CHARACTERIZATION OF DISTRIBUTED SYSTEMS

A distributed system is one in which components located at networked computers communicate and coordinate their actions only by-passing messages.

This definition leads to the following especially significant characteristics of distributed systems:

- concurrency of components : *Concurrency*: In a network of computers, concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary.
- lack of a global clock and : *No global clock*: When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks – there is no single global notion of the correct time. This is a direct consequence of the fact that the *only* communication is by sending messages through a network

independent failures of components : All computer systems can fail, and it is the responsibility of system designers to plan for the consequences of possible failures.

Examples of distributed systems

Finance and commerce *:* The growth of eCommerce as exemplified by companies such as Amazon and eBay, and underlying payments technologies such as PayPal;

The information society*:* The growth of the World Wide Web as a repository of information and knowledge; the development of web search engines such as Google and Yahoo to search this vast repository; user-generated content through sites such as YouTube, Wikipedia and Flickr;

Creative industries and entertainment*:* The emergence of online gaming as a novel and highly interactive form of entertainment; YouTube

Healthcare: The growth of health informatics as a discipline with its emphasis on online electronic patient records and related issues of privacy; the increasing role of telemedicine in supporting remote diagnosis or more advanced services such as remote surgery

Education : The emergence of e-learning through for example web-based tools such as virtual learning environments; associated support for distance learning;

Transport and logistics: The use of location technologies such as GPS in route finding systems and more general traffic management systems; the modern car itself as an example of a complex distributed system (also applies to other forms of transport such as aircraft); the development of web-based map services such as MapQuest, Google Maps and Google Earth

Science: The emergence of the Grid as a fundamental technology for eScience, including the use of complex networks of computers to support the storage, analysis and processing of (often very large quantities of) scientific data;

Environmental management: The use of (networked) sensor technology to both monitor and manage the natural environment, for example to provide early warning of natural disasters such as earthquakes, floods or tsunamis and to coordinate emergency response;

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

Web search

Web search has emerged as a major growth industry in the last decade, with recent figures indicating that the global number of searches has risen to over 10 billion per calendar month.

The task of a web search engine is to index the entire contents of the World Wide Web, encompassing a wide range of information styles including web pages, multimedia sources and (scanned) books.

This is a very complex task, as current estimates state that the Web consists of over 63 billion pages and one trillion unique web addresses.

Given that most search engines analyze the entire web content and then carry out sophisticated processing on this enormous database, this task itself represents a major challenge for distributed systems design.

Google, the market leader in web search technology, has put significant effort into the design of a sophisticated distributed system infrastructure to support search .

• an underlying physical infrastructure consisting of very large numbers of networked computers located at data centres all around the world;

• a distributed file system designed to support very large files and heavily optimized for the style of usage required by search and other Google applications (especially reading from files at high and sustained rates);

- an associated structured distributed storage system that offers fast access to very large datasets;
- a lock service that offers distributed system functions such as distributed locking and agreement;
- a programming model that supports the management of very large parallel and distributed computations across the underlying physical infrastructure.

Massively multiplayer online games (MMOGs)

Massively multiplayer online games offer an immersive experience whereby very large

numbers of users interact through the Internet with a persistent virtual world.

Example EVE, Online consists of a universe with over 5,000 star systems) and multifarious social and economic systems.

The number of players is also rising, with systems able to support over 50,000 simultaneous online players (and the total number of players perhaps ten times this figure).

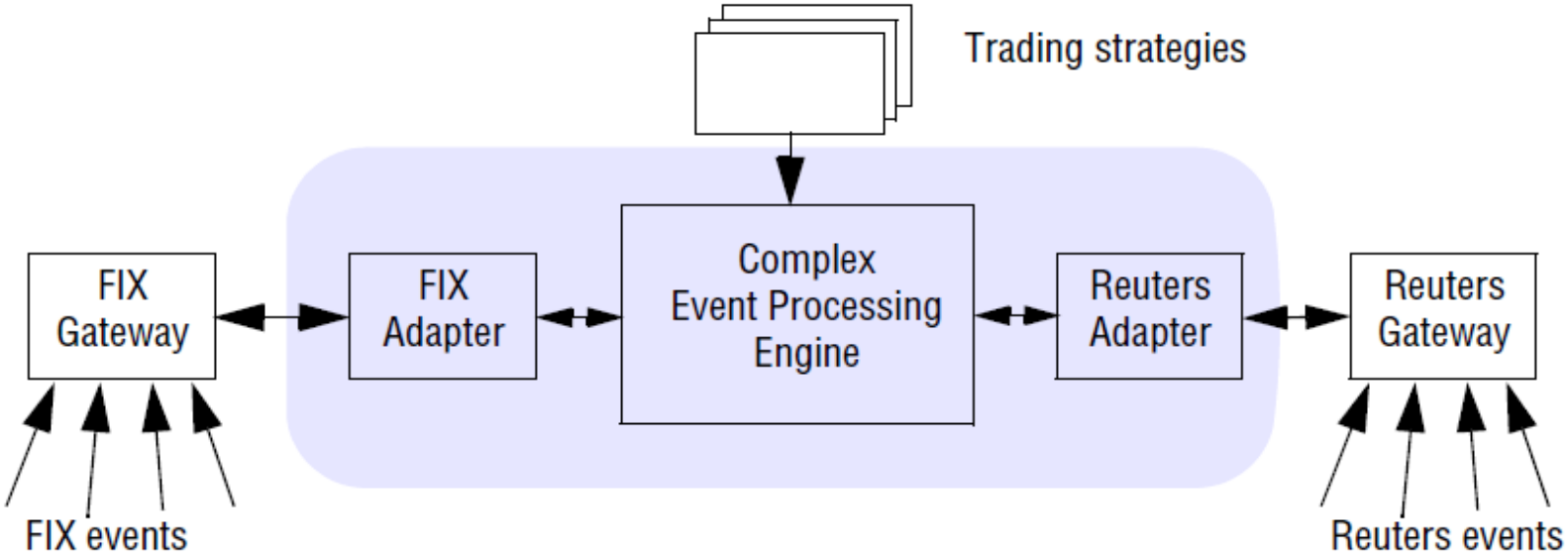
The engineering of MMOGs represents a major challenge for distributed systems technologies, particularly because of the need for fast response times to preserve the user experience of the game.

Financial trading

As a final example, we look at distributed systems support for financial trading markets.

The financial industry has long been at the cutting edge of distributed systems technology with its need, in particular, for real-time access to a wide range of information sources (for example, current share prices and trends, economic and political developments). The industry employs automated monitoring and trading applications

Figure 1.2 An example financial trading system



Firstly, the sources are typically in a variety of formats, such as Reuters market data events and FIX events (events following the specific format of the **Financial Information eXchange protocol**), and indeed from different event technologies, thus illustrating the problem of heterogeneity as encountered in most distributed systems.

Secondly, the trading system must deal with a variety of event streams, all arriving at rapid rates, and often requiring real-time processing to detect patterns that indicate trading opportunities.

This used to be a manual process but competitive pressures have led to increasing automation in terms of what is known as **Complex Event Processing** (CEP), which offers a way of composing event occurrences together into logical, temporal or spatial patterns.

This approach is primarily used to develop customized **algorithmic trading strategies** covering both buying and selling of stocks and shares, in particular looking for patterns that indicate a trading opportunity and then automatically responding by placing and managing orders.

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

Focus on resource sharing

In practice, patterns of resource sharing vary widely in their scope and in how closely users work together.

At one extreme, a search engine on the Web provides a facility to users throughout the world, users who need never come into contact with one another directly.

At the other extreme, in ***computer-supported cooperative working*** (CSCW), a group of users who cooperate directly share resources such as documents in a small, closed group.

The pattern of sharing and the geographic distribution of particular users determines what mechanisms the system must supply to coordinate users' actions.

We use the term *service* for a distinct part of a computer system that manages a collection of related resources and presents their functionality to users and applications.

For example, we access shared files through a file service; we send documents to printers through a printing service; we buy goods through an electronic payment service.

The only access we have to the service is via the set of operations that it exports.

For example, a **file service provides *read*, *write* and *delete* operations on files**.

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

The term *server* is probably familiar to most readers. It refers to a running program (a *process*) on a networked computer that accepts requests from programs running on other computers to perform a service and responds appropriately.

The requesting processes are referred to as *clients,* and the overall approach is known as *client-server computing*.

Challenges

Heterogeneity

The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity (that is, variety and difference) applies to all of the following:

- networks;
- computer hardware;
- operating systems;
- programming languages;
- implementations by different developers.

Although the Internet consists of many different sorts of network, their differences are masked by the fact that all of the computers attached to them use the Internet protocols to communicate with one another.

For example, a computer attached to an Ethernet has an implementation of the Internet protocols over the Ethernet, whereas a computer on a different sort of network will need an implementation of the Internet protocols for that network.

Middleware • The term *middleware* applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages.

The Common Object Request Broker (CORBA), is an example.

Some middleware, such as Java Remote Method Invocation (RMI), supports only a single programming language.

Most middleware is implemented over the Internet protocols, which themselves mask the differences of the underlying networks,

Heterogeneity and mobile code • The term *mobile code* is used to refer to program code that can be transferred from one computer to another and run at the destination – **Java applets are an example**.

Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system.

The ***virtual machine*** approach provides a way of making code executable on a variety of host computers: the compiler for a particular language generates code for a virtual machine instead of a particular hardware order code. For example, the Java compiler produces code for a Java virtual machine, which executes it by interpretation.

The Java virtual machine needs to be implemented once for each type of computer to enable Java programs to run.

Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and reimplemented in various ways.

The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.

They may be extended at the hardware level by the addition of computers to the network and at the software level by the introduction of new services and the reimplementation of old ones, enabling application programs to share resources.

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

In a distributed system, clients send requests to access data managed by servers, which involves sending information in messages over a network.

For example:

1. A doctor might request access to hospital patient data or send additions to that data.

2. In electronic commerce and banking, users send their credit card numbers across the Internet.

In both examples, the challenge is to send sensitive information in a message over a network in a secure manner.

But security is not just a matter of concealing the contents of messages – it also involves knowing for sure the identity of the user or other agent on whose behalf a message was sent.

In the first example, the server needs to know that the user is really a doctor, and in the second example, the user needs to be sure of the identity of the shop or bank with which they are dealing.

The second challenge here is to identify a remote user or other agent correctly

Following two security challenges have not yet been fully met:

Denial of service attacks: Another security problem is that a user may wish to disrupt a service for some reason.

This can be achieved by bombarding the service with such a large number of pointless requests that the serious users are unable to use it.

This is called a *denial of service* attack.

There have been several denial of service attacks on well-known web services.

Security of mobile code: Mobile code needs to be handled with care.

Consider someone who receives an executable program as an electronic mail attachment: the possible effects of running the program are unpredictable;

for example, it may seem to display an interesting picture but in reality it may access local resources

Scalability

Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as *scalable* if it will remain effective when there is a significant increase in the number of resources and the number of users.

Growth of the Internet (computers and web servers)

Date	*Computers*	*Web servers*	*Percentage*
1993, July	1,776,000	130	0.008
1995, July	6,642,000	23,500	0.4
1997, July	19,540,000	1,203,096	6
1999, July	56,218,000	6,598,697	12
2001, July	125,888,197	31,299,592	25
2003, July	~200,000,000	42,298,371	21
2005, July	353,284,187	67,571,581	19

Controlling the cost of physical resources: As the demand for a resource grows, it should be possible to extend the system, at reasonable cost, to meet it.

Controlling the performance loss: Consider the management of a set of data whose size is proportional to the number of users or resources in the system – for example, the table with the correspondence between the domain names of computers and their Internet addresses held by the Domain Name System, which is used mainly to look up DNS names such as www.amazon.com.

Algorithms that use hierarchic structures scale better than those that use linear structures. But even with hierarchic structures an increase in size will result in some loss in performance:

***Preventing software resources running out*:** An example of lack of scalability is shown by the numbers used as Internet (IP) addresses (computer addresses in the Internet).

In the late 1970s, it was decided to use 32 bits for this purpose, the supply of available Internet addresses is running out.

For this reason, a new version of the protocol with 128-bit Internet addresses is being adopted, and this will require modifications to many software components.

Avoiding performance bottlenecks: In general, algorithms should be decentralized to avoid having performance bottlenecks.

We illustrate this point with reference to the predecessor of the Domain Name System, in which the name table was kept in a single master file that could be downloaded to any computers that needed it.

That was fine when there were only a few hundred computers in the Internet, but it soon became a serious performance and administrative bottleneck.

The Domain Name System removed this bottleneck by partitioning the name table between servers located throughout the Internet and administered locally.

Failure handling

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation.

Detecting failures: Some failures can be detected. For example, checksums can be used to detect corrupted data in a message or a file.

It is difficult or even impossible to detect some other failures, such as a remote crashed server in the Internet.

The challenge is to manage in the presence of failures that cannot be detected but may be suspected.

Masking failures: Some failures that have been detected can be hidden or made less

severe. Two examples of hiding failures:

1. Messages can be retransmitted when they fail to arrive.

2. File data can be written to a pair of disks so that if one is corrupted, the other may still be correct.

Tolerating failures: Most of the services in the Internet do exhibit failures – it would not be practical for them to attempt to detect and hide all of the failures that might occur in such a large network with so many components.

.

Recovery from failures: Recovery involves the design of software so that the state of permanent data can be recovered or 'rolled back' after a server has crashed

Their clients can be designed to tolerate failures, which generally involves the users tolerating them as well.

For example, when a web browser cannot contact a web server, it does not make the user wait for ever while it keeps on trying – it informs the user about the problem, leaving them free to try again later.

***Redundancy*:** Services can be made to tolerate failures by the use of redundant components. Consider the following examples:

1. There should always be at least two different routes between any two routers in the Internet.

2. In the Domain Name System, every name table is replicated in at least two different servers.

3. A database may be replicated in several servers to ensure that the data remains accessible after the failure of any single server;

Concurrency

The process that manages a shared resource could take one client request at a time.

But that approach limits throughput.

Therefore services and applications generally allow multiple client requests to be processed concurrently.

To make this more concrete, suppose that each resource is encapsulated as an object and that invocations are executed in concurrent threads.

In this case it is possible that several threads may be executing concurrently within an object, in which case their operations on the object may conflict with one another and produce inconsistent results.

For example, if two concurrent bids at an auction are ‘Smith: $122’ and ‘Jones: $111’, and the corresponding operations are interleaved without any control, then they might get stored as ‘Smith: $111’ and ‘Jones: $122’.

For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent.

This can be achieved by standard techniques such as semaphores, which are used in most operating systems

Transparency

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components.

Access transparency enables local and remote resources to be accessed using identical operations.

Location transparency enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address).

Concurrency transparency enables several processes to operate concurrently using shared resources without interference between them.

Replication transparency enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.

Failure transparency enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.

Mobility transparency allows the movement of resources and clients within a system without affecting the operation of users or programs.

Performance transparency allows the system to be reconfigured to improve performance as loads vary.

Scaling transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms.

Quality of service

Once users are provided with the functionality that they require of a service, such as the file service in a distributed system, we can go on to ask about the quality of the service provided.

The main nonfunctional properties of systems that affect the quality of the service experienced by clients and users are *reliability*, *security* and *performance.*

for example, BBC iPlayer generally performs acceptably – but when networks are heavily loaded their performance can deteriorate, and no guarantees are provided. QoS applies to operating systems as well as networks.

Architectural models describe a system in terms of the computational and communication tasks performed by its computational elements; the computational elements being individual computers or aggregates of them supported by appropriate network interconnections.

Architectural models

The architecture of a system is its structure in terms of separately specified components and their interrelationships. The overall goal is to ensure that the structure will meet present and likely future demands on it.

Major concerns are to make the system reliable, manageable, adaptable and cost-effective.

we lay the groundwork for a thorough understanding of approaches such as client-server models, peer-to-peer approaches, distributed objects, distributed components, distributed event based systems and the key differences between these styles.

Communicating entities

From a system perspective, the answer is normally very clear in that the entities that communicate in a distributed system are typically *processes*, leading to the prevailing view of a distributed system as processes coupled with appropriate interprocess communication paradigms.

From a programming perspective, however, this is not enough, and more problem-oriented abstractions have been

proposed:

Objects: Objects have been introduced to enable and encourage the use of object oriented approaches in distributed systems (including both object-oriented design and object-oriented programming languages).

In distributed object-based approaches, a computation consists of a number of interacting objects representing

natural units of decomposition for the given problem domain. Objects are accessed via interfaces, with an associated interface definition language (or IDL) providing a specification of the methods defined on an object.

Java uses the Java Interface Definition Language (IDL) for defining interfaces in the context of remote method invocation, particularly in the context of Java RMI (Remote Method Invocation)

```
// Example.idl
module Example
{  interface Hello {
string sayHello();
};
};
```

Example is a module.

Hello is an interface with a single method **sayHello** that returns a string.

After defining this IDL, you would use an IDL compiler (like idlj) to generate Java code from it. The generated Java code includes the interface definition and additional supporting classes.

In Java, a module is a collection of related packages and resources organized together.

Interface: It is a collection of abstract methods. When a class implements an interface, it inherits the abstract methods declared in the interface

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

- **Objects** are instances of classes in object-oriented programming, representing entities with state and behavior.

- **Components** are modular and reusable parts of a software system, designed for modularity, reusability, and independent deployment. While objects are more closely associated with the principles of OOP, components are broader and may encompass multiple objects or other types of software elements within a larger system.

Component-based middleware often provides additional support for key areas such as deployment and support for server-side programming

Web Services :web services are XML-centered data exchange systems that use the internet for A2A (application-to-application) communication and interfacing. These processes involve programs, messages, documents, and/or objects. A key feature of web services is that applications can be written in various languages and are still able to communicate by exchanging data with one another via a web service between clients and servers. A client summons a web service by sending a request via XML, and the service then responses with an XML response. Web services are also often associated with SOA (Service-Oriented Architecture).

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

Communication paradigms •Consider three types of communication paradigm:

- interprocess communication;
- remote invocation;
- indirect communication.

Interprocess communication refers to the relatively low-level support for communication between processes in distributed systems, including message-passing primitives, direct access to the API offered by Internet protocols (socket programming) and support for multicast communication.

Remote invocation represents the most common communication paradigm in distributed systems, covering a range of techniques based on a two-way exchange between communicating entities in a distributed system and resulting in the calling of a remote operation, procedure or method,

Request-reply protocols: Request-reply protocols are effectively a pattern imposed

on an underlying message-passing service to support client-server computing. In particular, such protocols typically involve a pairwise exchange of messages from client to server and then from server back to client, with the first message containing an encoding of the operation to be executed at the server and also an array of bytes holding associated arguments and the second message containing any results of the operation, again encoded as an array of bytes.

This paradigm is rather primitive and only really used in embedded systems where performance is paramount. The approach is also used in the **HTTP** protocol

GET carries request parameter appended in URL string while POST carries request parameter in message body which makes it more secure way of transferring data from client to

Remote procedure calls: The concept of a remote procedure call (RPC), In RPC, procedures in processes on remote computers can be called as if they are procedures in the local address space.

The underlying RPC system then hides important aspects of distribution, including the encoding and decoding of parameters and results, the passing of messages and the preserving of the required semantics for the procedure call.

This approach directly and elegantly supports client-server computing with servers offering a set of operations through a service interface and clients calling these operations directly as if they were available locally.

Remote procedure calls (*RPCs*) enable *NFS* to transparently provide remote files to local computers.

Remote method invocation: Remote method invocation (RMI) strongly resembles remote procedure calls but in a world of distributed objects. With this approach, a calling object can invoke a method in a remote object. As with RPC, the underlying details are generally hidden from the user.

Group communication: Group communication is concerned with the delivery of messages to a set of recipients and hence is a multiparty communication paradigm supporting one-to-many communication. Group communication relies on the abstraction of a group which is represented in the system by a group identifier. Recipients elect to receive messages sent to a group by joining the group. Senders then send messages to the group via the group identifier, and hence do not need to know the recipients of the message.

Publish-subscribe systems: Many systems, such as the financial trading example , can be classified as information-dissemination systems wherein a large

number of producers (or publishers) distribute information items of interest (events) to a similarly large number of consumers (or subscribers).

Publish-subscribe systems all share the crucial feature of providing an intermediary service that efficiently ensures information generated by producers is routed to consumers who desire this information.

Message queues: Whereas publish-subscribe systems offer a one-to-many style of

communication, message queues offer a point-to-point service whereby producer

processes can send messages to a specified queue and consumer processes can

receive messages from the queue or be notified of the arrival of new messages in the

queue. Queues therefore offer an indirection between the producer and consumer

processes

A simple example of a message queue is an email inbox. When you send an email, it is placed in the recipient's inbox.

The recipient can then read the email at their convenience. This email inbox acts as a buffer between the sender and the recipient, decoupling them from each other.

Tuple spaces: Tuple spaces offer a further indirect communication service by

supporting a model whereby processes can place arbitrary items of structured data,

called tuples, in a persistent tuple space and other processes can either read or remove

such tuples from the tuple space by specifying patterns of interest

A **tuple space** is an implementation of the associative memory (high Speed Memory) paradigm for parallel/distributed computing.

It provides a repository of tuples that can be accessed concurrently. As an illustrative example, consider that there are a group of processors that produce pieces of data and a group of processors that use the data. Producers post their data as tuples in the space, and the consumers then retrieve data from the space that match a certain pattern.

Content-addressable memory (**CAM**) is a special type of computer memory used in certain very-high-speed searching applications. It is also known as **associative memory** or **associative storage** and compares input search data against a table of stored data, and returns the address of matching data.CAM is frequently used in networking devices where it speeds up forwarding information base and routing table operations.

Distributed shared memory: Distributed shared memory (DSM) systems provide an abstraction for sharing data between processes that do not share physical memory. Programmers are nevertheless presented with a familiar abstraction of reading or writing (shared) data structures as if they were in their own local address spaces, thus presenting a high level of distribution transparency. The underlying infrastructure must ensure a copy is provided in a timely manner and also deal with issues relating to synchronization and consistency of data.

distributed shared memory (DSM) is **a form of memory architecture where physically separated memories can be addressed as a single shared address space**.

we examine two architectural styles stemming from the role of individual processes: **client-server** and **peer-to-peer.**

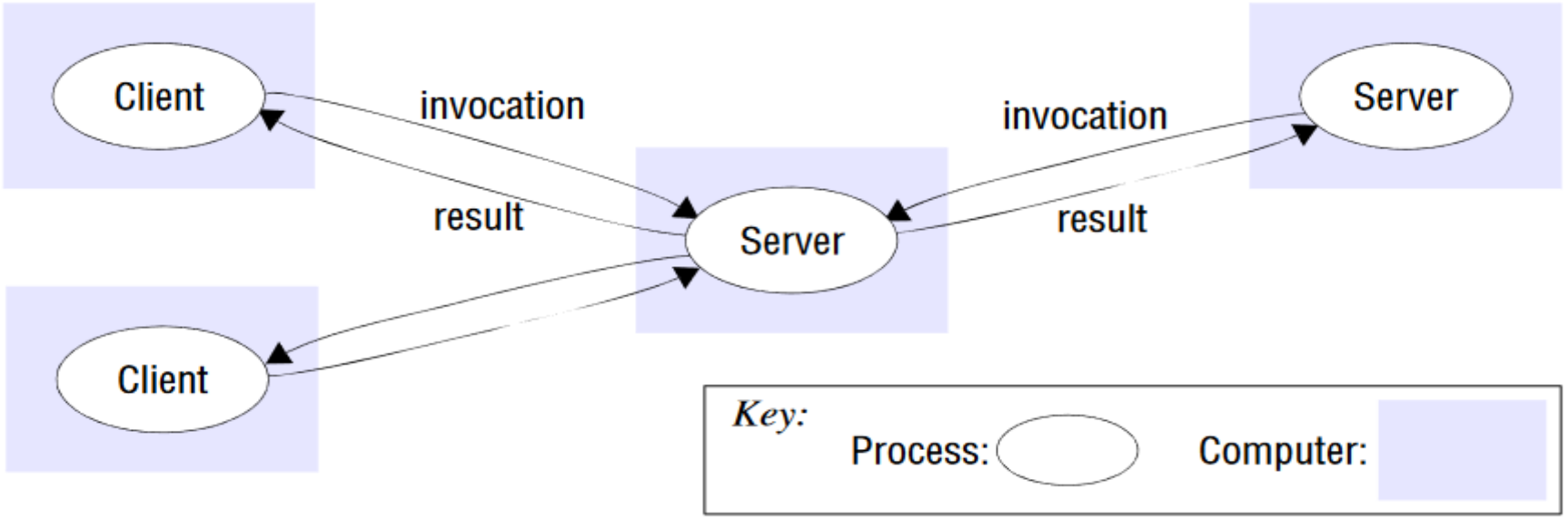
Client-server: This is the architecture that is most often cited when distributed systems are discussed. It is historically the most important and remains the most widely employed. Figure 2.3 illustrates the simple structure in which processes take on the roles of being clients or servers. In particular, client processes interact with individual server processes in potentially separate host computers in order to access the shared resources that they manage.

Servers may in turn be clients of other servers, as the figure indicates. For example, a web server is often a client of a local file server that manages the files in which the web pages are stored. Web servers and most other Internet services are clients of the DNS service, which translates Internet domain names to network addresses**.**

Another web-related example concerns search engines, which enable users to look up summaries of information available on web pages at sites throughout the Internet.

These summaries are made by programs called web crawlers, which run in the background at a search engine site using HTTP requests to access web servers throughout the Internet.

Thus a search engine is both a server and a client: it responds to queries from browser clients and it runs web crawlers that act as clients of other web servers



Peer-to-peer: In this architecture all of the processes involved in a task or activity play similar roles, interacting cooperatively as *peers* without any distinction between client and server processes or the computers on which they run. In practical terms, all participating processes run the same program and offer the same set of interfaces to each other.

While the client-server model offers a direct and relatively simple approach to the sharing of data and other resources, it scales poorly.

The centralization of service provision and management implied by placing a service at a single address does not scale well beyond the capacity of the computer that hosts the service and the bandwidth of its network connections.

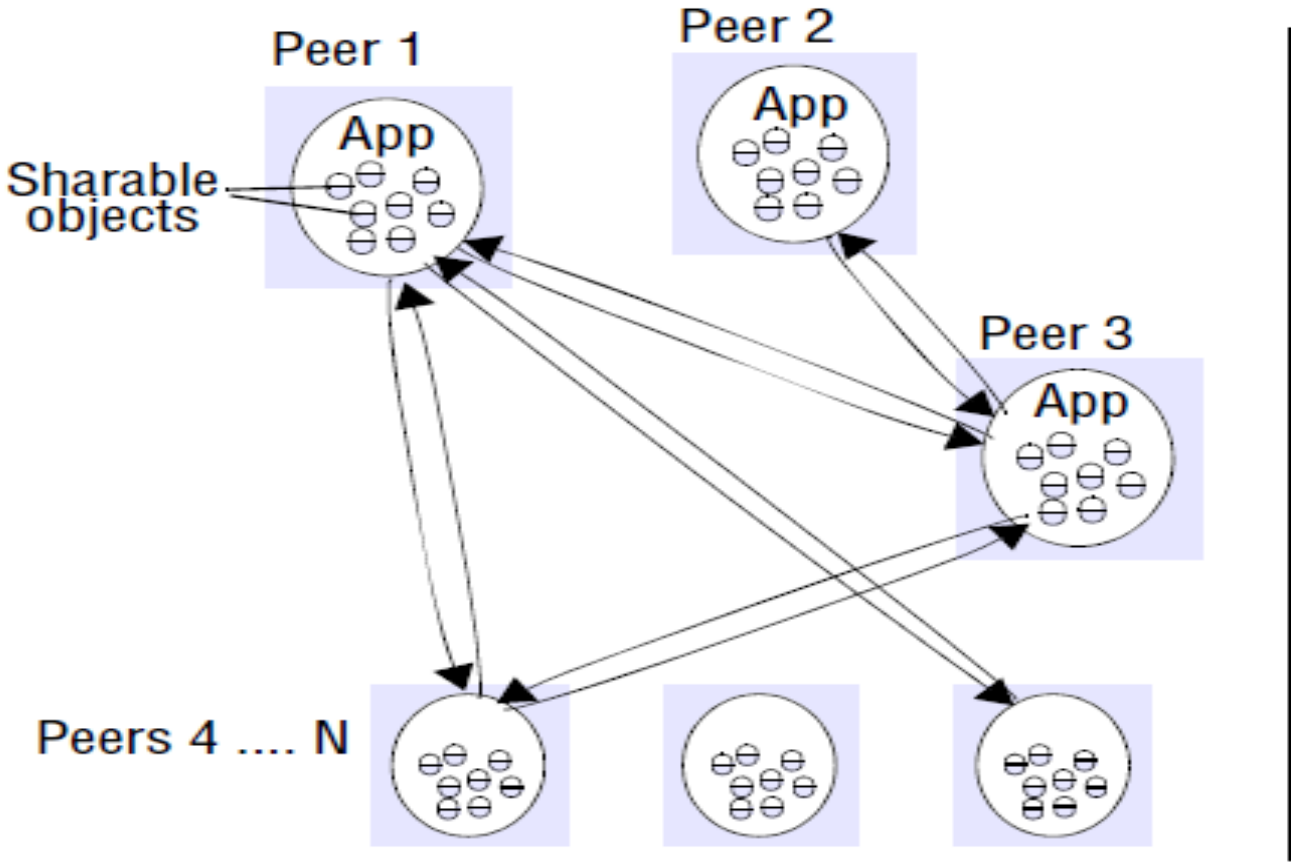
The aim of the peer-to-peer architecture is to exploit the resources (both data and hardware) in a large number of participating computers for the fulfilment of a given task or activity.

Peer-to-peer applications and systems have been successfully constructed that enable tens or hundreds of thousands of

computers to provide access to data and other resources that they collectively store and manage. One of the earliest instances was the Napster application for sharing digital music files.

Figure 2.4a illustrates the form of a peer-to-peer application.

Applications are composed of large numbers of peer processes running on separate computers and the pattern of communication between them depends entirely on application requirements.



Mapping of services to multiple servers: Services may be implemented as several server processes in separate host computers interacting as necessary to provide a service to client processes (Figure 2.4b).

The servers may partition the set of objects on which the service is based and distribute those objects between themselves, or they may maintain replicated copies of them on several hosts.

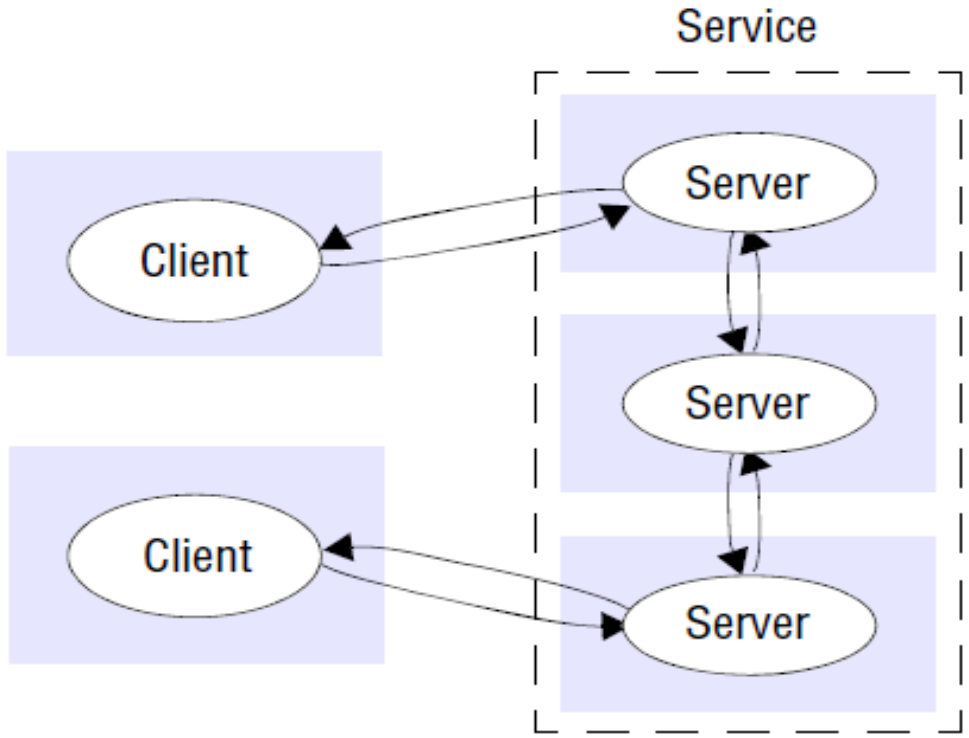
These two options are illustrated by the following examples.

The Web provides a common example of partitioned data in which each web server manages its own set of resources. A user can employ a browser to access a resource at any one of the servers.

An example of a service based on replicated data is the Sun Network Information Service (NIS), which is used to enable all the computers on a LAN to access the same user authentication data when users log in.

Each NIS server has its own replica of a common password file containing a list of users' login names and encrypted passwords

CHARACTERIZATION OF DISTRIBUTED SYSTEMS



Caching: A *cache* is a store of recently used data objects that is closer to one client or a particular set of clients than the objects themselves.

When a new object is received from a server it is added to the local cache store, replacing some existing objects if necessary.

When an object is needed by a client process, the caching service first checks the cache and supplies the object from there if an up-to-date copy is available.

If not, an up-to-date copy is fetched.

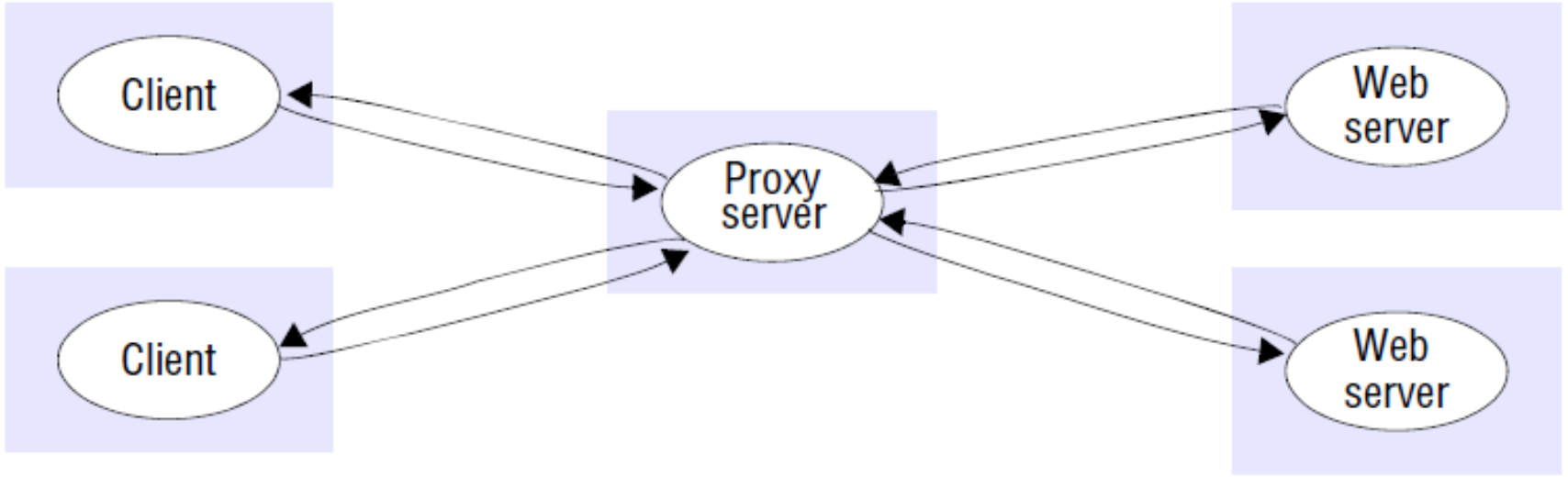
Caches may be co-located with each client or they may be located in a proxy server that can be shared by several clients.

Caches are used extensively in practice.

Web browsers maintain a cache of recently visited web pages and other web resources in the client's local file system, using a special HTTP request to check with the original server that cached pages are up-todate before displaying them.

Web proxy servers (Figure 2.5) provide a shared cache of web resources for the client machines at a site or across several sites.

The purpose of proxy servers is to increase the availability and performance of the service by reducing the load on the wide area network and web servers.



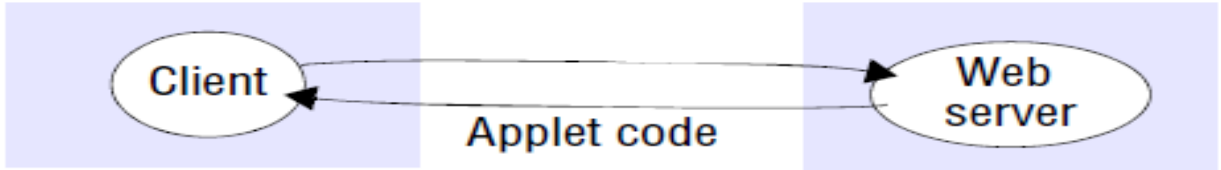
Mobile code: Applets are a well-known and widely used example of mobile code – the user running a browser selects a link to an applet whose code is stored on a web server; the code is downloaded to the browser and runs there, as shown in Figure 2.6.

An advantage of running the downloaded code locally is that it can give good interactive response since it does not suffer from the delays or variability of bandwidth associated with network communication.

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

Web applets

a) client request results in the downloading of applet code



b) client interacts with the applet

Mobile code is a potential security threat to the local resources in the destination computer.

Therefore browsers give applets limited access to local resources

Mobile agents: A mobile agent is a running program (including both code and data) that travels from one computer to another in a network carrying out a task on someone's behalf, such as collecting information, and eventually returning with the results.

A mobile agent may make many invocations to local resources at each site it visits – for example, accessing individual database entries.

Mobile agents might be used to install and maintain software on the computers within an organization or to compare the prices of products from a number of vendors by visiting each vendor's site and performing a series of database operations.

An early example of a similar idea is the so-called worm program developed at Xerox PARC [Shoch and Hupp 1982], which was designed to make use of idle computers in order to carry out intensive computations.

Mobile agents (like mobile code) are a potential security threat to the resources in computers that they visit. The environment receiving a mobile agent should decide which of the local resources it should be allowed to use, based on the identity of the user on whose behalf the agent is acting – their identity must be included in a secure way with the code and data of the mobile agent.

In addition, mobile agents can themselves be vulnerable – they may not be able to complete their task if they are refused access to the information they need.

The tasks performed by mobile agents can be performed by other means.

For example, web crawlers that need to access resources at web servers throughout the Internet work quite successfully by making remote invocations to server processes. For these reasons, the applicability of mobile agents may be limited.