

Distributed System Services

This chapter deals with aspects of system services that are unique to distributed systems: security and authentication, distributed shared file access, and computational load balancing.

14.1 Authentication and Security

A major problem with computer communication and distributed systems is that of trust. How do you know who is really sending you those bits over the network? And what do you allow them to do on your system?

14.1.1 Authentication

In distributed systems, services are rendered in response to incoming messages. For example, a file server may be requested to disclose the contents of a file, or to delete a file. Therefore it is important that the server know for sure who the client is. Authentication deals with such verification of identity.

Identity is established by passwords

The simple solution is to send the user name and password with every request. The server can then verify that this is the correct password for this user, and is so, it will respect the request. The problem is that an eavesdropper can obtain the user's password by monitoring the traffic on the network. Encrypting the password doesn't help at all: the eavesdropper can simply copy the encrypted password, without even knowing what the original password was!

Kerberos uses the password for encryption

The Kerberos authentication service is based on a secure authentication server (one that is locked in a room and nobody can temper with it), and on encryption. The server knows all passwords, but they are never transmitted across the network. Instead, they are used to generate encryption keys.

Background: encryption

Encryption deals with hiding the content of messages, so that only the intended recipient can read them. The idea is to apply some transformation on the text of the message. This transformation is guided by a secret key. The recipient also knows the key, and can therefore apply the inverse transformation. Eavesdroppers can obtain the transformed message, but don't know how to invert the transformation.

The login sequence is more or less as follows:

1. The client workstation where the user is trying to log in sends the user name U to the server.
2. The Kerberos server does the following:
 - (a) It looks up the user's password p , and uses a one-way function to create an encryption key K_p from it. One way functions are functions that it is hard to reverse, meaning that it is easy to compute K_p from p , but virtually impossible to compute p from K_p .
 - (b) It generates a new session key K_s for this login session.
 - (c) It bundles the session key with the user name: $\{U, K_s\}$.
 - (d) It uses its own secret encryption key K_k to encrypt this. We express such encryption by the notation $\{U, K_s\}_{K_k}$. Note that this is something that nobody can forge, because it is encrypted using the server's key.
 - (e) It bundles the session key with the created unforgeable ticket, creating $\{K_s, \{U, K_s\}_{K_k}\}$.
 - (f) Finally, the whole thing is encrypted using the user-key that was generated from the user's password, leading to $\{K_s, \{U, K_s\}_{K_k}\}_{K_p}$. This is sent back to the client.
3. The client does the following steps:
 - (a) It prompts the user for his password p , immediately computes K_p , and erases the password. Thus the password only exists in the client's memory for a short time.
 - (b) Using K_p , the client decrypts the message it got from the server, and obtains K_s and $\{U, K_s\}_{K_k}$.
 - (c) It erases the user key K_p .

The session key is used to get other keys

Now, the client can send authenticated requests to the server. Each request is composed of two parts: the request itself, R , encrypted using K_s , and the unforgeable ticket. Thus the message sent is $\{R_{K_s}, \{U, K_s\}_{K_k}\}$. When the server receives such a request, it decrypts the ticket using its secret key K_k , and finds U and K_s . If this works, the server knows that the request indeed came from user U , because only user U 's password could be used to decrypt the previous message and get the ticket. Then the server uses the session key K_s to decrypt the request itself. Thus even if someone spies on the network and manages to copy the ticket, they will not be able to use it because they cannot obtain the session key necessary to encrypt the actual requests.

However, an eavesdropper can copy the whole request message and retransmit it, causing it to be executed twice. Therefore the original session key K_s is not used for any real requests, and the Kerberos server does not provide any real services. All it does is to provide keys for *other* servers. Thus the only requests allowed are things like "give me a key for the file server". Kerberos will send the allocated key K_f to the client encrypted by K_s , and also send it to the file server. The client will then be able to use K_f to convince the file server of its identity, and to perform operations on files. An eavesdropper will be able to cause another key to be allocated, but will not be able to use it.

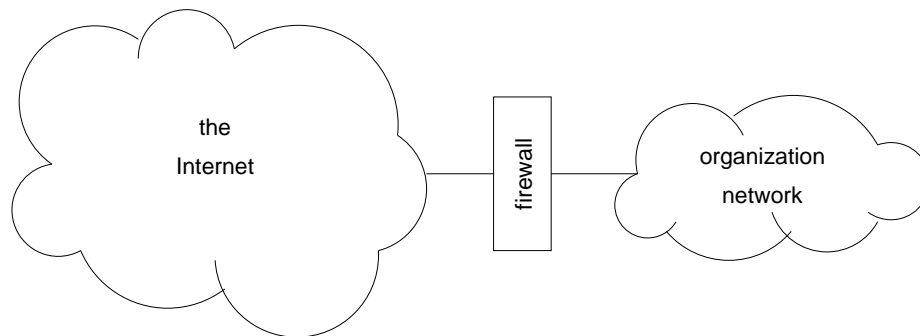
To read more: Kerberos is used in DCE, and is described in Tanenbaum's book on distributed operating systems [10, sect. 10.6].

14.1.2 Security

Kerberos can be used within the confines of a single organization, as it is based on a trusted third party: the authentication server. But on the Internet in general nobody trusts anyone. Therefore we need mechanisms to prevent malicious actions by intruders.

Security is administered by firewalls

An organization is typically connected to other organizations via a router. The router forwards outgoing communications from the internal network to the external Internet, and vice versa. This router is therefore ideally placed to control incoming packets, and stop those that look suspicious. Such a router that protects the internal network from bad things that may try to enter it is called a firewall.



The question, of course, is how to identify “bad things”. Simple firewalls are just packet filters: they filter out certain packets based on some simple criterion. Criteria are usually expressed as rules that make a decision based on three inputs: the source IP address, and destination IP address, and the service being requested. For example, there can be a rule saying that datagrams from any address to the mail server requesting mail service are allowed, but requests for any other service should be dropped. As another example, if the organization has experienced break-in attempts coming from a certain IP address, the firewall can be programmed to discard any future packets coming from that address.

As can be expected, such solutions are rather limited: they may often block packets that are perfectly benign, and on the other hand they may miss packets that are part of a complex attack. A more advanced technology for filtering is to use *stateful inspection*. In this case, the firewall actually follows that state of various connections, based on knowledge of the communication protocols being used. This makes it easier to specify rules, and also supports rules that are more precise in nature. For example, the firewall can be programmed to keep track of TCP connections. If an internal host creates a TCP connection to an external host, the data about the external host is retained. Thus incoming TCP datagrams belonging to a connection that was initiated by an internal host can be identified and forwarded to the host, whereas other TCP packets will be dropped.

14.2 Networked File Systems

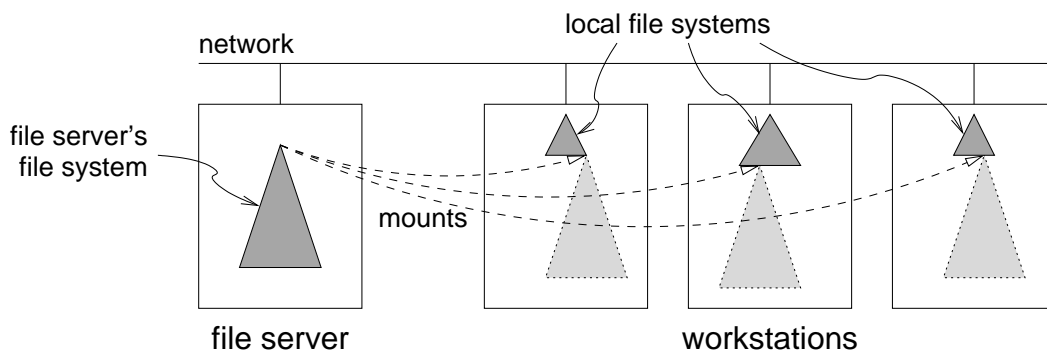
One of the most prominent examples of distributing the services of an operating system across a network is the use of networked file systems such as Sun’s NFS. In fact, NFS has become a de-facto industry standard for networked Unix workstations. Its design also provides a detailed example of the client-server paradigm.

To read more: Distributed file systems are described in detail in Silberschatz chap. 17 [9] and Tanenbaum chapter 5 [10]. NFS is described by Tanenbaum [10, Sect. 5.2.5]. It is also described in detail, with an emphasis on the protocols used, by Comer [6, Chap. 23 & 24].

Remote file systems are made available by mounting

The major feature provided by NFS is support for the creation of a uniform file-system name space. You can log on to any workstation, and always see your files in the same way. But your files are not really there — they actually reside on a file server in the machine room.

The way NFS supports the illusion of your files being available wherever you are is by mounting the file server's file system onto the local file system of the workstation you are using. One only needs to specify the remoteness when mounting; thereafter it is handled transparently during traversal of the directory tree. In the case of a file server the same file system is mounted on all other machines, but more diverse patterns are possible.



Exercise 205 *The uniformity you see is an illusion because actually the file systems on the workstations are not identical — only the part under the mount point. Can you think of an example of a part of the file system where differences will show up?*

Mounting a file system means that the root directory of the mounted file system is identified with a leaf directory in the file system on which it is mounted. For example, a directory called `/mnt/fs` on the local machine can be used to host the root of the file server's file system. When trying to access a file called `/mnt/fs/a/b/c`, the local file system will traverse the local root and `mnt` directories normally (as described in Section 5.2.1). Upon reaching the directory `/mnt/fs`, it will find that this is a *mount point* for a remote file system. It will then parse the rest of the path by sending requests to access `/a`, `/a/b`, and `/a/b/c` to the remote file server. Each such request is called a *lookup*.

Exercise 206 *What happens if the local directory serving as a mount point (`/mnt/fs` in the above example) is not empty?*

Exercise 207 *Can the client send the whole path (`/a/b/c` in the above example) at once, instead of one component at a time? What are the implications?*

NFS servers are stateless

We normally consider the file system to maintain the state of each file: whether it is open, where we are in the file, etc. When implemented on a remote server, such an approach implies that the server is *stateful*. Thus the server is cognizant of sessions, and can use this knowledge to perform various optimizations. However, such a design is also vulnerable to failures: if clients fail, server may be stuck with open files that are never cleaned up; if a server fails and comes up again, clients will lose their connections and their open files. In addition, a stateful server is less scalable because it needs to maintain state for every client.

The NFS design therefore uses *stateless* servers. The remote file server does not maintain any data about which client has opened what file, and where it is in the file.

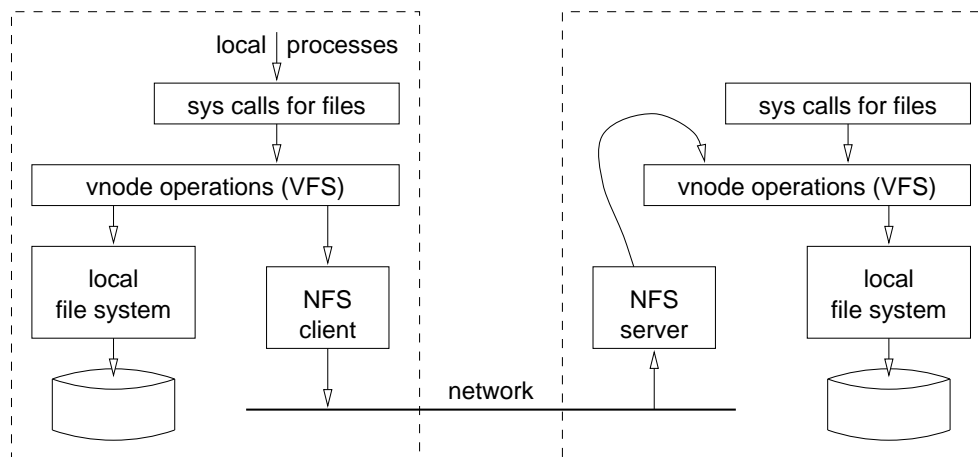
To interact with stateless servers, each operation must be self contained. In particular,

- There is no need for open and close operations at the server level. However, there is an open operation on the client, that parses the file's path name and retrieves a handle to it, as described below.
- Operations must be *idempotent*, meaning that if they are repeated, they produce the same result. For example, the system call to read the next 100 bytes in the file is not idempotent — if repeated, it will return a different set of 100 bytes each time. But the call to read the 100 bytes at offset 300 is idempotent, and will always return the same 100 bytes. The reason for this requirement is that a request may be sent twice by mistake due to networking difficulties, and this should not lead to wrong semantics.

The price of using stateless servers is a certain reduction in performance, because it is sometimes necessary to redo certain operations, and less optimizations are possible.

The virtual file system handles local and remote operations uniformly

Handling of local and remote accesses is mediated by the vfs (virtual file system). Each file or directory is represented by a *vnode* in the vfs; this is like a virtual inode. Mapping a path to a vnode is done by lookup on each component of the path, which may traverse multiple servers due to cascading mounts. The final vnode contains an indication of the server where the file resides, which may be local. If it is local, it is accessed using the normal (Unix) file system. If it is remote, the NFS client is invoked to do an RPC to the appropriate NFS server. That server injects a request to its vnode layer, where it is served locally, and then returns the result.



State is confined to the NFS client

For remote access, the NFS client contains a handle that allows direct access to the remote object. This handle is an *opaque object* created by the server: the client receives it when parsing a file or directory name, stores it, and sends it back to the server to identify the file or directory in subsequent operations. The content of the handle is only meaningful to the server, and is used to encode the object's ID. Note that using such handles does not violate the principle of stateless servers. The server generates handles, but does not keep track of them, and may revoke them based on a timestamp contained in the handle itself. If this happens a new handle has to be obtained. This is done transparently to the application.

The NFS client also keeps pertinent state information, such as the position in the file. In each access, the client sends the current position to the server, and receives the updated position together with the response.

The semantics are fuzzy due to caching

Both file attributes (inodes) and file data blocks are cached on client nodes to reduce communication and improve performance. The question is what happens if multiple clients access the same file at the same time. Desirable options are

Unix semantics: this approach is based on uniprocessor Unix systems, where a single image of the file is shared by all processes (that is, there is a single inode and single copy of each block in the buffer cache). Therefore modifications are immediately visible to all sharing processes.

Session semantics: in this approach each user gets a separate copy of the file data, that is only returned when the file is closed. It is used in the Andrew file system, and provides support for disconnected operation.

The semantics of NFS are somewhat less well-defined, being the result of implementation considerations. In particular, shared write access can lead to data loss. How-

ever, this is a problematic access pattern in any case. NFS does not provide any means to lock files and guarantee exclusive access.

Exercise 208 *Can you envision a scenario in which data is lost?*

14.3 Load Balancing

Computational servers provide cycles

Just like file servers that provide a service of storing data, so computational servers provide a service of hosting computations. They provide the CPU cycles and physical memory needed to perform the computation. The program code is provided by the client, and executed on the server.

Computation can be done on a server using mobile code, e.g. a Java application. But another method is to use process migration. In this approach, the process is started locally on the client. Later, if it turns out that the client cannot provide adequate resources, and that better performance would be obtained on a server, the process is moved to that server. In fact, in a network of connected PCs and workstations all of them can be both clients and servers: Whenever any machine has cycles (and memory) to spare, it can host computations from other overloaded machines.

Exercise 209 *Are there any restrictions on where processes can be migrated?*

Migration should be amortized by lots of computation

There are two major questions involved in migration for load balancing: which process to migrate, and where to migrate it.

The considerations for choosing a process for migration involve its size. This has two dimensions. In the space dimension, we would like to migrate small processes: the smaller the used part of the process's address space, the less data that has to be copied to the new location. In the time dimension, we would like to migrate long processes, that will continue to compute for a long time after they are migrated. Processes that terminate soon after being migrated waste the resources invested in moving them, and would have done better staying where they were.

Luckily, the common distributions of job runtimes are especially well suited for this. As noted in Section 2.3, job runtimes are well modeled by a heavy-tailed distribution. This means that a small number of processes dominate the CPU usage. Moving only one such process can make all the difference.

Moreover, it is relatively easy to identify these processes: they are the oldest ones available. This is because a process that has already run for a long time can be assigned to the tail of the distribution. The distribution of process lifetimes has the property that if a process is in its tail, it is expected to run for even longer, more so than processes that have only run for a short time, and (as far as we know) are not from the tail of the distribution. (This was explained in more detail on page 186.)

Details: estimating remaining runtime

A seemingly good model for process runtimes, at least for those over a second long, is that they follow a Pareto distribution with parameter near -1 :

$$\Pr(r > t) = 1/t$$

The conditional distribution describing the runtime, given that we already know that it is more than a certain number T , is then

$$\Pr(r > t | r > T) = T/t$$

Thus, if the current age of a process is T , the probability that it will run for more than $2T$ time is about $1/2$. In other words, the median of the expected remaining running time grows linearly with the current running time.

To read more: The discussion of process lifetimes and their interaction with migration is based on Harchol-Balter and Downey [7].

Choosing a destination can be based on very little information

Choosing a destination node seems to be straightforward, but expensive. We want to migrate the process to a node that is significantly less loaded than the one on which it is currently. The simple solution is therefore to query that status of all other nodes, and choose the least loaded one (assuming the difference in loads is large enough).

A more economical solution is to randomly query only a subset of the nodes. It turns out that querying a rather small subset (in fact, a small constant number of nodes, independent of the system size!) is enough in order to find a suitable migration destination with high probability. An additional optimization is to invert the initiative: have underloaded nodes scour the system for more work, rather than having overloaded ones waste their precious time on the administrative work involved in setting up a migration.

Example: the Mosix system

The Mosix system is a modified Unix, with support for process migration. At the heart of the system lies a randomized information dispersal algorithm. Each node in the system maintains a short load vector, with information regarding the load on a small set of nodes [5]. The first entry in the vector is the node's own load. Other entries contain data that it has received about the load on other nodes.

At certain intervals, say once a minute, each node sends its load vector to some randomly chosen other node. A node that received such a message merges it into its own load vector. This is done by interleaving the top halves of the two load vectors, and deleting the bottom halves. Thus the retained data is the most up-to-date available.

Exercise 210 *Isn't there a danger that all these messages will overload the system?*

The information in the load vector is used to obtain a notion of the general load on the system, and to find migration partners. When a node finds that its load differs significantly from the perceived load on the system, it either tries to offload processes to some underloaded node, or solicits additional processes from some overloaded node. Thus load is moved among the nodes leading to good load balancing within a short time.

The surprising thing about the Mosix algorithm is that it works so well despite using very little information (the typical size of the load vector is only four entries). This turns out to have a solid mathematical backing. An abstract model for random assignment of processes is the throwing of balls into a set of urns. If n balls are thrown at random into n urns, one should not expect them to spread out evenly. In fact, the probability that an urn stay empty is $(1 - \frac{1}{n})^n$, which for large n tends to $1/e$ — i.e. more than a third of the urns stay empty! At the other extreme, the urn that gets the most balls is expected to have about $\log n$ balls in it. But if we do not choose the urns at random one at a time, but rather select two each time and place the ball in the emptier of the two, the expected number of balls in the fullest urn drops to $\log \log n$ [3]. This is analogous to the Mosix algorithm, that migrates processes to the less loaded of a small subset of randomly chosen nodes.

In addition to its efficient assignment algorithm, Mosix employs a couple of other heuristic optimizations. One is the preferential migration of “chronic forgers” — processes that fork many other processes. Migrating such processes effectively migrates their future offspring as well. Another optimization is to amortize the cost of migration by insisting that a process complete a certain amount of work before being eligible for migration. This prevents situations in which processes spend much of their time migrating, and do not make any progress.

The actual migration is done by stopping the process and restarting it

Negotiating the migration of a process is only part of the problem. Once the decision to migrate has been made, the actual migration should be done. Of course, the process itself should continue its execution as if nothing had happened.

To achieve this magic, the process is first blocked. The operating systems on the source and destination nodes then cooperate to create a perfect copy of the original process on the destination node. This includes not only its address space, but also its description in various system tables. Once everything is in place, the process is restarted on the new node.

Exercise 211 *Can a process nevertheless discover that it had been migrated?*

Exercise 212 *Does all the process's data have to be copied before the process is restarted?*

Special care is needed to support location-sensitive services

But some features may not be movable. For example, the process may have opened some files that are available locally on the original node, but are not available on

the new node. It might perform I/O to the console of the original node, and moving this to the console of the new node would be inappropriate. Worst of all, it might have network connections with remote processes somewhere on the Internet, and it is unreasonable to update all of them about the migration.

The solution to such problems is to maintain a presence on the original node, to handle the location-specific issues. Data is forwarded between the body of the process and its home-node representative as needed by the operating system kernel.

To read more: Full details about the Mosix system are available in Barak, Guday, and Wheeler [4]. More recent publications include [2, 8, 1].

Bibliography

- [1] L. Amar, A. Barak, and A. Shiloh, “*The MOSIX direct file system access method for supporting scalable cluster file systems*”. *Cluster Computing* **7(2)**, pp. 141–150, Apr 2004.
- [2] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren, “*An opportunity cost approach for job assignment in a scalable computing cluster*”. *IEEE Trans. Parallel & Distributed Syst.* **11(7)**, pp. 760–768, Jul 2000.
- [3] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, “*Balanced allocations*”. In *26th Ann. Symp. Theory of Computing*, pp. 593–602, May 1994.
- [4] A. Barak, S. Guday, and R. G. Wheeler, *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag, 1993. Lect. Notes Comput. Sci. vol. 672.
- [5] A. Barak and A. Shiloh, “*A distributed load-balancing policy for a multicomputer*”. *Software — Pract. & Exp.* **15(9)**, pp. 901–913, Sep 1985.
- [6] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications*. Prentice Hall, 2nd ed., 1996.
- [7] M. Harchol-Balter and A. B. Downey, “*Exploiting process lifetime distributions for dynamic load balancing*”. *ACM Trans. Comput. Syst.* **15(3)**, pp. 253–285, Aug 1997.
- [8] R. Lavi and A. Barak, “*The home model and competitive algorithms for load balancing in a computing cluster*”. In *21st Intl. Conf. Distributed Comput. Syst.*, pp. 127–134, Apr 2001.
- [9] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Addison-Wesley, 5th ed., 1998.
- [10] A. S. Tanenbaum, *Distributed Operating Systems*. Prentice Hall, 1995.