuickSilver distributed file services: An architecture for horizontal growth. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 23–27, March 1988. Also IBM Research Report RJ 5578, April 1987.

[5] CCITT. Recommendation X.500: The Directory, December 1988.

[6] George A. Champine, Daniel E. Geer Jr., and William N. Ruh. Project athena as a distributed computer system. *IEEE Computer*, 23(9):40–51, September 1990.

[7] Douglas Comer, Ralph E. Droms, and Thomas P. Murtagh. An experimental implementation of the Tilde naming system. *Computing Systems*, 4(3):487–515, Fall 1990.

[8] Robert J. Fowler. *Decentralized Object Finding Using Forwarding Addresses*. PhD thesis, University of Washington, December 1985. Department of Computer Science technical report 85-12-1.

[9] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating System Principles*, pages 150–159, December 1979. Pacific Grove, California.

[10] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, December 1989.

[11] Andy Hisgen, Andrew Birrell, Timothy Mann, Michael Schroeder, and Garret Swart. Availability and consistency tradeoffs in the Echo distributed file system. In *Proceedings of the 2nd IEEE Workshop on Workstation Operating Systems*, pages 49–54, September 1989.

[12] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[13] Butler W. Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 33–48, 1983.

[14] Butler W. Lampson. Designing a global name service. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, August 1985.

[15] Paul Mockapetris. Domain names - concepts and facilities. DARPA Internet RFC 1034, November 1987.

[16] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):289–299, 1986.

[17] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communication of the ACM*, 21(12):993–999, December 1978.

[18] B. Clifford Neuman. Issues of scale in large distributed operating systems. Generals Report, Department of Computer Science, University of Washington, May 1988.

[19] B. Clifford Neuman. Proxy-based authorization and accounting for distributed systems. Technical Report 91-02-01, Department of Computer Science and Engineering, University of Washington, March 1991.

[20] B. Clifford Neuman. The Prospero File System: A global file system based on the Virtual System Model. In *Proceedings of the Workshop on File Systems*, May 1992.

[21] David Notkin, Andrew P. Black, Edward D. Lazowska, Henry M. Levy, Jan Sanislo, and John Zahorjan. Interconnecting heterogeneous computer systems. *Communications of the ACM*, 31(3):258–273, March 1988.

[22] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, 21(2):23–35, February 1988.

[23] Larry L. Peterson. The Profile naming service. *ACM Transactions on Computer Systems*, 6(4):341–364, November 1988.

[24] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9: A distributed system. In *Proceedings of Spring 1991 EurOpen*, May 1991.

[25] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network File System. In *Proceedings of the Summer 1985 Usenix Conference*, pages 119–130, June 1985.

[26] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5):9–21, May 1990.

[27] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with Grapevine: The growth of a distributed system. *ACM Transactions on Computer Systems*, 2(1):3–23, February 1984.

[28] M. F. Schwartz. The networked resource discovery project. In *Proceedings of the IFIP XI World Congress*, pages 827–832, August 1989. San Francisco.

[29] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 Usenix Conference*, pages 191–201, February 1988. Dallas, Texas.

[30] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experience with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):47–63, December 1990.

[31] Douglas B. Terry, Mark Painter, David W. Riggle, and Songnian Zhou. The Berkeley internet domain server. In *Proceedings of the 1984 Usenix Summer Conference*, pages 23–31, June 1984.

[32] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The Locus distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–70, October 1983.