**UNIT-1**

**Distributed system** in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages.

Our definition of distributed systems has the following significant consequences:

* Concurrency
* No global clock
* Independent failures

The prime motivation for constructing and using distributed systems stems from a desire to share resources.

**Examples of distributed systems:**

**1) Web search:** 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.

* ***Finance and commerce:*** The growth of eCommerce as exemplified by companies such as Amazon and eBay, and underlying payments technologies such as PayPal; the associated emergence of online banking and trading and also complex information dissemination systems for financial markets.
* ***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;
* ***Creative industries and entertainment:*** The emergence of online gaming as a novel and highly interactive form of entertainment; the availability of music and film in the home through networked media centers and more widely in the Internet via downloadable or streaming content; the role of user-generated content (as mentioned above) as a new form of creativity, for example via services such as YouTube; the creation of new forms of art and entertainment enabled by emergent (including networked) technologies.
* ***Healthcare****:* The growth of health informatics as a discipline with its emphasis on online electronic patient records and related issues of privacy;
* ***Education:***The emergence of e-learning through for example web-based tools such as virtual learning environments; associated support for distance learning; support for collaborative or community-based learning.
* ***Transport and logistics:***The use of location technologies such as GPS in route finding systems and more general traffic management systems;
* ***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 scientific data;
* ***Environmental management:***The use of (networked) sensor technology to both monitor and manage the natural environment.

**2) 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. 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. Other challenges include the real-time propagation of events to the many players and maintaining a consistent view of the shared world.

**3) Financial trading**

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.



***An example financial trading system***

**Trends in distributed systems:**

Distributed systems are undergoing a period of significant change and this can be traced back to a number of influential trends:

**•** The emergence of pervasive networking technology;

**•** The emergence of ubiquitous computing coupled with the desire to support user mobility in distributed systems;

**•** The increasing demand for multimedia services;

**•** The view of distributed systems as a utility.

**Pervasive networking and the modern Internet**

The modern Internet is a vast interconnected collection of computer networks of many different types, with the range of types increasing all the time and now including, for example, a wide range of wireless communication technologies such as WiFi, WiMAX, Bluetooth and third-generation mobile phone networks. The net result is that networking has become a pervasive resource and devices can be connected at any time and in any place.

The Internet is also a very large distributed system. It enables users, wherever they are, to make use of services such as the World Wide Web, email and file transfer. The set of services is open-ended – it can be extended by the addition of server computers and new types of service.. The role of a *firewall* is to protect an intranet by preventing unauthorized messages from leaving or entering. A firewall is implemented by filtering incoming and outgoing messages. Filtering might be done by source or destination, or a firewall might allow only those messages related to email and web access to pass into or out of the intranet that it protects. Internet Service Providers (ISPs) are companies that provide broadband links and other types of connection to individual users and small organizations, enabling them to access services anywhere in the Internet as well as providing local services such as email and web hosting. The intranets are linked together by backbones. A *backbone* is a network link with a high transmission capacity, employing satellite connections, fiber optic cables and other high-bandwidth circuits.

The implementation of the Internet and the services that it supports has entailed the development of practical solutions to many distributed system issues.

**Mobile and ubiquitous computing**

Technological advances in device miniaturization and wireless networking have led increasingly to the integration of small and portable computing devices into distributed systems. These devices include:

**•** Laptop computers.

**•** Handheld devices, including mobile phones, smart phones, GPS-enabled devices, pagers, personal digital assistants (PDAs), video cameras and digital cameras.

**•** Wearable devices, such as smart watches with functionality similar to a PDA.

**•** Devices embedded in appliances such as washing machines, hi-fi systems, cars and refrigerators.

**Mobile computing** is the performance of computing tasks while the user is on the move, or visiting places other than their usual environment. Mobility introduces a number of challenges for distributed systems, including the need to deal with variable connectivity and indeed disconnection, and the need to maintain operation in the face of device mobility.

***Ubiquitous computing***is the harnessing of many small, cheap computational devices that are present in users’ physical environments, including the home, office and even natural settings. Ubiquitous and mobile computing overlap, since the mobile user can in principle benefit from computers that are everywhere. Ubiquitous computing could benefit users while they remain in a single environment such as the home or a hospital.

**Distributed multimedia systems:**

Multimedia support can usefully be defined as the ability to support a range of media types in an integrated manner. One can expect a distributed system to support the storage, transmission and presentation of what are often referred to as discrete media types, such as pictures or text messages. A distributed multimedia system should be able to perform the same functions for continuous media types such as audio and video; that is, it should be able to store and locate audio or video files, to transmit them across the network to support the presentation of the media types to the user and optionally also to share the media types across a group of users.

***Webcasting***is an application of distributed multimedia technology. Webcasting is the ability to broadcast continuous media, typically audio or video, over the Internet. Distributed multimedia applications such as webcasting place considerable demands on the underlying distributed infrastructure in terms of:

**•** providing support for an (extensible) range of encoding and encryption formats, such as the MPEG series of standards (including for example the popular MP3 standard otherwise known as MPEG-1, Audio Layer 3) and HDTV;

**•** providing a range of mechanisms to ensure that the desired quality of service can be met;

**•** providing associated resource management strategies, including appropriate scheduling policies to support the desired quality of service;

**•** providing adaptation strategies to deal with the inevitable situation in open systems where quality of service cannot be met or sustained.

**Distributed computing as a utility:**

With the increasing maturity of distributed systems infrastructure, a number of companies are promoting the view of distributed resources as a commodity or utility, drawing the analogy between distributed resources and other utilities such as water or electricity. With this model, resources are provided by appropriate service suppliers and effectively rented rather than owned by the end user. This model applies to both physical resources and more logical services

**Focus on resource sharing:**

We routinely share hardware resources such as printers, data resources such as files, and resources with more specific functionality such as search engines. The fact that services restrict resource access to a well-defined set of operations is in part standard software engineering practice. Resources in a distributed system are physically encapsulated within computers and can only be accessed from other computers by means of communication. For effective sharing, each resource must be managed by a program that offers a communication interface enabling the resource to be accessed and updated reliably and consistently.

**Challenges:**

**1) 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.

**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.

**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.

**2) Openness:**

The openness of a computer system is the characteristic that determines whether the system can be extended and re-implemented 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. Openness cannot be achieved unless the specification and documentation of the key software interfaces of the components of a system are made available to software developers.

To summarize:

• Open systems are characterized by the fact that their key interfaces are published.

• Open distributed systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.

• Open distributed systems can be constructed from heterogeneous hardware and software, possibly from different vendors. But the conformance of each component to the published standard must be carefully tested and verified if the system is to work correctly.

**3) Security:**

Security for information resources has three components: confidentiality (protection against disclosure to unauthorized individuals), integrity (protection against alteration or corruption), and availability (protection against interference with the means to access the resources). In a distributed system, clients send requests to access data managed by servers, which involves sending information in messages over a network the following two security challenges, have not yet been fully met:

1. Denial of Service
2. Security of mobile code

**4) 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.

The design of scalable distributed systems presents the following challenges:

* Controlling the cost of physical resources
* Controlling the performance loss
* Preventing software resources running out
* Avoiding performance bottlenecks

**5) Failure handling**

Failures in a distributed system are partial – that is, some components fail while others continue to function.

The following techniques for dealing with failures:

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

***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.

***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; the servers can be designed to detect faults in their peers; when a fault is detected in one server, clients are redirected to the remaining servers.

**6) 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. 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.

**7) 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. The implications of transparency are a major influence on the design of the system software.

***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.

**The World Wide Web:**

The World Wide Web is an evolving system for publishing and accessing resources and services across the Internet. The Web is an *open* system: it can be extended and implemented in new ways without disturbing its existing functionality. First, its operation is based on communication standards and document or content standards that are freely published and widely implemented. Second, the Web is open with respect to the types of resource that can be published and shared on it.

The Web is based on three main standard technological components:

**•** The HyperText Markup Language (HTML), a language for specifying the contents and layout of pages as they are displayed by web browsers;

**•** Uniform Resource Locators (URLs), also known as Uniform Resource Identifiers (URIs), which identify documents and other resources stored as part of the Web;

**•** A client-server system architecture, with standard rules for interaction (the HyperText Transfer Protocol – HTTP) by which browsers and other clients fetch documents and other resources from web servers.

**System Models:**

***Physical models***are the most explicit way in which to describe a system; they capture the hardware composition of a system in terms of the computers and their interconnecting 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.

***Fundamental models***take an abstract perspective in order to examine individual aspects of a distributed system. we introduce fundamental models that examine three important aspects of distributed systems: *interaction models*, which consider the structure and sequencing of the communication between the elements of the system; *failure models*, which consider the ways in which a system may fail to operate correctly and; *security models*, which consider how the system is protected against attempts to interfere with its correct operation or to steal its data.

**PHYSICAL MODELS:**

A physical model is a representation of the underlying hardware elements of a distributed system that abstracts away from specific details of the computer and networking technologies employed.

**Baseline physical model:** A distributed system is one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. This leads to a minimal physical model of a distributed system as an extensible set of computer nodes interconnected by a computer network for the required passing of messages.

Beyond this baseline model, we can usefully identify three generations of distributed systems.

1. **Early distributed systems:** Such systems emerged in the late 1970s and early 1980s in response to the emergence of local area networking technology, usually Ethernet. These systems typically consisted of between 10 and 100 nodes interconnected by a local area network, with limited Internet connectivity and supported a small range of services such as shared local printers and file servers as well as email and file transfer across the Internet.
2. **Internet-scale distributed systems:** Building on this foundation, larger-scale distributed systems started to emerge in the 1990s in response to the dramatic growth of the Internet during this time.
3. **Contemporary distributed systems:** In the above systems, nodes were typically desktop computers and therefore relatively static, discrete and autonomous.

The key trends identified have resulted in significant further developments in physical models:

* The emergence of mobile computing has led to physical models where nodes such as laptops or smart phones may move from location to location in a distributed system, leading to the need for added capabilities such as service discovery and support for spontaneous interoperation.
* The emergence of ubiquitous computing has led to a move from discrete nodes to architectures where computers are embedded in everyday objects and in the surrounding environment.
* The emergence of cloud computing and, in particular, cluster architectures has led to a move from autonomous nodes performing a given role to pools of nodes that together provide a given service.

**Distributed systems of systems:** A system of systems can be defined as a complex system consisting of a series of subsystems that are systems in their own right and that come together to perform a particular task or tasks.

**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.

**1. Architectural elements**

To understand the fundamental building blocks of a distributed system, it is necessary to consider four key questions:

**•** What are the ***entities*** that are communicating in the distributed system?

**•** How do they communicate, or, more specifically, what ***communication paradigm***is used?

**•** What ***roles and responsibilities*** do they have in the overall architecture?

**•** How are they mapped on to the ***physical distributed infrastructure*** (what is their *placement*)?

**A. Communicating entities:** From a system perspective, the entities are 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 , with two caveats:

**•** In some primitive environments, such as sensor networks, the underlying operating systems may not support process abstractions, and hence the entities that communicate in such systems are *nodes*.

**•** In most distributed system environments, processes are supplemented by *threads*, so, strictly speaking, it is threads that are the endpoints of communication.

At one level, this is sufficient to model a distributed system and indeed the fundamental models. From a programming perspective, however, this is not enough, and more problem-oriented abstractions have been proposed:

1. ***Objects*:** Objects have been introduced to enable and encourage the use of object-oriented approaches in distributed systems. 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.
2. ***Components*:** Components resemble objects in that they offer problem-oriented abstractions for building distributed systems and are also accessed through interfaces. The key difference is that components specify not only their (provided) interfaces but also the assumptions they make in terms of other components/interfaces that must be present for a component to fulfill its function.
3. ***Web services*:** Web services are closely related to objects and components, again taking an approach based on encapsulation of behavior and access through interfaces.

**B.Communication paradigms •** We now turn our attention to how entities communicate in a distributed system, and 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:

1. ***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 pair-wise exchange of messages from client to server and then from server back to client.
2. ***Remote procedure calls*:** 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.
3. ***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.

In contrast, a number of techniques have emerged whereby communication is indirect, through a third entity, allowing a strong degree of decoupling between senders and receivers. In particular:

**•** Senders do not need to know who they are sending to (*space uncoupling*).

**•** Senders and receivers do not need to exist at the same time (*time uncoupling*).

***Indirect communication:*** Key techniques for indirect communication include:

***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.

***Publish-subscribe systems*:** Many systems 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).

***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.

***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.

***Distributed shared memory*:** Distributed shared memory (DSM) systems provide an abstraction for sharing data between processes that do not share physical memory.

**C.Roles and responsibilities:** In a distributed system processes – or indeed objects, components or services, including web services– interact with each other to perform a useful activity, for example, to support a chat session. Two architectural styles stemming from the role of individual processes: **client-server** and **peer-to-peer**.

**Placement •** Placement is crucial in terms of determining the properties of the distributed system, most obviously with regard to performance but also to other aspects, such as reliability and security.

We therefore focus mainly on the following placement strategies, which can significantly alter the characteristics of a given design:

**•** Mapping of services to multiple servers;

**•** caching;

**•** Mobile code;

**•** Mobile agents.

**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. 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..

A more closely coupled type of multiple-server architecture is the cluster. A cluster is constructed from up to thousands of commodity processing boards, and service processing can be partitioned or replicated between them.

**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.

**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.

**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

**2. Architectural patterns:**

Architectural patterns build on the more primitive architectural elements and provide composite recurring structures that have been shown to work well in given circumstances. We present several key architectural patterns in distributed systems, including layering and tiered architectures and the related concept of thin clients

**Layering •** In a layered approach, a complex system is partitioned into a number of layers, with a given layer making use of the services offered by the layer below. A given layer therefore offers a software abstraction, with higher layers being unaware of implementation details, or indeed of any other layers beneath them.



***Software and hardware service layers in distributed systems***

The important terms *platform* and *middleware*, which we define as follows:

A **platform** for distributed systems and applications consists of the lowest-level hardware and software layers. These low-level layers provide services to the layers above them, which are implemented independently in each computer, bringing the system’s programming interface up to a level that facilitates communication and coordination between processes.

**Middleware** was defined as a layer of software whose purpose is to mask heterogeneity and to provide a convenient programming model to application programmers. Middleware is represented by processes or objects in a set of computers that interact with each other to implement communication and resource-sharing support for distributed applications.

**Tiered architecture •** Tiered architectures are complementary to layering. Whereas layering deals with the vertical organization of services into layers of abstraction, tiering is a technique to organize functionality of a given layer and place this functionality into appropriate servers and, as a secondary consideration, on to physical nodes.

Let us first examine the concepts of two- and three-tiered architecture. Consider the functional decomposition of a given application, as follows:

**•** The **presentation logic**, which is concerned with handling user interaction and updating the view of the application as presented to the user;

**•** The **application logic**, which is concerned with the detailed application-specific processing associated with the application (also referred to as the business logic, although the concept is not limited only to business applications);

**•** The **data logic**, which is concerned with the persistent storage of the application, typically in a database management system.

In the two-tier solution, the three aspects mentioned above must be partitioned into two processes, the client and the server. This is most commonly done by splitting the application logic, with some residing in the client and the remainder in the server. The advantage of this scheme is low latency in terms of interaction, with only one exchange of messages to invoke an operation. The disadvantage is the splitting of application logic across a process boundary, with the consequent restriction on which parts of the logic can be directly invoked from which other part.



In the three-tier solution, there is a one-to-one mapping from logical elements to physical servers and hence, for example, the application logic is held in one place, which in turn can enhance maintainability of the software. Each tier also has a well-defined role;

**Thin clients •** The trend in distributed computing is towards moving complexity away from the end-user device towards services in the Internet. This trend has given rise to interest in the concept of a *thin client*, enabling access to sophisticated networked services, provided for example by a cloud solution, with few assumptions or demands on the client device. More specifically, the term thin client refers to a software layer that supports a window-based user interface that is local to the user while executing application programs or, more generally, accessing services on a remote computer.

The main drawback of the thin client architecture is in highly interactive graphical activities such as CAD and image processing, where the delays experienced by users are increased to unacceptable levels by the need to transfer image and vector information between the thin client and the application process, due to both network and operating system latencies.

**Other commonly occurring patterns •** A large number of architectural patterns have now been identified and documented. Here are a few key examples:

The ***proxy* pattern** is a commonly recurring pattern in distributed systems designed particularly to support location transparency in remote procedure calls or remote method invocation.

The use of ***brokerage***in web services can usefully be viewed as an architectural pattern supporting interoperability in potentially complex distributed infrastructures. In particular, this pattern consists of the trio of service provider, service requester and service broker

***Reflection***is a pattern that is increasingly being used in distributed systems as a means of supporting both introspection (the dynamic discovery of properties of the system) and intercession (the ability to dynamically modify structure or behavior).

**3. Associated middleware solutions:** The task of middleware is to provide a higher-level programming abstraction for the development of distributed systems and, through layering, to abstract over heterogeneity in the underlying infrastructure to promote interoperability and portability.



***Categories of middleware***

**FUNDAMENTAL MODELS:**

In general, such a fundamental model should contain only the essential ingredients that we need to consider in order to understand and reason about some aspects of a system’s behavior. The purpose of such a model is:

**•** To make explicit all the relevant assumptions about the systems we are modeling.

**•** To make generalizations concerning what is possible or impossible, given those assumptions.

The aspects of distributed systems that we wish to capture in our fundamental models are intended to help us to discuss and reason about:

***Interaction*:** Computation occurs within processes; the processes interact by passing messages, resulting in communication (information flow) and coordination (synchronization and ordering of activities) between processes.

***Failure*:** The correct operation of a distributed system is threatened whenever a fault occurs in any of the computers on which it runs or in the network that connects them. Our model defines and classifies the faults.

***Security*:** The modular nature of distributed systems and their openness exposes them to attack by both external and internal agents. Our security model defines and classifies the forms that such attacks may take, providing a basis for the analysis of threats to a system and for the design of systems that are able to resist them.

**1. Interaction model**

System architectures indicate that fundamentally distributed systems are composed of many processes, interacting in complex ways. For example:

**•** Multiple server processes may cooperate with one another to provide a service;

**•** A set of peer processes may cooperate with one another to achieve a common goal:

Their behavior and state can be described by a *distributed algorithm* – a definition of the steps to be taken by each of the processes of which the system is composed, *including the transmission of messages between them*. Messages are transmitted between processes to transfer information between them and to coordinate their activity.

Interacting processes perform all of the activity in a distributed system. Each process has its own state, consisting of the set of data that it can access and update, including the variables in its program. The state belonging to each process is completely private – that is, it cannot be accessed or updated by any other process.

The two significant factors affecting interacting processes in a distributed system are:

**•** Communication performance is often a limiting characteristic.

**•** It is impossible to maintain a single global notion of time.

**Performance of communication channels •** Communication over a computer network has the following performance characteristics relating to latency, bandwidth and jitter:

**•** The **delay** between the start of a message’s transmission from one process and the beginning of its receipt by another is referred to as *latency*.

**•** The ***bandwidth***of a computer network is the total amount of information that can be transmitted over it in a given time.

**• *Jitter***is the variation in the time taken to deliver a series of messages. Jitter is relevant to multimedia data.

**Computer clocks and timing events •** Each computer in a distributed system has its own internal clock, which can be used by local processes to obtain the value of the current time. Therefore two processes running on different computers can each associate timestamps with their events. The term *clock drift rate* refers to the rate at which a computer clock deviates from a perfect reference clock.

**Two variants of the interaction model •** In a distributed system it is hard to set limits on the time that can be taken for process execution, message delivery or clock drift. Two opposing extreme positions provide a pair of simple models – the first has a strong assumption of time and the second makes no assumptions about time:

***Synchronous distributed systems***: synchronous distributed system to be one in which the following bounds are defined:

**•** The time to execute each step of a process has known lower and upper bounds.

**•** Each message transmitted over a channel is received within a known bounded time.

**•** Each process has a local clock whose drift rate from real time has a known bound.

***Asynchronous distributed systems***: An asynchronous distributed system is one in which there are no bounds on:

**•** Process execution speeds – for example, one process step may take only a picoseconds and another a century; all that can be said is that each step may take an arbitrarily long time.

**•** Message transmission delays – for example, one message from process A to process B may be delivered in negligible time and another may take several years. In other words, a message may be received after an arbitrarily long time.

**•** Clock drift rates – again, the drift rate of a clock is arbitrary.

**Event ordering •** In many cases, we are interested in knowing whether an event (sending or receiving a message) at one process occurred before, after or concurrently with another event at another process. The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks.

**2. Failure model**

In a distributed system both processes and communication channels may fail – that is, they may depart from what is considered to be correct or desirable behavior. The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures. A taxonomy that distinguishes between the failures of processes and communication channels. These are presented under the headings omission failures, arbitrary failures and timing failures.

**Omission failures •** The faults classified as *omission failures* refer to cases when a process or communication channel fails to perform actions that it is supposed to do.

1. **Process omission failures:** The chief omission failure of a process is to crash. When we say that a process has crashed we mean that it has halted and will not execute any further steps of its program ever. A process crash is called *fail-stop* if other processes can detect certainly that the process has crashed.
2. **Communication omission failures:** The communication channel produces an omission failure if it does not transport a message from *p*’s outgoing message buffer to *q*’s incoming message buffer. This is known as ‘dropping messages’ and is generally caused by lack of buffer space at the receiver or at an intervening gateway, or by a network transmission error, detected by a checksum carried with the message data.

Failures can be categorized according to their severity. All of the failures we have described so far are *benign* failures. Most failures in distributed systems are benign. Benign failures include failures of omission as well as timing failures and performance failures.

**Arbitrary failures •** The term *arbitrary* or *Byzantine* failure is used to describe the worst possible failure semantics, in which any type of error may occur. For example, a process may set wrong values in its data items, or it may return a wrong value in response to an invocation. Communication channels can suffer from arbitrary failures;

**Timing failures •** Timing failures are applicable in synchronous distributed systems where time limits are set on process execution time, message delivery time and clock drift rate.

**Masking failures •** Each component in a distributed system is generally constructed from a collection of other components. It is possible to construct reliable services from components that exhibit failures. Knowledge of the failure characteristics of a component can enable a new service to be designed to mask the failure of the components on which it depends.

A service *masks* a failure either by hiding it altogether or by converting it into a more acceptable type of failure.

**Reliability of one-to-one communication •** Although a basic communication channel can exhibit the omission failures, it is possible to use it to build a communication service that masks some of those failures.

The term *reliable communication* is defined in terms of validity and integrity as follows:

***Validity*:** Any message in the outgoing message buffer is eventually delivered to the incoming message buffer.

***Integrity*:** The message received is identical to one sent, and no messages are delivered twice.

The threats to integrity come from two independent sources:

**•** Any protocol that retransmits messages but does not reject a message that arrives twice. Protocols can attach sequence numbers to messages so as to detect those that are delivered twice.

**•** Malicious users that may inject spurious messages, replay old messages or tamper with messages. Security measures can be taken to maintain the integrity property in the face of such attacks.

**OPERATING SYSTEM SUPPORT:**

The task of any operating system is to provide problem-oriented abstractions of the underlying physical resources – the processors, memory, networks, and storage media. It takes over the physical resources on a single node and manages them to present these resource abstractions through the system-call interface. An operating system that produces a single system image like this for all the resources in a distributed system is called a ***distributed operating system.***

**Middleware and network operating systems •** In fact, there are no distributed operating systems in general use, only network operating systems such as UNIX, Mac OS and Windows. This is likely to remain the case, for two main reasons. The first is that users have much invested in their application software, which often meets their current problem-solving needs;

The second reason against the adoption of distributed operating systems is that users tend to prefer to have a degree of autonomy for their machines, even in a closely knit organization.

**The operating system layer:**

****

Our goal is to examine the impact of particular OS mechanisms on middleware’s ability to deliver distributed resource sharing to users. Kernels and the client and server processes that execute upon them are the chief architectural components that concern us. Kernels and server processes are the components that manage resources and present clients with an interface to the resources.

We require at least the following of them:

***Encapsulation*:** They should provide a useful service interface to their resources – that is, a set of operations that meet their clients’ needs.

***Protection*:** Resources require protection from illegitimate accesses – for example, files are protected from being read by users without read permissions, and device registers are protected from application processes.

***Concurrent processing*:** Clients may share resources and access them concurrently. Resource managers are responsible for achieving concurrency transparency.

A combination of libraries, kernels and servers may be called upon to perform the following invocation related tasks:

***Communication*:** Operation parameters and results have to be passed to and from resource managers, over a network or within a computer.

***Scheduling*:** When an operation is invoked, its processing must be scheduled within the kernel or server.

****

***Core OS functionality***

The core OS components and their responsibilities are:

***Process manager*:** Creation of and operations upon processes. A process is a unit of resource management, including an address space and one or more threads.

***Thread manager*:** Thread creation, synchronization and scheduling. Threads are schedulable activities attached to processes.

***Communication manager*:** Communication between threads attached to different processes on the same computer.

***Memory manager*:** Management of physical and virtual memory.

***Supervisor*:** Dispatching of interrupts, system call traps and other exceptions; control of memory management unit and hardware caches; processor and floating-point unit register manipulations. This is known as the Hardware Abstraction Layer in Windows.

**Protection:**

Protecting the file consists of two sub-problems. The first is to ensure that each of the file’s two operations can be performed only by clients with the right to perform it. A complete solution to this resource-protection sub-problem in a distributed system requires cryptographic techniques.

**Kernels and protection •** The kernel is a program that is distinguished by the facts that it remains loaded from system initialization and its code is executed with complete access privileges for the physical resources on its host computer. In particular, it can control the memory management unit and set the processor registers so that no other code may access the machine’s physical resources except in acceptable ways.

A kernel process executes with the processor in *supervisor* (privileged) mode; the kernel arranges that other processes execute in *user* (unprivileged) mode.

The kernel also sets up *address spaces* to protect itself and other processes from the accesses of an aberrant process, and to provide processes with their required virtual memory layout. An address space is a collection of ranges of virtual memory locations, in each of which a specified combination of memory access rights applies, such as read-only or read-write. A process cannot access memory outside its address space. The terms *user process* or *user-level process* are normally used to describe one that executes in user mode and has a user-level address space.

**Processes and threads:**

A process consists of an execution environment together with one or more threads. A *thread* is the operating system abstraction of an activity. An *execution environment* is the unit of resource management: a collection of local kernel managed resources to which its threads have access. An execution environment primarily consists of:

**•** An address space;

**•** thread synchronization and communication resources such as semaphores and communication interfaces (for example, sockets);

**•** Higher-level resources such as open files and windows.

Threads can be created and destroyed dynamically, as needed. The central aim of having multiple threads of execution is to maximize the degree of concurrent execution between operations, thus enabling the overlap of computation with input and output, and enabling concurrent processing on multiprocessors.

An execution environment provides protection from threads outside it, so that the data and other resources contained in it are by default inaccessible to threads residing in other execution environments.

**1. Address spaces:**

An address space is a unit of management of a process’s virtual memory. It is large and consists of one or more *regions*, separated by inaccessible areas of virtual memory. A region is an area of contiguous virtual memory that is accessible by the threads of the owning process. Each region is specified by the following properties:

**•** Its extent (lowest virtual address and size);

**•** Read/write/execute permissions for the process’s threads;

**•** Whether it can be grown upwards or downwards.

The provision of an indefinite number of regions is motivated by several factors. One of these is the need to support a separate stack for each thread. Another motivation is to enable files in general – not just the text and data sections of binary files – to be mapped into the address space. A *mapped file* is one that is accessed as an array of bytes in memory. The virtual memory system ensures that accesses made in memory are reflected in the underlying file storage.

The need to share memory between processes, or between processes and the kernel, is another factor leading to extra regions in the address space. A *shared memory region* is one that is backed by the same physical memory as one or more regions belonging to other address spaces. The uses of shared regions include the following:

***Libraries*:** Library code can be very large and would waste considerable memory if it was loaded separately into every process that used it. Instead, a single copy of the library code can be shared by being mapped as a region in the address spaces of processes that require it.

***Kernel*:** Often the kernel code and data are mapped into every address space at the same location. When a process makes a system call or an exception occurs, there is no need to switch to a new set of address mappings.

***Data sharing and communication*:** Two processes, or a process and the kernel, might need to share data in order to cooperate on some task.

**2 Creation of a new process:**

The creation of a new process has traditionally been an indivisible operation provided by the operating system. For a distributed system, the design of the process-creation mechanism has to take into account the utilization of multiple computers; consequently, the process-support infrastructure is divided into separate system services.

The creation of a new process can be separated into two independent aspects:

**•** The choice of a target host,

**•** The creation of an execution environment.

**Choice of process host •**. In general, process allocation policies range from always running new processes at their originator’s workstation to sharing the processing load between a set of computers. Two policy categories for load sharing.

The ***transfer policy***determines whether to situate a new process locally or remotely.

The ***location policy***determines which node should host a new process selected for transfer. This decision may depend on the relative loads of nodes, on their machine architectures or on any specialized resources they may possess.

Process location policies may be *static* or *adaptive*. The former operate without regard to the current state of the system.

Load-sharing systems may be centralized, hierarchical or decentralized. In the first case there is one *load manager* component, and in the second there are several, organized in a tree structure. Load managers collect information about the nodes and use it to allocate new processes to nodes. In hierarchical systems, managers make process allocation decisions as far down the tree as possible, but managers may transfer processes to one another, via a common ancestor, under certain load conditions. In a decentralized load-sharing system, nodes exchange information with one another directly to make allocation decisions.

In *sender-initiated* load-sharing algorithms, the node that requires a new process to be created is responsible for initiating the transfer decision. It typically initiates a transfer when its own load crosses a threshold. By contrast, in *receiver-initiated* algorithms, a node whose load is below a given threshold advertises its existence to other nodes so that relatively loaded nodes can transfer work to it.

*Migratory* load-sharing systems can shift load at any time, not just when a new process is created. They use a mechanism called *process migration*: the transfer of an executing process from one node to another.

**Creation of a new execution environment •** Once the host computer has been selected, a new process requires an execution environment consisting of an address space with initialized contents.

There are two approaches to defining and initializing the address space of a newly created process. The first approach is used where the address space is of a statically defined format. Address space regions are initialized from an executable file or filled with zeros as appropriate.

Alternatively, the address space can be defined with respect to an existing execution environment.

Mach [Accetta *et al.* 1986] and Chorus [Rozier *et al.* 1988, 1990], for example, apply an optimization called *copy-on-write* when an inherited region is copied from the parent. The region is copied, but no physical copying takes place by default. The page frames that make up the inherited region are shared between the two address spaces. A page in the region is only physically copied when one or another process attempts to modify it.

**3. Threads:**

The server has a pool of one or more threads, each of which repeatedly removes a request from a queue of received requests and processes it.

Consider the *maximum* server throughput, measured in client requests handled per second, for different numbers of threads. If a single thread has to perform all processing, then the turnaround time for handling any request is on average 2 + 8 = 10 milliseconds, so this server can handle 100 client requests per second. Any new request messages that arrive while the server is handling a request are queued at the server port.

Now consider what happens if the server pool contains two threads. We assume that threads are independently schedulable – that is, one thread can be scheduled when another becomes blocked for I/O. Then thread number two can process a second request while thread number one is blocked, and vice versa. This increases the server throughput.

The throughput can be increased by using a shared-memory multiprocessor to ease the processor bottleneck. A multi-threaded process maps naturally onto a shared memory multiprocessor. The shared execution environment can be implemented in shared memory, and the multiple threads can be scheduled to run on the multiple processors.

**Architectures for multi-threaded servers •** Figure shows one of the possible threading architectures, the *worker pool architecture*. In its simplest form, the server creates a fixed pool of ‘worker’ threads to process the requests when it starts up. The module marked ‘receipt and queuing’ in Figure is typically implemented by an ‘I/O’ thread, which receives requests from a collection of sockets or ports and places them on a shared request queue for retrieval by the workers.



A disadvantage of this architecture is its inflexibility: Another disadvantage is the high level of switching between the I/O and worker threads as they manipulate the shared queue.

In the ***thread-per-request architecture*** the I/O thread spawns a new worker thread for each request, and that worker destroys it when it has processed the request against its designated remote object. This architecture has the advantage that the threads do not contend for a shared queue, and throughput is potentially maximized because the I/O thread can create as many workers as there are outstanding requests. Its disadvantage is the overhead of the thread creation and destruction operations.



***Alternative server threading architectures***

The ***thread-per-connection architecture***associates a thread with each connection. The server creates a new worker thread when a client makes a connection and destroys the thread when the client closes the connection

The ***thread-per-object architecture*** associates a thread with each remote object. An I/O thread receives requests and queues them for the workers, but this time there is a per-object queue.

**Threads within clients •** Threads can be useful for clients as well as servers. The first thread generates results to be passed to a server by remote method invocation, but does not require a reply. Remote method invocations typically block the caller, even when there is strictly no need to wait. This client process can incorporate a second thread, which performs the remote method invocations and blocks while the first thread is able to continue computing further results. The first thread places its results in buffers, which are emptied by the second thread. It is only blocked when all the buffers are full.

**Threads versus multiple processes •** Figure shows some of the main state components that must be maintained for execution environments and threads, respectively. An execution environment has an address space, communication interfaces such as sockets, and higher-level resources such as open files and thread synchronization objects such as semaphores; it also lists the threads associated with it. A thread has a scheduling priority, an execution state (such as *BLOCKED* or *RUNNABLE*), and saved processor register values when the thread is *BLOCKED*, and state concerning the thread’s software interrupt handling. A *software interrupt* is an event that causes a thread to be interrupted (similar to the case of a hardware interrupt). If the thread has assigned a handler procedure, control is transferred to it. UNIX signals are examples of software interrupts.



The figure shows that an execution environment and the threads belonging to it are both associated with pages belonging to the address space held in main memory, and data and instructions held in hardware caches.

We can summarize a comparison of processes and threads as follows:

**•** Creating a new thread within an existing process is cheaper than creating a process.

**•** More importantly, switching to a different thread within the same process is cheaper than switching between threads belonging to different processes.

**•** Threads within a process may share data and other resources conveniently and efficiently compared with separate processes.

**•** But, by the same token, threads within a process are not protected from one another.

The second performance advantage of threads concerns *switching* between threads – that is, running one thread instead of another at a given processor. Switching between threads sharing the same execution environment is considerably cheaper than switching between threads belonging to different processes. The overheads associated with thread switching are related to scheduling and context switching.

A processor context comprises the values of the processor registers such as the program counter, and the current hardware protection domain: the address space and the processor protection mode (supervisor or user). A *context switch* is the transition between contexts that takes place when switching between threads, or when a single thread makes a system call or takes another type of exception. It involves the following:

**•** The saving of the processor’s original register state, and the loading of the new state;

**•** In some cases, a transfer to a new protection domain – this is known as a *domain transition*.

Switching between threads sharing the same execution environment entirely at user level involves no domain transition and is relatively cheap. Switching to the kernel, or to another thread belonging to the same execution environment via the kernel, involves a domain transition.

**Threads programming •** Threads programming is concurrent programming, as traditionally studied in, the field of operating systems. The following concurrent programming concepts are explained fully: *race conditions*, *critical sections*, *monitors*, *condition variables* and *semaphores*.

**Thread lifetimes •** A new thread is created on the same Java virtual machine (JVM) as its creator, in the *SUSPENDED* state. After it is made *RUNNABLE* with the *start()* method, it executes the *run()* method of an object designated in its constructor. The JVM and the threads on top of it all execute in a process on top of the underlying operating system. Threads can be assigned a priority, so that a Java implementation that supports priorities will run a particular thread in preference to any thread with lower priority. A thread ends its life when it returns from the *run()* method or when its *destroy()* method is called.

Programs can manage threads in groups. Every thread belongs to one group, which it is assigned at the time of its creation. Thread groups are useful when several applications coexist on the same JVM. Thread groups also facilitate control of the relative priorities of threads.

**Thread synchronization •** Programming a multi-threaded process requires great care. The main difficult issues are the sharing of objects and the techniques used for thread coordination and cooperation. Each thread’s local variables in methods are private to it – threads have private stacks

Java provides the *synchronized* keyword for programmers to designate the well known monitor construct for thread coordination. Programmers designate either entire methods or arbitrary blocks of code as belonging to a monitor associated with an individual object. The monitor’s guarantee is that at most one thread can execute within it at any time.

Java allows threads to be blocked and woken up via arbitrary objects that act as condition variables. A thread that needs to block awaiting a certain condition calls an object’s *wait()* method. All objects implement this method, since it belongs to Java’s root *Object* class. Another thread calls *notify()* to unblock at most one thread or *notifyAll()* to unblock all threads waiting on that object. Both notification methods also belong to the *Object* class.

**Thread scheduling •** An important distinction is between preemptive and non-preemptive scheduling of threads. In *preemptive scheduling*, a thread may be suspended at any point to make way for another thread, even when the preempted thread would otherwise continue running. In *non-preemptive scheduling*, a thread runs until it makes a call to the threading system, when the system may deschedule it and schedule another thread to run.

**Threads implementation •**. A threads runtime library organizes the scheduling of threads. A thread would block the process, and therefore all threads within it, if it made a blocking system call, so the asynchronous (non-blocking) I/O facilities of the underlying kernel are exploited. Similarly, the implementation can utilize the kernel-provided timers and software interrupt facilities to time slice between threads.

When no kernel support for multi-threaded processes is provided, a user-level threads implementation suffers from the following problems:

**•** The threads within a process cannot take advantage of a multiprocessor.

**•** A thread that takes a page fault blocks the entire process and all threads within it.

**•** Threads within different processes cannot be scheduled according to a single scheme of relative prioritization.

User-level threads implementations, on the other hand, have significant advantages over kernel-level implementations:

**•** Certain thread operations are significantly less costly.

**•** Given that the thread-scheduling module is implemented outside the kernel, it can be customized or changed to suit particular application requirements.

**•** Many more user-level threads can be supported than could reasonably be provided by default by a kernel.

Figure shows that a process notifies the kernel when either of two types of event occurs: when a virtual processor is ‘idle’ and no longer needed, or when an extra virtual processor is required.



***Scheduler activations***

Figure also shows that the kernel notifies the process when any of four types of event occurs. Scheduler *activation* (SA) is a call from the kernel to a process, which notifies the process’s scheduler of an event. The four types of event that the kernel notifies the user-level scheduler of are as follows:

***Virtual processor allocated*:** The kernel has assigned a new virtual processor to the process, and this is the first time slice upon it; the scheduler can load the SA with the context of a *READY* thread, which can thus recommence execution.

***SA blocked*:** An SA has blocked in the kernel, and the kernel is using a fresh SA to notify the scheduler; the scheduler sets the state of the corresponding thread to *BLOCKED* and can allocate a *READY* thread to the notifying SA.

***SA unblocked*:** An SA that was blocked in the kernel has become unblocked and is ready to execute at user level again; the scheduler can now return the corresponding thread to the *READY* list.

***SA preempted*:** The kernel has taken away the specified SA from the process; the scheduler places the preempted thread in the *READY* list and reevaluates the thread allocation.

**Communication and invocation:**

Communication as part of the implementation of what we have called an *invocation* – a construct, such as a remote method invocation, remote procedure call or event notification, whose purpose is to bring about an operation on a resource in a different address space.

**Communication primitives •** Some kernels designed for distributed systems have provided communication primitives tailored to the types of invocation. Amoeba, for example, provides *doOperation*, *getRequest* and *sendReply* as primitives. Amoeba, the V system and Chorus provide group communication primitives Developers typically implement middleware over sockets giving access to Internet standard protocols – often connected sockets using TCP but sometimes unconnected UDP sockets. The principal reasons for using sockets are portability and interoperability: middleware is required to operate over as many widely used operating systems as possible, and all common operating systems, such as UNIX and the Windows family, provide similar socket APIs giving access to TCP and UDP protocols.

Despite the widespread use of TCP and UDP sockets provided by common kernels, research continues to be carried out into lower-cost communication primitives in experimental kernels.

**Protocols and openness •** One of the main requirements of the operating system is to provide standard protocols that enable interworking between middleware implementations on different platforms. Several research kernels developed in the 1980s incorporated their own network protocols tuned to RPC interactions – notably Amoeba RPC, VMTP and Sprite RPC. These kernels provide message passing between local processes only, and leave network protocol processing to a server that runs on top of the kernel.

Protocols are normally arranged in a *stack* of layers. Many operating systems allow new layers to be integrated statically, by including a layer such as IrDA as a permanently installed protocol ‘driver’. By contrast, *dynamic protocol composition* is a technique whereby a protocol stack can be composed on the fly to meet the requirements of a particular application, and to utilize whichever physical layers are available given the platform’s current connectivity.

**1. Invocation performance:**

Invocation performance is a critical factor in distributed system design. The more designers separate functionality between address spaces, the more remote invocations are required. Clients and servers may make many millions of invocation-related operations in their lifetimes, so small fractions of milliseconds count in invocation costs.

**Invocation costs •** Calling a conventional procedure or invoking a conventional method, making a system call, sending a message, remote procedure calling and remote method invocation are all examples of invocation mechanisms. Each mechanism causes code to be executed outside the scope of the calling procedure or object. Each involves, in general, the communication of arguments to this code and the return of data values to the caller. Invocation mechanisms can be either synchronous, as for example in the case of conventional and remote procedure calls, or asynchronous.

The important performance-related distinctions between invocation mechanisms, apart from whether or not they are synchronous, are whether they involve a domain transition (that is, whether they cross an address space), whether they involve communication across a network and whether they involve thread scheduling and switching.

**Invocation over the network •** A *null RPC* (and similarly, a *null RMI*) is defined as an RPC without parameters that executes a null procedure and returns no values. Its execution involves an exchange of messages carrying some system data but no user data. The time taken by a null RPC between user processes connected by a LAN is on the order of a tenth of a millisecond.

Recall that the steps in an RPC are as follows (RMI involves similar steps):

**•** A client stub marshals the call arguments into a message, sends the request message and receives and unmarshals the reply.

**•** At the server, a worker thread receives the incoming request, or an I/O thread receives the request and passes it to a worker thread; in either case, the worker calls the appropriate server stub.

**•** The server stub unmarshals the request message, calls the designated procedure, and marshals and sends the reply.

The following are the main components accounting for remote invocation delay, besides network transmission times:

***Marshalling*:** Marshalling and unmarshalling, which involve copying and converting data, create a significant overhead as the amount of data grows.

***Data copying*:** Potentially, even after marshalling, message data is copied several times in the course of an RPC:

1. across the user–kernel boundary, between the client or server address space and kernel buffers;

2. Across each protocol layer (for example, RPC/UDP/IP/Ethernet);

3. between the network interface and kernel buffers.

Transfers between the network interface and main memory are usually handled by direct memory access (DMA). The processor handles the other copies.

***Packet initialization*:** This involves initializing protocol headers and trailers, including checksums. The cost is therefore proportional, in part, to the amount of data sent.

***Thread scheduling and context switching*:** These may occur as follows:

1. Several system calls are made during an RPC, as stubs invoke the kernel’s communication operations.

2. One or more server threads are scheduled.

3. If the operating system employs a separate network manager process, then each *Send* involves a context switch to one of its threads.

***Waiting for acknowledgements*:** The choice of RPC protocol may influence delay, particularly when large amounts of data are sent.

**Memory sharing •** Shared regions may be used for rapid communication between a user process and the kernel, or between user processes. Data is communicated by writing to and reading from the shared region. Data is thus passed efficiently, without being copied to and from the kernel’s address space. But system calls and software interrupts may be required for synchronization, such as when the user process has written data that should be transmitted, or when the kernel has written data for the user process to consume.

**Choice of protocol •** The delay that a client experiences during request-reply interactions over TCP is not necessarily worse than for UDP and in fact is sometimes better, particularly for large messages. However, care is required when implementing request-reply interactions on top of a protocol such as TCP, which was not specifically designed for this purpose. In particular, TCP’s buffering behavior can hinder good performance, and its connection overheads put it at a disadvantage compared with UDP, unless enough requests are made over a single connection to render the overhead per request negligible.

**Invocation within a computer • M**ost cross-address-space invocation took place within a computer and not, as might be expected in a client-server installation, between computers. The trend towards placing service functionality inside user-level servers means that more and more invocations will be to a local process. This is especially so as caching is pursued aggressively if the data needed by a client is liable to be held in a local server. The cost of an RPC within a computer is growing in importance as a system performance parameter. These considerations suggest that this local case should be optimized.

**Discussion of LRPC •** There is little doubt that LRPC is more efficient than RPC for the local case, as long as enough invocations take place to offset the memory management costs.

A client stub examines a bit set at bind time that records whether the server is local or remote, and proceeds to use LRPC or RPC, respectively. The application is unaware of which is used. In later work several performance improvements, which are addressed particularly to multiprocessor operation. The improvements largely concern avoiding traps to the kernel and scheduling processors in such a way as to avoid unnecessary domain transitions. For example, if a processor is idling in the server’s memory management context at the time a client thread attempts to invoke a server procedure, then the thread should be transferred to that processor. This avoids a domain transition; at the same time, the client’s processor may be reused by another thread in the client. These enhancements involve an implementation of two-level (user and kernel) thread scheduling.

**2 Asynchronous operations:**

A common technique to defeat high latencies is asynchronous operation, which arises in two programming models: concurrent invocations and asynchronous invocations. These models are largely in the domain of middleware rather than operating system kernel design, but it is useful to consider them here, while we are examining the topic of invocation performance.

**Making invocations concurrently •** In the first model, the middleware provides only blocking invocations, but the application spawns multiple threads to perform blocking invocations concurrently. A good example of such an application is a web browser.

In the concurrent case, the first client thread marshals the arguments and calls the *Send* operation. The second thread then immediately makes the second invocation. Each thread waits to receive its results. The total time taken is liable to be lower than in the serialized case.

**Asynchronous invocations •** An *asynchronous invocation* is one that is performed asynchronously with respect to the caller. That is, it is made with a non-blocking call, which returns as soon as the invocation request message has been created and is ready for dispatch.

Sometimes the client does not require any response. An asynchronous operation returns an object called a *promise*. Eventually, when the invocation succeeds or is deemed to have failed, the Mercury system places the status and any return values in the promise. The caller uses the *claim* operation to obtain the results from the promise. The claim operation blocks until the promise is ready, whereupon it returns the results or exceptions from the call. The *ready* operation is available for testing a promise without blocking – it returns *true* or *false* according to whether the promise is ready or blocked.

**Persistent asynchronous invocations •** Traditional asynchronous invocation mechanisms such as Mercury invocations and CORBA *oneway* invocations are implemented upon TCP streams and fail if a stream breaks – that is, if the network link is down or the target host crashes.

But a more developed form of the asynchronous invocation model, which we shall call *persistent asynchronous invocation*, is becoming increasingly relevant because of disconnected operation. This model is similar to Mercury in terms of the programming operations it provides, but the difference is in its failure semantics. A conventional invocation mechanism (synchronous or asynchronous) is designed to fail after a given number of timeouts have occurred, but these short-term timeouts are often not appropriate where disconnections or very high latencies occur.

A system for persistent asynchronous invocation tries indefinitely to perform the invocation, until it is known to have succeeded or failed, or until the application cancels the invocation.

As its name suggests, QRPC queues outgoing invocation requests in a stable log while there is no network connection and schedules their dispatch over the network to servers when there is a connection. Similarly, it queues invocation results from servers in what we can consider to be the client’s invocation ‘mailbox’ until the client reconnects and collects them. Requests and results may be compressed when they are queued, before their transmission over a low-bandwidth network.

Programming with an asynchronous invocation system (persistent or otherwise) raises the issue of how users can continue using the applications on their client device while the results of invocations are still not known.

**Operating system architecture:**

An open distributed system should make it possible to:

**•** run only that system software at each computer that is necessary for it to carry out its particular role in the system architecture –

**•** allow the software (and the computer) implementing any particular service to be changed independently of other facilities;

**•** allow for alternatives of the same service to be provided, when this is required to suit different users or applications;

**•** introduce new services without harming the integrity of existing ones.

The separation of fixed resource management *mechanisms* from resource management *policies*, which vary from application to application and service to service, has been a guiding principle in operating system design for a long time

**Monolithic kernels and microkernels •** There are two key examples of kernel design: the so-called *monolithic* and *microkernel* approaches. These designs differ primarily in the decision as to what functionality belongs in the kernel and what is to be left to server processes that can be dynamically loaded to run on top of it. Although microkernels have not been deployed widely, it is instructive to understand their advantages and disadvantages compared with the typical kernels found today.

The UNIX operating system kernel has been called *monolithic*. This term is meant to suggest that it is *massive* – it performs all basic operating system functions and takes up in the order of megabytes of code and data – and that it is *undifferentiated*, i.e. it is coded in a non-modular way. The result is that to a large extent it is *intractable*: altering any individual software component to adapt it to changing requirements is difficult.

By contrast, in the case of a microkernel design the kernel provides only the most basic abstractions, principally address spaces, threads and *local* inter process communication; *all* other system services are provided by servers that are dynamically loaded at precisely those computers in the distributed system that require them. Clients access these system services using the kernel’s message-based invocation mechanisms.

The microkernel appears as a layer between the hardware layer and a layer consisting of major system components called *subsystems*. If performance is the main goal, rather than portability, then middleware may use the facilities of the microkernel directly. Otherwise, it uses a language runtime support subsystem, or a higher-level operating system interface provided by an operating system emulation subsystem*.* Each of these, in turn, is implemented by a combination of library procedures linked into applications and a set of servers running on top of the microkernel.



**Comparison •** The chief advantages of a microkernel-based operating system are its extensibility and its ability to enforce modularity behind memory protection boundaries. In addition, a relatively small kernel is more likely to be free of bugs than one that is larger and more complex.

The advantage of a monolithic design is the relative efficiency with which operations can be invoked. System calls may be more expensive than conventional procedures, but even using the techniques we examined in the previous section, an invocation to a separate user-level address space on the same node is more costly still.

**Some hybrid approaches •** Two of the original microkernels, Mach and Chorus, began their developmental life running servers only as user processes. In this configuration, modularity is hardware-enforced through address spaces. Where servers require direct access to hardware, special system calls can be provided for these privileged processes, which map device registers and buffers into their address spaces. The kernel turns interrupts into messages, which enables user-level servers to handle interrupts.

Because of performance problems, the Chorus and Mach microkernel designs eventually changed to allow servers to be loaded dynamically either into the kernel address space or into a user-level address space. In each case, clients interact with servers using the same inter process communication calls. A developer can thus debug a server at user level and then, when the development is deemed complete, allow the server to run inside the kernel’s address space in order to optimize system performance. But such a server then threatens the integrity of the system, should it turn out still to contain bugs.

Operating systems such as Nemesis exploit the fact that, even at the hardware level, an address space is not necessarily also a single protection domain. The kernel coexists in a single address space with all dynamically loaded system modules and all applications. When it loads an application, the kernel places the application’s code and data in regions chosen from those that are available at runtime.

The advent of processors with 64-bit addressing has made single-address-space operating systems particularly attractive, since they support very large address spaces that can accommodate many applications.

The kernel of a single-address-space operating system sets the protection attributes on individual regions within the address space to restrict access by user-level code. User-level code still runs with the processor in a particular protection context (determined by settings in the processor and memory management unit), **w**hich gives it full access to its own regions and only selectively shared access to others. The saving of a single address space, compared with using multiple address spaces, is that the kernel need never flush any caches when it implements a domain transition.