the Availability Digest

Distributed Systems: Principles and Paradigms

January 2008

Distributed Systems: Principles and Paradigms[1] is a thorough description of the theory and practice behind distributed systems. Authored by Andrew Tanenbaum and Maarten Van Steen, Professors of Computer Sciences at Vrije University in Amsterdam, The Netherlands, this book deals with the myriad issues that must be faced when implementing distributed systems.

The authors define a distributed system as *a collection of independent computers that appears to its users as a single coherent system.*

Inherent in this definition is the concept of *transparency*. There are several aspects of transparency applicable to a distributed system, all related to its system resources:

- Access – Hide the differences in data representation and how a resource is accessed.
- Location – Hide where a resource is located.
- Migration – Hide that a resource may move to another location.
- Relocation – Hide that a resource may be moved to another location while in use.
- Replication – Hide that a resource is replicated.
- Concurrency – Hide that a resource may be shared by several competitive users.
- Failure – Hide the failure and recovery of a resource.

Another major issue faced in distributed systems is scalability. A distributed system:

- must be scalable with respect to its size – more users and resources can be easily added.
- must be geographically scalable – users and resources may be far apart.
- must be administratively scalable – system management is easy even if the system spans many independent organizations.

Common misassumptions in the design of distributed systems include:

- The network is reliable.
- The network is secure.
- The network is homogeneous.
- The topology does not change.
- Latency is zero.
- Bandwidth is infinite.
- Transport cost is zero.
- There is one administrator.

[1] A. S. Tanenbaum, M. Van Steen, *Distributed Systems: Principles and Paradigms*, Pearson Prentice Hall; 2007.

This book explores various distributed system attributes that are associated with transparency and scalability and that relate to problems that are caused by one or more of the above assumptions being false.

Distributed System Architectures

The authors point out that there are two aspects to a distributed system architecture – the hardware configuration and the software architecture. Software architectural styles include:

- *Layered architectures*, in which components are organized into layers. A component can invoke the services of an underlying component but not the other way around.

- *Object-based architectures*, in which components (the *objects*) are organized in a mesh and interact via remote procedure calls.

- *Data-centered architectures*, in which components communicate through a common repository.

- *Event-based architectures*, in which components communicate via events, such as in a publish/subscribe system.

The client/server model of component interaction is described in detail and is applied to each of these architectures. It is noted that in peer-to-peer architectures, a process may act as both a client and a server.

Self-management of distributed systems is briefly explored with several examples.

Processes

A process is a program in execution. In centralized systems, the management and scheduling of processes are the most important issues with which to deal. However, in distributed systems, there are many additional issues, such as concurrent processing within a process, virtualization, and process migration.

Concurrent processing within a process is typically implemented by threads. The authors discuss in detail user threads, kernel threads, and lightweight processes. An important property of threads is that they provide a means to allow blocking system calls without blocking the entire process.

In distributed systems, threads are particularly important for clients that would otherwise be blocked due to communication latency as they make a request to a remote server and await its response. As in centralized systems, threads are important to servers to let them process multiple requests at a time.

Virtualization allows many processes to run on a single physical server in their own virtual machines. Virtualization is important in today's large data centers to support the "new legacy" applications as hardware and operating systems evolve and to consolidate the number of servers required to support an enterprise's needs. The authors describe four approaches to virtualization:

- An interface between hardware and software consisting of machine instructions that can be invoked by a program.
- An interface between hardware and software consisting of machine instructions that can be invoked only by privileged programs such as an operating system.
- An interface consisting of system calls as offered by an operating system.
- An interface consisting of library calls.

Process migration is important to optimize performance through load balancing and by moving parts of a client or server process closer to the data that is being processed. Various models for process migration are described. These models allow code to be moved even while it is active. One common model is to move blocks of code casually, resending those blocks that have changed. When most of the code has been moved, a brief stop-and-copy phase is entered in which any remaining code blocks are moved.

Communication

Interprocess communication is at the heart of all distributed systems. The authors look at three widely-used models for Interprocess communication: remote procedure calls (RPCs), message-oriented middleware (MOM), and data streaming.

RPCs use client and server stubs to pass procedure calls to the server. The syntax for the procedure calls is defined in an IDL (interface definition language). RPC mechanisms are inherently synchronous in that the client is blocked after making a procedure call until the procedure completes. However, some RPC systems provide facilities for asynchronous client operation in which the client is released immediately after sending its request and is later interrupted with the reply. The DCE (Distributed Computing Environment) RPC is presented as an example.

MOM supports both transient and persistent communications. With transient messaging, the receiver must be up and running in order to receive the message. With persistent messaging, the receiving process does not have to be running when the sending process transmits a message. Rather, a sender moves a message to a send queue; and the MOM engine sends the queued message to a receiver queue. At its leisure (perhaps after it is once again running), the receiving process reads the message from its queue and processes it. IBM's WebSphere MQ (originally known as MQ series) is used as an example of MOM.

Data streaming is used to send temporally sensitive data. This might include video and audio signals. A particular problem in data streaming is synchronizing related streams. Examples of related streams are video with its associated audio, multichannel stereo audio, and audio associated with slides in Web casts. Several solutions to stream synchronization are covered.

Finally, multicasting, in which information is sent to multiple receivers, is covered.

Naming

Names are used to share resources, to uniquely identify entities, and to refer to locations, among other uses. Names may be flat (such as an IP address) or may be structured (such as a URL).

Naming services may be distributed across multiple machines for performance and reliability. The authors explore various ways in which names can be resolved to the physical entity. For instance, finding the physical resource corresponding to an IP address (a flat name) on a LAN is done via an ARP (address resolution protocol) request. The requester broadcasts a request message that asks who has the IP address, and that entity responds with its physical address.

An example of the resolution of structured names is the use of distributed DNS (Domain Name System) servers to resolve Web URL names. DNS is described in detail.

Home agents can be used to resolve mobile names to their locations. For instance, a cell phone is registered with a Home Location Register (HLR) that is kept informed of the location of the cell phone via the cell-phone network. The HLR is used by the network to resolve the cell-phone number to a location (i.e., its closest cell tower).

An entity can be described by a set of its attributes rather than by a name. The use of LDAP (Lightweight Directory Access Protocol) is described as an example of resolving attribute-based names.

Synchronization

There are three major areas in which distributed systems must synchronize with each other. One is time synchronization. Another is in the competition for common resources. The third is the election of coordinators.

The authors explain several algorithms for keeping geographically-separated systems in time synchronization. They start by describing current techniques for maintaining accurate time within a system, including atomic clocks, GPS, and radio stations that broadcast the time. Algorithms for synchronizing remote systems with a system that has an accurate clock or for simply synchronizing systems with no reference to an accurate clock are discussed. Primary among these is NTP, the Network Time Protocol, in common use today.

Also explained in some depth are Lamport logical clocks that can not only maintain systems in time synchronism, but can also guarantee the correct ordering of the processing of events across the network.

Methods for controlling access to common resources using mutual exclusion mechanisms are analyzed in some detail. The authors distinguish between four mutual exclusion algorithms:

- Centralized, in which one process in the system is elected to be the coordinator.
- Decentralized, in which the coordinator is replicated across the network to solve the problem of coordinator failure. Mutual access is granted by a majority vote of all coordinators.
- Distributed, in which all processes that may want to access the resource participate in the decision of which process will get access to the resource next. Decisions are based on the time of requests, with the earliest request getting access.
- Token, in which a token is passed among all possible requesters. The requester that currently has the token has exclusive access to the resource.

Many distributed algorithms require that there be one process that performs some specific role. A centralized mutual access coordinator is one example. Another example is a master node in a data replication network. Several algorithms are given for an election process to choose this node either initially or following the failure of the current process performing the role. Special attention is given to wireless environments and very large systems.

Data Consistency and Replication

There are many reasons why a database needs to be replicated. Among these are performance (locating data close to those who will use it most), and reliability, or the ability to suffer the loss of a database copy.

A major problem with replicated databases is keeping the database copies consistent. If synchronous replication is used, it is guaranteed that all database copies will be consistent. However, this approach imposes a potentially unacceptable performance penalty in many applications.

It is important to define the level of consistency that is required by an application. This is done via a *consistency model*. For instance, the consistency model might specify:

- a minimum deviation in numerical values between the replicas.

- a minimum deviation in staleness between the replicas.
- a minimum deviation with respect to the ordering of update operations.

Events that have a causal relationship must be applied in the same order. Concurrent events that are not related are not so constrained.

The placement of permanent and transient database replicas is considered. Transient replicas are those requested by a client or a server to temporarily improve performance. Replication models that pull (poll) or push data to replicas are described as are those that distribute data by unicasting (point-to-point) or multicasting changes or change notices.

Considerations with respect to maintaining a local replica by caching a portion of a database are described.

Special attention is given to *client-centric* models for data replication. Client-centric models ensure that any client will always see an update that is the same or later than that seen on its last access even if the client accesses different database copies (such as a mobile user). These models ensure *eventual consistency* in that, in the absence of updates, the databases will eventually converge. Implementation schemes for this class of data replication model are given.

Though the possibility of data collisions is acknowledged, there is no discussion of data-collision detection or resolution. However, architectures are proposed that will preclude data collisions. One is a master/slave configuration in which all updates are sent to a master node, where they are applied to the database. The master node then replicates the changes to all slaves. The other is also a master/slave configuration in which the slave makes updates to its local copy with the permission of the master. These updates are then replicated to the master, which will then replicate the updates to the other slave systems.

Fault Tolerance

As opposed to single-machine systems, a distributed system can suffer partial failures. Fault tolerance is the ability for a distributed system to survive a partial failure. The authors consider several failure modes:

- Crash Failure – A server halts but is working properly until it halts.
- Omission Failure – A server fails to receive an incoming message or fails to send an outgoing message.
- Timing Failure – A server's response time is excessive.
- Response Failure – A server's response is incorrect.
- Arbitrary Failure – (Byzantine Failure) A server produces arbitrary responses at arbitrary times.

The authors also consider transient faults, intermittent faults, and permanent faults.

Fault masking is accomplished via redundancy. The book focuses on process resilience through flat process groups or process groups organized as a hierarchy. The detection of a failure and the recovery from any of the failure modes listed above by a process group is described. This is complicated if the process group is not stable – that is, processes may be entering and leaving the group as the process group is active.

Reliable communications between processes is a necessary condition for fault tolerance. A great deal of attention is given to reliable multicasting to achieve this end. Reliable multicasting requires that receivers can detect a message loss and can request that the message be retransmitted.

The distributed two-phase commit protocol for transactions is explained. This protocol allows the reliable application of a group of changes to a set of database copies. It requires a transaction coordinator to coordinate the actions of the distributed systems. Described also is a three-phase commit protocol that will survive the failure of the transaction coordinator.

Once a failure has occurred, it is essential that the process in which the failure occurred can recover to a correct state. This is done via periodic checkpoints of the process' state to stable storage and via an optional message log to be able to roll forward the process state from its last checkpoint. (There is no mention of checkpointing to other processes rather than to disk as is done in HP NonStop server process pairs.)

A brief description is given of recovery-oriented computing (ROC).[2] ROC is implemented by recursively rebooting the failed processes or subsystems. The smallest such unit is first rebooted. If this does not restore the system, its parent unit is rebooted. This process continues until the system is restored to operation or until the entire system is rebooted.

Security

Security is one of the most difficult aspects of distributed systems. It needs to be pervasive throughout the system, and a single design flaw may render it useless.

Security threats include:

- Interception – An unauthorized party has gained access to a service or to data.
- Interruption – Services or data become unavailable, unusable, or destroyed. A denial-of-service attack is an interruption.
- Modification – Data has undergone unauthorized changes, or services have been modified so that they do not adhere to their original specifications.
- Fabrication – Additional activity or data is created that would not otherwise exist.

Once a security policy has been established that describes which actions the entities in the system are allowed to take and which are prohibited, the security mechanisms can be designed. Security mechanisms include:

- Encryption – the transformation of data into a form that an attacker cannot understand.
- Authentication – the verification of the identity of a person, a process, or other entity that is requesting services (a client).
- Authorization – a determination that an authenticated client is authorized to perform a requested action.
- Auditing – a record of actions taken by clients.

There are three ways in which the security mechanisms can focus on ensuring the security of the system: protection of the data, control of operations that can be executed by a client, and control of the users. Each of these is evaluated by the authors.

Extensive attention is given to encryption methods. Secret keys and public keys are described along with the mathematics behind their encryption algorithms. The use of channels secured by encryption for authentication is explored in great detail, as are digital signatures. Kerberos is used as an example implementation.

Other topics discussed are access control lists, key management (especially the difficult problem of initial key establishment and key distribution), and secure group management.

[2] Recovery-Oriented Computing, *Availability Digest*; February, 2007.
Microrebooting for Fast Recovery, *Availability Digest*; March, 2007.

Examples

The authors provide four in-depth examples of the use of these principles in distributed systems:

- Distributed Object-Based Systems, using Enterprise Java Beans (EJBs) and CORBA (Common Object Request Broker Architecture) as examples.
- Distributed File Systems, using NFS (Network File System) as an example.
- Distributed Web-Based Systems, describing HTML, XML, HTTP, and SOAP and using the Apache Web Server as an example.
- Distributed Coordination-Based Systems (publish-subscribe systems), using as an example TIB/Rendezvous.

For each of these system models, the application of the paradigms for processes, communication, naming, synchronization, replication, fault-tolerance, and security is described in some detail.

Summary

Distributed Systems: Principles and Paradigms is an exhaustive discourse on the technology that goes into building effective distributed systems. Though the book borders on the erudite rather than the practical (the authors often lapse into notational descriptions that fortunately can be ignored by the theoretically-challenged), it makes extensive use of well-known systems as examples to demonstrate the principles described. It is a must reference for any serious practitioner of distributed systems.