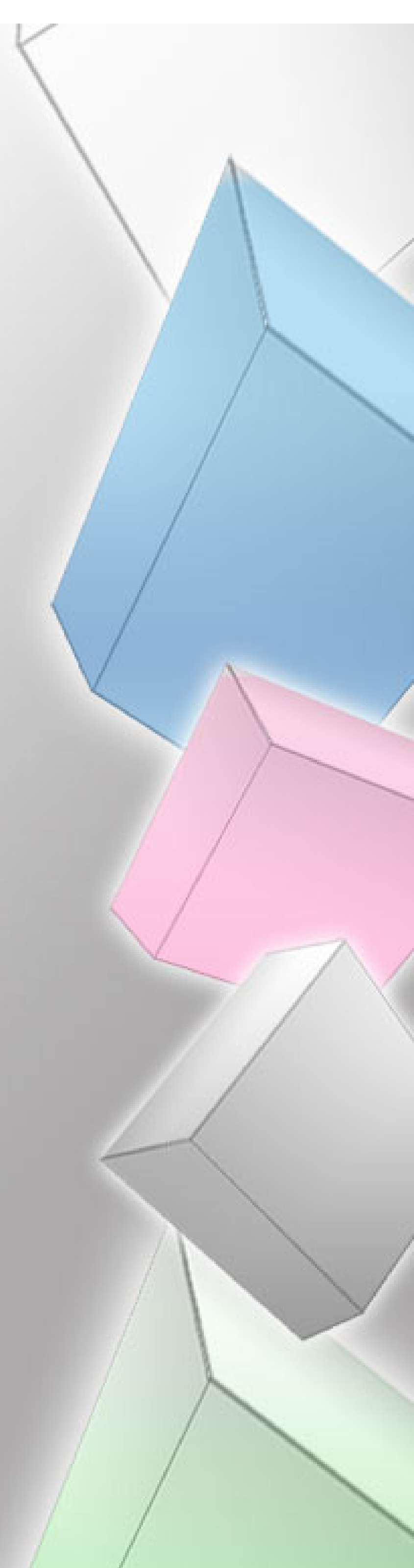
Advanced Topics in Operating Systems

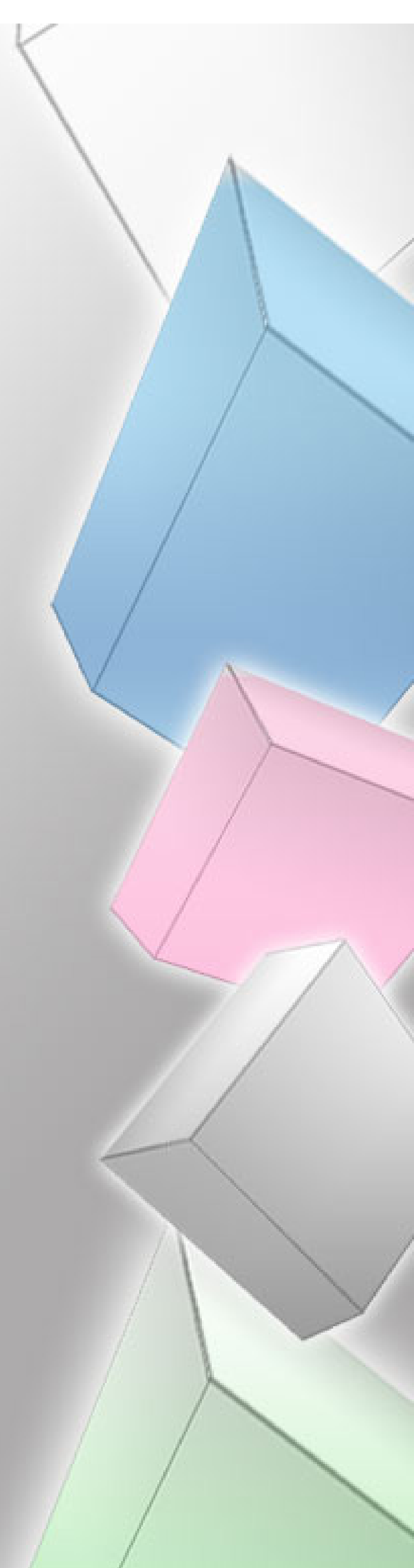
MSc in Computer Science
UNYT-UoG

Dr. Marenglen Biba
18-19-20 December 2009



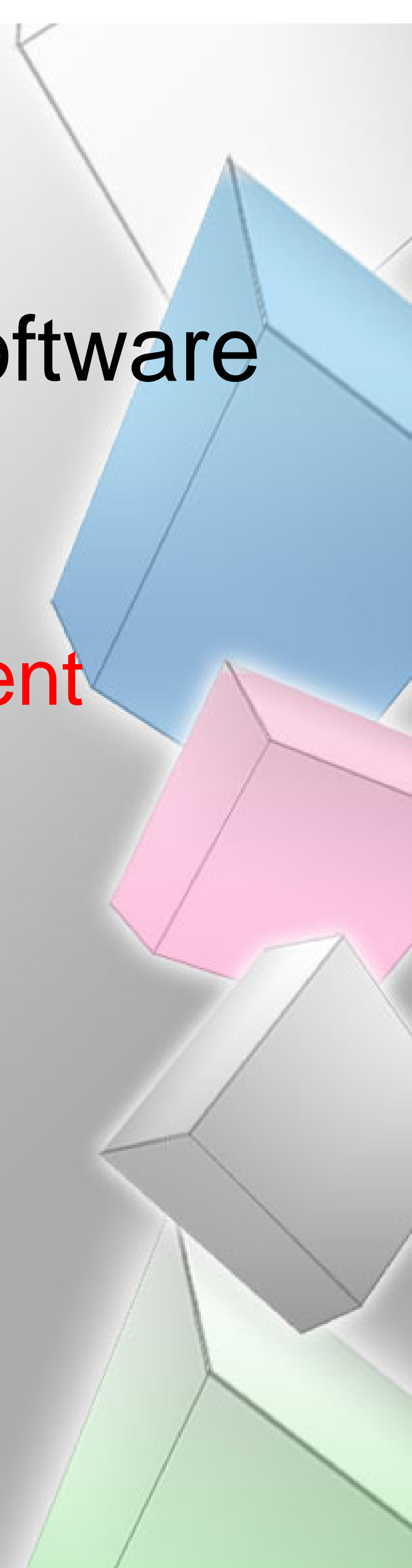
Lesson 1

01: Introduction
02: Architectures
03: Processes
04: Communication
05: Naming
06: Synchronization
07: Consistency & Replication
08: Fault Tolerance
09: Security
10: Distributed Object-Based Systems
11: Distributed File Systems
12: Distributed Web-Based Systems
13: Distributed Coordination-Based Systems



Distributed Systems

- A distributed system is a piece of software that ensures that:
  - a collection of independent computers appears to its users as a single coherent system
- Two aspects:
  - independent computers
  - single system => middleware.



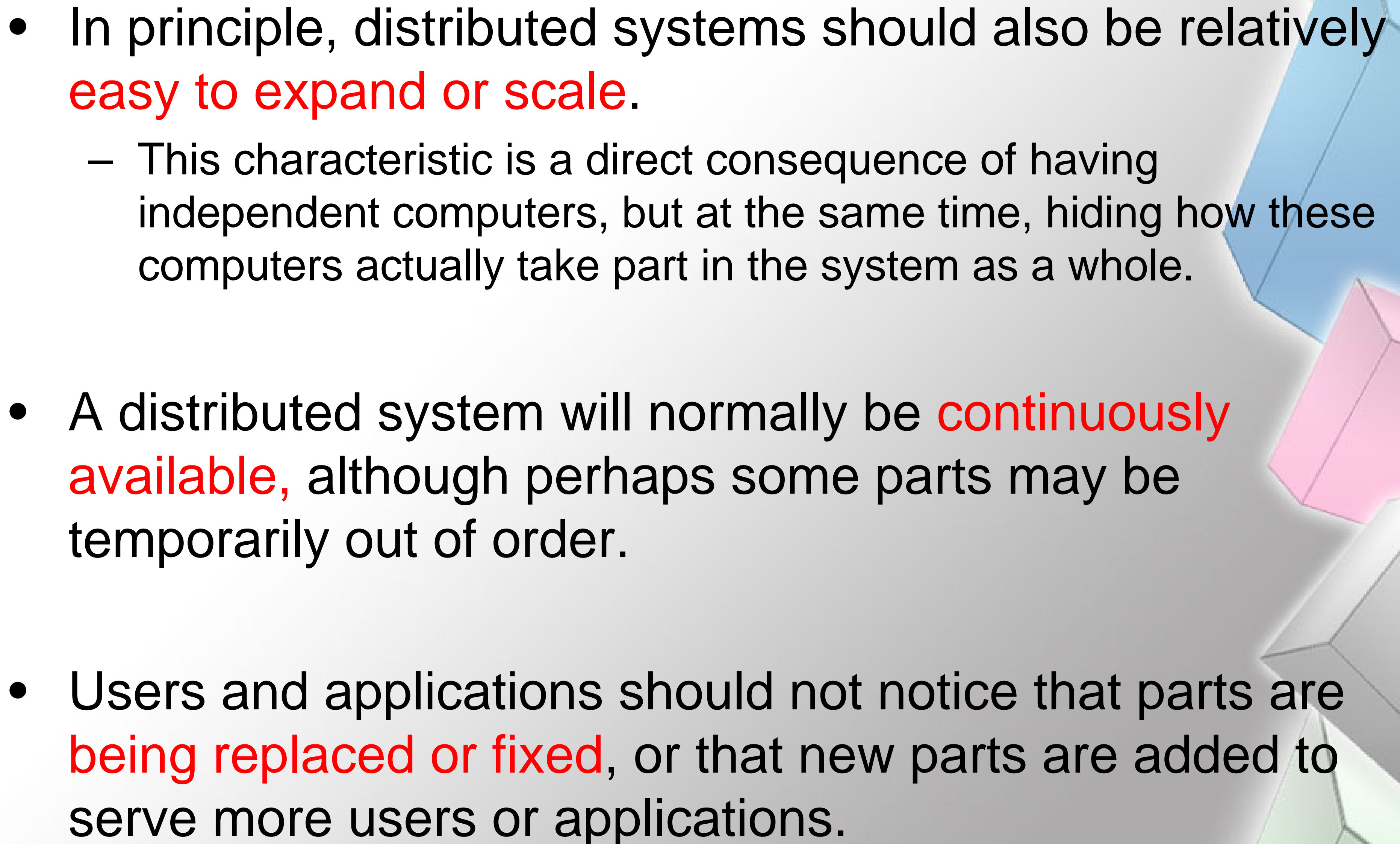
Distributed Systems

- One important characteristic of DSs is that differences between the various computers and the ways in which they communicate are mostly hidden from users.

- The same holds for the internal organization of the distributed system.

- Another important characteristic is that users and applications can interact with a distributed system in a consistent and uniform way, regardless of where and when interaction takes place.

Distributed Systems



- In principle, distributed systems should also be relatively easy to expand or scale.
  - This characteristic is a direct consequence of having independent computers, but at the same time, hiding how these computers actually take part in the system as a whole.
- A distributed system will normally be continuously available, although perhaps some parts may be temporarily out of order.
- Users and applications should not notice that parts are being replaced or fixed, or that new parts are added to serve more users or applications.

Middleware

- In order to support heterogeneous computers and networks while offering a single-system view, distributed systems are often organized by means of a layer of software-that is, logically placed between:
  - a higher-level layer consisting of users and applications,
  - and a layer underneath consisting of operating systems and basic communication facilities,
- Accordingly, such a distributed system is sometimes called middleware.

Middleware

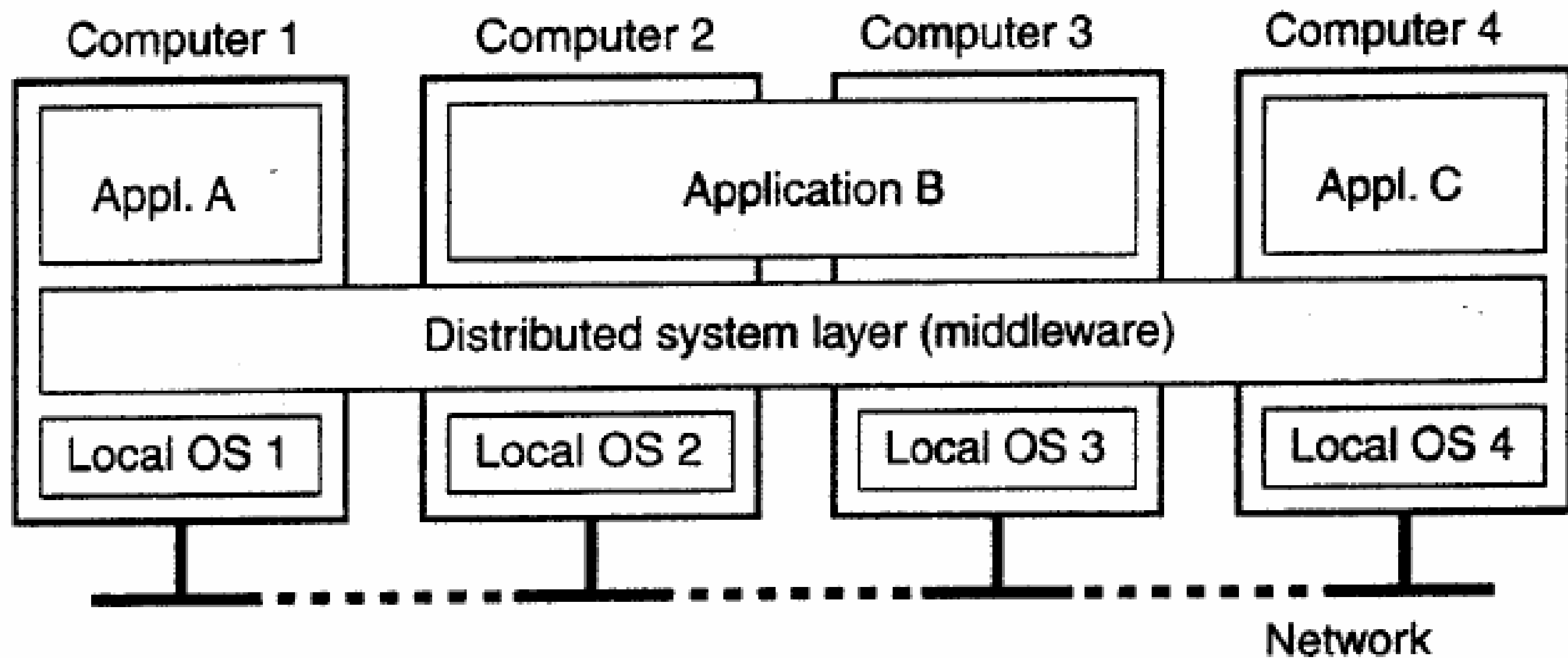
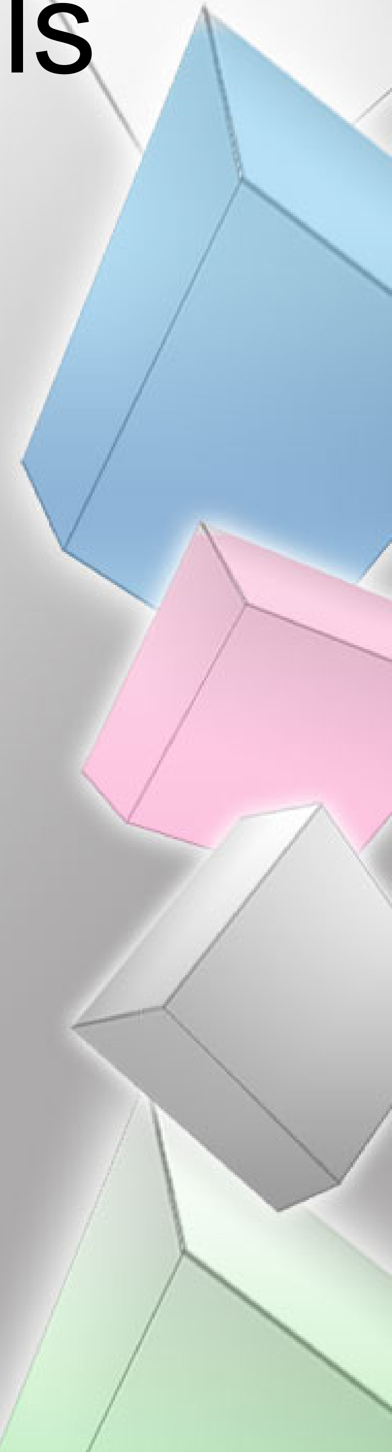


Figure I-I. A distributed system organized as middleware. The middleware layer extends over multiple machines, and offers each application the same interface.

Distributed Systems Goals

- Making resources available
- Distribution transparency
- Openness
- Scalability



Making resources available

- 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.

- Resources can be just about anything, but typical examples include things like printers, computers, storage facilities, data, files, Web pages, and networks, to name just a few.

Making resources available

- There are many reasons for wanting to share resources.
- One obvious reason is that of economics. For example, it is cheaper to let a printer be shared by several users in a small office than having to buy and maintain a separate printer for each user.
- Likewise, it makes economic sense to share costly resources such as supercomputers, high-performance storage systems, image setters, and other expensive peripherals.

Making resources available

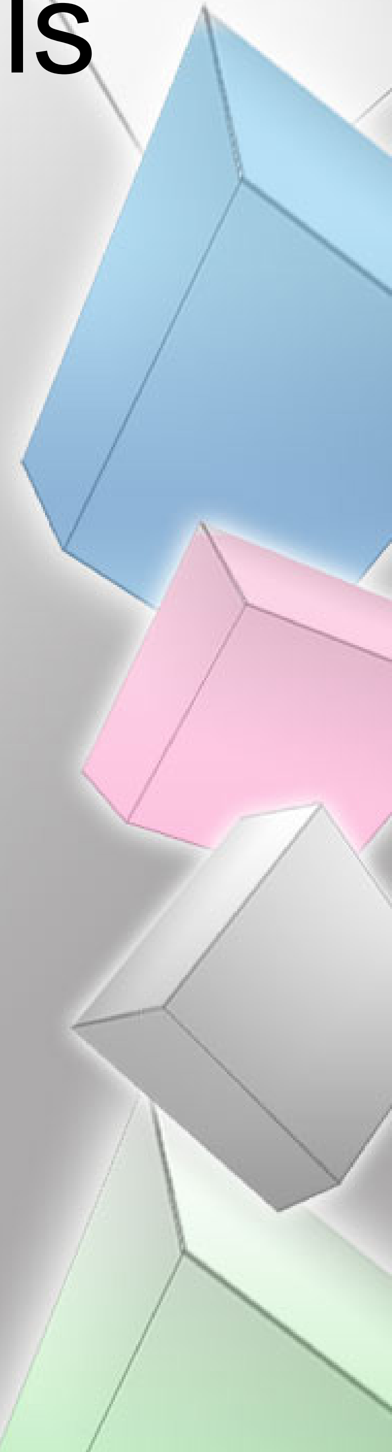
- Connecting users and resources also makes it easier to collaborate and exchange information, as is clearly illustrated by the success of the Internet with its simple protocols for exchanging files, mail. documents, audio, and video.

- The connectivity of the Internet is now leading to numerous virtual organizations in which geographically widely-dispersed groups of people work together by means of groupware, that is, software for collaborative editing, teleconferencing, and so on.

- Likewise, the Internet connectivity has enabled electronic commerce allowing us to buy and sell all kinds of goods without actually having to go to a store or even leave home.

Distributed Systems Goals

- Making resources available
- Distribution transparency
- Openness
- Scalability



Distribution transparency

- An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers.

- A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent.

- There are different kinds of transparency in distributed systems.

Distribution transparency

Transp.	Description
Access	Hides differences in data representation and invocation mechanisms
Location	Hides where an object resides
Migration	Hides from an object the ability of a system to change that object's location
Relocation	Hides from a client the ability of a system to change the location of an object to which the client is bound
Replication	Hides the fact that an object or its state may be replicated and that replicas reside at different locations
Concurrency	Hides the coordination of activities between objects to achieve consistency at a higher level
Failure	Hides failure and possible recovery of objects

Access transparency

- Access transparency deals with hiding differences in data representation and the way that resources can be accessed by users.
- At a basic level, we wish to hide differences in machine architectures, but more important is that we reach agreement on how data is to be represented by different machines and operating systems.
  - For example, a distributed system may have computer systems that run different operating systems, each having their own file-naming conventions.
- Differences in naming conventions, as well as how files can be manipulated, should all be hidden from users and applications.

Location transparency

- 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.

- In particular, location transparency can be achieved by assigning only logical names to resources, that is, names in which the location of a resource is not secretly encoded.

- An example of such a name is the URL *http://www.prenhall.com/index.html.* which gives no clue about the location of Prentice Hall's main Web server. The URL also gives no clue as to whether *index.html* has always been at its current location or was recently moved there.

- Distributed systems in which resources can be moved without affecting how those resources can be accessed are said to provide migration transparency.

Replication transparency

- Replication plays a very important role in distributed systems.
- For example, resources may be replicated to increase availability or to improve performance by placing a copy close to the place where it is accessed.
- Replication transparency deals with hiding the fact that several copies of a resource exist.
- To hide replication from users, it is necessary that all replicas have the same name. Consequently, a system that supports replication transparency should generally support location transparency as well, because it would otherwise be impossible to refer to replicas at different locations.

Concurrency transparency

- An important goal of distributed systems is to allow sharing of resources.
  - In many cases, sharing resources is done in a cooperative way, as in the case of communication.
- However. there are also many examples of competitive sharing of resources.
  - For example, two independent users may each have stored their files on the same file server or may be accessing the same tables in a shared database. In such cases, it is important that each user does not notice that the other is making use of the same resource.
- This phenomenon is called concurrency transparency.
- An important issue is that concurrent access to a shared resource leaves that resource in a consistent state. Consistency can be achieved through locking mechanisms,

Failure Trasparency

- A popular alternative definition of a distributed system, due to Leslie Lamport, is:

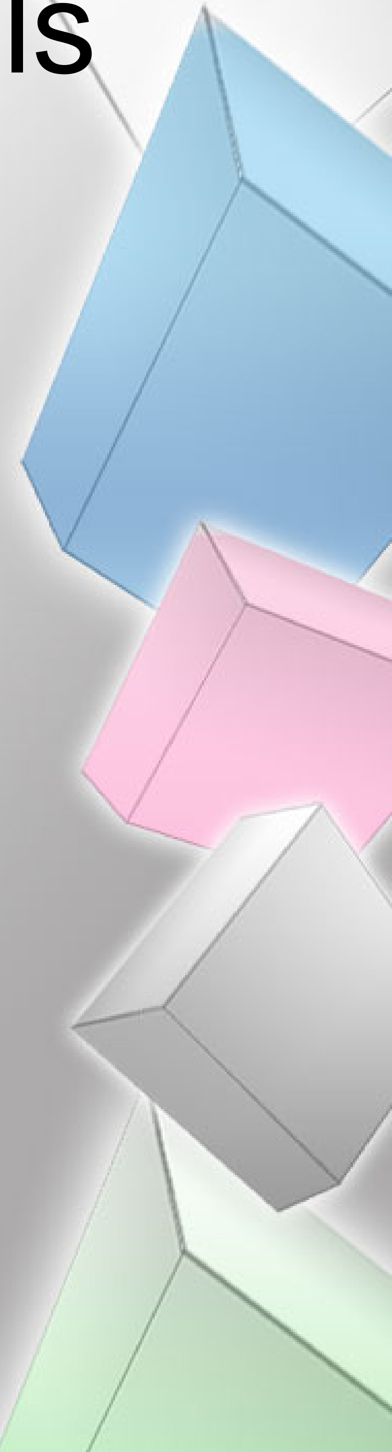
  "You know you have one when the crash of a computer you've never heard of stops you from getting any work done."

- This description puts the finger on another important issue of distributed systems design: dealing with failures.

- 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.

Distributed Systems Goals

- Making resources available
- Distribution transparency
- Openness
- Scalability



Openness

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

- For example, in computer networks, standard rules govern the format, contents, and meaning of messages sent and received. Such rules are formalized in protocols.

- In distributed systems, services are generally specified through interfaces, which are often described in an Interface Definition Language (IDL).

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.

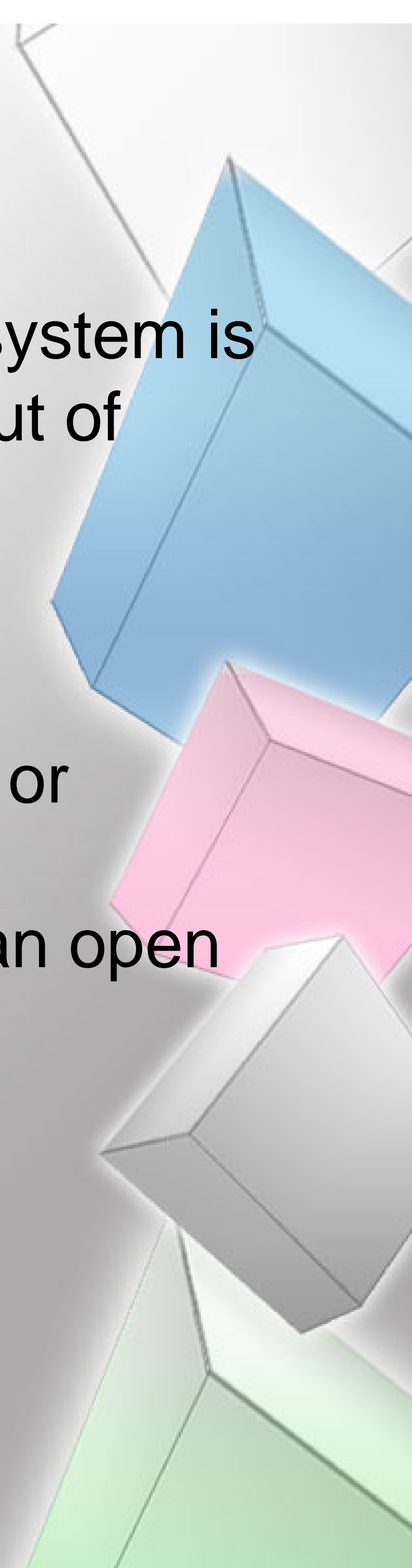
Interfaces

- If properly specified, an interface definition allows an arbitrary process that needs a certain interface to talk to another process that provides that interface.
- It also allows two independent parties to build completely different implementations of those interfaces, leading to two separate distributed systems that operate in exactly the same way.
- Proper specifications are complete and neutral.
  - Complete means that everything that is necessary to make an implementation has indeed been specified.
  - Important is the fact that specifications do not prescribe what an implementation should look like: they should be neutral.

Extensibility

- Another important goal for an open distributed system is that it should be easy to configure the system out of different components (possibly from different developers).

- Also, it should be easy to add new components or replace existing ones without affecting those components that stay in place. In other words, an open distributed system should also be extensible.

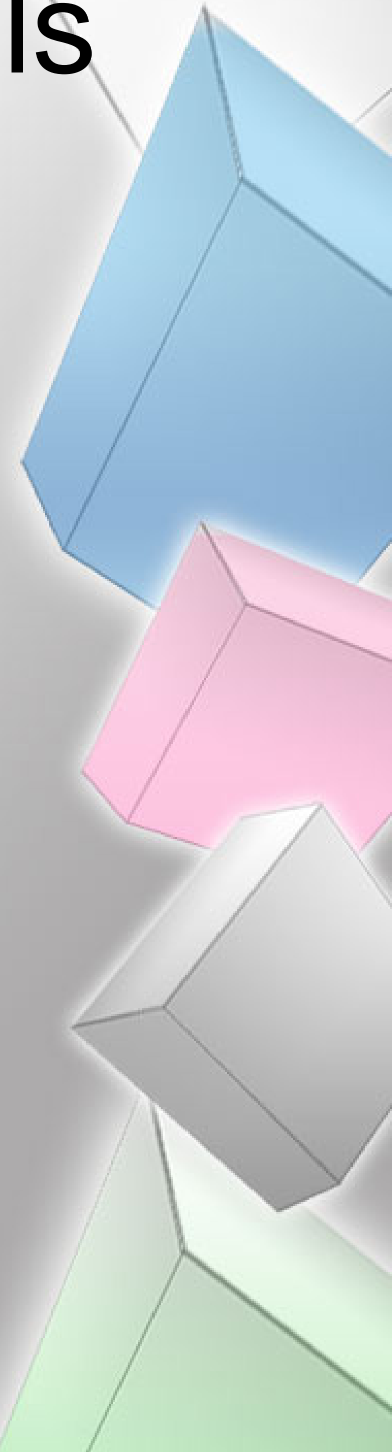


Separating Policy from Mechanism

- To achieve flexibility in open distributed systems, it is crucial that the system is organized as a collection of relatively small and easily replaceable or adaptable components.
- Many older and even contemporary systems are constructed using a **monolithic approach** in which components are only logically separated but implemented as one huge program.
- What we need is a separation between policy and mechanism.
- In the case of Web caching, for example, a browser should ideally provide facilities for only storing documents, and at the same time allow users to decide which documents are stored and for how long (Think of Firefox for example).

Distributed Systems Goals

- Making resources available
- Distribution transparency
- Openness
- Scalability



Scalability

- 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.
- Unfortunately, a system that is scalable in one or more of these dimensions often exhibits some loss of performance as the system scales up.

Challenges of scalability

- At least three components:
  - Number of users and/or processes (size scalability)
  - Maximum distance between nodes (geographical scalability)
  - Number of administrative domains (administrative scalability)

Observation

- Most systems account only, to a certain extent, for size scalability. The (non)solution: powerful servers.
- Today, the challenge lies in geographical and administrative scalability.

Scalability problems

- Let’s consider scaling with respect to size.
- If more users or resources need to be supported, we are often confronted with the limitations of centralized services, data, an

  algorithms.

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

Figure 1-3. Examples of scalability limitations.

Centralized Services

- For example, many services are centralized in the sense that they are implemented by means of only a single server running on a specific machine in the distributed system.

- The problem with this scheme is obvious: the server can become a bottleneck as the number of users and applications grows.

- Even if we have virtually unlimited processing and storage capacity, communication with that server will eventually prohibit further growth.

Centralized Services

- Unfortunately. using only a single server is sometimes unavoidable.
- Imagine that we have a service for managing highly confidential information such as medical records, bank accounts. and so on.
  - In such cases, it may be best to implement that service by means of a single server in a highly secured separate room, and protected from other parts of the distributed system through special network components.
  - Copying the server to several locations to enhance performance maybe out of the question as it would make the service less secure.

- How should we keep track of the telephone numbers and addresses of 50 million people?
- Imagine how the Internet would work if its Domain Name System (DNS) was still implemented as a single table.

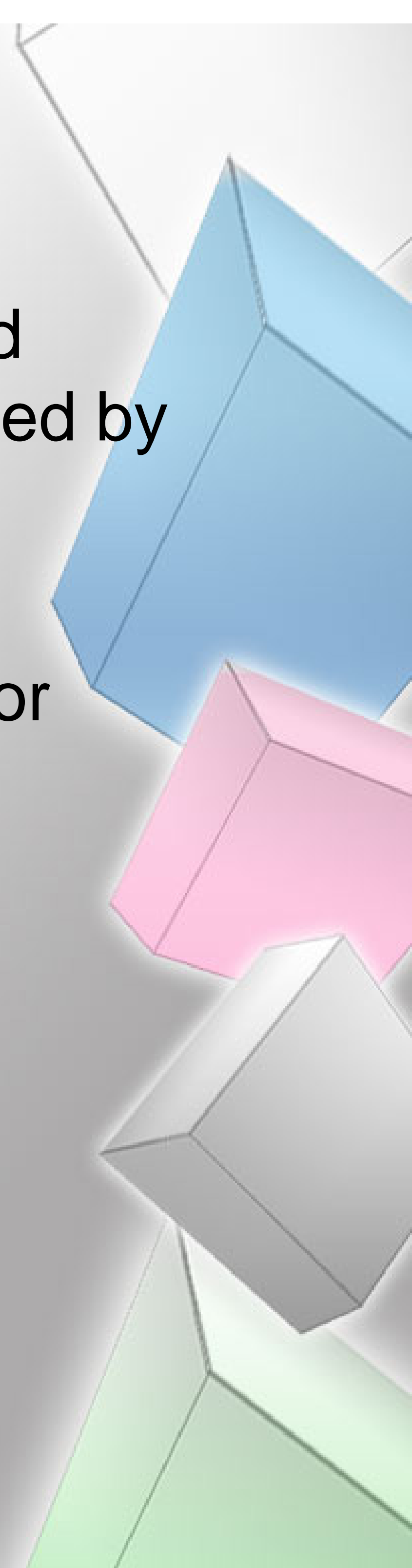
Decentralized Services

- Only decentralized algorithms should be used!
- These algorithms generally have the following characteristics, which distinguish them from centralized algorithms:
  1. No machine has complete information about the system state.
  2. Machines make decisions based only on local information,
  3. Failure of one machine does not ruin the algorithm.
  4. There is no implicit assumption that a global clock exists.

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
  - replication



Hiding communication latencies

- Avoid waiting for responses; do something else:
  - Make use of asynchronous communication
  - Have separate handler for incoming response
  - Problem: not every application fits this model
    - In interactive applications when a user sends a request he will generally have nothing better to do than to wait for the answer.
    - In such cases, a much better solution is to reduce the overall communication, for example, by moving part of the computation that is normally done at the server to the client process requesting the service.

Reducing overall communication

- A typical case where this approach works is accessing databases using forms.
  - Filling in forms can be done by sending a separate message for each field, and waiting for an acknowledgment from the server.
  - For example, the server may check for syntactic errors before accepting an entry.

- A much better solution is to ship the code for filling in the form, and possibly checking the entries, to the client, and have the client return a completed form.
  - This approach of shipping code is now widely supported by the Web in the form of Java applets and Javascript.

Shipping code to clients

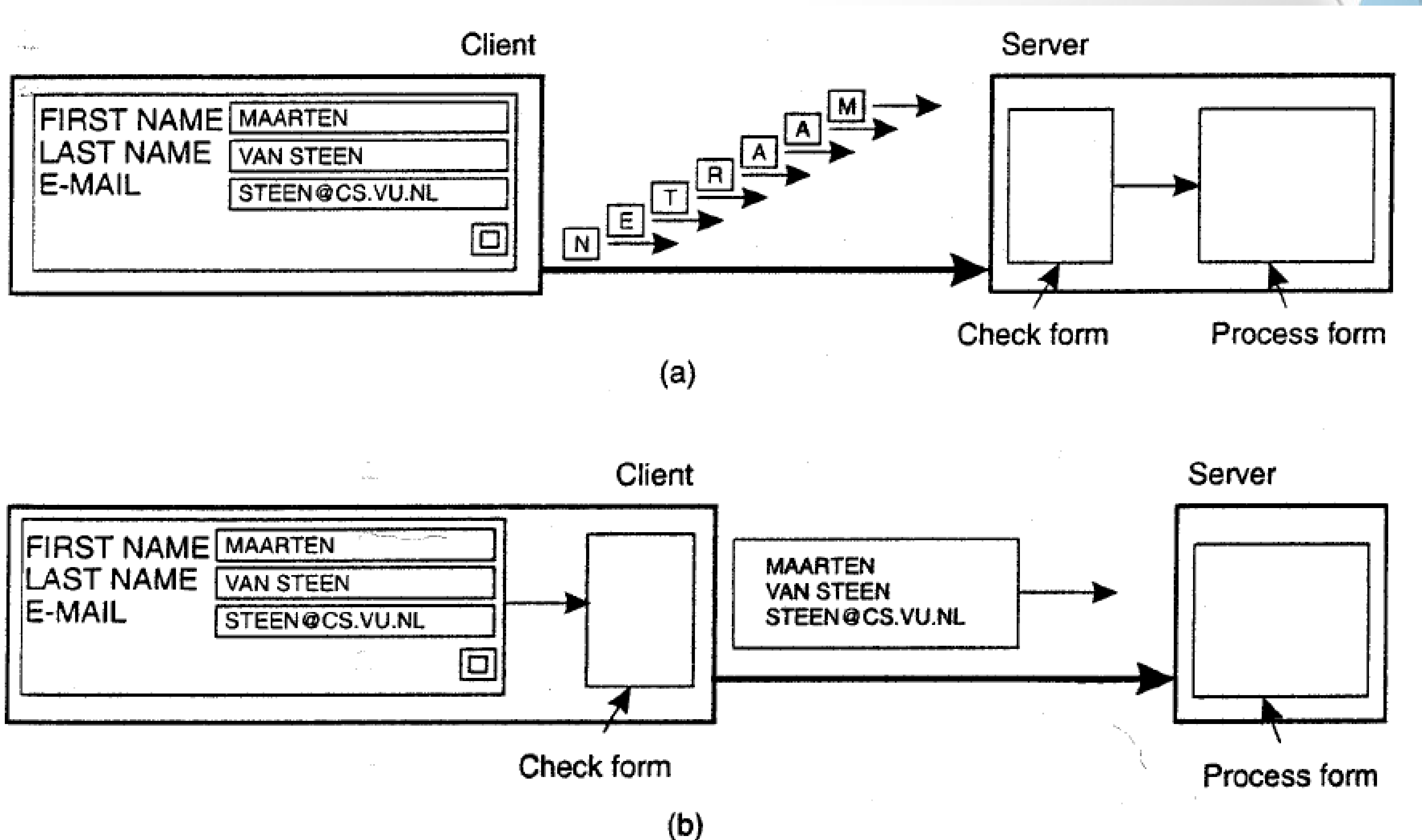


Figure 1-4. The difference between letting (a) a server or (b) a client check forms as they are being filled.

Distribution

- Another important scaling technique is distribution.
- Distribution involves taking a component, splitting it into smaller parts, an subsequently spreading those parts across the system.
- An excellent example of distribution is the Internet Domain Name System (DNS).
  - The DNS name space is hierarchically organized into a tree of **domains,** which are divided into non overlapping **zones.**
  - The names in each zone are handled by a single name server.
  - One can think of each path name, being the name of a host in the Internet, and thus associated with a network address of that host.
  - Basically, resolving a name means returning the network address of the associated host.

Domain Name System

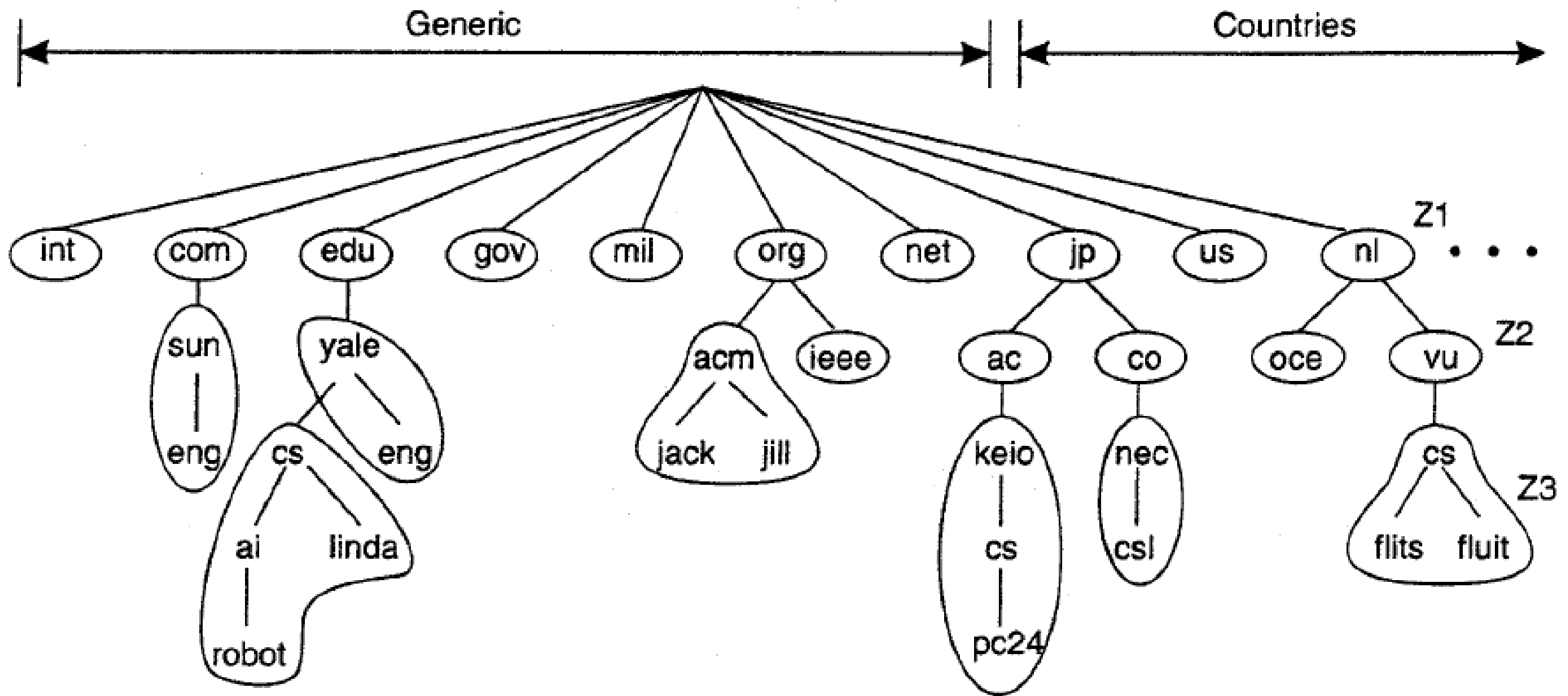


Figure 1-5. An example of dividing the DNS name space into zones.

WWW

- As another example, consider the World Wide Web. To most users, the Web appears to be an enormous document-based information system in which each document has its own unique name in the form of a URL.
  - Conceptually, it may even appear as if there is only a single server.
  - However, the Web is physically distributed across a large number of servers, each handling a number of Web documents.
  - The name of the server handling a document is encoded into that document's URL.
  - It is only because of this distribution of documents that the Web has been capable of scaling to its current size.

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.
- Also, in geographically widely-dispersed systems, having a copy nearby can hide much of the communication latency problems.
- Make copies of data available at different machines:
  - Replicated file servers and databases
  - Mirrored Web sites
  - Web caches (in browsers and proxies)
  - File caching (at server and client)

Caching and replication

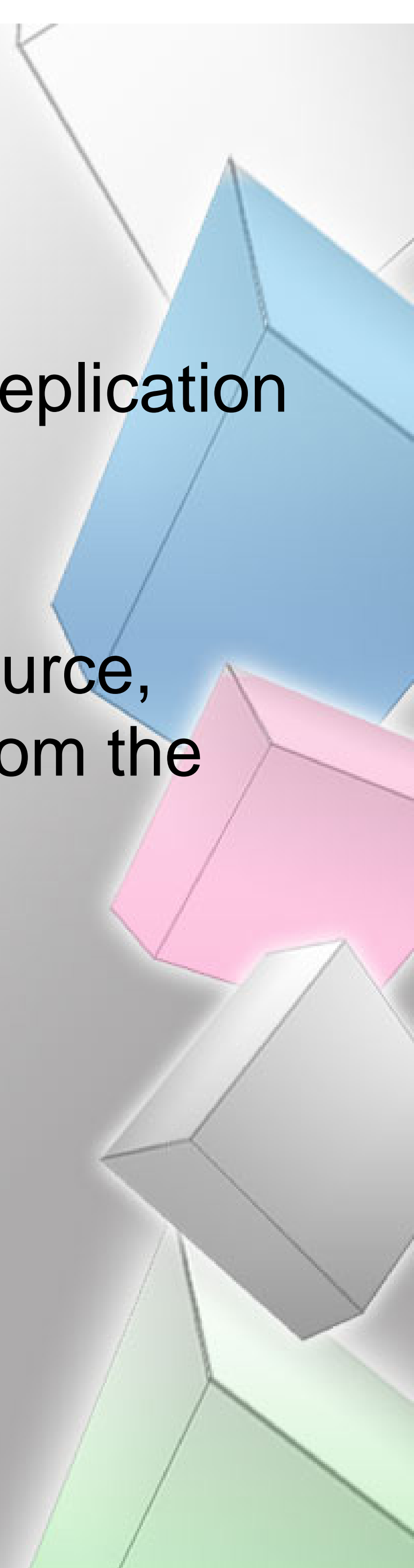
- Caching is a special form of replication, although the distinction between the two is often hard to make or even artificial.
- As in the case of replication, caching results in making a copy of a resource, generally in the proximity of the client accessing that resource.
- However, in contrast to replication, caching is a decision made by the client of a resource, and not by the owner of a resource.
- Also, caching happens on demand whereas replication is often planned in advance.

Replication, oops
copies are not equal ☺

- 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.



Inconsistency

- To what extent inconsistencies can be tolerated depends highly on the usage of a resource.
  - For example, many Web users find it acceptable that their browser returns a cached document of which the validity has not been checked for the last few minutes.
  - However, there are also many cases in which strong consistency guarantees need to be met, such as in the case of electronic stock exchanges and auctions.
- The problem with strong consistency is that an update must be immediately propagated to all other copies.
  - **Moreover, if two updates happen concurrently, it is often also required that each copy is updated in the same order!!!.**
- Situations such as these generally require some global synchronization mechanism.
- Unfortunately, such mechanisms are extremely hard or even impossible to implement in a scalable way, as she insists that photons and electrical signals obey a speed limit of 187 *miles/msec* (the speed of light).

Is Scaling really feasible? ☹/☺

- When considering these scaling techniques, one could argue that size scalability is the least problematic from a technical point of view.
  - In many cases, simply increasing the capacity of a machine will the save the day (at least temporarily and perhaps at significant costs).
- Geographical scalability is a much tougher problem as Mother Nature is getting in our way.
- Nevertheless, practice shows that combining distribution, replication, and caching techniques with different forms of consistency will often prove sufficient in many cases.

Pitfalls

- Developing distributed systems can be a formidable task.

- As we will see many times throughout this course, there are so many issues to consider at the same time that it seems that only complexity can be the result.

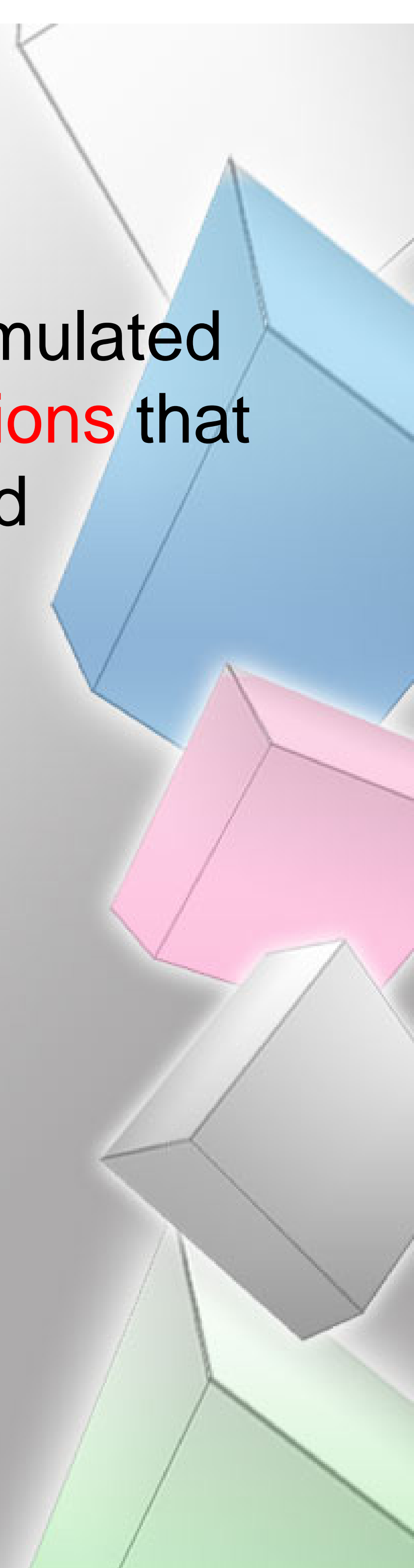
- However, by following a number of design principles, distributed systems can be developed that strongly adhere to the goals we set out here.

Principles: hold them tight

- Many principles follow the basic rules of decent software engineering and will not be repeated here.
- However, distributed systems differ from traditional software because components are dispersed across a network.
- Not taking this dispersion into account during design time is what makes so many systems needlessly complex and results in mistakes that need to be patched later on.

False assumptions

- Peter Deutsch, then at Sun Microsystems, formulated these mistakes as the following false assumptions that everyone makes when developing a distributed application for the first time:
    1. The network is reliable.
    2. The network is secure.
    3. The network is homogeneous.
    4. The topology does not change.
    5. Latency is zero.
    6. Bandwidth is infinite.
    7. Transport cost is zero.
    8. There is one administrator.

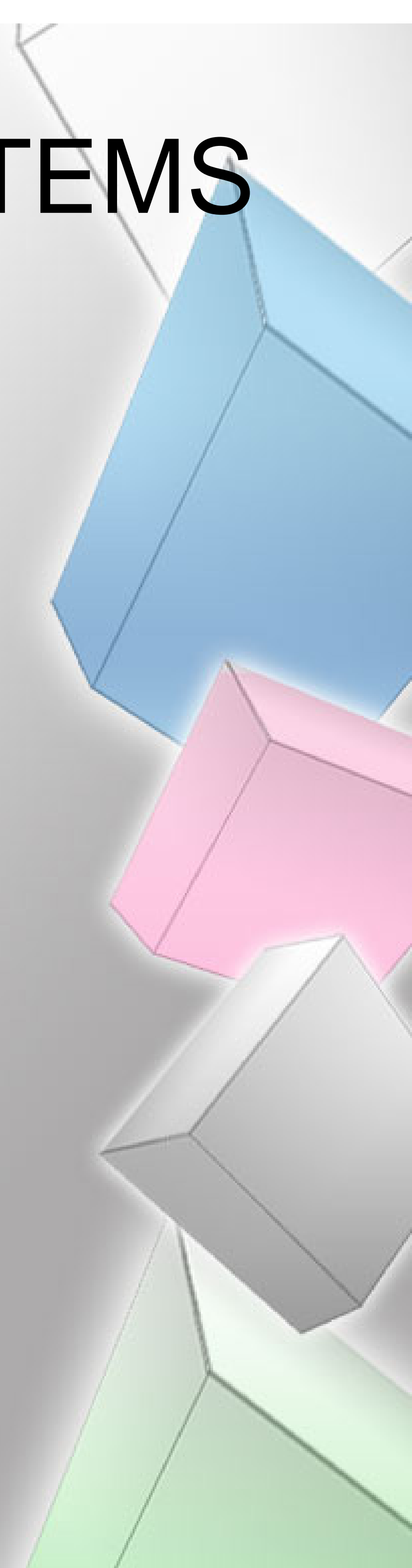


Short bio

- L Peter Deutsch or Peter Deutsch (born Laurence Peter Deutsch) is the founder of Aladdin Enterprises and creator of Ghostscript, a free software PostScript and Pdf interpreter.
- Deutsch's other work includes the definitive Smalltalk implementation that, among other innovations, inspired Java just-in-time technology 15 or-so years later.
- He also wrote the PDP-1 Lisp 1.5 implementation, Basic PDP-1 LISP, "while still in short pants" between the age of 12-15 years old.
- He is also the author of a number of RFCs, and the The Eight Fallacies of Distributed Computing.
- Deutsch received a Ph.D. in Computer Science from the University of California, Berkeley in 1973[1]. He has done stints at Xerox PARC and Sun Microsystems. In 1994 he was inducted as a Fellow of the Association for Computing Machinery.
- Deutsch changed his legal first name from Laurence to L on September 12, 2007[2].
- His published work and other public references before that time generally use the name L. Peter Deutsch (with a dot after the L).

TYPES OF DISTRIBUTED SYSTEMS

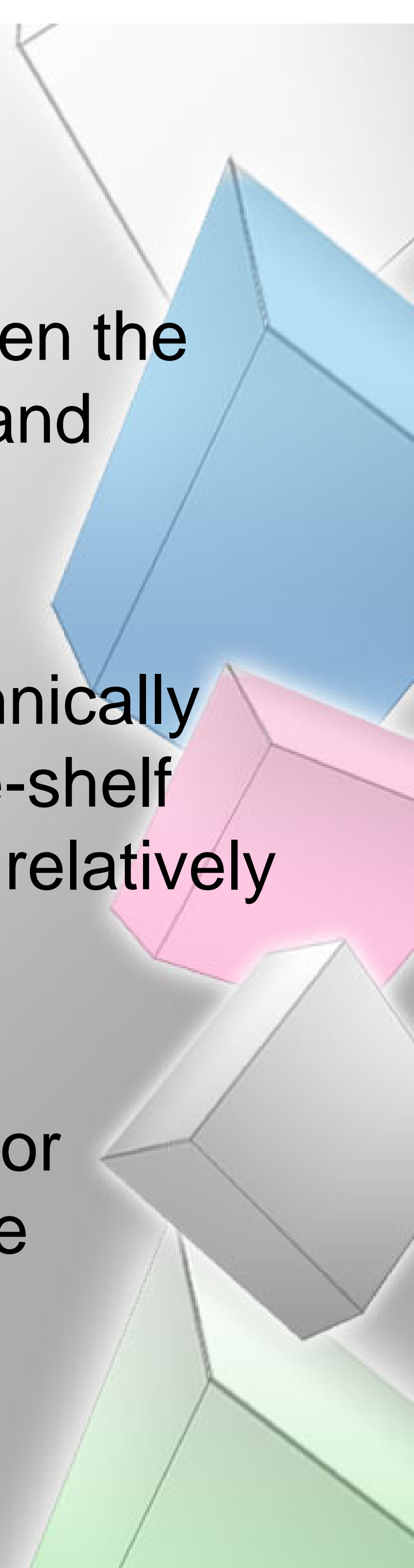
- Distributed Computing Systems
- Distributed Information Systems
- Distributed Pervasive Systems



Distributed Computing Systems

- An important class of distributed systems is the one used for high-performance computing tasks.
- In cluster computing the underlying hardware consists of a collection of similar workstations or PCs, closely connected by means of a highspeed local-area network.
  - In addition, each node runs the same operating system.
- The situation becomes quite different in the case of grid computing.
- Grids consist of distributed systems that are often constructed as 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.

Cluster Computing



- Cluster computing systems became popular when the price/performance ratio of personal computers and workstations improved.

- At a certain point, it became financially and technically attractive to build a supercomputer using off-the-shelf technology by simply hooking up a collection of relatively simple computers in a high-speed network.

- In virtually all cases, cluster computing is used for parallel programming in which a single (compute intensive) program is run in parallel on multiple machines.

Cluster Computing: Linux-based Beowulf clusters

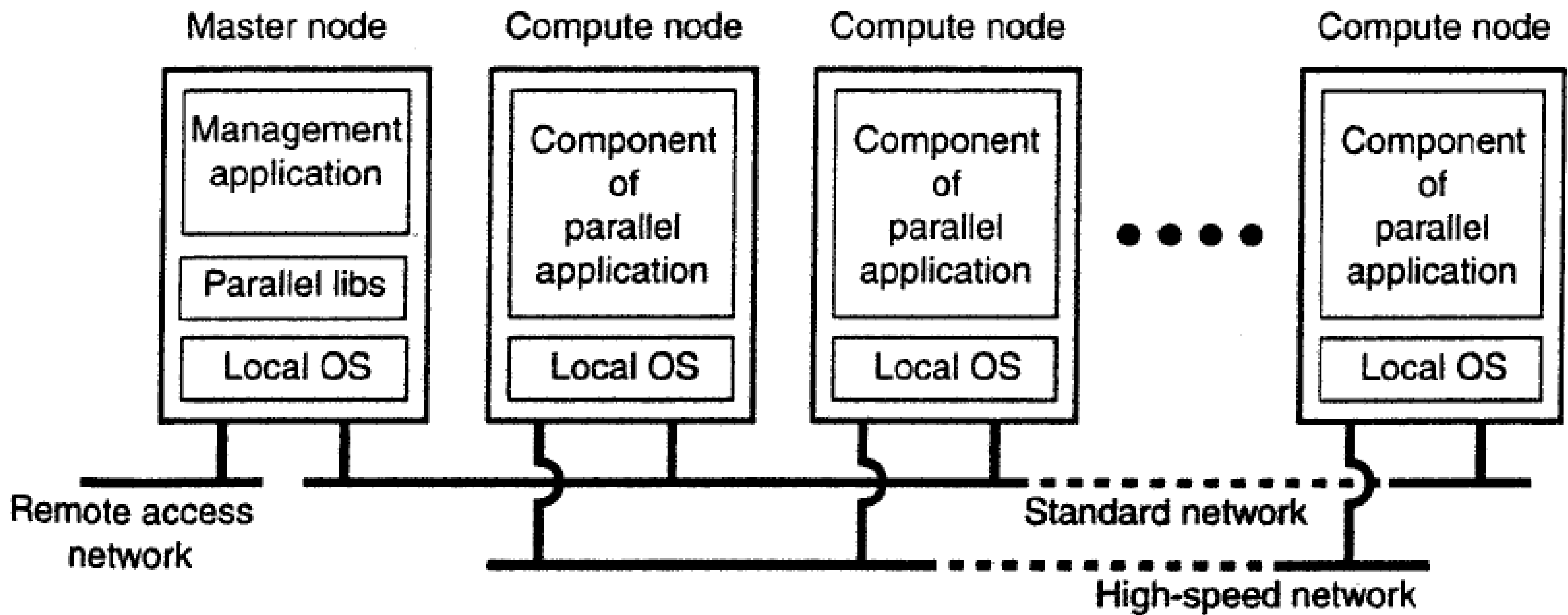


Figure 1-6. An example of a cluster computing system.

Cluster computing

Existing image web application

WWW

End user

Web server

Image metadata database

< Serving images >

< Image similarity queries >

Image file server

< Image data when adding images >

isk-daemon

isk-daemon

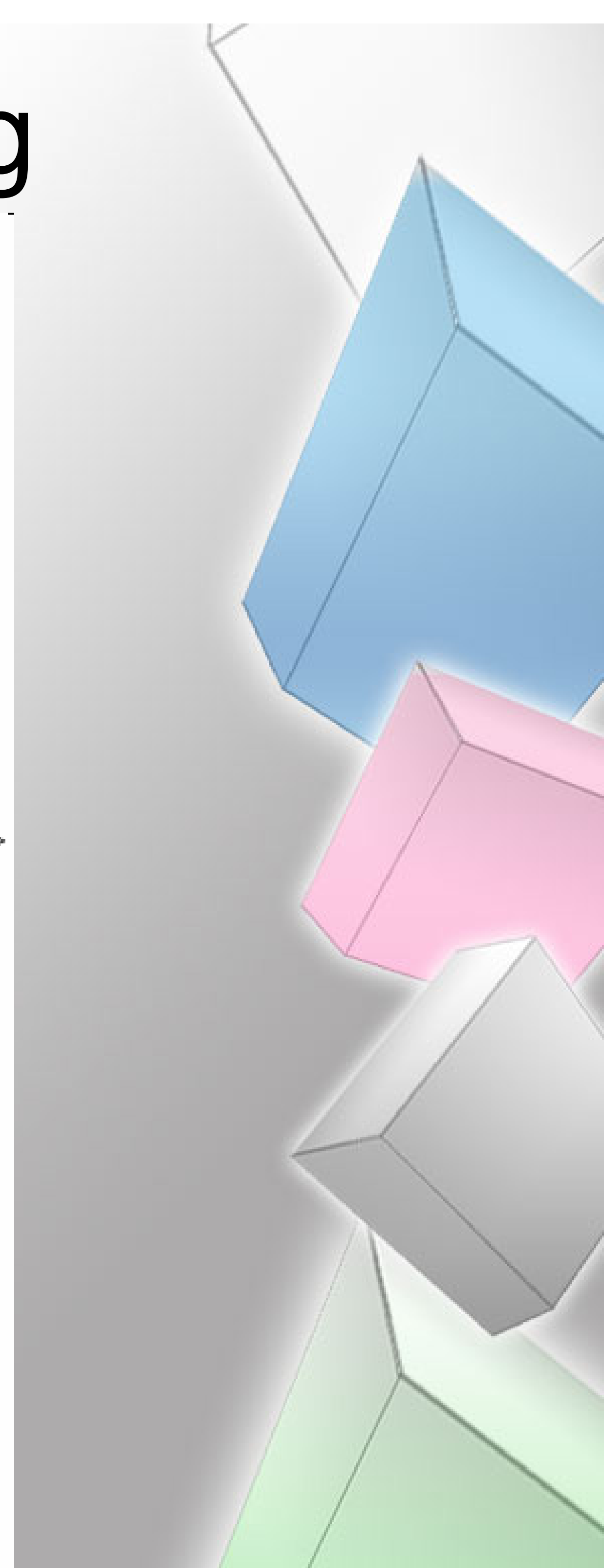
isk-daemon

Isk-daemon cluster

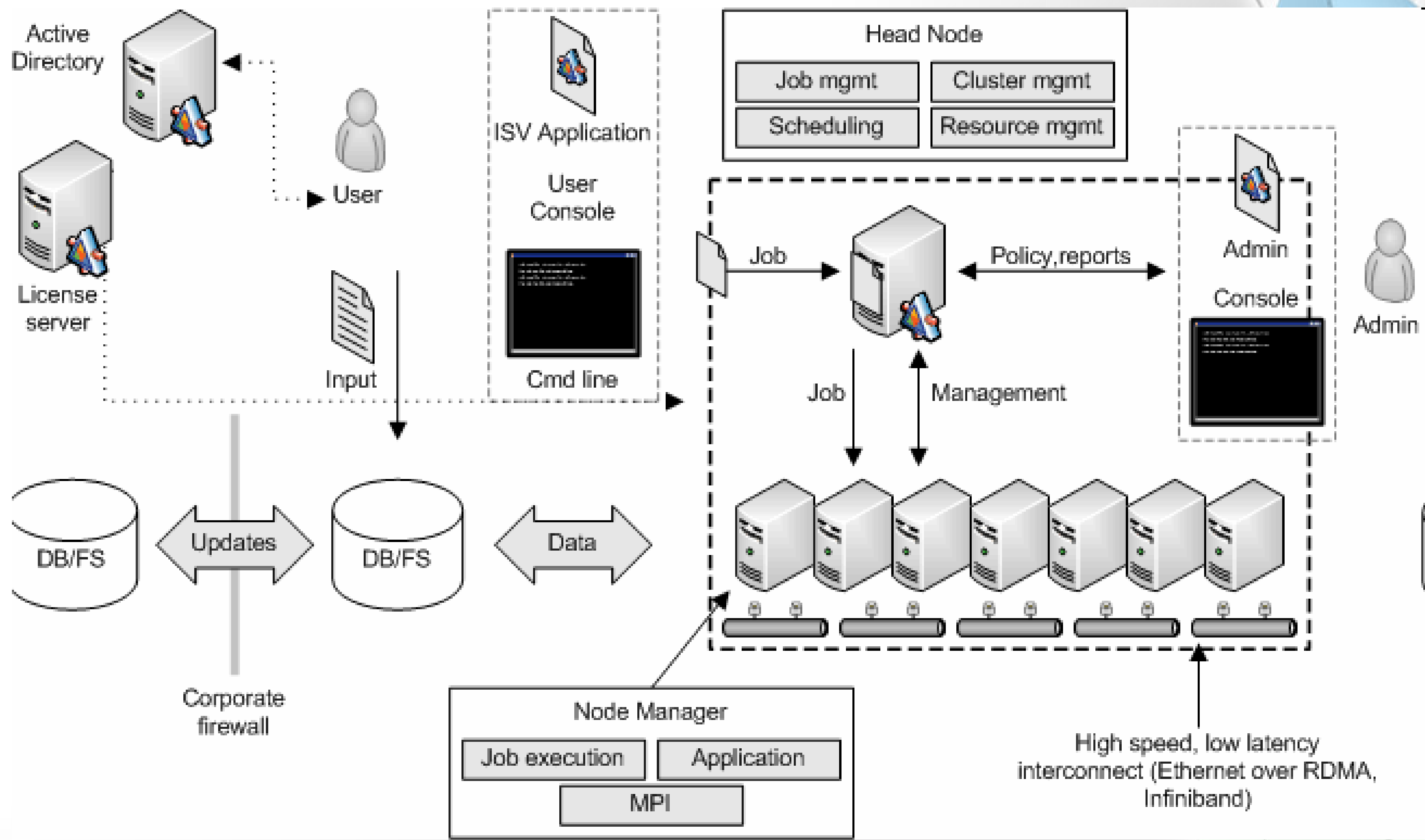
Memcached instance

Memcached instance

Memcached cluster



Cluster Architecture



Cluster Computing: MOSIX

- MOSIX is an on-line management system targeted for high performance computing on Linux clusters, multi-clusters and Clouds.
- It supports both interactive processes and batch jobs.
- MOSIX can be viewed as a multi-cluster operating system that incorporates automatic resource discovery and dynamic workload distribution, commonly found on single computers with multiple processors.
- http://www.mosix.org

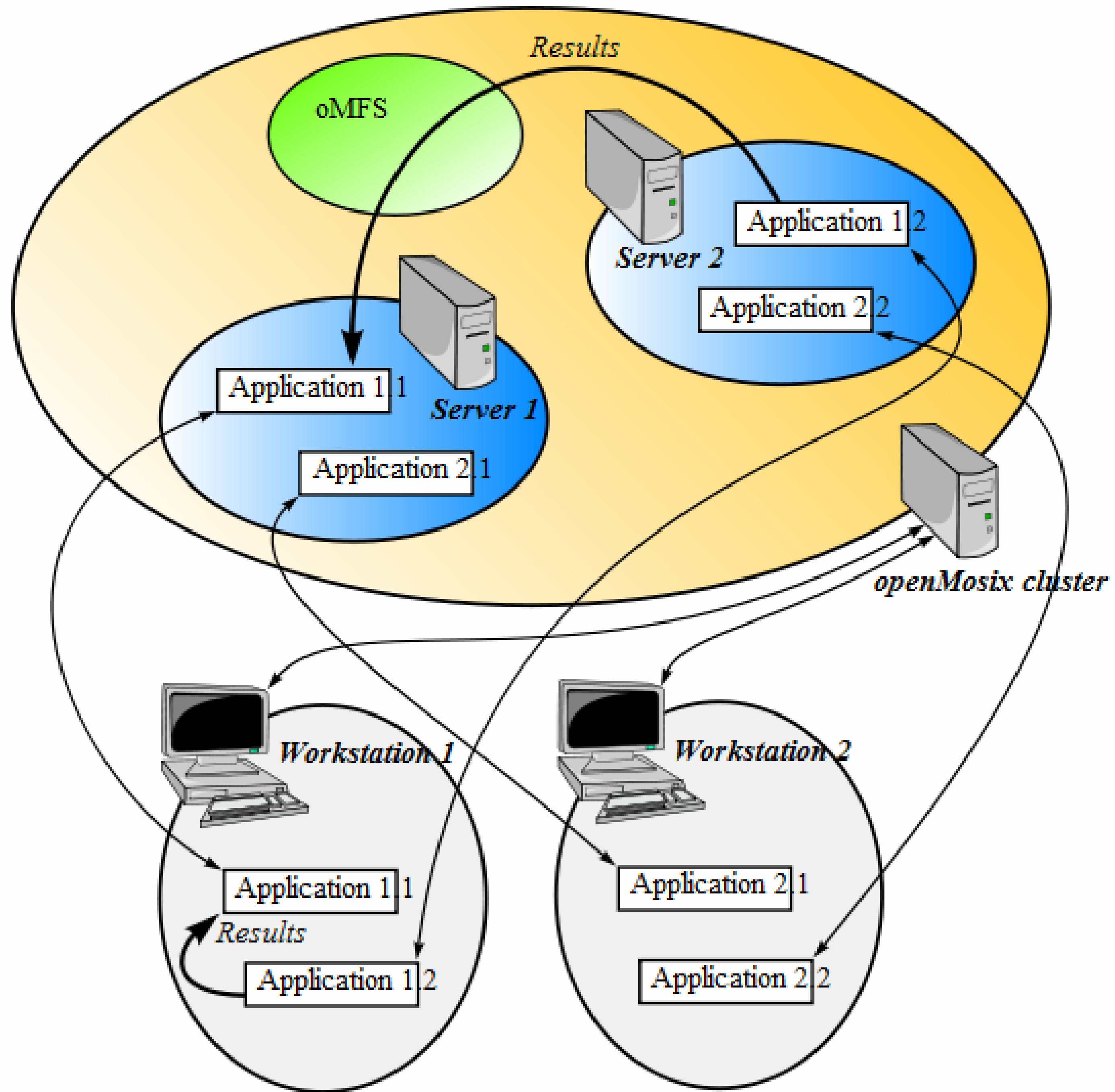
MOSIX

- **Efficient utilization of system-wide resources**- by automatic resource discovery and load-balancing.
- **Running applications with unpredictable resource requirements or run times.**
- **Running long processes** - which are automatically sent to nodes in remote clusters and are migrated back when these nodes are disconnected.
- **Combining nodes of different speeds** - by migrating processes among nodes based on their respective speeds, current load and available memory.

Applications:

- **Scientific applications** - genomic, protein sequences, molecular dynamics, quantum dynamics, nano-technology and other parallel HPC applications.
- **Engineering applications** - CFD, weather forecasting, crash simulations, oil industry, ASIC design, pharmaceutical and other HPC applications.
- **Financial modeling**, rendering farms, compilation farms.

OpenMosix: free



LinuxPMI

- LinuxPMI (Linux Process Migration Infrastructure) is a Linux Kernel extension for multi-system-image (in contrast to a single-system image) clustering.
- The project is a continuation of the abandoned openMosix clustering project.
- It is in many ways similar in principle to how a multiuser operating system manages workload on a multicpu system;
  - however, LinuxPMI can have machines (nodes) with several CPU's.
  - You can also add or remove nodes to a running cluster thus expanding the total computing power of the system.
- In short it means that many computers are able to work as one large computer;
  - however, there is no master machine, so each machine can be used as an individual workstation.

LinuxPMI

How LinuxPMI works is perhaps best described by an example:

You may have a network of 10 computers and there is one user working on each machine (node).

9 users are working on simple tasks that do not necessary use their machine to the full potential.

1 user is working on a program that spawns a lot of jobs that would normally overload the computer.

Since we have 9 computers on the network that have a lot of free resources, they can basically take over some of the jobs from the one computer that would normally be overloaded.

In other words, LinuxPMI will migrate jobs from busy computers to computers that are able to perform the same task faster.

Even if all 10 users were using their machines for heavy tasks, it could be that not all machines are fully occupied at the same time, and LinuxPMI will use these to reduce load on other machines.

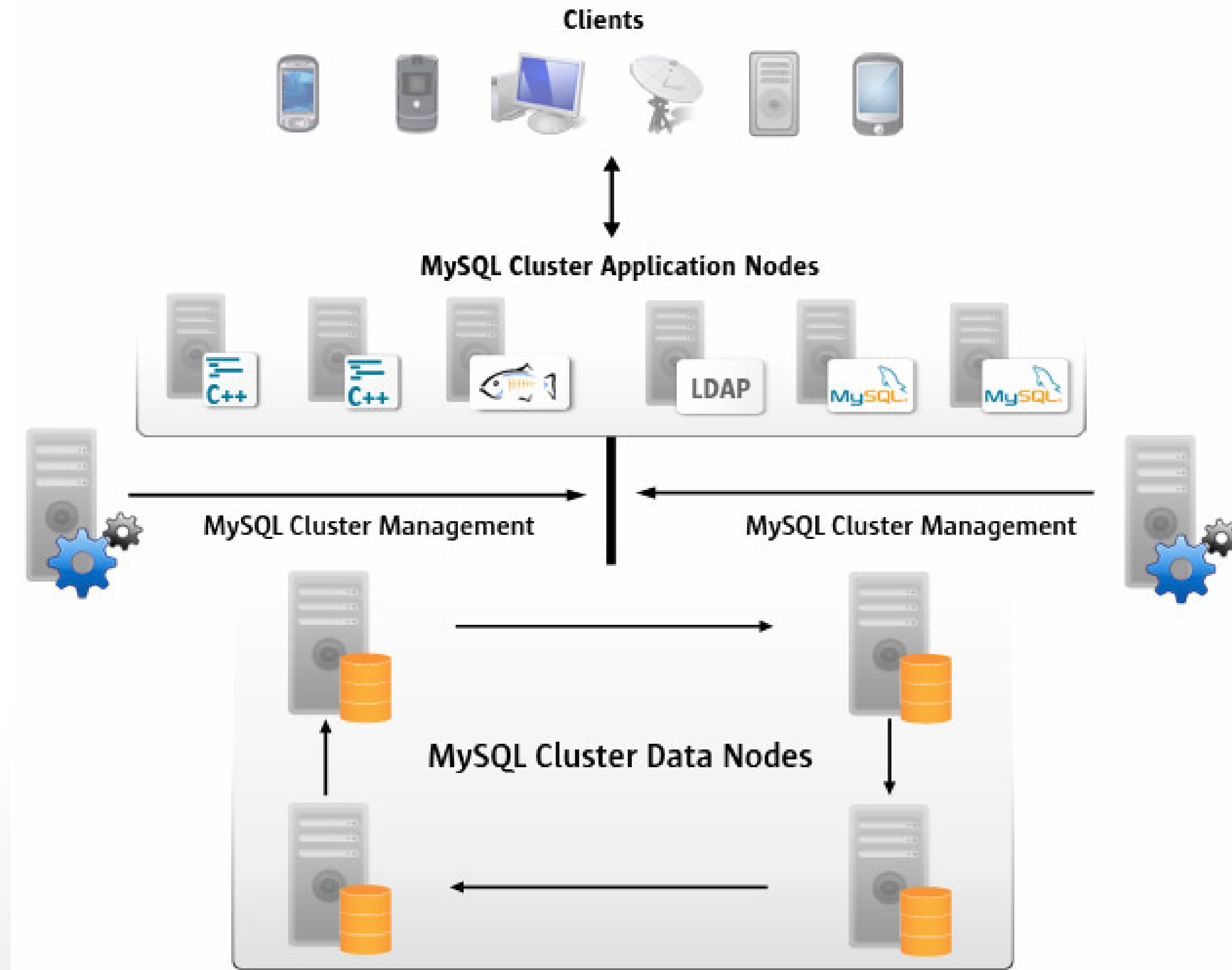
OpenSSI

- OpenSSI is an open source single-system image clustering system.
- It allows a collection of computers to be treated as one large system, allowing applications running on any one machine access to the resources of all the machines in the cluster.
- Processes running on any node have full access to the resources of all nodes. Processes can be migrated from node to node automatically to balance system utilization. Inbound network connections can be directed to the least loaded node available.
- It is possible to create an OpenSSI cluster with no single point of failure, for example the file system can be mirrored between two nodes, so if one node crashes the process accessing the file will *fail over* to the other node. Alternatively the cluster can be designed in such a manner that every node has direct access to the file system.

MySQL cluster

- MySQL Cluster is a high availability database which leverages a shared-nothing data storage architecture.
- The system consists of multiple nodes which can be distributed across hosts to ensure continuous availability in the event of a data node, hardware or network failure.
- MySQL Cluster Carrier Grade Edition uses a storage engine, consisting of a set of data nodes to store data, which is accessed through a native C++ API, Java, LDAP or standard SQL interface.

MySQL Cluster



MySQL Cluster

- By storing and distributing data in a shared-nothing architecture, i.e. without the use of a shared-disk, if a Data Node happens to fail, there will always at least one additional Data Node storing the same information.
  - This allows for requests and transactions to continue to be satisfied without interruption.
  - Data Nodes can also be added on-line, allowing for unprecedented scalability of data capacity and processing.

- **Application Nodes** are the applications connecting to the database. This can take the form of an application leveraging the high performance NDB API or the use of MySQL Servers which perform the function of SQL interfaces into the data stored within a cluster. Thus, applications can simultaneously access the data in MySQL Cluster using a rich set of interfaces, such as SQL, LDAP and web services. Moreover, additional Application Nodes can be added online.

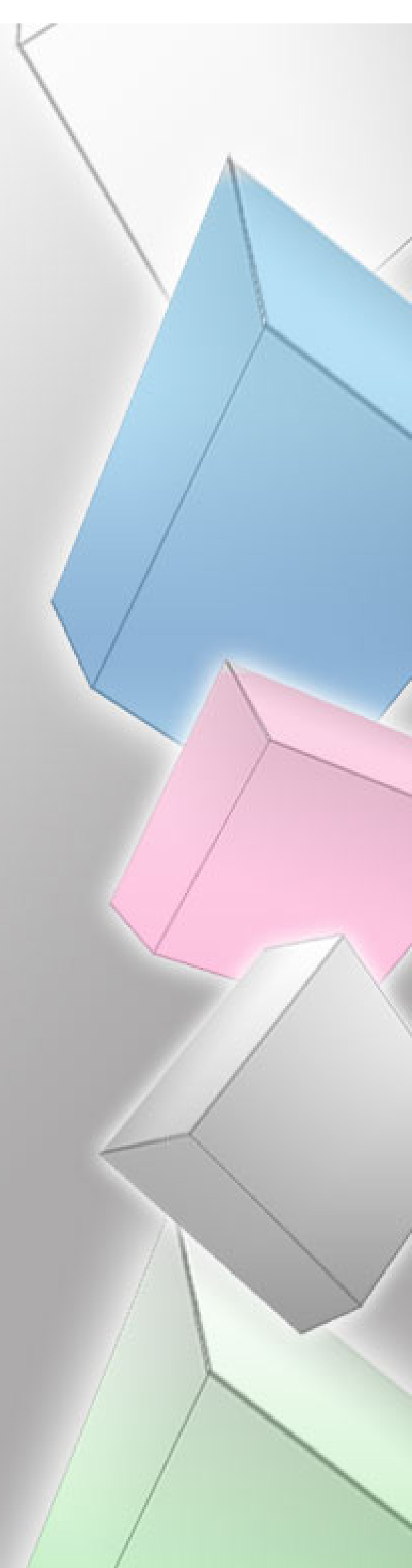
MySQL Cluster

- **Management Nodes** are responsible for managing the cluster and make cluster configuration information available to other nodes. The Management Nodes are used at startup and when there is a system reconfiguration. Management Nodes can be stopped and restarted without affecting the ongoing execution of the Data and Application Nodes. By default, the Management Nodes also provides arbitration services, in the event of a network failure leading to a "split-brain", or a cluster exhibiting "network-partitioning".

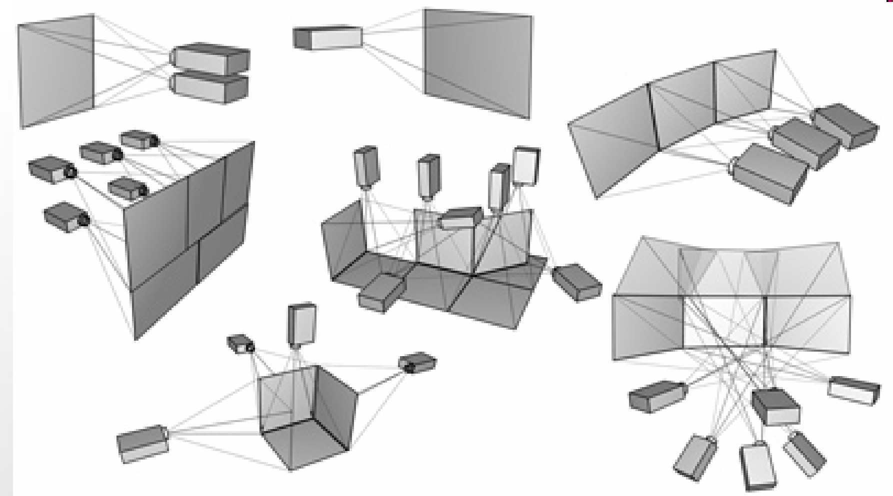
- With this distributed architecture, where dependencies have been minimized, applications continue to run and data remains consistent, even if any one of the Data, Application, or Management Nodes fail.

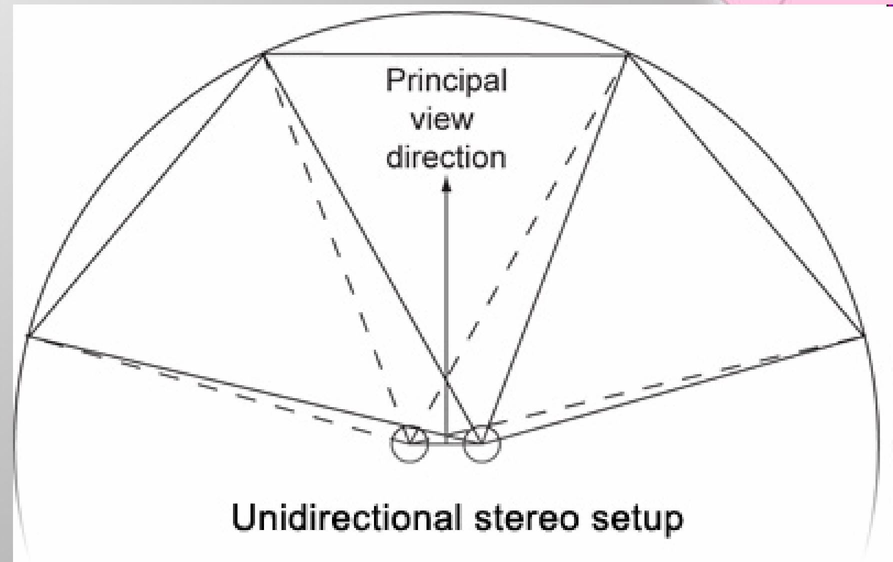
Clusters at NASA

Example: **The "Tholos" Virtual Reality Dome Theater**



Example: **The "Tholos" Virtual Reality Dome Theater**

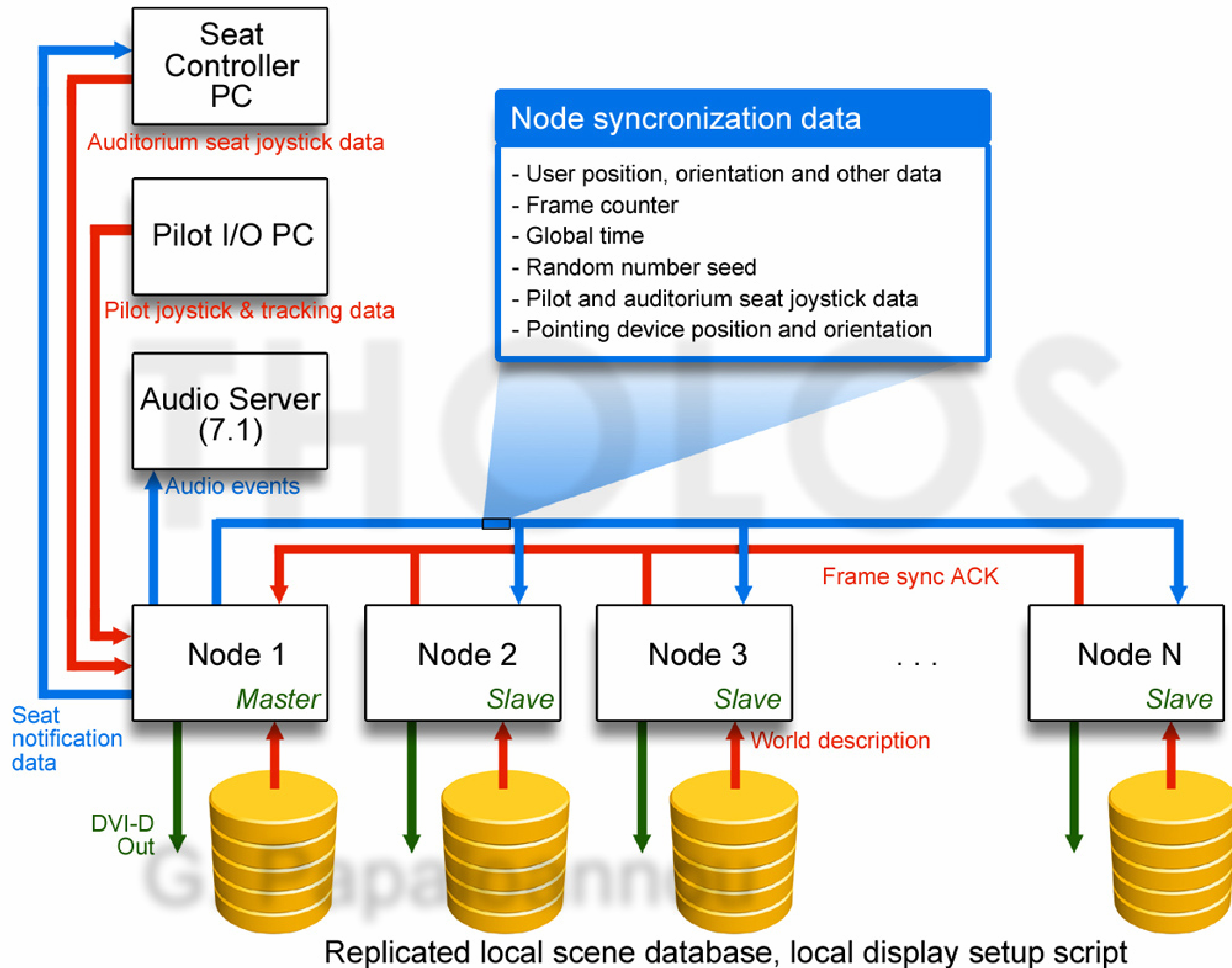




Tholos computing system

- In order to drive a multi-display environment, multiple graphics outputs need to be provided and synchronized to generate partial views of the same panorama (12 in our case). Graphics outputs have to be frame-locked at a physical layer and swap-locked in process level.
- Due to the high amount of rendered and simulation data, the corresponding processes that drive each display output need to run in parallel.
- The obvious viable solution for satisfying the rendering demands of such a display system is a virtual reality cluster.
- For the Tholos VR system, an asymmetric master/slave cluster configuration was designed and implemented, which provides a highly parallel execution and has almost zero scaling overhead (frame lag) when adding new nodes.

Example: **The "Tholos" Virtual Reality Dome Theater**



Too Fast, Too Furious

Rank	Rmax Rpeak (Tflops)	Name	Computer Processor cores	Maker	Site Country, Year
1	1759.00 2331.00	*Jaguar*	**Cray XT5** 224162 (Opteron)	Cray	Oak Ridge National Laboratory United States, 2008
2	1042.00 1375.78	*Roadrunner*	**BladeCenter QS22/LS21** 122400 (Cell/Opteron)	IBM	Los Alamos National Laboratory United States, 2008
3	831.70 1028.85	*Kraken*	**Cray XT5** 98928 (Opteron)	Cray	National Institute for Computational Sciences United States, 2008
4	825.50 1002.70	*JUGENE*	**Blue Gene/P Solution** 294912 (Power)	IBM	Jülich Research Centre Germany, 2009
5	563.10 1206.19	*Tianhe-I*	**NUDT TH-1** 71680 (Xeon), InfiniBand	NUDT	National SuperComputer Center China, 2009
6	544.30 673.26	*Pleiades*	**SGI Altix ICE 8200EX** 56320 (Xeon), InfiniBand	SGI	NASA/Ames Research Center United States, 2008
7	478.20 596.38	*Blue Gene/L*	**eServer Blue Gene Solution** 212992 (Power)	IBM	Lawrence Livermore National Laboratory United States, 2007
8	458.61 557.06	*Intrepid*[4]	**Blue Gene/P Solution** 163840 (Power)	IBM	Argonne National Laboratory United States, 2007
9	433.20 579.38	*Ranger*	**Sun Constellation System** 62976 (Opteron), Infiniband	Sun	Texas Advanced Computing Center United States, 2008
10	423.90 487.74	*Red Sky*	**Sun Constellation System** 41616 (Xeon), InfiniBand	Sun	Sandia National Laboratories United States, 2009

The Fast and the Furious

- Jaguar is a petascale supercomputer built by Cray at Oak Ridge National Laboratory in Oak Ridge, Tennessee.
- As of November 2009, it was the world's fastest computer with a peak performance of more than 1,750 teraflops (1.75 petaflops). A Cray XT5 system, Jaguar has 224,256 Opteron processor cores, and operated with a version of Linux.

Cray-XT5

- The XT5 family run the Cray Linux Environment, formerly known as UNICOS. This incorporates SUSE Linux Enterprise Server and Cray's Compute Node Linux.

- UNICOS is the name of a range of Unix-like operating system variants developed by Cray for its supercomputers. UNICOS is the successor of the Cray Operating System (COS). It provides network clustering and source code compatibility layers for some other Unixes.

- Compute Node Linux (CNL) is a runtime environment based on the Linux kernel for the Cray XT3, Cray XT4 and Cray XT5 supercomputer systems based on SUSE Linux Enterprise Server. CNL forms part of the UNICOS/lc operating system from release 2.0 onwards (UNICOS/lc was later renamed Cray Linux Environment). As of November 2008 systems running CNL are ranked 2nd, 7th and 8th among the fastest supercomputers.

IBM Roadrunner

- Roadrunner is a supercomputer built by IBM at the Los Alamos National Laboratory in New Mexico, USA. Currently the world's second fastest computer, the US$133-million Roadrunner is designed for a peak performance of 1.7 petaflops, achieving 1.026 on May 25, 2008 and to be the world's first TOP500 Linpack sustained 1.0 petaflops system. It is a one-of-a-kind supercomputer, built from off the shelf parts, with many novel design features.

- In November 2008, it reached a top performance of 1.456 petaflops, retaining its top spot in the TOP500 list. It is also the fourth-most energy-efficient supercomputer in the world on the Supermicro Green500 list, with an operational rate of 444.94 megaflops per watt of power used

IBM Roadrunner

IBM BladeCenter

Magerit Supercomputer housed by Supercomputing and Visualization Center of Madrid (CeSViMa), Technical University of Madrid (UPM)

Distributed System at BladeRunner

- The Roadrunner uses Red Hat Enterprise Linux along with Fedora as its operating systems and is managed with xCAT distributed computing software.
- It also uses the Open MPI Message Passing Interface implementation.

xCAT Extreme Cloud Administration Toolkit

- xCAT (Extreme Cloud Administration Toolkit) is open-source distributed computing management software. It achieved recognition in June 2008 for having been used with the IBM Roadrunner, which set a computing speed record at that time.
- xCAT offers complete and ideal management for HPC clusters, RenderFarms, Grids, WebFarms, Online Gaming Infrastructure, Clouds, Datacenters, and whatever tomorrow's buzzwords may be.
- It is agile, extendable, and based on years of system administration best practices and experience.
- http://xcat.sourceforge.net

Grid Computing Systems

- A characteristic feature of cluster computing is its homogeneity.

- In most cases, the computers in a cluster are largely the same, they all have the same operating system, and are all connected through the same network.

- In contrast, grid computing systems have a high degree of heterogeneity: no assumptions are made concerning hardware, operating systems, networks, administrative domains, security policies, etc.

Grid computing

- A key issue in a grid computing system is that resources from different organizations are brought together to allow the collaboration of a group of people or institutions.
- Such a collaboration is realized in the form of a virtual organization.
- The people belonging to the same virtual organization have access rights to the resources that are provided to that organization.
- Typically, resources consist of compute servers (including supercomputers, possibly implemented as cluster computers), storage facilities, and databases.
- In addition, special networked devices such as telescopes, sensors, etc., can be provided as well.

Grid Architecture

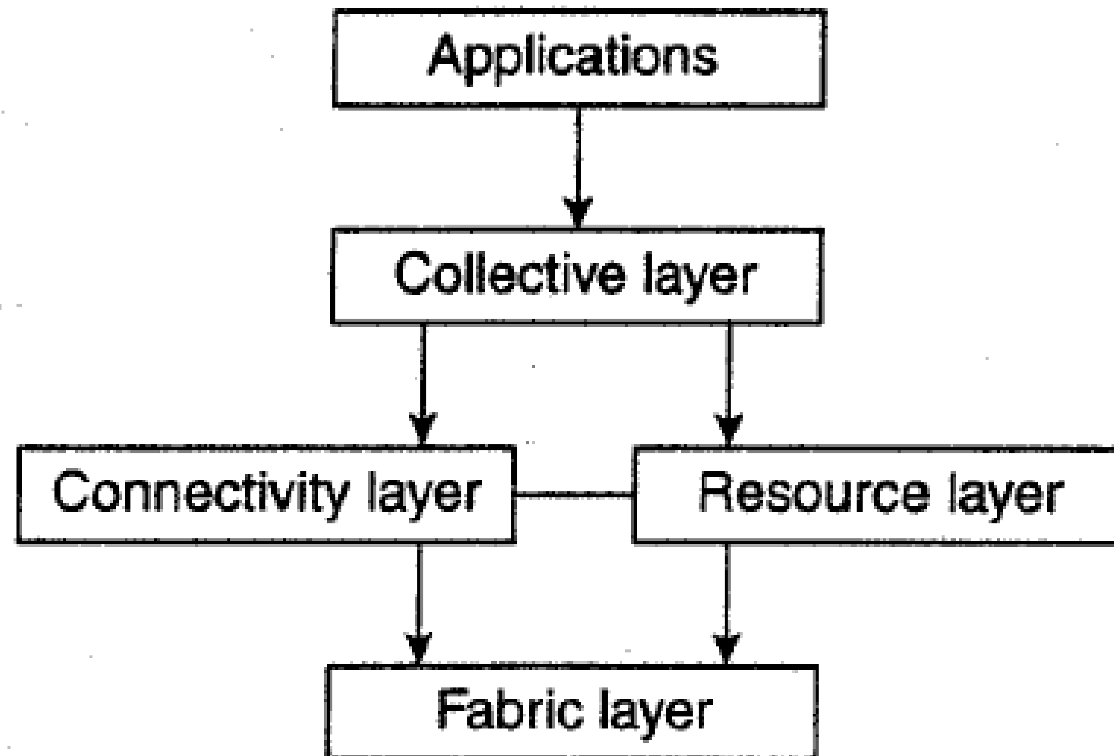
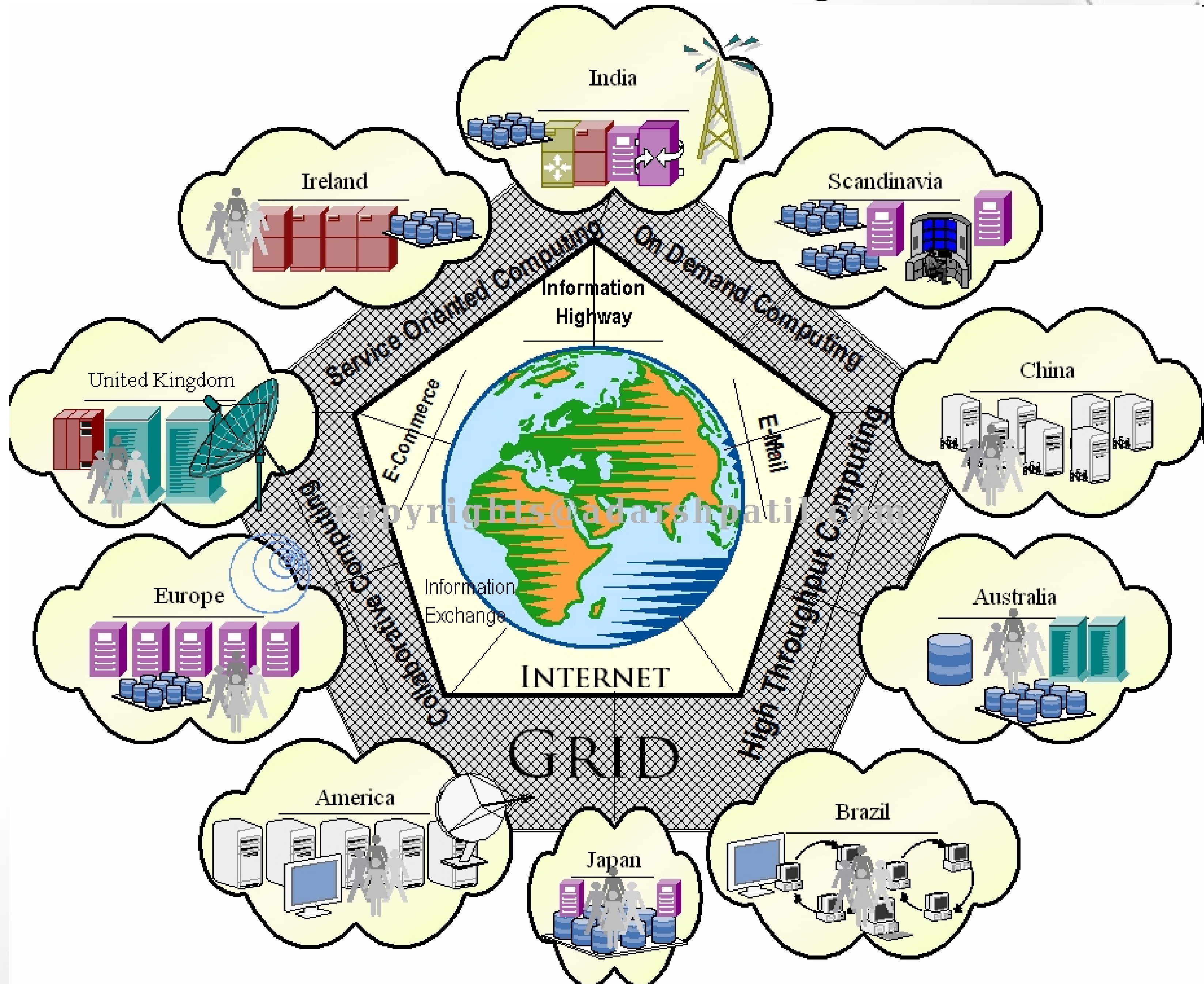


Figure 1-7. A layered architecture for grid computing systems.

Grid Computing*

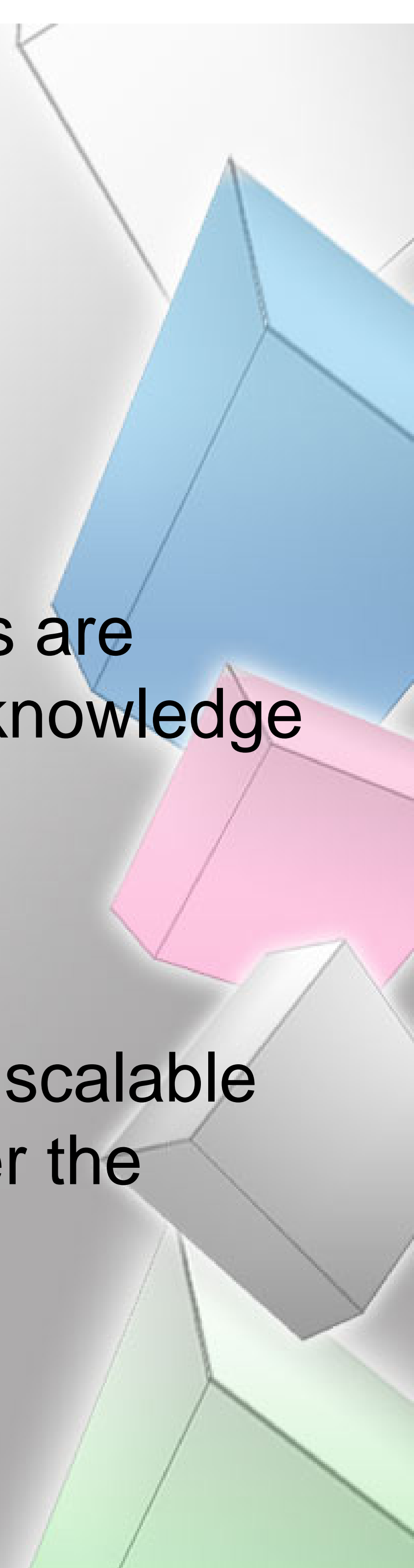


*From Adarsh Patil

Middleware for grid computing

- At the core of Grid computing is the middleware: it consists of a series of software components that realize the interface between the distributed resources on one side, and the applications on the other side. These components include resource discovery, job scheduling, authentication and authorization, data logging, data transfer and replication etc.
- Some middlewares are:
  - Globus: developed by the Globus alliance as a basic and fundamental technology behind the grid.
  - LCG: developed by the CERN grid group (LCG).
  - gLite: a light-weighted middleware developed by the EGEE collaboration (see below).
- More at:
  - http://www.gridcomputing.com/

Cloud Computing

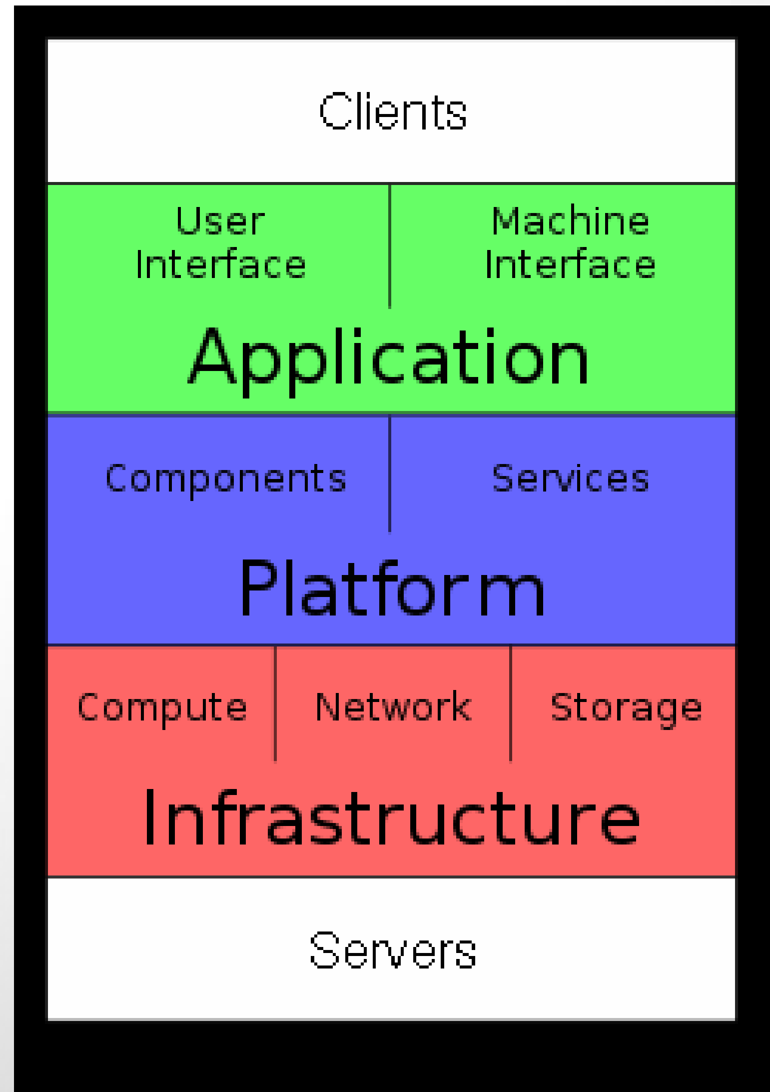


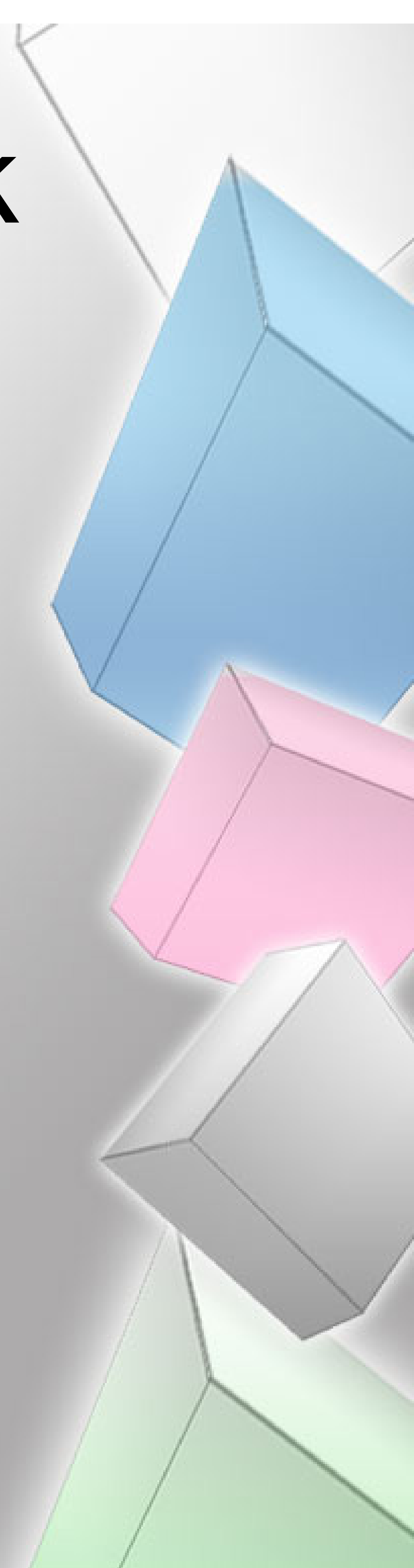
- Cloud computing is Internet- ("cloud-") based development and use of computer technology ("computing").

- In concept, it is a paradigm shift whereby details are abstracted from the users who no longer need knowledge of, expertise in, or control over the technology infrastructure "in the cloud" that supports them.

- It typically involves the provision of dynamically scalable and often virtualized resources as a service over the Internet.

Cloud computing stack





Cloud computing: clients

- A cloud client consists of computer hardware and/or computer software that relies on cloud computing for application delivery, or that is specifically designed for delivery of cloud services and that, in either case, is essentially useless without it.

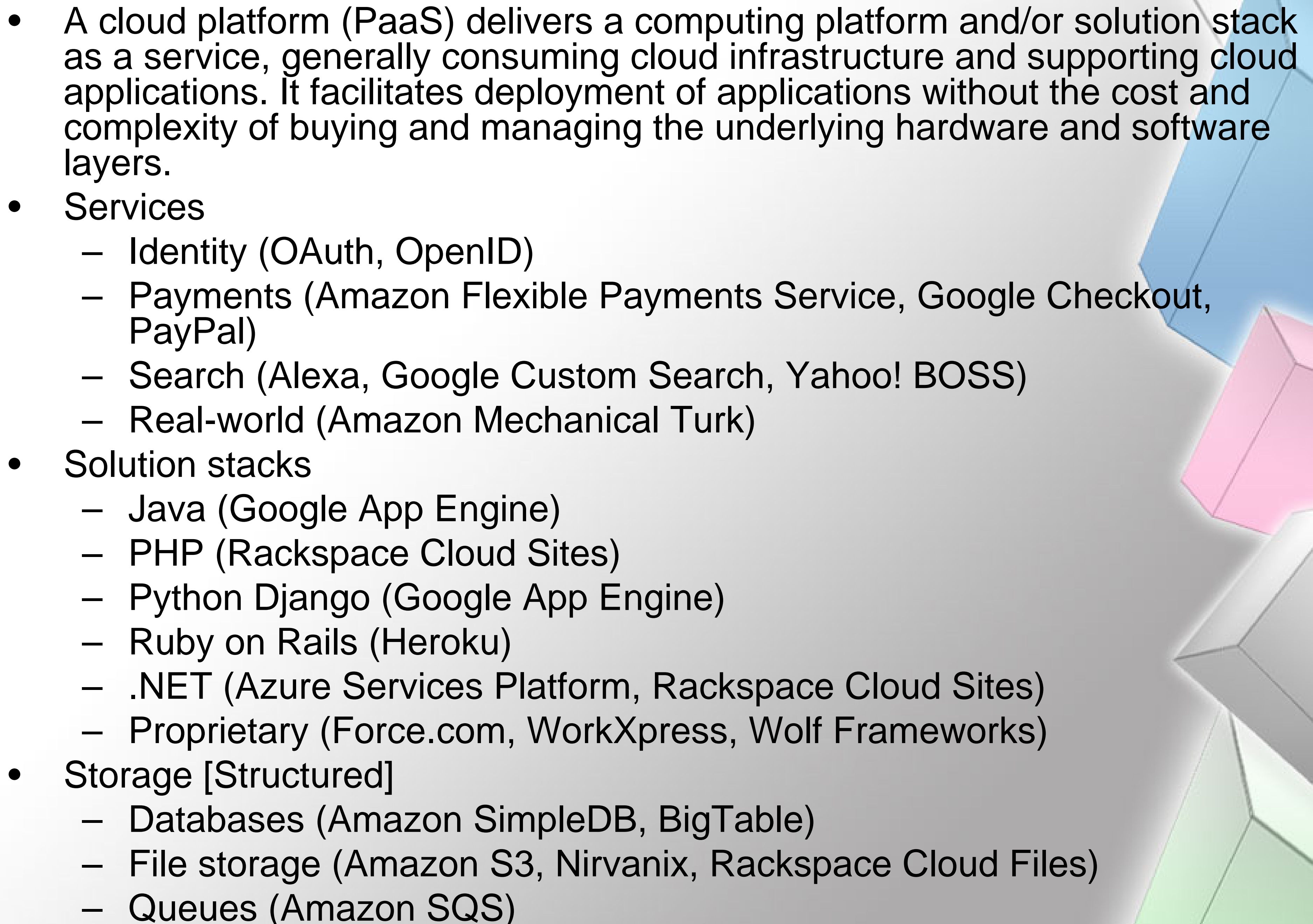
- For example:
- * Mobile (Linux based - Palm Pre-WebOS Linux Kernel, Android-Linux Kernel, iPhone-Darwin Linux Kernel, Microsoft based - Windows Mobile)[46][47][48]
- Thin client (CherryPal, Wyse, Zonbu, gOS-based systems)
- Thick client / Web browser (Internet Explorer, Mozilla Firefox, Google Chrome, WebKit)

Cloud computing: applications

- A cloud application leverages cloud computing in software architecture, often eliminating the need to install and run the application on the customer's own computer, thus alleviating the burden of software maintenance, ongoing operation, and support.
- For example:

- Peer-to-peer / volunteer computing (BOINC, Skype)
- Web applications (Webmail, Facebook, Twitter, YouTube)
- Security as a service (MessageLabs, Purewire, ScanSafe, Zscaler)
- Software as a service (A2Zapps.com, Google Apps, Salesforce,Learn.com, Zoho, BigGyan.com)
- Software plus services (Microsoft Online Services)
- Storage [Distributed]
- Content distribution (BitTorrent, Amazon CloudFront)
- Synchronisation (Dropbox, Live Mesh, SpiderOak, ZumoDrive)

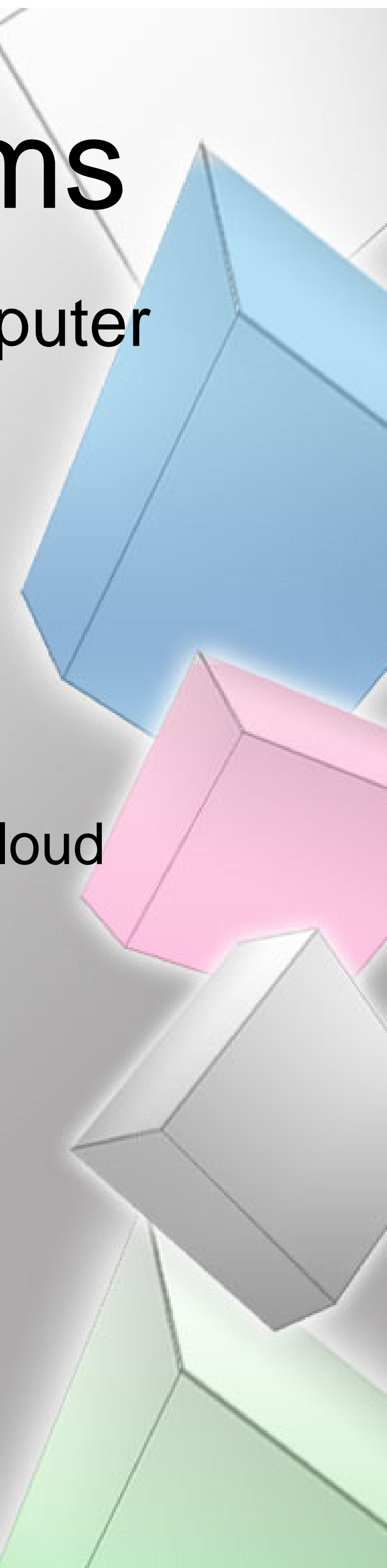
Cloud computing: Platforms



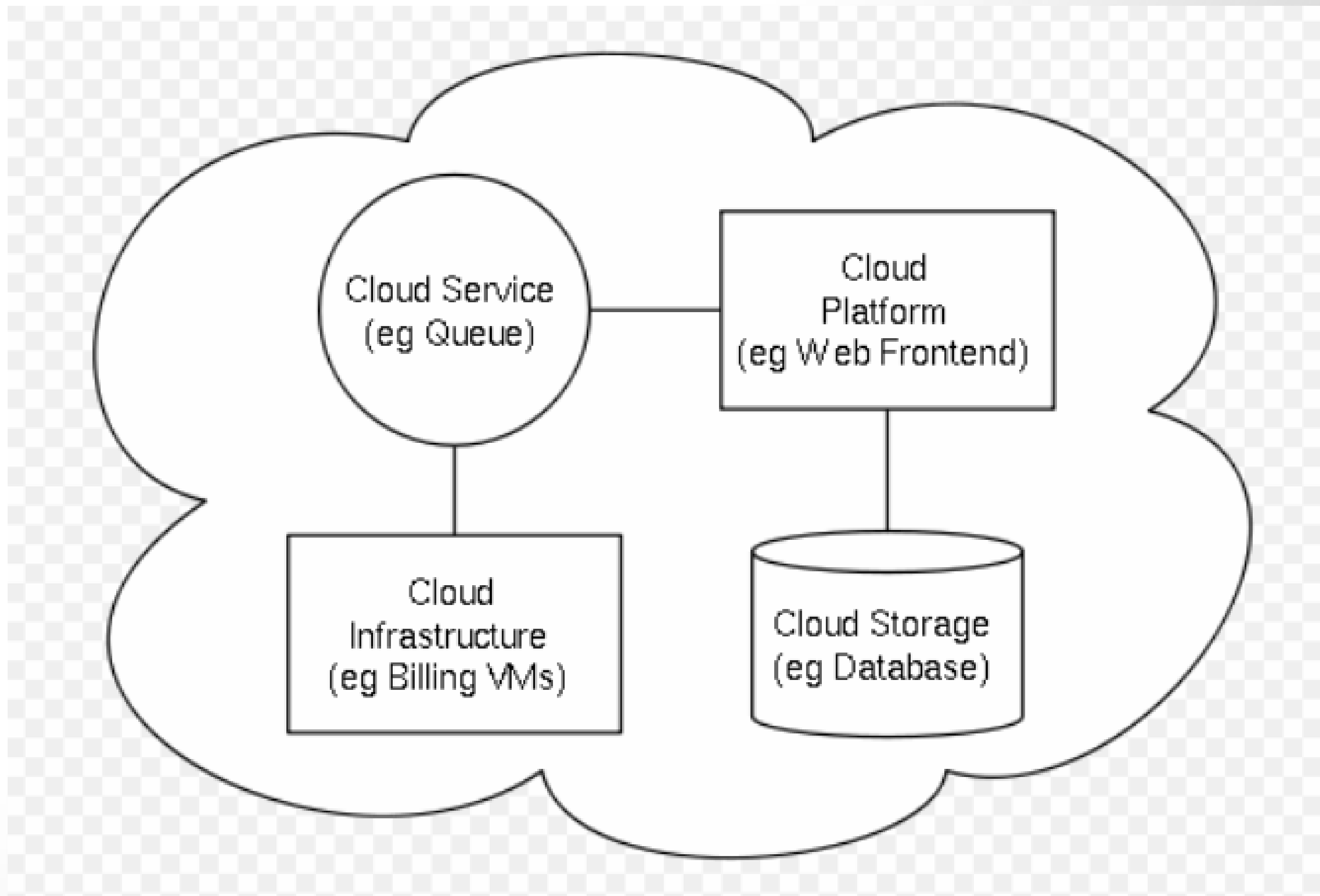
- A cloud platform (PaaS) delivers a computing platform and/or solution stack as a service, generally consuming cloud infrastructure and supporting cloud applications. It facilitates deployment of applications without the cost and complexity of buying and managing the underlying hardware and software layers.
- Services
  - Identity (OAuth, OpenID)
  - Payments (Amazon Flexible Payments Service, Google Checkout, PayPal)
  - Search (Alexa, Google Custom Search, Yahoo! BOSS)
  - Real-world (Amazon Mechanical Turk)
- Solution stacks
  - Java (Google App Engine)
  - PHP (Rackspace Cloud Sites)
  - Python Django (Google App Engine)
  - Ruby on Rails (Heroku)
  - .NET (Azure Services Platform, Rackspace Cloud Sites)
  - Proprietary (Force.com, WorkXpress, Wolf Frameworks)
- Storage [Structured]
  - Databases (Amazon SimpleDB, BigTable)
  - File storage (Amazon S3, Nirvanix, Rackspace Cloud Files)
  - Queues (Amazon SQS)

Cloud computing: Platforms

- Cloud infrastructure (IaaS) is the delivery of computer infrastructure, typically a platform virtualization environment, as a service.
- For example:
  - Compute (Amazon CloudWatch, RightScale)
  - Physical machines)
  - Virtual machines (Amazon EC2, GoGrid, Rackspace Cloud Servers)
  - OS-level virtualisation
  - Network (Amazon VPC)
  - Storage [Raw] (Amazon EBS)



Clouding



Cloud Computing

- Typical cloud computing providers deliver common business applications online which are accessed from a web browser, while the software and data are stored on the servers.

- These applications are broadly divided into the following categories: Software as a Service (SaaS), Utility Computing, Web Services, Platform as a Service (PaaS), Managed Service Providers (MSP), Service Commerce, and Internet Integration.

- The name cloud computing was inspired by the cloud symbol that is often used to represent the Internet in flow charts and diagrams.

Cloud Computing

- In general, cloud computing customers do not own the physical infrastructure, instead avoiding capital expenditure by renting usage from a third-party provider.
- They consume resources as a service and pay only for resources that they use. Many cloud-computing offerings employ the utility computing model, which is analogous to how traditional utility services (such as electricity) are consumed, whereas others bill on a subscription basis.
- Sharing "perishable and intangible" computing power among multiple tenants can improve utilization rates, as servers are not unnecessarily left idle (which can reduce costs significantly while increasing the speed of application development). A side-effect of this approach is that overall computer usage rises dramatically, as customers do not have to engineer for peak load limits.
- In addition, "increased high-speed bandwidth" makes it possible to receive the same response times from centralized infrastructure at other sites

Distributed Information Systems

- Another important class of distributed systems is found in organizations that were confronted with a wealth of networked applications, but for which interoperability turned out to be a painful experience.

- 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.

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).
- The vast amount of distributed systems in use today are forms of traditional information systems, that now integrate legacy systems.

Transaction Processing Systems

- A transaction is a collection of operations on the state of an object (database, object composition, etc.) that satisfies the following properties (ACID):
- Atomicity: All operations either succeed, or all of them fail. When the transaction fails, the state of the object will remain unaffected by the transaction.
- Consistency: A transaction establishes a valid state transition. This does not exclude the possibility of invalid, intermediate states during the transaction's execution.
- Isolation: Concurrent transactions do not interfere with each other. It appears to each transaction T that other transactions occur either before T, or after T, but never both.
- Durability: After the execution of a transaction, its effects are made permanent: changes to the state survive failures.

Nested Transaction

- Nested transactions are important in distributed systems, for they provide a natural way of distributing a transaction across multiple machines.

- They follow a *logical* division of the work of the original transaction. For example, a transaction for planning a trip by which three different flights need to be reserved can be logically split up into three subtransactions.

- Each of these subtransactions can be managed separately and independent of the other two.

Nested Transaction

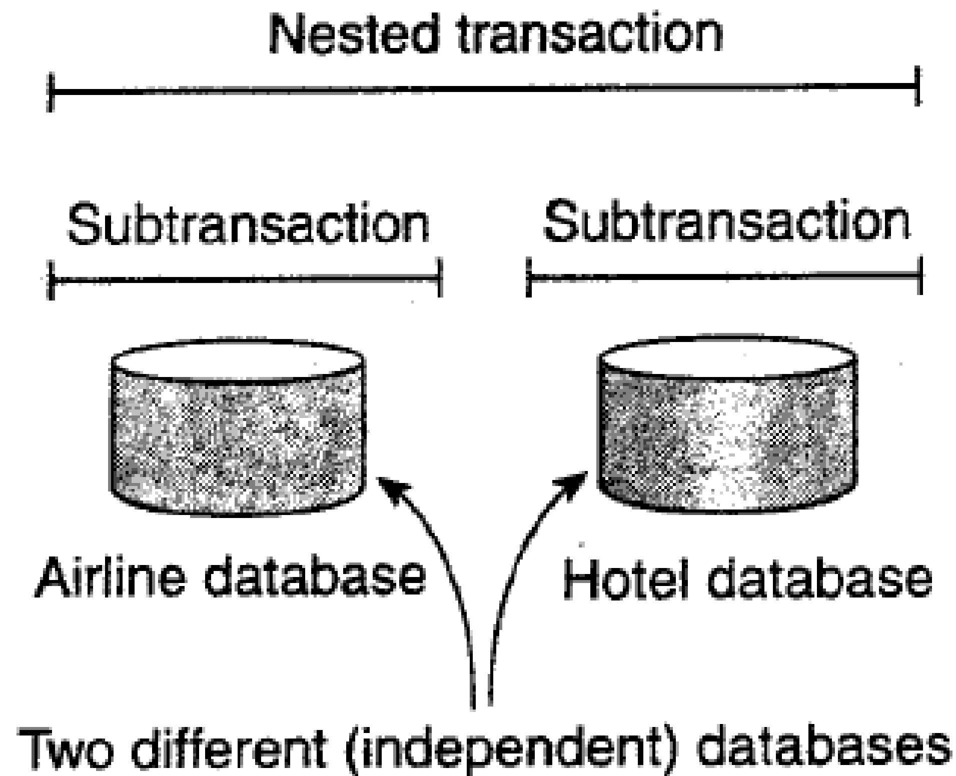


Figure 1-9. A nested transaction.

Transaction Processing Monitor

- In the early days of enterprise middleware systems, the component that handled distributed (or nested) transactions formed the core for integrating applications at the server or database level.

- This component was called a transaction processing monitor or TP monitor for short. Its main task was to allow an application to access multiple server/databases by offering it a transactional programming model

Transaction Processing Monitor

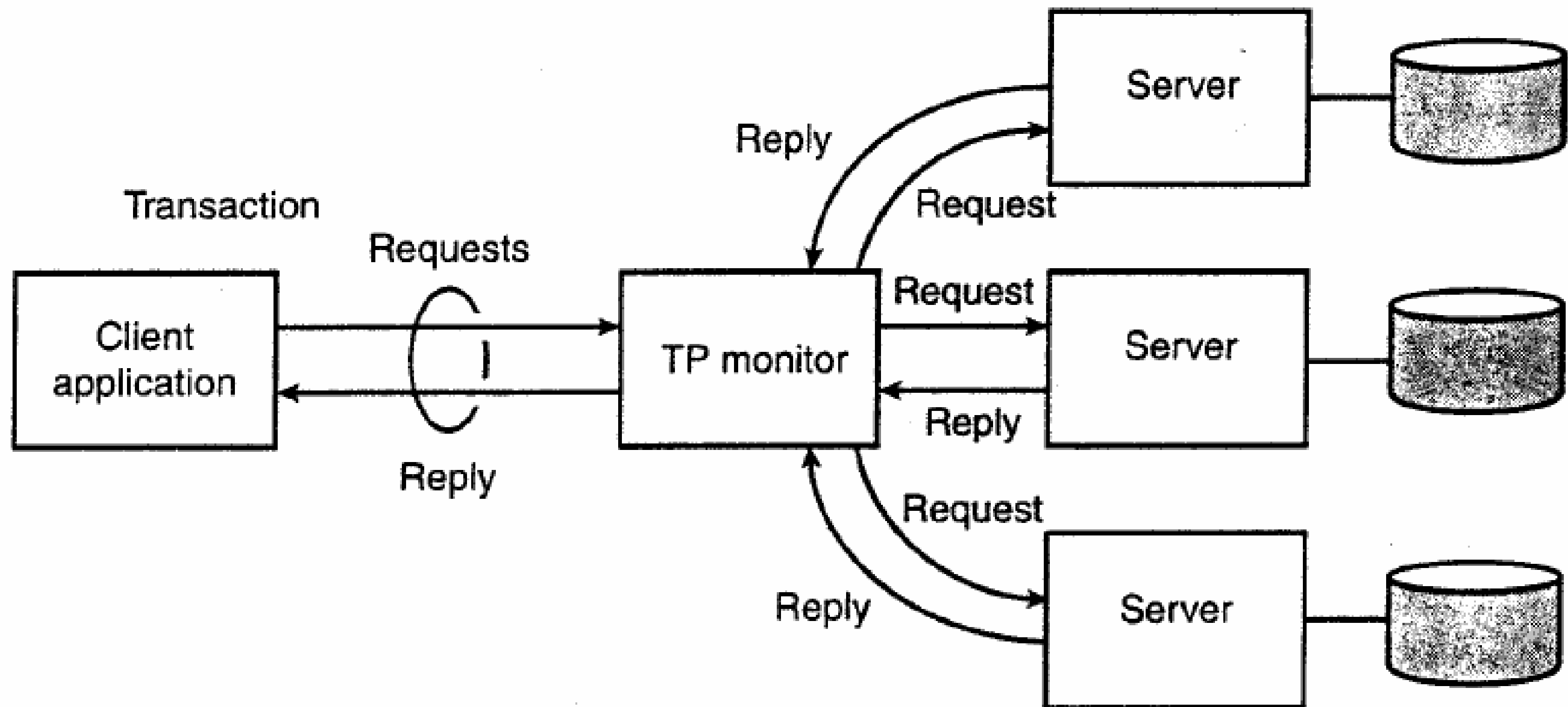


Figure 1-10. The role of a TP monitor in distributed systems.

Enterprise Application Integration

- The more applications became decoupled from the databases they were built upon, the more evident it became that facilities were needed to **integrate applications independent from their databases**.
- In particular, 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.
- This need for interapplication communication led to many different **communication models**, which we will discuss in detail in this course (and for which reason we shall keep it brief for now).
- **The main idea was that existing applications could directly exchange information.**

Enterprise Application Integration

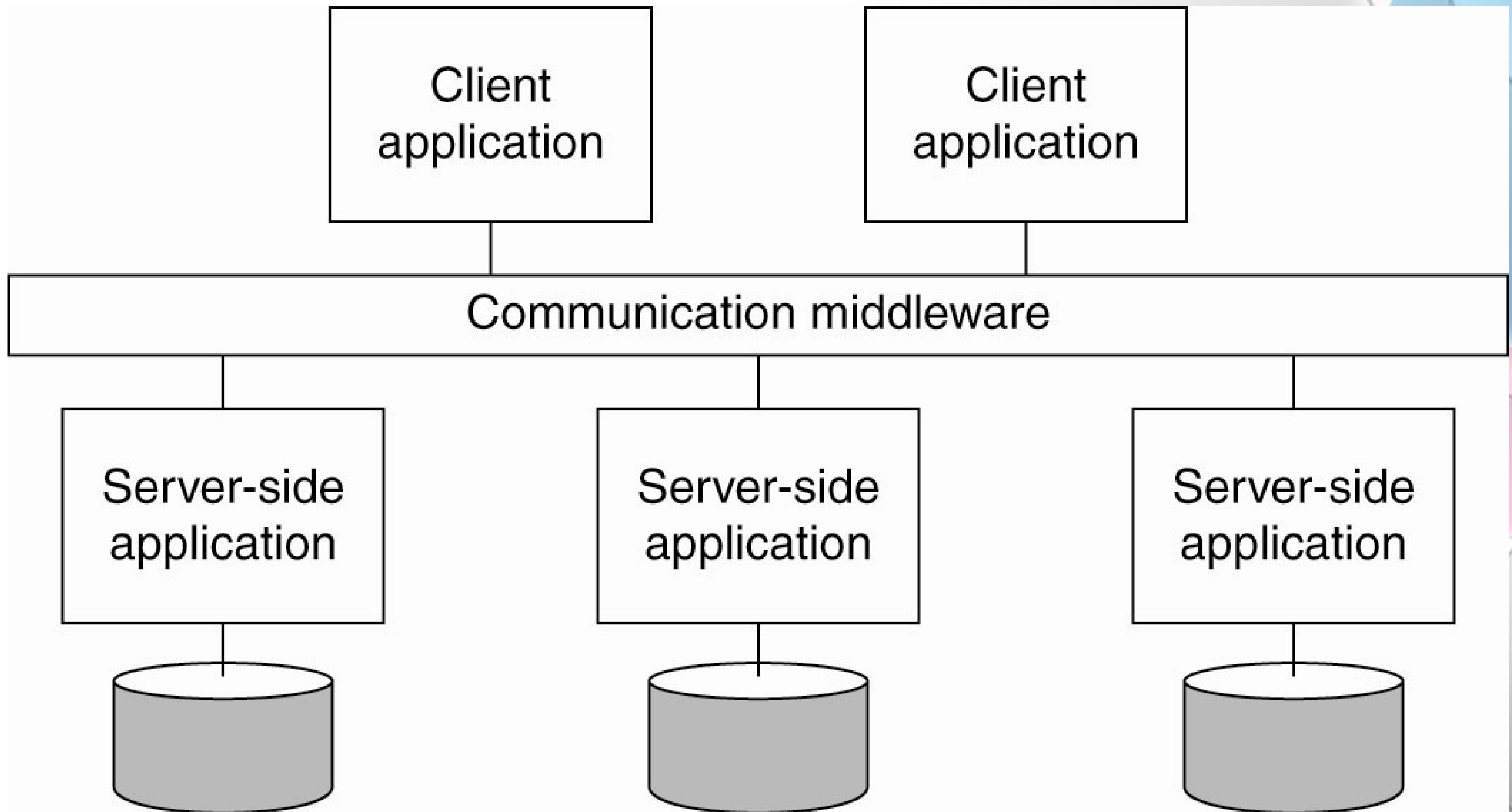


Figure 1-11. Middleware as a communication facilitator in enterprise application integration.

Communication Middleware

- Several types of communication middleware exist. 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.
- 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.

Disadvantages of RPCs and RMIs

- 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 points, often described by means of a subject.
  - 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.
  - These so-called publish/subscribe systems form an important and expanding class of distributed systems.

Distributed Pervasive Systems

- The distributed systems we have been discussing so far are largely characterized by their stability: nodes are fixed and have a more or less permanent and high-quality connection to a network.
- To a certain extent, this stability has been realized through the various techniques that are discussed in this course and which aim at achieving distribution transparency.
- However, 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.

Distributed Pervasive Systems

Some requirements

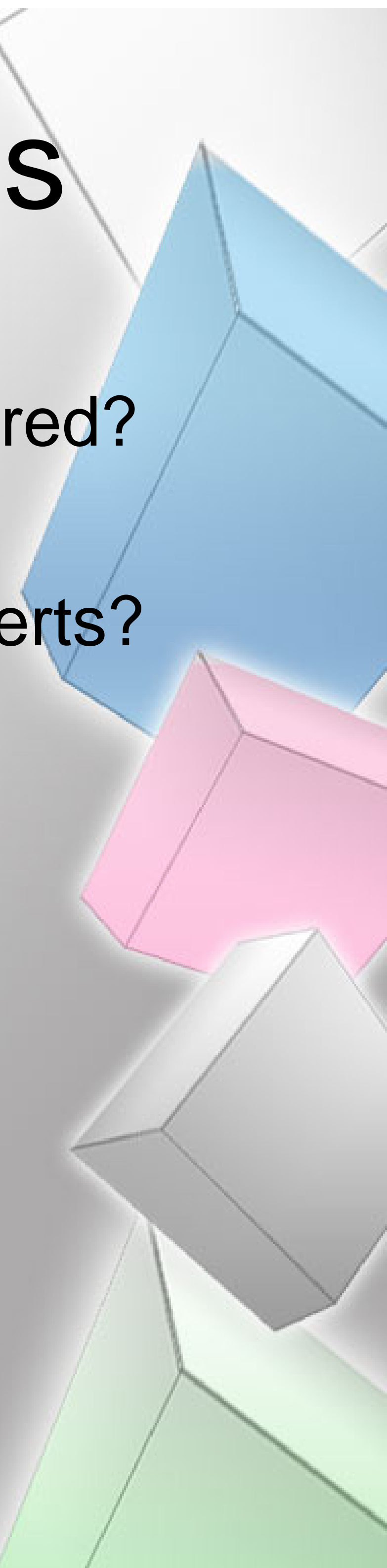
- Contextual change: The system is part of an environment in which changes should be immediately accounted for.
- Ad hoc composition: Each node may be used in a very different ways by different users. Requires ease-of-configuration.
- Sharing is the default: Nodes come and go, providing sharable services and information. Calls again for simplicity.

Home Systems

- An increasingly popular type of pervasive system, but which may perhaps be the least constrained, are systems built around home networks.
- These systems generally consist of one or more personal computers, but more importantly integrate typical consumer electronics such as TVs, audio and video equipment. Gaming devices, (smart) phones, PDAs, and other personal wearables into a single system.
- **In addition, we can expect that all kinds of devices such as kitchen appliances, surveillance cameras, clocks, controllers for lighting, and so on, will all be hooked up into a single distributed system. Welcome to the future ☺**

Electronic health systems

- Devices are physically close to a person:
  - Where and how should monitored data be stored?
  - How can we prevent loss of crucial data?
  - What is needed to generate and propagate alerts?
  - How can security be enforced?
  - How can physicians provide online feedback?



Electronic health systems

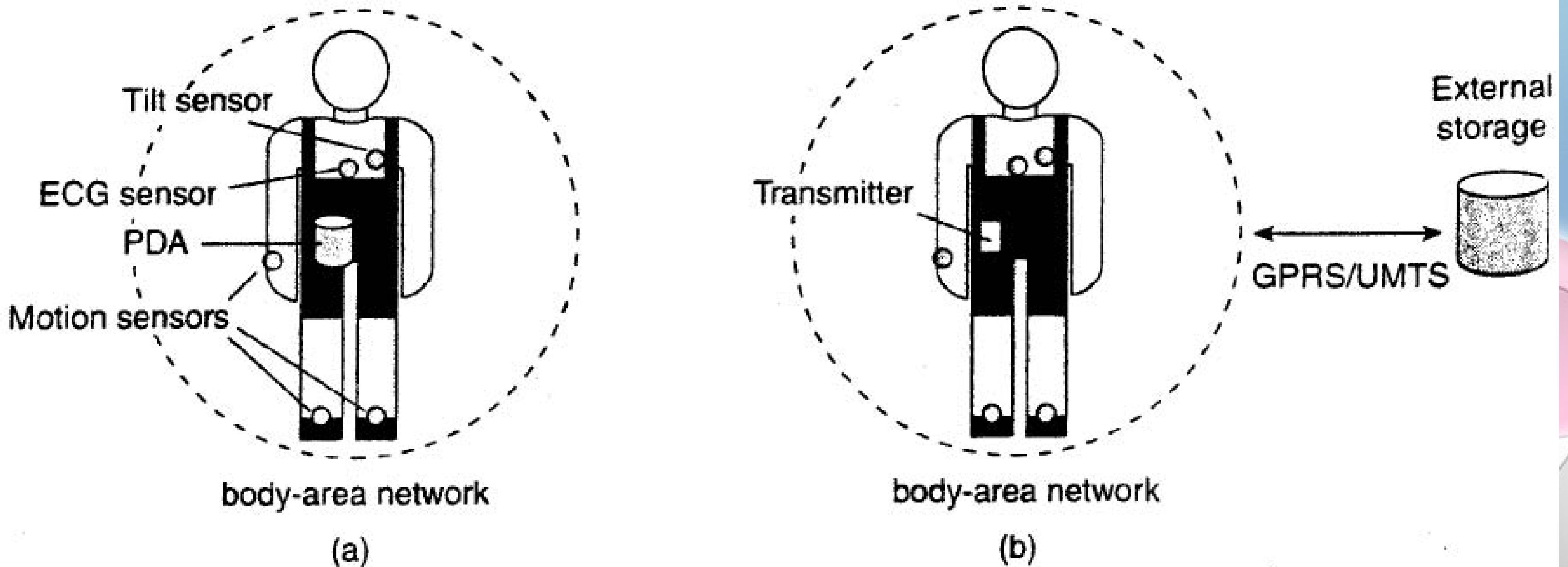


Figure 1-12. Monitoring a person in a pervasive electronic health care system, using (a) a local hub or (b) a continuous wireless connection.

Sensor Networks

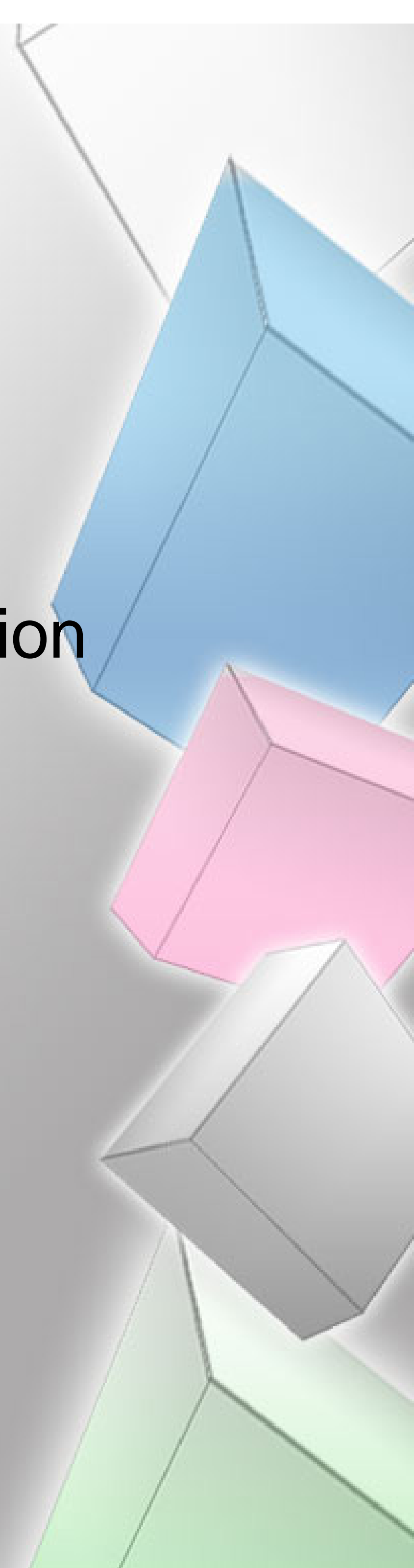
- Another example of pervasive systems is sensor networks.
- These networks in many cases form part of the enabling technology for pervasiveness and we see that many solutions for sensor networks return in pervasive applications.
- What makes sensor networks interesting from a distributed system's perspective is that in virtually all cases they are used for processing information.
  - In this sense, they do more than just provide communication services. which is what traditional computer networks are all about.

Sensor Networks

Characteristics

The nodes to which sensors are attached are:

- Many (10s-1000s)
- Simple (small memory/compute/communication capacity)
- Often battery-powered (or even battery-less)



Sensor Networks as DSs

- The relation with distributed systems can be made clear by considering sensor networks as distributed databases.

- This view is quite common and easy to understand when realizing that many sensor networks are deployed for measurement and surveillance applications.

- In these cases, an operator would like to extract information from (a part of) the network by simply issuing queries such as "What is the northbound traffic load on Autostrada Tirane-Durres?"

Sensor Networks (2)

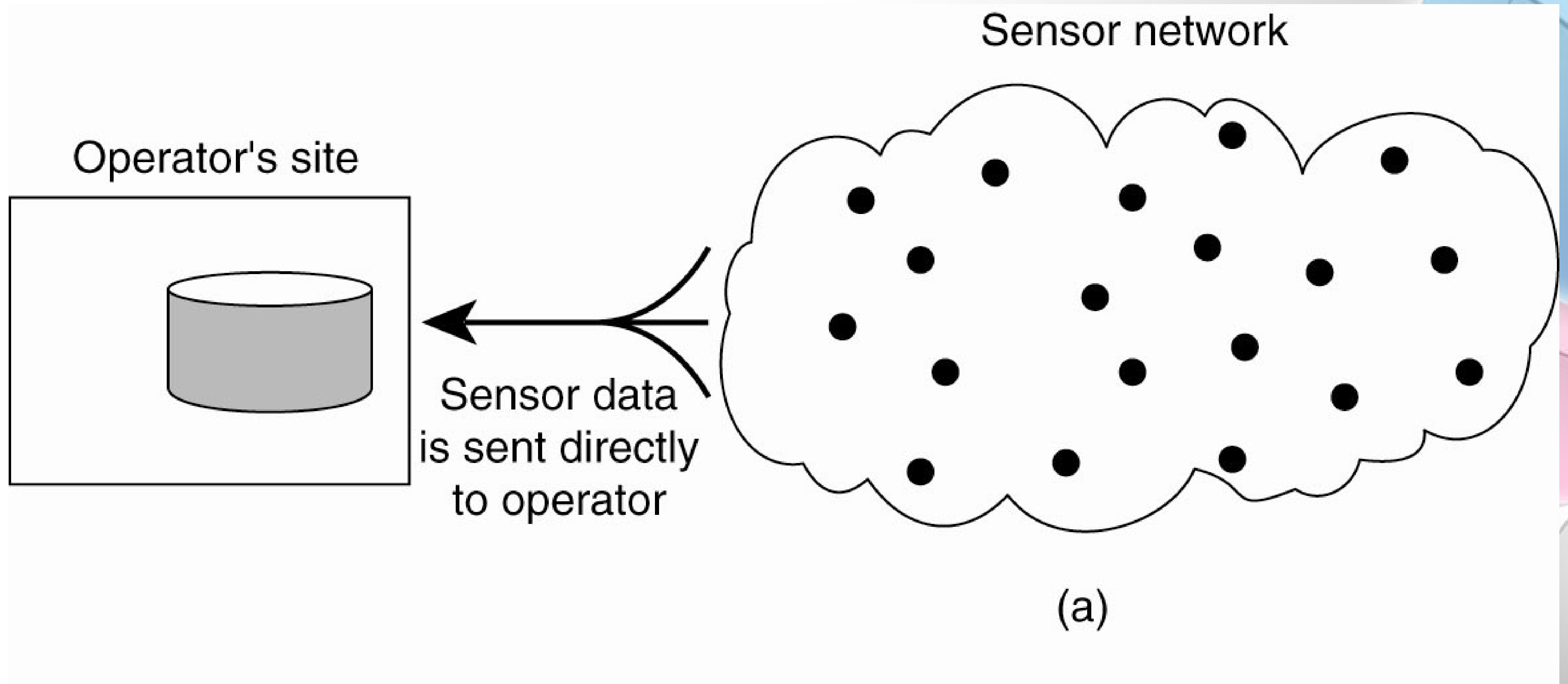


Figure 1-13. Organizing a sensor network database, while storing and processing data (a) only at the operator's site or …

Sensor Networks (3)

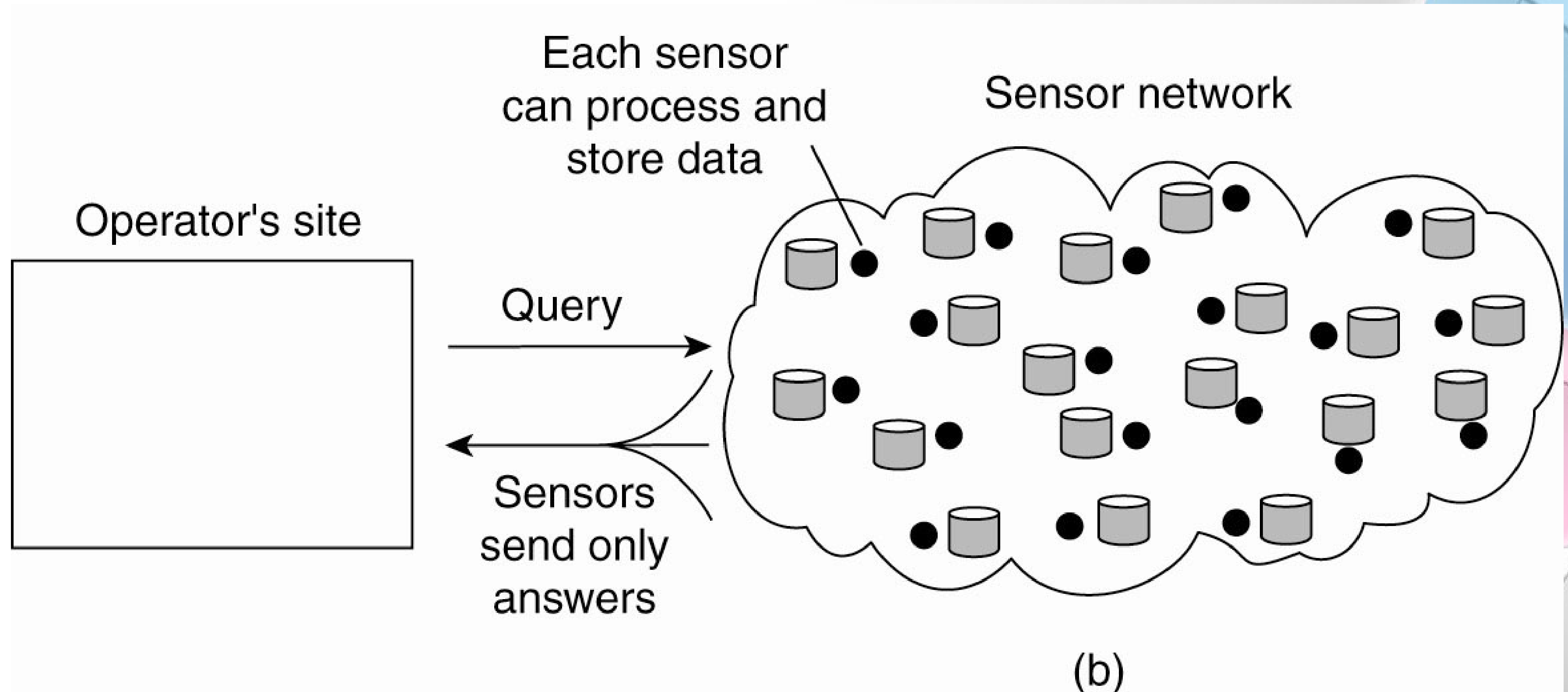


Figure 1-13. Organizing a sensor network database, while storing and processing data (b) only at the sensors.

End of Lesson 1

- Readings
  - *Distributed Systems, Principles and Paradigms,* Chapter I.