Introduction to Distributed Systems

Distributed Systems
Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica: Scienza e Ingegneria (DISI)
Alma Mater Studiorum—Università di Bologna a Cesena

Academic Year 2012/2013

Outline

1. Definitions
2. Motivations & Goals
3. Sorts of Distributed Systems

These Slides Contain Material from [Tanenbaum and van Steen, 2007]

Slides were made kindly available by the authors of the book

- Such slides shortly introduced the topics developed in the book [Tanenbaum and van Steen, 2007] adopted here as the main book of the course
- Most of the material from those slides has been re-used in the following, and integrated with new material according to the personal view of the teacher of this course
- Every problem or mistake contained in these slides, however, should be attributed to the sole responsibility of the teacher of this course

Definition of a Distributed System

A distributed system is

A collection of independent computers that appears to its users as a single coherent system [Tanenbaum and van Steen, 2007]

User's view

- This is a possible definition, which accounts for an observational / user-oriented view
- We may also call it the *computer scientist* definition of a distributed system
- From an engineering viewpoint, is a sort of *a posteriori* definition

Distributed System: Another Definition

A distributed system is

A collection of autonomous computational entities conceived as a single coherent system by its designer

Engineer's view

- This is another possible definition, which accounts for a constructive, design-oriented view
- We may also call it the *computer engineer* definition of a distributed system
- From an engineering viewpoint, is a sort of *a priori* definition

Definition of Distributed System: Some Remarks

A distributed system is made of a multiplicity of *components*

- Independent / autonomous computational entities (computers, microprocessors, . . .)
- No assumptions on their individual nature, structure, behaviour, . . .
- Heterogeneity

A distributed system can be seen as a single coherent system

- According to either the user's view or the engineer's view—or both
- Need for coherence over multiplicity and heterogeneity

Main Issues of Distributed System

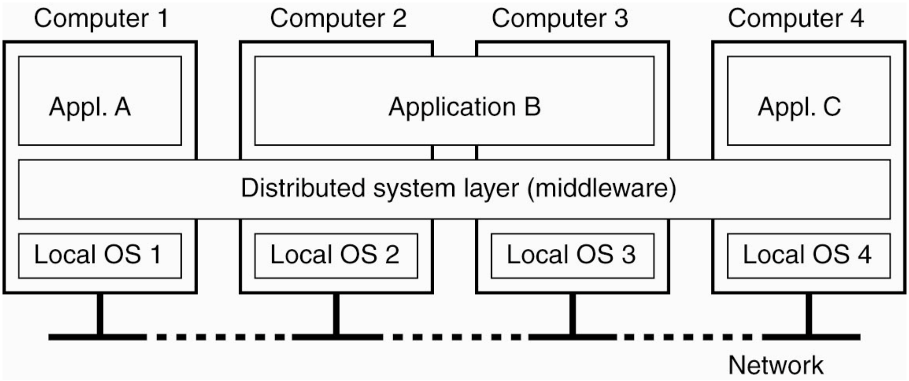
Collaboration

- Many autonomous entities should work altogether as a single coherent system

Amalgamation

- Many heterogeneous entities should look altogether as a single uniform system

An Architectural View of Distributed Systems



A distributed system organized as middleware. The middleware layer extends over multiple machines, and offers each application the same interface [Tanenbaum and van Steen, 2007]

An Architectural View of Distributed Systems: Remark

Moving from a view of non-distributed systems

- Trying to extend the same old interpretation of systems
- Good for preserving good habits
- Bad for looking for new ideas and new problems

Middleware: A Principled Solution

Collaboration & heterogeneity

Solution through separation

- The middleware layer enables meaningful interaction between autonomous distributed components
  - communication issues like syntax, semantics, ...
- The middleware layer hides differences in technology, structure, behaviour, ...
  - provides for a common shared interface for both applications and components—like, operating systems

Outline

1 Definitions

2 Motivations & Goals
- Motivations
- Goals

3 Sorts of Distributed Systems
- Distributed Computing Systems
- Distributed Information Systems
- Distributed Pervasive Systems

What Made Computational Systems *Distributed*?

At the very beginning

- Computer were huge & expensive machines
- Computer were islands
  - Computer science as the art of computer programming was born upon such machines

Then drastic advances in Electronics and TLC occurred

- Microprocessor technology made computational entities more and more powerful and cheap
- High-speed computer networks made interconnection of computational entities possible at a wide range of scales and speeds
  - The scope and goal of computer science changed dramatically

↓ from *centralised* (single-processor) systems

→ to *decentralised* (multi-processor), distributed systems

Why Should We Build in a Distributed System?

A Distributed System is Easy to Build

- Hardware, software, and network components are easy to find & use
  - and to be put together somehow
- However, at a first sight, distribution apparently introduces problems, rather than solving them
  - why should we build a system as a distributed system?
  - it is not easy to make a distributed system *actually work*

Outline

1 Definitions

2 Motivations & Goals
- Motivations
- Goals

3 Sorts of Distributed Systems
- Distributed Computing Systems
- Distributed Information Systems
- Distributed Pervasive Systems

Making Distributed System Worth the Effort

Four goals for a distributed system

- Making (distributed, remote) resources available for use
- Allowing distribution of resources to be hidden whenever unnecessary
- Promoting openness
- Promoting scalability

Making Resources Available

Resouces are physically distributed

- A good reason to build a distributed system is to make them distributed resources available as they would belong to a single system

What is a resource?

Anything that . . .

- . . . could be connected to a computational system
- . . . anyone could legitimately use

E.g., printers, scanners, storage devices, distributed sensors, . . .

By making interaction possible between users and resources, distributed systems are *enablers* of collaboration, sharing, information exchange, . . .

Distribution Transparency

Physical distribution is not a feature, sometimes

- A good reason to build a distributed system is to make physical distribution *irrelevant* from the user's viewpoint

Transparency

- Hiding non-relevant properties of the system's components and structure is called *transparency*
- There exists a number of different and useful sorts of transparency, according to the property hidden to the user's perception

By hiding non-relevant properties to users, distributed systems provide users with a *higher level of abstraction*

Types of Transparency in a Distributed System

Transparency	**Description**
Access	Hide 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

Different forms of transparency in a distributed system (ISO, 1995) [Tanenbaum and van Steen, 2007]

Access Transparency

Heterogeneity in representation and use

- Different data representation
- Different component structure
- Different resource usage interface / protocols

All of them should be hidden from the user's view, whenever possible & non-relevant

Providing a homogeneous view over heterogeneity

- Distributed systems should hide heterogeneity, by providing uniform/homogeneous access to data, components, resources

Location Transparency

Location of users and resources

- Often, the physical location of a resource is not relevant for its use by the users—nor, viceversa, the location of users
  - e.g., the position of a storage facility, or of a single printer in a cluster of printers in a lab

Hiding physical distribution of users and resources

- Distributed systems should hide physical distribution, whenever possible & non-relevant

Naming is essential

- There should exist a system of logical identifiers, not bound to physical location
  - e.g., URLs

Migration Transparency

Resources might be mobile

- Locations change within a distributed system
- which has to maintain its coherence anyway

A distributed system should allow *migration* of resources

- Without losing coherence
- Without losing functionality

Also users might move

- This aspect is not typically accounted by the classification
- but is relevant as well

Relocation Transparency

Resources should be still accessible when moving

- *Migration* should not prevent users to access resources, while they are changing their location
- *Relocation* transparency in some sense is a specialised version of migration transparency

Distributed systems could be useful to allow access to mobile resources by mobile users, by hiding changes in location (migration transparency), even while changes are actually occurring (relocation transparency).

Replication Transparency

Sometimes *replication* helps

Like, for instance,

- in providing a local, faster accessible copy of data to local agents/users
- in promoting tolerance to failures

Whatever the motivation behind replication . . .

. . . replication is not something a user should worry about

- all replicas should be accessible in the same, transparent way
- so they should have the same name
- and should be essentially in the same "state", so to be apparently one and the same thing for each and every user

Distributed systems could exploit replication techniques for many reasons but should at the same time hide it to users.

Concurrency Transparency

Activity in a distributed systems involves independent entities

- Users and resources are distributed, and work autonomously, in a concurrent way
- For instance, two users may try to exploit the same resource at the same time
- Typically, no user need to be bothered with these facts—like, "another user is accessing the same database you are accessing just now", who cares?

Concurrency in activities should be hidden to users

- While shared access to resources could be done cooperatively, it is often the case that users should access competitively to resources
- A distributed system could take care of this, by defining access policies governing concurrent sharing of resources
- Possibly, transparently to the users

Concurrency & Consistency

The problem of *consistency*

- When many users access the same resource concurrently, consistency of its state is in jeopardy—but should be ensured anyway
- A distributed system should take care of ensuring resource safety even under concurrent accesses
- As usual, transparently to the users

A distributed systems should take care of allowing transparent concurrent access to resources, while ensuring consistency of resources.

Failure in Distributed Systems

Failure in a distributed system is essentially a failure *somewhere*

- "*You know you have [a distributed system] when the crash of a computer you've never heard of stops you from getting any work done.*" (L. Lamport)
- Distribution might be either a source of problems or a blessing
- It means that a failure could occur anywhere, but also that a part of the system is likely keep on working
    - Distributed failure is hard to control
    - Partial failure is possible, and much better than total failure of centralised systems

Distribution should work as a *feature*.

Failure Transparency

What does this mean?

- Masking failures under realistic assumptions
- Hiding failure of resources to users

Being failure transparent is a hard problem

- E.g., the problem of latency
  - how do we distinguish between a dead resource and a very slow one?
  - Is “silence” from a resource originated by slowness, deliberate choice, resource failure, or network failure?

Distributed systems should exploit distribution to reduce the impact of partial failure onto the overall system, hiding failures to users as much and often as possible.

Degree of Transparency in a Distributed System

Hiding distribution is not always the best idea

- For instance, users may move and be subject to different time zones—this could lead to funny situations, if hidden
- Also, you should know that a file server is located in Italy or in Japan when choosing from where you will download the nth 250-zillion-of-orribytes patch for your Windows operating system from home

Trade-off between transparency and information

- It typically concerns performance, but is not limited to this
- Location-awareness is often a feature
- Every engineer should find out the precise *degree of transparency* its distributed system should feature, by taking into account other issues like performance, understandability, . . .

Openness

What is openness?

- Essentially, the property of working with a number and sort of components that is not set once and for all at design time
- Open systems are fundamentally *unpredictable*
- Open systems are typically *designed to be open*

Designing over unpredictability requires predictable items

- Something needs to be *a priori* shared between the system and the (potential) components
- Like, standard rules for services syntax and semantics, message interchange, . . .

Interfaces for Open Systems

Interfaces to specify service syntax

- IDL (Interface Definition Languages) to define how interface are specified
- They capture syntax, rather than semantics—often, they do not specify the protocol, too

Issues for Open Systems

Interoperability

- Interoperability measures how easy / difficult is to make one component / system work with different ones based on some standard-based specifications

Portability

- Portability measures how much an application (or, a portion of it) can be moved to a different distributed system and keep working

Extensibility

- Extensibility measures how easy / difficult is to add new components and functionality to an existing distributed system

Separating Policy from Mechanism

Openness mandates for a clean architecture

- External interfaces are not enough
- Components should be small and focussed enough to be easily modified / replaced
- Internal specifications should be as neat as the external ones

Components providing mechanisms should not impose policies

- Mechanisms should be neutral, and open to different policy specifications
- Policies should be encapsulated into other components or delegated to users
- Separation between mechanisms and policies should be enforced

Scalability

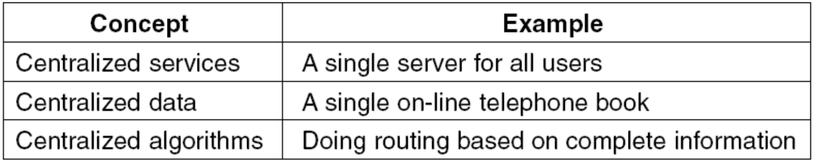
World-wide scale changes everything

- Often, few realistic assumptions can be done on the actual “size” of a distributed system at design time
- There, *size* might mean actual size in number of components, but also in geographical distribution

Dimensions of scalability [Neuman, 1994]

- A system might scale up when the number of users and resources grows
- A system might scale up when the geographical distribution of users and resources extends
- A system might scale up when it spans over a growing number of distinct administrative domains

Scalability Problems: Scaling with Respect to Size



Concept	Example
Centralized services	A single server for all users
Centralized data	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Examples of scalability limitations [Tanenbaum and van Steen, 2007]

Centralisation

Making things centralised might be necessary

- Even though a single server is a bottleneck, it could be a necessity in case of security problems
- Even though a single collection of data is a bottleneck, it could still be needed if replication is insecure
- Sometimes, the most efficient algorithm from a theoretical viewpoint might be a centralised one

However, centralisation hinder scalability, and should be avoided in general in distributed systems whenever possible

Decentralised vs. Centralised Algorithms

The trouble with centralised algorithms

- Data should flow from the whole network to and from the place where the centralised algorithm works
- The network would be overloaded
- Any transmission problem would cause problems to the overall algorithm

→ Only decentralised algorithms should be used in distributed systems

Characteristics of decentralised 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

Scalability Problems: Geographical Scalability

The trouble with communication

- Communication in LAN is typically synchronous—this does not scale up to WAN: e.g., how do I set up timeouts?
- Communication in WAN is typically unreliable, and typically point-to-point—LAN broadcasting no longer an option: e.g., how do I locate a service?

Shared troubles with size scalability

- Centralisation is still a mess

Administration / organisation troubles

- E.g.: within a single domain, users and components might be trusted: however, trust does not cross domain boundaries
- Distributed systems typically extend over multiple administration / organisation domains
    - security measures are needed that may hinder scalability
    - policies may conflict

Techniques for Geographical Scalability

Three Basic Techniques [Neuman, 1994]

- Hiding communication latency
  - asynchronous communication
- Distribution
- Replication

Scaling Techniques: Hiding Communication Latency I

The basic idea

Try to avoid wasting time waiting for remote responses to service requests whenever possible

Asynchronous communication

This basically means using *asynchronous communication* for requests whenever possible

- a request is sent by the application
- the application does not stop waiting for a response
- when a response come in, the application is interrupted and a handler is called to complete the request

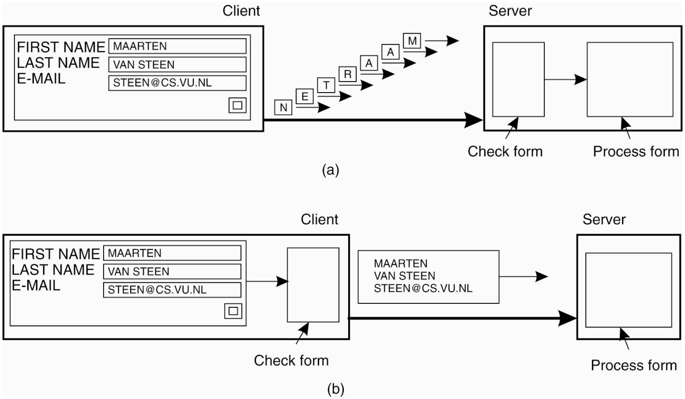
Scaling Techniques: Hiding Communication Latency II

Problem

Sometimes, asynchronous communication is not feasible

- like in Web application when a user is just waiting for the response
- alternative techniques like shipping code are needed—e.g., Javascript or Java Applets

Scaling Techniques: Code Shipping Example



The difference between letting (a) a server or (b) a client check forms as they are being filled [Tanenbaum and van Steen, 2007]

Scaling Techniques: Distribution

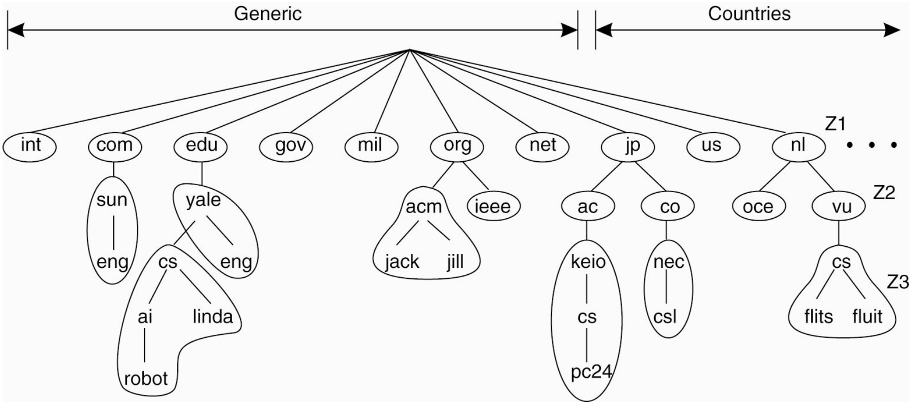
The basic idea

Taking a component, splitting it into parts, and spreading the parts across the system

Example: The Domain Name System (DNS)

- the DNS is hierarchically organised into a tree of *domains*
- domains are divided in non-overlapping *zones*
- the names in each zone are in charge of a single server
- e.g., `apice.unibo.it`
- the *naming service* is thus distributed across several machines, without centralisation

Scaling Techniques: Distribution Example



An example of dividing the DNS space into zones
[Tanenbaum and van Steen, 2007]

Scaling Techniques: Replication

The basic idea

- When degradation of performance occurs, *replicating* components across a distributed system may increase availability and solve problems of latency
- Replication typically involves making a copy of a resource form the original location to a location in the proximity of the (potential) users

Caching

- is a special form of replication
- caching is making a copy of a resource, like replication
- however, caching is a decision by the client of a resource, replication by the owner of a resource

The Problem of Consistency

Duplicating a resource introduces consistency problems

- involving both caching and replication
- inconsistency is technically unavoidable, whenever copying a resource in a distributed setting
- the point is how much inconsistency could a system tolerate, and how it could be hidden from users and components of a distributed system

Scalability Problems: Administrative Scalability

The trouble with organisation

- Maybe the most difficult problem: many non-technical problems to be solved, such as policy of organisation and human collaboration

A successful approach: Ignoring administrative domains

- Users take over control: peer-to-peer technologies
- Only a partial solution, nevertheless, something should be done

Pitfalls of Distributed Systems

False assumptions made by first time developer (by Peter Deutsch)

- 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

These false assumptions typically produces all mistakes in the engineering of distributed systems

Pitfalls of Distributed Systems: Remarks

Such (false) assumptions relate to properties unique to distributed systems

- reliability of the network
- security of the network
- heterogeneity of the network
- topology of the network
- latency
- bandwidth
- transport costs
- administrative domains

When engineering non-distributed systems, such problems are likely not to show up.

Sorts of Distributed Systems

Three Classes of Distributed Systems

- Distributed Computing Systems
- Distributed Information Systems
- Distributed Pervasive Systems

Outline

1 Definitions

2 Motivations & Goals
- Motivations
- Goals

3 Sorts of Distributed Systems
- Distributed Computing Systems
- Distributed Information Systems
- Distributed Pervasive Systems

Distributed Computing Systems

The main characteristic

- Using a multiplicity of distributed computers to perform high-performance tasks

Two classes

- Cluster Computing Systems
- Grid Computing Systems

Cluster Computing Systems

The basic idea

- A collection of similar workstations / PCs
- running the same OS
- located in the same area
- interconnected through a high-speed LAN

Motivation

- The ever increasing price / performance ration of computers makes it cheaper to build a supercomputer by putting together many simple computers, rather than buying a high-performance one
- Also, robustness is higher, maintenance and incremental addition of computing power is easier

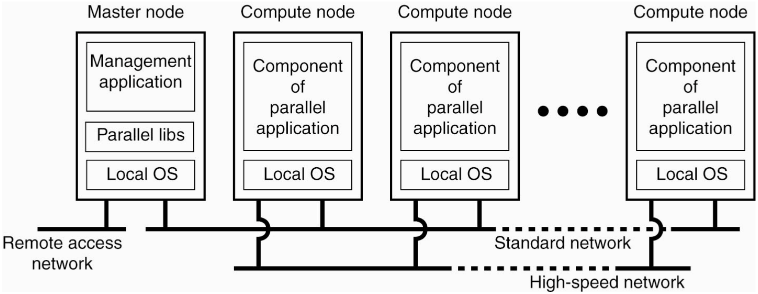
Usage

- Parallel programming
- Typically, a single computationally-intensive program is run in parallel on multiple machines

An Example of Cluster Computing Systems

Beowulf clusters

- Linux-based
- Each cluster is a collection of computing nodes controlled and accessed by a single master node



[Tanenbaum and van Steen, 2007]

Cluster vs. Grid Computing Systems

Homogeneity vs. heterogeneity

- Homogeneity
  - computers in a cluster are typically similar
  - computers in a cluster have the same OS
  - computers in a cluster are connected to the same (local) network
- In essence, cluster computer systems are homogeneous
- Grid computer systems instead are typically heterogeneous

Grid Computing Systems

The main idea

- Resources from different organisations are brought together to promote collaboration between individuals, groups, or institutions, by passing organisation boundaries
- Collaboration is built in the form of a *virtual organisation*
    - essentially, a new virtual organisational entity including people from existing organisations
    - accessing resources made available by participating organisations
    - including servers, databases, hard disks, . . .
- By their very nature, grid computer systems deal with different administrative domains

Architecture of a Grid Computing System I

A layered architecture for a grid computing system [Foster et al., 2001]

Fabric layer — interface to local resources at a specific site

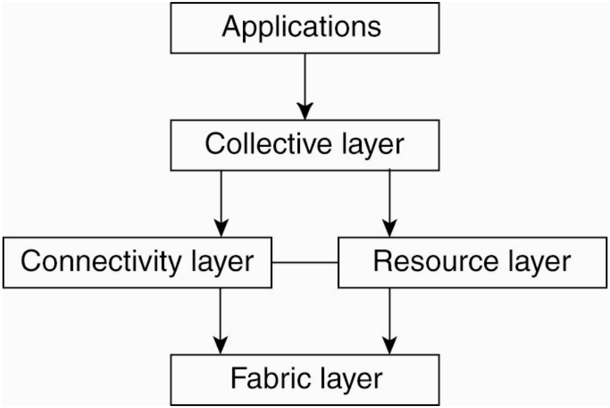
Connectivity layer — communication protocols for grid transactions spanning over multiple resources, plus security protocols for authentication

Resource layer — management of single resources—e.g., access control

Collective layer — handling access to multiple resources—resource discovery, allocation, . . .

Application layer — applications operating in the virtual organisation

Architecture of a Grid Computing System II



[Tanenbaum and van Steen, 2007]

Architecture of a Grid Computing System III

Grid middleware layer

- The core of a grid middeleware layer is represented by connectivity, resource, and collective layers
- Altogether, they provide uniform access to otherwise dispersed resources

Outline

1 Definitions

2 Motivations & Goals
- Motivations
- Goals

3 Sorts of Distributed Systems
- Distributed Computing Systems
- Distributed Information Systems
- Distributed Pervasive Systems

Distributed Information Systems

Origin

- Many separate networked applications to be integrated
- Structural problems of interoperability

Sorts

- Several non-interoperating servers shared by a number of clients: distributed queries, distributed transactions

$\rightarrow$ Transaction Processing Systems

- Several sophisticated applications – not only databases, but also processing components – requiring to directly communicate with each other

$\rightarrow$ Enterprise Application Integration (EAI)

Transaction Processing Systems

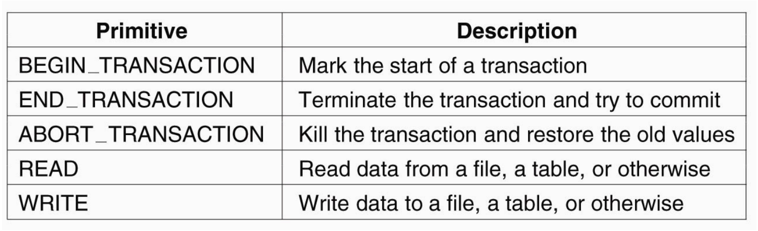
Distributed transactions for distributed databases

- Operations on databases are usually performed in terms of *transactions*
- When databases are distributed, transactions should be distributed
- Special primitives from the distributed system or the runtime system

ACID properties

Atomic	the transaction occurs invisibly to the outside world
Consistent	the transaction does not violate system invariants
Isolated	Concurrent transactions do not interfere with each other
Durable	Once a transaction commits, its effects are permanent

Example Primitives for Transactions

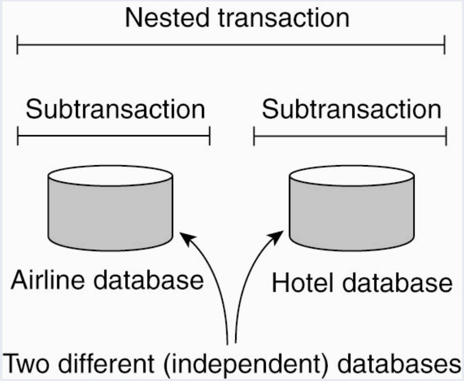


Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

[Tanenbaum and van Steen, 2007]

Nested Transactions

A nested transaction is made of a number of subtransactions



[Tanenbaum and van Steen, 2007]

Nesting in transactions could be arbitrarily deep

The Problem with Nested Transactions I

Durability of nested and sub-transactions

- A whole nested transaction should exhibit ACID properties
- So, if a subtransaction fails, all subtransactions till there should be undone, even though they already committed
- The effects of subtransactions could not be really durable if the whole transanction does not succeed

→ Durability here refers to the top-level transaction

The Problem with Nested Transactions II

Private copy of the world as a solution

- All transactions are performed over a copy of the data, so subtransactions could keep ACIDity in the *local world*
- The effect of a successful nested transaction would be propagated only after it succeeds

→ In case, the copy of the world transformed becomes the world

→ In any case, transactions are ACID

Nested Transactions in Distributed Information Systems

Nested transactions are a natural way for distributing transactions

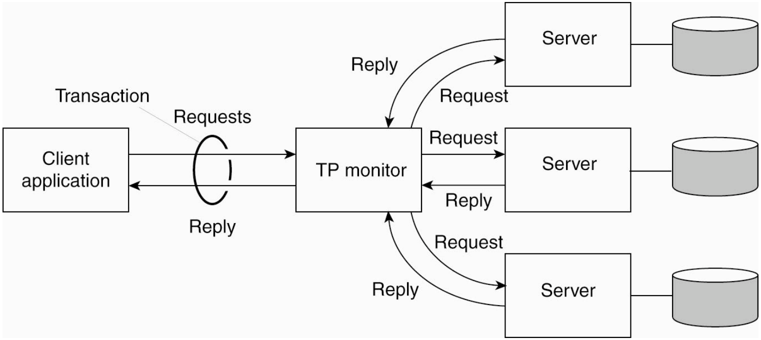
- "Leaf" subtransactions are usual transactions over single servers
- Distributed transactions are nested transactions
- The effects of subtransactions could not be really durable if the whole transanction does not succeed

→ Durability here refers to the top-level transaction

An early solution: TP monitor

- Transaction processing monitor (or, TP monitor)
- to allow applications to access multiple DB servers
- with a transactional semantics

TP Monitor



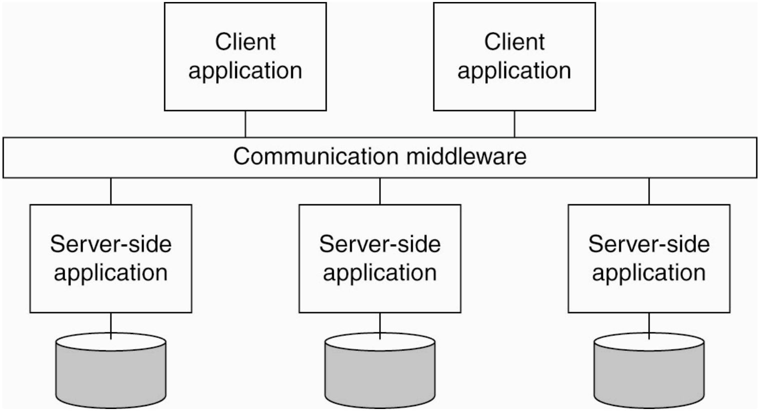
[Tanenbaum and van Steen, 2007]

Enterprise Application Integration

It is not only a matter of accessing distributed databases

- Integration should happen at the application level, too
- Beyond data integration, process integration
- Application should interact and communicate meaningfully with each other

Middleware as a Communication Facilitator



[Tanenbaum and van Steen, 2007]

Sorts of Communication Middleware

Different communication middeware support different sorts of communication

RPC	Remote Procedure Call
RMI	Remote Method Invocation
MOM	Message-Oriented Middleware
	Publish & Subscribe

Outline

1 Definitions

2 Motivations & Goals
- Motivations
- Goals

3 Sorts of Distributed Systems
- Distributed Computing Systems
- Distributed Information Systems
- Distributed Pervasive Systems

Distributed Systems with Instability

What happens when instability is the default condition?

- Like, with mobile devices with batteries and sporadic network connection?
- Like, in modern *distributed pervasive systems*?

Main features

- A distributed pervasive system is part of our surroundings
- A distributed pervasive system generally lacks of a human administrative control

Requirements for Pervasive Distributed Systems [Grimm et al., 2004]

Three points

- Embrace contextual changes
- Encourage ad hoc composition
- Recognise sharing as the default

Remarks

- A device must be continually aware of the fact that its environment may change at any time
- Many devices in pervasive system will be used in different ways by different users
- Devices generally join the system in order to access (provide) information: information should then be easy to read, store, manage, and share

Home Systems

Systems built around home networks

- No way to ask people to act as a competent network / system administrator

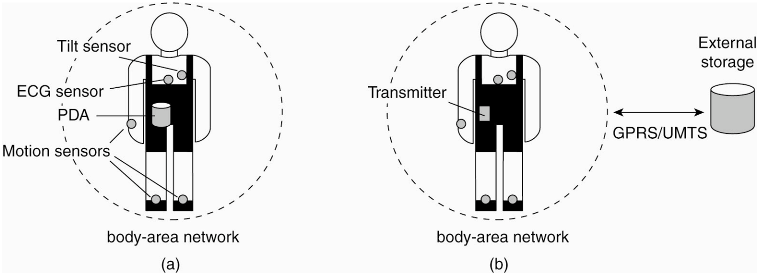
$\rightarrow$ home systems should be self-configuring and self-maintaining in essence

Systems built around personal information

- Huge amount of heterogeneous personal information to be managed,
- coming from heterogeneous sources from inside and outside the home system

Health Care Systems

- Personal systems built around a Body Area Network
- Possibly, minimising impact on the person—like, preventing free motion



[Tanenbaum and van Steen, 2007]

Health Care Systems: Questions to be Addressed

- Where and how should monitored data be stored?
- How can we prevent loss of crucial data?
- What infrastructure is needed to generate and propagate alerts?
- How can physicians provide online feedback?
- How can extreme robustness of the monitoring system be realized?
- What are the security issues and how can the proper policies be enforced?

Sensor Networks

An enabling technology for pervasive systems

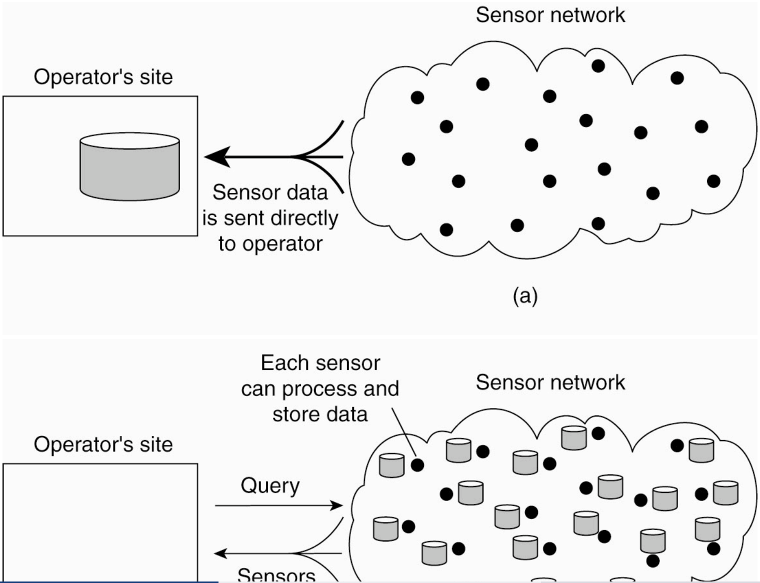
- Clouds of spatially distributed sensors—from tens to thousands of nodes with a sensing device
- Acquiring, processing and transmitting environmental information

A possible view: distributed databases

- Distributed sources of information
- that can possibly be queried along time
- Two possible extremes: either sensors just send information in without cooperating, or they do all the computation and return the results

↑ Both solutions are bad ones, since they require either too much network consumption, or too much node power consumption

Architecture of Sensor Networks: Two Extremes


Sensor network
Operator's site
Sensor data is sent directly to operator
(a)
Each sensor can process and store data
Sensor network
Operator's site
Query
Sensors

Sensor Networks: A Solution

In-network data-processing

- Setting up a tree network among sensors
- Passing queries through the sensor tree
- Aggregating the results at the different levels of the tree

Questions to be addressed

- How do we (dynamically) set up an efficient tree in a sensor network?
- How does aggregation of results take place? Can it be controlled?
- What happens when network links fail?

Summing Up

There are good reasons to build up distributed systems

- Several problems, and several opportunities, too
- Systems are inherently complex, nowadays, and distributed systems may help hiding some complexity and improving understanding and ease of use

Distributed systems introduces / reveals new dimensions of computational systems

- When they are ignored, suddenly lead to severe problems
- To account for them, a new discipline for system engineering is required

Diverse sorts of distributed systems exist

- Depending on both the environment where they are developed, the goals they have to achieve, and the level of the available technologies
- Different models, methodologies and technologies are to be used to design and develop different sorts of distributed systems

References I

Foster, I., Kesselman, C., and Tuecke, S. (2001).
The anatomy of the grid: Enabling scalable virtual organizations.
International Journal of High Performance Computing Applications, 15(3):200–222.

Grimm, R., Davis, J., Lemar, E., Macbeth, A., Swanson, S., Anderson, T., Bershad, B., Borriello, G., Gribble, S., and Wetherall, D. (2004).
System support for pervasive applications.
ACM Transactions on Computer Systems, 22(4):421–486.

Neuman, B. C. (1994).
Scale in distributed systems.
In Casavant, T. L. and Singhal, M., editors, *Readings in Distributed Computing Systems*, pages 463–489. IEEE CS Press, Los Alamitos, CA, USA.

References II

Tanenbaum, A. S. and van Steen, M. (2007).
Distributed Systems. Principles and Paradigms.
Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition.

Introduction to Distributed Systems

Distributed Systems
Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica: Scienza e Ingegneria (DISI)
Alma Mater Studiorum—Università di Bologna a Cesena

Academic Year 2012/2013