بسمه تعالی

سیستمهای توزیع شده

مدرس : علیرضا تقی زاده

کتاب مرجع

Distributed systems: principles and paradigms

Andrew S. Tanenbaum, Maarten Van Steen

Second Edition, 2007 Pearson Education. Inc.

2

Chapter 1:
INTRODUCTION

Before 1980s:

- large and expensive mainframes.

Around mid 1980s:

- microprocessors, high-speed computer networks.

➢ It is now not only feasible, but easy, to put together computing systems composed of large numbers of computers connected by a high-speed network.

"distributed systems, in contrast to the previous centralized systems"

3

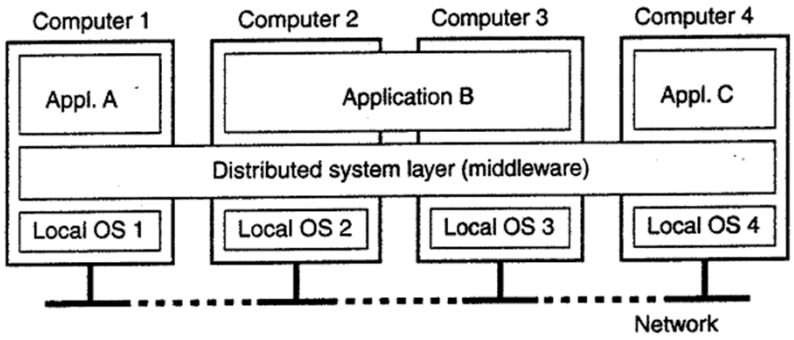
DEFINITION

A distributed system

is a collection of independent computers that appears to its users as a single coherent system.

4

LAYERED MODEL



5

CHARACTERISTICS

1- Differences between the various computers and the ways in which they communicate are mostly hidden from users.

2- Relatively easy to expand or scale.

3- Continuously available, although perhaps some parts may be temporarily out of order.

6

GOALS & ISSUES

1. ACCESSIBILITY
2. TRANSPARENCY
3. OPENNESS
4. SCALABILITY

7

MAKING RESOURCES ACCESSIBLE

The main goal of a distributed system is to make it easy for the users (and applications) to access remote resources, and to share them in a controlled and efficient way.

+ Economics
+ Collaborate and exchange information.

- Security is becoming increasingly important.

8

DISTRIBUTION TRANSPARENCY

- ✓ Access transparency deals with hiding differences in data representation and the way that resources can be accessed by users.
- ✓ Location transparency refers to the fact that users cannot tell where a resource is physically located in the system. Naming plays an important role in achieving location transparency.

9

DISTRIBUTION TRANSPARENCY

- ✓ Migration transparency in which resources can be moved without affecting how those resources can be accessed.
- ✓ Relocation transparency is the situation in which resources can be relocated *while* they are being accessed without the user or application noticing anything.
- ✓ Replication transparency deals with hiding the fact that several copies of a resource exist.

10

DISTRIBUTION TRANSPARENCY

- ✓ Concurrency transparency that each user does not notice that the other is making use of the same resource.
- ✓ Making a distributed system failure transparent means that a user does not notice that a resource (he has possibly never heard of) fails to work properly, and that the system subsequently recovers from that failure.

11

DEGREE OF TRANSPARENCY

There is a trade-off between a high degree of transparency and the performance of a system.

- ✓ A wide-area distributed system that connects two processes in two points far from each other, suffers from several hundreds of milliseconds using a computer network
- ✓ For example, Attempting to mask a transient server failure before trying another one may slow down the system as a whole.

12

DEGREE OF TRANSPARENCY

As distributed systems are expanding to devices that people carry around, and where the very notion of location and context awareness is becoming increasingly important, it may be best to actually *expose* distribution rather than trying to hide it.

- As a simple example, consider an office worker who wants to print a file from her notebook computer.

13

OPENNESS

- Another important goal of distributed systems is openness. An **open distributed** system is a system that offers services according to standard rules that describe the syntax and semantics of those services.

14

OPENNESS

- ✓ In distributed systems, services are generally specified through interfaces, which are often described in an Interface Definition Language (IDL).
- ✓ Interface definitions written in an IDL nearly always capture only the syntax of services. In other words, they specify precisely the names of the functions that are available together with types of the parameters, return values, possible exceptions that can be raised, and so on.
- ✓ The hard part is specifying precisely what those services do, that is, the semantics of interfaces. In practice, such specifications are always given in an informal way by means of natural language.

15

OPENNESS

Proper specifications are **complete** and **neutral**:

- ➢ Complete means that everything that is necessary to make an implementation has indeed been specified.
- ➢ Just as important is the fact that specifications do not prescribe what an implementation should look like: they should be neutral.

16

OPENNESS

Completeness and neutrality are important for **interoperability** and **portability.**

- ✓ Interoperability characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard.
- ✓ Portability characterizes to what extent an application developed for a distributed system *A* can be executed without modification, on a different distributed system *B* that implements the same interfaces as *A.*

17

SCALABILITY

Scalability is one of the most important design goals for developers of distributed systems. Scalability of a system can be measured along at least three different dimensions:

First, a system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system.

Second, a geographically scalable system is one in which the users and resources may lie far apart.

Third, a system can be administratively scalable, meaning that it can still be easy to manage even if it spans many independent administrative organizations.

18

DECENTRALIZED ALGORITHMS

A decentralized algorithms generally have the following characteristics, which distinguish them from centralized algorithms:

- No machine has complete information about the system state.
- Machines make decisions based only on local information.
- Failure of one machine does not ruin the algorithm.
- There is no implicit assumption that a global clock exists.

19

SCALABILITY ISSUES (Geographical)

- It is currently hard to scale existing distributed systems that were designed for local-area networks.
- Communication in wide-area networks is inherently unreliable, and virtually always point-to-point. In contrast, local-area networks generally provide highly reliable communication facilities based on broadcasting, making it much easier to develop distributed systems.

20

SCALABILITY ISSUES (administrative)

A difficult, and in many cases open question is how to scale a distributed system across multiple, independent administrative domains.

A major problem that needs to be solved is that of conflicting policies with respect to resource usage (and payment), management, and security.

21

SCALING TECHNIQUES

In most cases, scalability problems in distributed systems appear as performance problems caused by limited capacity of servers and network. There are now basically only three techniques for scaling:

- Hiding communication latencies,
- Distribution,
- and Replication

22

HIDING COMMUNICATION LATENCIES

Hiding communication latencies is important to achieve geographical scalability. The basic idea is simple: try to avoid waiting for responses to remote (and potentially distant) service requests as much as possible.

- For example, when a service has been requested at a remote machine, an alternative to waiting for a reply from the server is to do other useful work at the requester's side.

23

HIDING COMMUNICATION LATENCIES

There are many applications that cannot make effective use of asynchronous communication. For example, interactive applications.

- A much better solution is to reduce the overall communication, for example, by moving part of the computation to the client.

24

HIDING COMMUNICATION LATENCIES

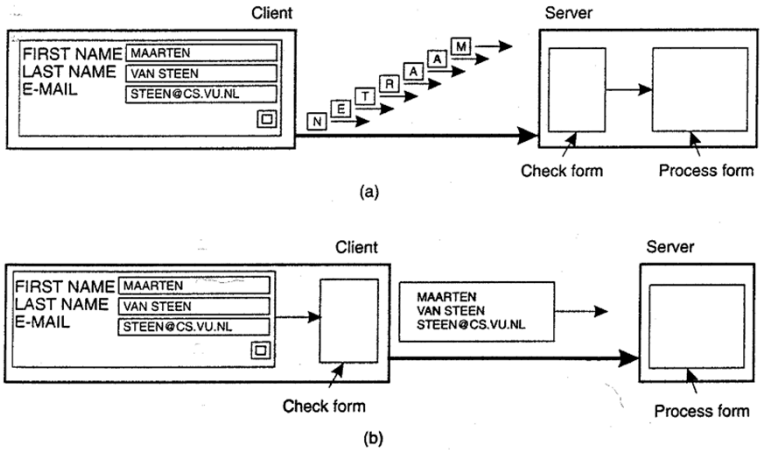


Figure 1-4. The difference between letting (a) a server or (b) a client check forms as they are being filled.

25

DISTRIBUTION

Distribution involves taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system.

- An excellent example of distribution is the Internet Domain Name System (DNS).

26

DISTRIBUTION

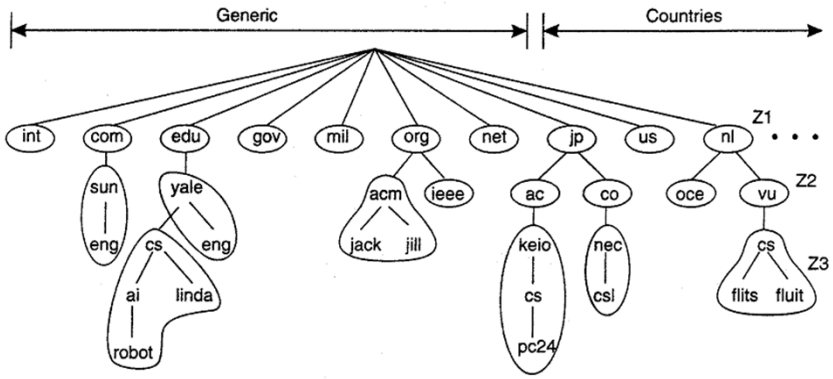


Figure 1-5. An example of dividing the DNS name space into zones.

27

Replication

Considering that scalability problems often appear in the form of performance degradation, it is generally a good idea to actually **replicate** components across a distributed system.

- Replication not only increases availability, but also helps to balance the load between components leading to better performance.

28

Replication vs. Caching

Caching is a special form of replication, although the distinction between the two is often hard to make or even artificial.

- As in the case of replication, caching results in making a copy of a resource, generally in the proximity of the client accessing that resource.
- However, in contrast to replication, caching is a decision made by the client of a resource, and not by the owner of a resource.

29

Replication issues

- There is one serious drawback to caching and replication that may adversely affect scalability. Because we now have multiple copies of a resource, modifying one copy makes that copy different from the others. Consequently, caching and replication leads to consistency problems.

30

Sum up

- One could argue that size scalability is the least problematic from a technical point of view.
- Geographical scalability is a much tougher problem as Mother Nature is getting in our way.
- Finally, administrative scalability seems to be the most difficult one, rarely also because we need to solve nontechnical problems (e.g., politics of organizations and human collaboration).

31

Types of distributed systems

- Distributed computing systems
- Distributed information systems
- Distributed embedded systems

32

Distributed computing systems

An important class of distributed systems is the one used for high performance computing tasks.

- **Cluster computing:** the underlying hardware consists of a collection of similar workstations or PCs, closely connected by means of a high speed local area network.
- **Grid computing:** a federation of computer systems, where each system may fall under a different administrative domain, and may be very different when it comes to hardware, software, and deployed network technology.

33

Cluster computing

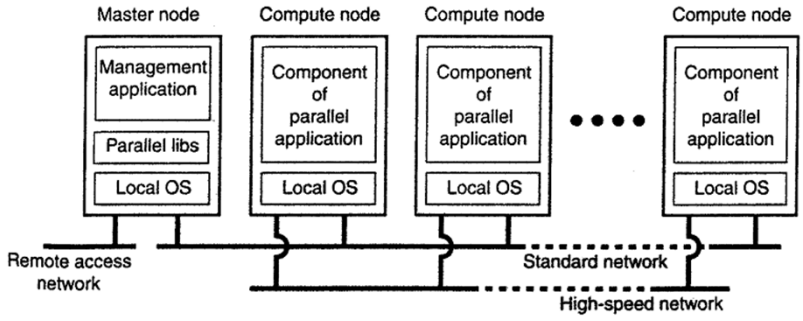


Figure 1-6. An example of a cluster computing system.

Linux-based Beowulf cluster

34

Grid Computing

Software for realizing grid computing evolves around providing access to resources from different administrative domains. Focus is often on architectural issues. An architecture proposed by Foster et al. (2001) is shown below:

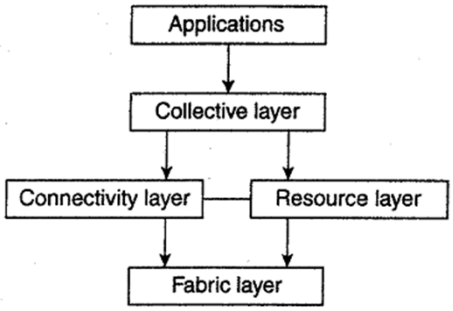


Figure 1-7. A layered architecture for grid computing systems.

35

Grid Computing

- *Fabric layer* provides interfaces to local resources at a specific site. (e.g. querying the state and capabilities of a resource, along with functions for actual resource management such as locking resources).
- *Connectivity layer* consists of communication protocols for supporting grid transactions that span the usage of multiple resources.
- *Resource layer* is responsible for managing a single resource. It uses the functions provided by the connectivity layer and calls directly the interfaces made available by the fabric layer.

36

Grid Computing

- *Collective layer* deals with handling access to multiple resources and typically consists of services for resource discovery, allocation and scheduling of tasks onto multiple resources, data replication, and so on.
- *Application layer* consists of the applications that operate within a virtual organization and which make use of the grid computing environment.

Typically the collective, connectivity, and resource layer form the heart of what could be called a grid middleware layer. These layers jointly provide access to and management of resources that are potentially dispersed across multiple sites.

37

Distributed Information Systems

Many of the existing middleware solutions are the result of working with an infrastructure in which it was easier to integrate applications into an enterprise-wide information system. We can distinguish several levels at which integration took place:

- In many cases, a networked application simply consisted of a server running that application (often including a database) and making it available to remote programs, called clients. Integration at the lowest level would allow clients to wrap a number of requests, possibly for different servers, into a single larger request and have it executed as a **distributed transaction.** The key idea was that all, or none of the requests would be executed.

38

Distributed Information Systems

- As applications became more sophisticated and were gradually separated into independent components (notably distinguishing database components from processing components), it became clear that integration should also take place by letting applications communicate directly with each other. This has now led to a huge industry that concentrates on **enterprise application integration** (EAI).

39

Transaction Processing Systems

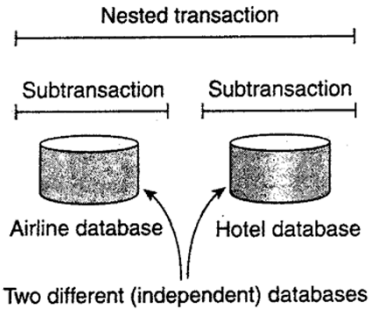
Programming using transactions requires special primitives that must either supplied by the underlying distributed system or by the language runtime system. The characteristic feature of a transaction is either all of these operations are executed or none are executed. More specifically, transactions are:

1. **Atomic**: To the outside world, the transaction happens indivisibly.
2. **Consistent**: The transaction does not violate system invariants.
3. **Isolated**: Concurrent transactions do not interfere with each other.
4. **Durable**: Once a transaction commits, the changes are permanent.

40

Nested transaction

- A nested transaction is constructed from a number of sub-transactions. The top-level transaction may fork off children that run in parallel with one another, on different machines, to gain performance or simplify programming.



a transaction for planning a trip

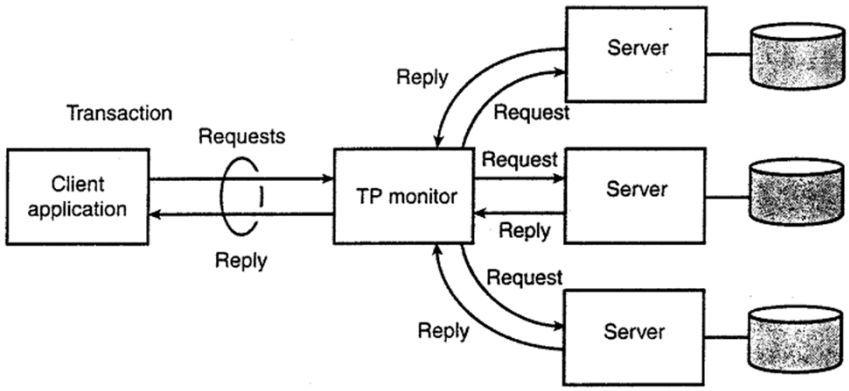
41

Nested transaction

- The permanence referred to applies only to top-level transactions.
- When any transaction or subtransaction starts, it is conceptually given a private copy of all data in the entire system for it to manipulate as it wishes. If it aborts, its private universe just vanishes, as if it had never existed. If it commits, its private universe replaces the parent's universe.
- Likewise, if an enclosing (higher-level) transaction aborts, all its underlying subtransactions have to be aborted as well.

42

Transactional programming model



43

Enterprise Application Integration

In modern environment, application components should be able to communicate directly with each other and not merely by means of the request/reply behavior that was supported by transaction processing systems. Several types of communication middleware exist.

1. RPC
2. RMI

44

Remote Procedure Call (RPC)

- With remote procedure calls (RPC), an application component can effectively send a request to another application component by doing a local procedure call, which results in the request being packaged as a message and sent to the callee.
- Likewise, the result will be sent back and returned to the application as the result of the procedure call.

45

Remote Method Invocations (RMI)

- As the popularity of object technology increased, techniques were developed to allow calls to remote objects, leading to what is known as remote method invocations (RMI).
- An RMI is essentially the same as an RPC, except that it operates on objects instead of applications.

46

Message-oriented Middleware

- RPC and RMI have the disadvantage that the caller and callee both need to be up and running at the time of communication. In addition, they need to know exactly how to refer to each other.
- This tight coupling is often experienced as a serious drawback, and has led to what is known as message-oriented middleware, or simply MOM. In this case, applications simply send messages to logical contact point. Likewise, applications can indicate their interest for a specific type of message, after which the communication middleware will take care that those messages are delivered to those applications.

47

Distributed embedded systems

- Matters have become very different with the introduction of mobile and embedded computing devices. We are now confronted with distributed systems in which instability is the default behavior.
- The devices in these, what we refer to as distributed pervasive systems, are often characterized by being small, battery-powered, mobile, and having only a wireless connection, although not all these characteristics apply to all devices.

48

Distributed embedded systems

Requirements for pervasive applications:

1. **Embrace contextual changes**: means that a device must be continuously be aware of the fact that its environment may change all the time.
2. **Encourage ad hoc composition**: refers to the fact that many devices in pervasive systems will be used in very different ways by different users.
3. **Recognize sharing as the default**: means to easily read, store, manage, and share information.

49

Some examples of pervasive systems

- **Home Systems**: should be completely self-configuring and self-managing.
- **Electronic Health Care Systems**: such a network should at worst only minimally hinder a person. To this end, the network should be able to operate while a person is moving, with no strings (i.e., wires) attached to immobile devices.
- **Sensor Networks:** their limited resources, restricted communication capabilities, and constrained power consumption demand that efficiency be high on the list of design criteria.

50