System Models

Three important and complementary ways in which the design of distributed systems can usefully be described and discussed:

1. Physical models
2. Architectural models
3. Fundamental models

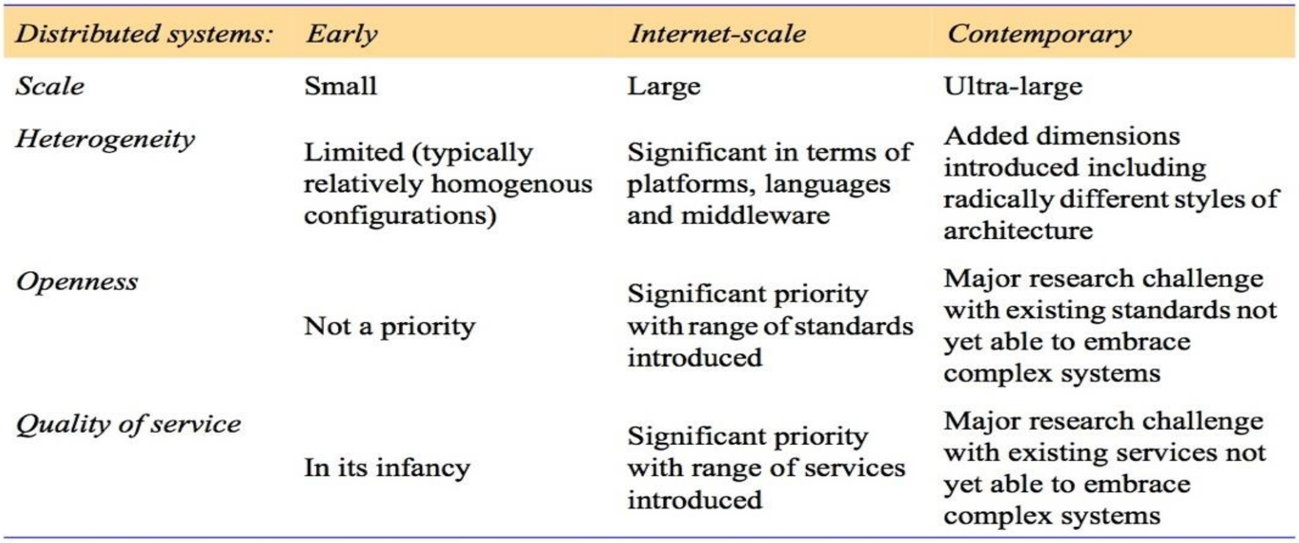
1. Physical Models

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 was defined as 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, there are **three generations of distributed systems**.



Distributed systems:	*Early*	*Internet-scale*	*Contemporary*
Scale	Small	Large	Ultra-large
Heterogeneity	Limited (typically relatively homogenous configurations)	Significant in terms of platforms, languages and middleware	Added dimensions introduced including radically different styles of architecture
Openness	Not a priority	Significant priority with range of standards introduced	Major research challenge with existing standards not yet able to embrace complex systems
Quality of service	In its infancy	Significant priority with range of services introduced	Major research challenge with existing services not yet able to embrace complex systems

Further developments in physical models:

- **Mobile computing** - led to physical models where nodes such as laptops or smart phones may move from location to location in a distributed system.
- **Ubiquitous computing** - led to a move from discrete nodes to architectures where computers are embedded in everyday objects and in the surrounding environment. **Example**: washing machines.
- **Cloud computing** - cluster architectures has led to a move from autonomous nodes performing a given role to pools of nodes that together provide a given service **Example**: search service offered by Google.

Distributed systems of systems/ ultra-large-scale (ULS) distributed systems

A system of systems (mirroring the view of the Internet as a network of networks) 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.

2. 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.

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?

Communicating entities

1.System-oriented entities
2.Problem-oriented entities

Communication paradigms

Three types of communication paradigm:

1.Interprocess communication

- refers to the low-level support for communication between processes in distributed systems

2.Remote invocation

- cover 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.

3.Indirect communication

Key techniques for indirect communication include:

a.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.

b.Publish-subscribe systems/ distributed event-based systems

- Many systems, such as the financial trading 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).

c.Message queues

- 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.

d.Tuple spaces/Generative communication

- 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.

e.Distributed shared memory

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

Two architectural styles

1.Client-server Architecture

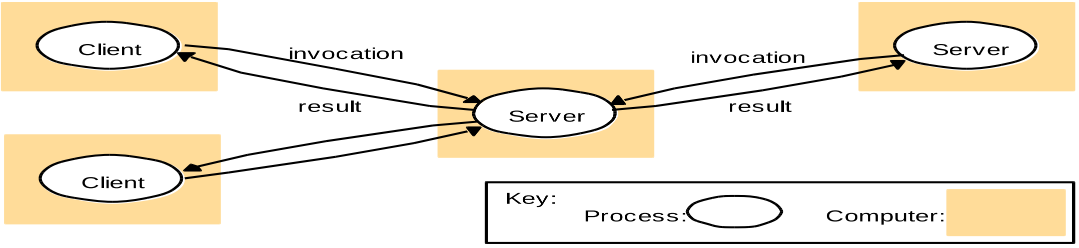
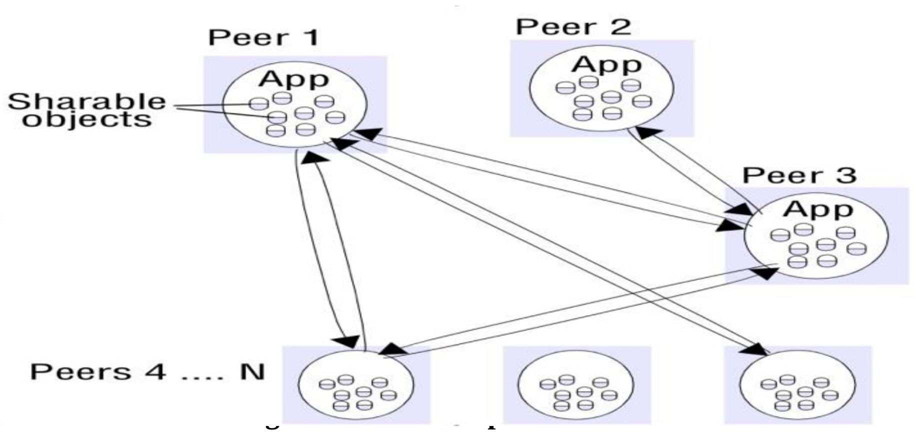


Figure 2.3 Clients invoke individual Servers

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.

2.Peer-to-peer Architecture



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.

Architectural Patterns

Architectural patterns build on the more primitive architectural elements. They are not themselves necessarily complete solutions but rather offer partial insights that, when combined with other patterns, lead the designer to a solution for a given problem domain.

1.Layering Architectural Pattern

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.

In terms of distributed systems, this equates to a **vertical organization of services** into service layers. A distributed service can be provided by one or more server processes, interacting with each other and with client processes.

Given the complexity of distributed systems, it is often helpful to organize such services into layers. A common view of a layered architecture in Figure 2.7 introduces the important terms platform and middleware, which we define as follows:

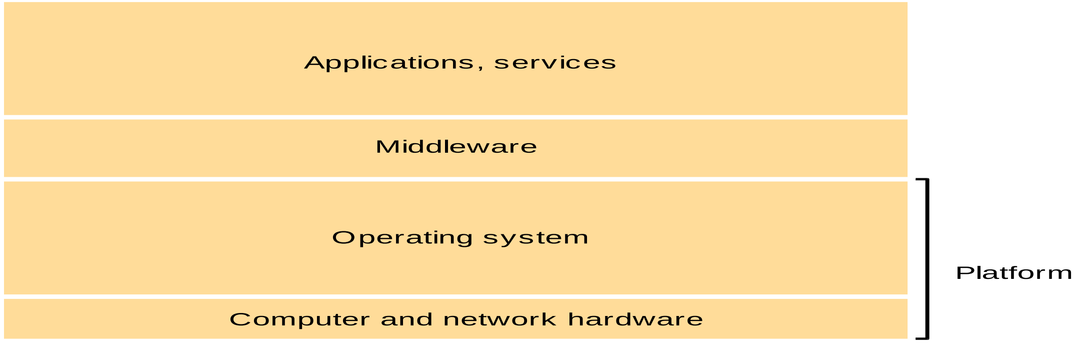


Figure 2.7 Software and hardware service layers in distributed systems

2.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.

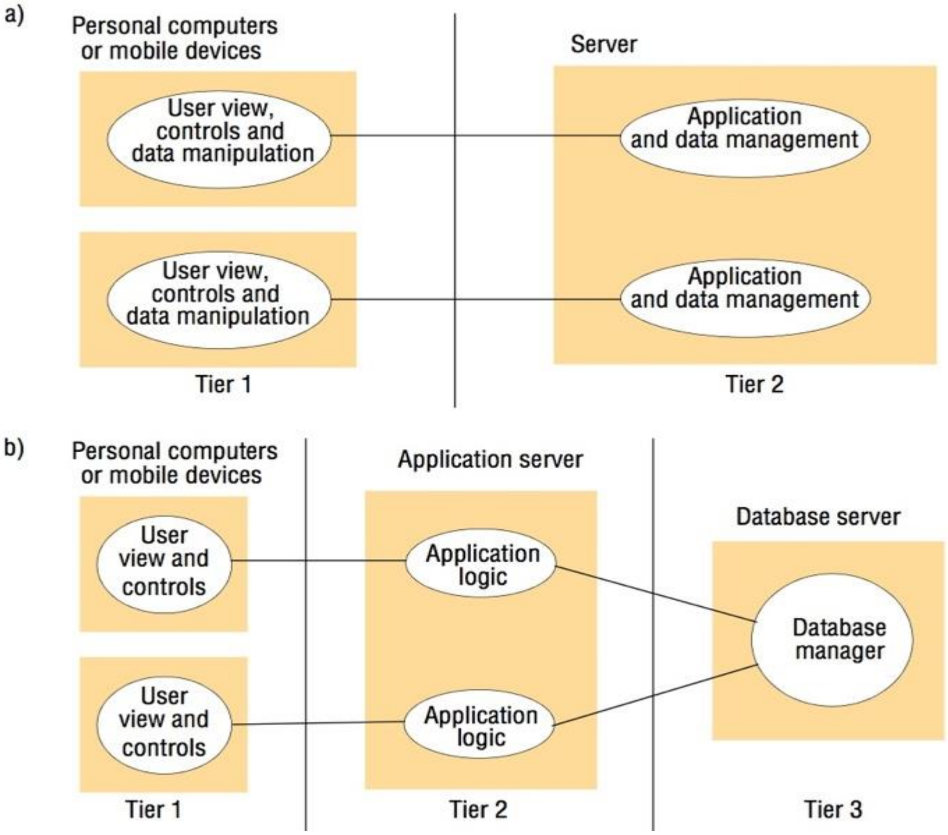
Two- and three-tiered architecture

To illustrate this, consider the functional decomposition of a given application, as follows:

Presentation logic, which is concerned with handling user interaction and updating the view of the application as presented to the user;

Application logic, which is concerned with the detailed application-specific processing associated with the application (also referred to as the **business logic**);

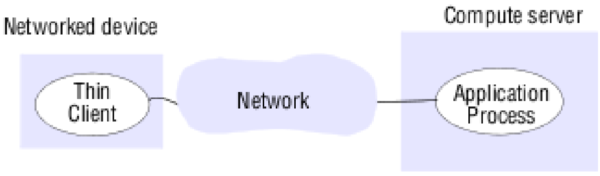
Data logic, which is concerned with the persistent storage of the application, typically in a database management system.



3.Thin clients

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.

Figure 2.10 Thin clients and computer servers



4.Other commonly occurring patterns

a.Proxy pattern

A commonly recurring pattern in distributed systems designed particularly to support location transparency in remote procedure calls or remote method invocation. With this approach, a proxy is created in the local address space to represent the remote object. This proxy offers exactly the same interface as the remote object, and the programmer makes calls on this proxy object and hence does not need to be aware of the distributed nature of the interaction.

b.Web Service Architectural Pattern

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 (a service that matches services provided to those requested), as shown in Figure 2.11.

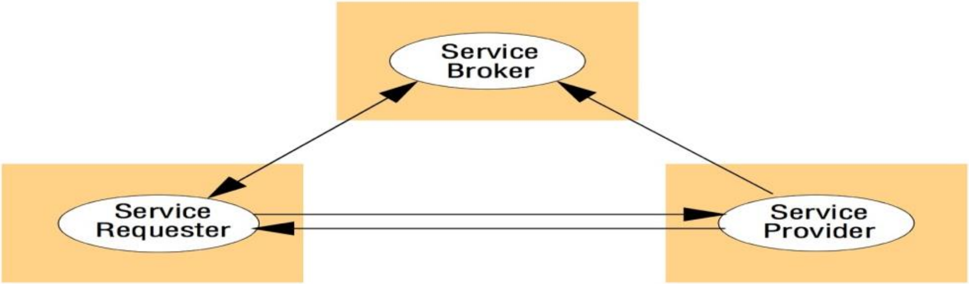


Figure 2.11 Web Service Architectural Pattern

c.Reflection

A pattern that is increasingly 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 behaviour).

3. Fundamental Models

1.Interaction Models

Computation occurs within processes; the processes interact by passing messages, resulting in communication (information flow) and coordination (synchronization and ordering of activities) between processes. Distributed systems are composed of many processes, interacting in complex ways.

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

Two variants of the interaction model

a.Synchronous distributed systems
b.Asynchronous distributed systems

a.Synchronous distributed systems

Hadzilacos and Toueg [1994] define a 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.

b.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 picosecond 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.

2.Failure Model

The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures. Hadzilacos and Toueg [1994] provide a taxonomy that distinguishes between the failures of processes and communication channels. These are presented under the headings:

a) Omission failures
b) Arbitrary failures
c) Timing failures
d) Masking failures

a)Omission failures

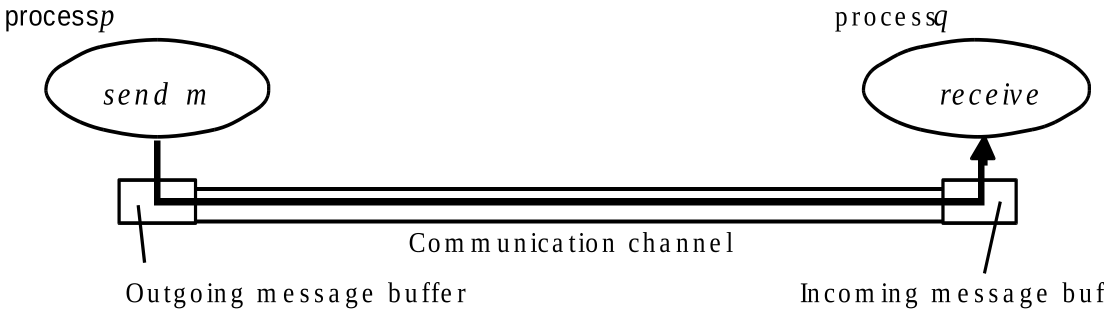
The faults classified as omission failures refer to cases when a process or communication channel fails to perform actions that it is supposed to do.

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.

Communication omission failures

Consider the communication primitives send and receive. A process p performs a send by inserting the message m in its outgoing message buffer. The communication channel transports m to q's incoming message buffer. Process q performs a receive by taking m from its incoming message buffer and delivering it (see Figure 2.14). The outgoing and incoming message buffers are typically provided by the operating system.



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.

Hadzilacos and Toueg [1994] refer to the loss of messages between the sending process and the outgoing message buffer as **send-omission failures**, to loss of messages between the incoming message buffer and the receiving process as **receive-omission failures**, and to loss of messages in between as **channel omission failures**.

b)Arbitrary failures/ Byzantine failure

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. An arbitrary failure of a process is one in which it arbitrarily omits intended processing steps or takes unintended processing steps.

c)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. Any one of these failures may result in responses being unavailable to clients within a specified time interval.

In an **asynchronous distributed system**, an overloaded server may respond too slowly, but we cannot say that it has a timing failure since no guarantee has been offered.

d)Masking failures

A service masks a failure either by hiding it altogether or by converting it into a more acceptable type of failure. For an example of the latter, checksums are used to mask corrupted messages, effectively converting an arbitrary failure into an omission failure. Masking can be done by means of replication.

3.Security model

The security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.

Protecting objects

Figure 2.17 Objects and principals

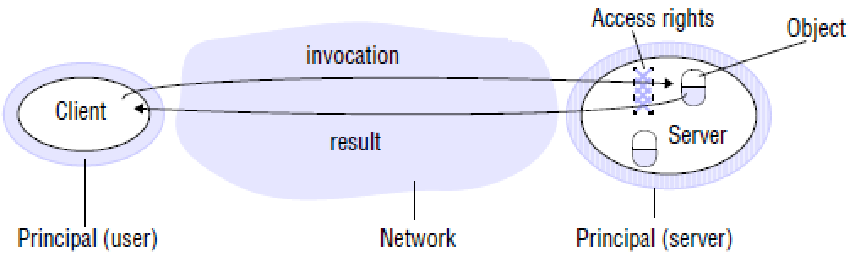


Figure 2.17 shows a server that manages a collection of objects on behalf of some users. The users can run client programs that send invocations to the server to perform operations on the objects. The server carries out the operation specified in each invocation and sends the result to the client.

Objects are intended to be used in different ways by different users. For example, some objects may hold a user's private data, such as their mailbox, and other objects may hold shared data such as web pages. To support this, **access rights** specify who is allowed to perform the operations of an object – for example, who is allowed to read or to write its state.

The users are the beneficiaries of access rights. We do so by associating with each invocation and each result the authority on which it is issued. Such an authority is called a **principal**. A principal may be a user or a process.

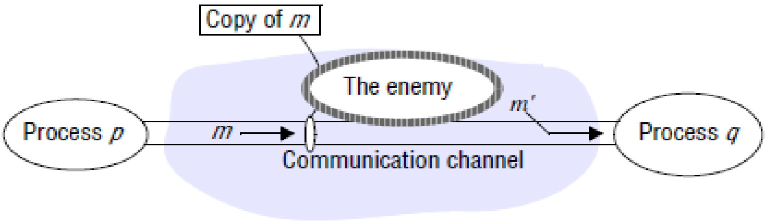
Securing processes and their interactions

Processes interact by sending messages. The messages are exposed to attack because the network and the communication service that they use are open.

Distributed systems are often deployed and used in tasks that are likely to be subject to external attacks by hostile users. This is especially true for applications that handle financial transactions, confidential or classified information or any other information whose secrecy or integrity is crucial. The following discussion introduces a model for the analysis of security threats.

The enemy/The Adversary

Figure 2.18 The enemy



To model security threats, we postulate an enemy that is capable of sending any message to any process and reading or copying any message sent between a pair of processes, as shown in Figure 2.18. Such attacks can be made simply by using a computer connected to a network to run a program that reads network messages addressed to other computers on the network, or a program that generates messages that make false requests to services, purporting to come from authorized users.

The threats from a potential enemy include

a)Threats to processes

b)Threats to communication channels

Defeating security threats

a)Cryptography and shared secrets

Suppose that a pair of processes (for example, a particular client and a particular server) share a secret; that is, they both know the secret but no other process in the distributed system knows it.

Cryptography is the science of keeping messages secure, and encryption is the process of scrambling a message in such a way as to hide its contents.

b)Authentication

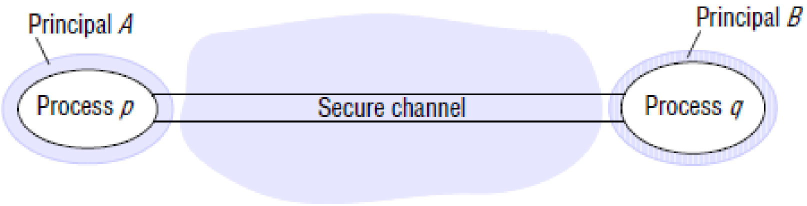
Authentication of messages means proving the identities supplied by their senders. The basic authentication technique is to include in a message an encrypted portion that contains enough of the contents of the message to guarantee its authenticity.

Secure channels

A secure channel is a communication channel connecting a pair of processes, each of which acts on behalf of a principal, as shown in Figure 2.19. A secure channel has the following properties:

- Each of the processes knows reliably the identity of the principal on whose behalf the other process is executing.
- A secure channel ensures the privacy and integrity (protection against tampering) of the data transmitted across it.
- Each message includes a physical or logical timestamp to prevent messages from being replayed or reordered.

Figure 2.19 Secure channels



c)Other possible threats from an enemy

Denial of service

This is a form of attack in which the enemy interferes with the activities of authorized users by making excessive and pointless invocations on services or message transmissions in a network, resulting in overloading of physical resources (network bandwidth, server processing capacity). Such attacks are usually made with the intention of delaying or preventing actions by other users.

Mobile code

Mobile code receives and executes program code from elsewhere, such as the email attachment. Such code may easily play a Trojan horse role, purporting to fulfil an innocent purpose but in fact including code that accesses or modifies resources that are legitimately available to the host process but not to the originator of the code.

4. Interprocess Communication

Interprocess communication is concerned with the characteristics of protocols for communication between processes in a distributed system. Interprocess communication in the Internet provides both datagram and stream communication.

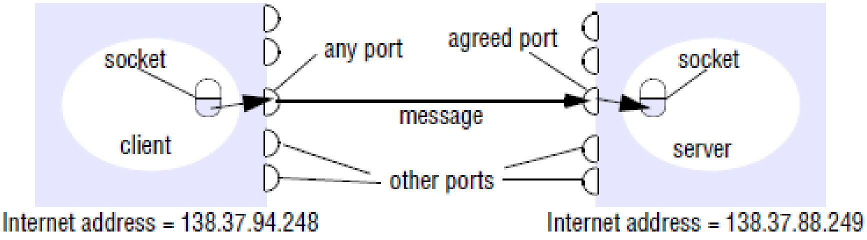
4.2 The API for the Internet protocols
4.3 External data representation and marshalling
4.3 Multicast communication
4.5 Network virtualization: Overlay networks
4.6 Case study: MPI

The API for the Internet protocols

Sockets

Both forms of communication (UDP and TCP) use the socket abstraction, which provides an endpoint for communication between processes. Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process, as illustrated in Figure 4.2. For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs.

Figure 4.2 Sockets and ports


socket
client
any port
agreed port
message
other ports
socket
server
Internet address = 138.37.94.248
Internet address = 138.37.88.249

UDP datagram communication

A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries. If a failure occurs, the message may not arrive.

Failure model for UDP datagrams

UDP datagrams suffer from the following failures:

- **Omission failures**: Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination.
- **Ordering**: Messages can sometimes be delivered out of sender order.

Java API for UDP datagrams

The Java API provides datagram communication by means of two classes: DatagramPacket and DatagramSocket.

- **DatagramPacket**: This class provides a constructor that makes an instance out of an array of bytes comprising a message, the length of the message and the Internet address and local port number of the destination socket, as follows:
- **DatagramSocket**: This class supports sockets for sending and receiving UDP datagrams. It provides a constructor that takes a port number as its argument, for use by processes that need to use a particular port.

TCP stream communication

Failure model

- To satisfy the **integrity** property of reliable communication,
  - TCP streams use **checksums** to detect and reject corrupt packets and **sequence numbers** to detect and reject duplicate packets.
- For the sake of the **validity** property,
  - TCP streams use **timeouts and retransmissions** to deal with lost packets.

Java API for TCP streams

The Java interface to TCP streams is provided in the classes ServerSocket and Socket:

- **ServerSocket**: This class is intended for use by a server to create a socket at a server port for listening for connect requests from clients. Its accept method gets a connect request from the queue or, if the queue is empty, blocks until one arrives.
- **Socket**: This class is for use by a pair of processes with a connection. The client uses a constructor to create a socket, specifying the DNS hostname and port of a server. It can throw an UnknownHostException if the hostname is wrong or an IOException if an IO error occurs.

External data representation

The information stored in running programs is represented as data structures, whereas the information in messages consists of sequences of bytes. Irrespective of the form of communication used, the data structure must be **flattened** (converted to a sequence of bytes) before transmission and rebuilt on arrival.

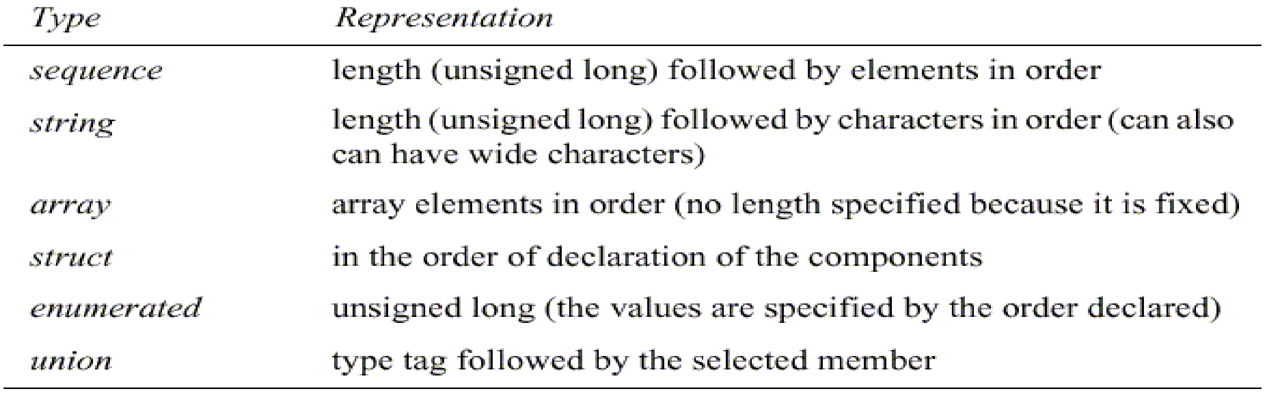
An agreed standard for the representation of data structures and primitive values is called an **external data representation**.

Three alternative approaches to external data representation and marshalling are discussed:

1. CORBA's common data representation
2. Java's object serialization
3. XML (Extensible Markup Language)

1.CORBA's Common Data Representation (CDR)

- CORBA CDR is the external data representation defined with CORBA 2.0.
- It consists 15 **primitive types**:
    - Short (16 bit)
    - Long (32 bit)
    - Unsigned short
    - Unsigned long
    - Float(32 bit)
    - Double(64 bit)
    - Char
    - Boolean(TRUE,FALSE)
    - Octet(8 bit)
    - Any(can represent any basic or constructed type)
- **Constructed types** are shown below:



Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

2.Java's object serialization

- In Java RMI, both object and primitive data values may be passed as arguments and results of method invocation.
- An object is an instance of a Java class.
  Example, the Java class equivalent to the Person struct
    - Public class Person implements Serializable {
        - Private String name;
        - Private String place;
        - Private int year;
        - Public Person(String aName ,String aPlace, int aYear) {
            - name = aName;
            - place = aPlace;
            - year = aYear;
    - }}
    - //followed by methods for accessing the instance variables
- In Java, the term **serialization** refers to the activity of flattening an object or a connected set of objects into a serial form that is suitable for storing on disk or transmitting in a message, for example, as an argument or the result of an RMI. **Deserialization** consists of restoring the state of an object or a set of objects from their

serialized form.

3.XML (Extensible Markup Language)

- XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web.
- XML documents, being textual, can be read by humans. In practice, most XML documents are generated and read by XML processing software, but the ability to read XML can be useful when things go wrong.

XML elements and attributes

- Figure 4.10 shows the XML definition of the Person structure. It shows that XML consists of tags and character data. The character data, for example Smith or 1984, is the actual data. As in HTML, the structure of an XML document is defined by pairs of tags enclosed in angle brackets.

Elements

An element in XML consists of a portion of character data surrounded by matching start and end tags.

Attributes

A start tag may optionally include pairs of associated attribute names and values such as id="123456789", as shown above. The syntax is the same as for HTML, in which an attribute name is followed by an equal sign and an attribute value in quotes. Multiple attribute values are separated by spaces.

XML namespaces

An XML namespace is a set of names for a collection of element types and attributes that is referenced by a URL. Any element that makes use of an XML namespace can specify that namespace as an attribute called xmlns, whose value is a URL referring to the file containing the namespace definitions.

XML schemas

An XML schema defines the elements and attributes that can appear in a document, how the elements are nested and the order and number of elements, and whether an element is empty or can include text. The sender of a SOAP message may use an XML schema to encode it, and the recipient will use the same XML schema to validate and decode it.

Document type definitions

Document type definitions (DTDs) were provided as a part of the XML 1.0 specification for defining the structure of XML documents and are still widely used for that purpose. It cannot describe data types and its definitions are global, preventing element names from being duplicated.

Multicast communication

A multicast operation is more appropriate – this is an operation that sends a single message from one process to each of the members of a group of processes, usually in such a way that the membership of the group is transparent to the sender.

IP multicast

IP multicast is built on top of the Internet Protocol (IP). IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group. The sender is unaware of the identities of the individual recipients and of the size of the group. Being a member of a multicast group allows a computer to receive IP packets sent to the group. The membership of multicast groups is dynamic, allowing computers to join or leave at any time and to join an arbitrary number of groups.

Reliability and ordering of multicast

When a multicast on a local area network uses the multicasting capabilities of the network to allow a single datagram to arrive at multiple recipients, any one of those recipients may drop the message because its buffer is full.

Another factor is that any process may fail. If a multicast router fails, the group members beyond that router will not receive the multicast message, although local members may do so.

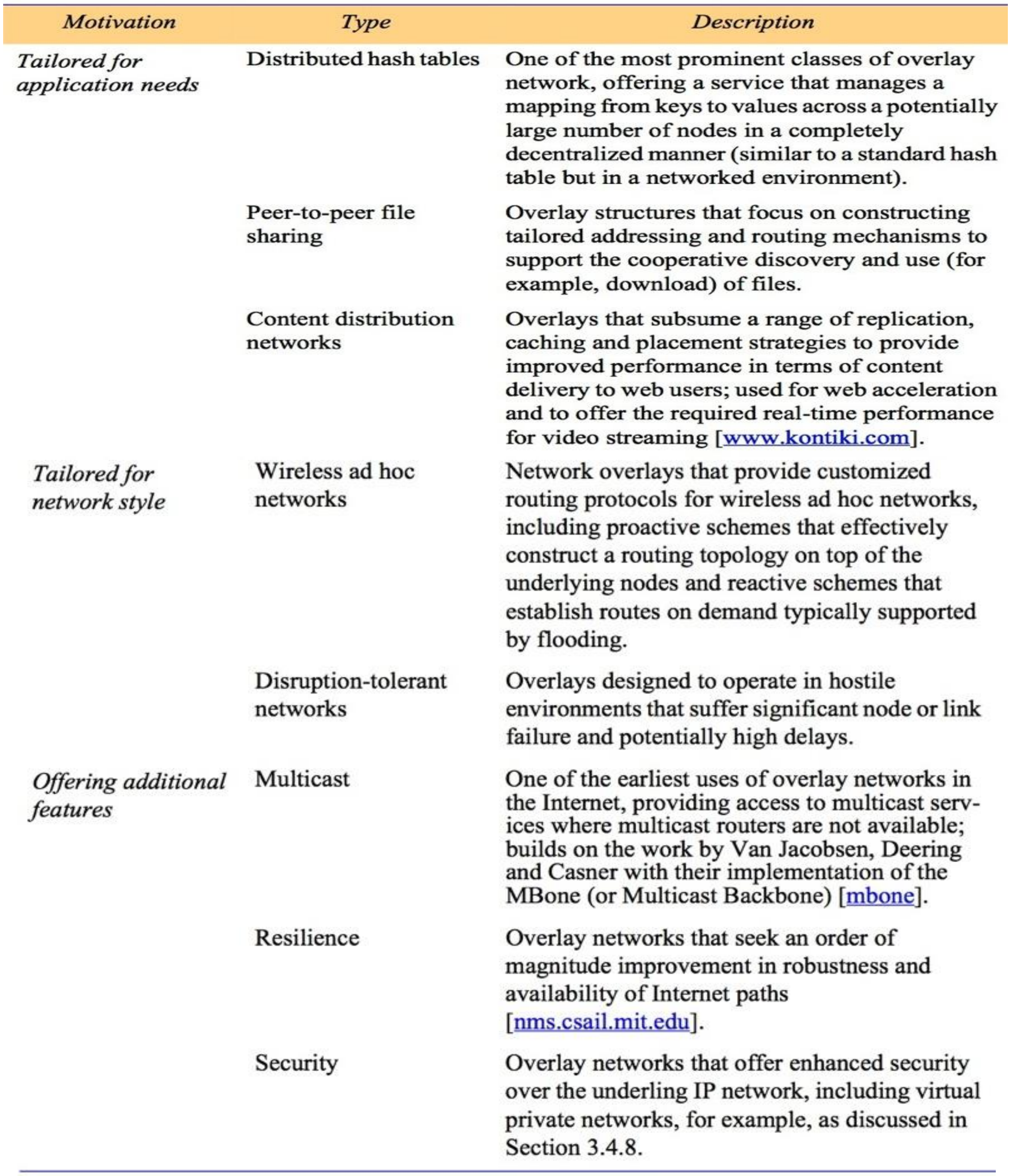
Ordering is another issue. IP packets sent over an internetwork do not necessarily arrive in the order in which they were sent, with the possible effect that some group members receive datagrams from a single sender in a different order from other group members. In addition, messages sent by two different processes will not necessarily arrive in the same order at all the members of the group.

Overlay networks

An overlay network is a virtual network consisting of nodes and virtual links, which sits on top of an underlying network (such as an IP network) and offers something that is not otherwise provided:

- a service that is tailored towards the needs of a class of application or a particular higher-level service – for example, multimedia content distribution;
- more efficient operation in a given networked environment – for example routing in an ad hoc network;
- an additional feature – for example, multicast or secure communication.

Types of Overlays



Motivation	*Type*	*Description*
Tailored for application needs	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).
	Peer-to-peer file sharing	Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.
	Content distribution networks	Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [www.kontiki.com].
Tailored for network style	Wireless ad hoc networks	Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.
	Disruption-tolerant networks	Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.
Offering additional features	Multicast	One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobsen, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [mbone].
	Resilience	Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [nms.csail.mit.edu].
	Security	Overlay networks that offer enhanced security over the underling IP network, including virtual private networks, for example, as discussed in Section 3.4.8.

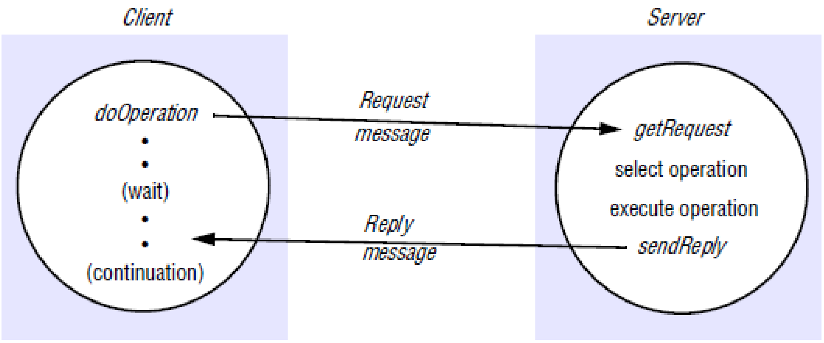
Request-reply protocols

Request-reply communication is synchronous because the client process blocks until the reply arrives from the server. It can also be reliable because the reply from the server is effectively an acknowledgement to the client. Asynchronous request-reply communication is an alternative that may be useful in situations where clients can afford to retrieve replies later.

Request-reply protocol

The protocol is based on a trio of **communication primitives**, doOperation, getRequest and sendReply, as shown in Figure 5.2. This request-reply protocol matches requests to replies. It may be designed to provide certain delivery guarantees.

Figure 5.2 Request-reply communication



- **doOperation** - used by clients to invoke remote operations. Its arguments specify the remote server and which operation to invoke, together with additional information (arguments) required by the operation. Its result is a byte array containing the reply.
- **getRequest** - used by a server process to acquire service requests
- **sendReply** - When the server has invoked the specified operation, it then uses sendReply to send the reply message to the client. When the reply message is received by the client the original doOperation is unblocked and execution of the client program continues.

  The information to be transmitted in a request/reply message is shown in Figure 5.4.
- **First field** - whether the message is a Request or a Reply message.
- **Second field** - requestId, contains a message identifier. A doOperation in the client generates a requestId for each request message, and the server copies these IDs into the corresponding reply messages. This enables doOperation to check that a reply message is the result of the current request, not a delayed earlier call.
- **Third field** - a remote reference.
- **Fourth field** - identifier for the operation to be invoked.

Remote procedure call

The remote procedure call (RPC) is a procedure call to distributed environments, allowing a calling process to call a procedure in a remote node as if it is local.

Design issues for RPC

1. Programming with interfaces

The interface of a module specifies the procedures and the variables that can be accessed from other modules. So long as its interface remains the same, the implementation may be changed without affecting the users of the module.

Interfaces in distributed systems

In the client-server model, each server provides a set of procedures that are available for use by clients. The term **service interface** is used to refer to the specification of the procedures offered by a server, defining the types of the arguments of each of the procedures.

2. RPC call semantics

Fault Tolerance Measures

- **Retry request message**: Controls whether to retransmit the request message until either a reply is received or the server is assumed to have failed.
- **Duplicate filtering**: Controls when retransmissions are used and whether to filter out duplicate requests at the server.
- **Retransmission of results**: Controls whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.

RPC invocation semantics

- **Maybe semantics**: With maybe semantics, the remote procedure call may be executed once or not at all.
- **At-least-once semantics**: With at-least-once semantics, the invoker receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception informing it that no result was received.
- **At-most-once semantics**: With at-most-once semantics, the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received, in which case the procedure will have been executed either once or not at all.

3. Transparency

The originators of RPC, aimed to make remote procedure calls as much like local procedure calls as possible, with no distinction in syntax between a local and a remote procedure call. All the necessary calls to marshalling and message-passing procedures were hidden from the programmer making the call.

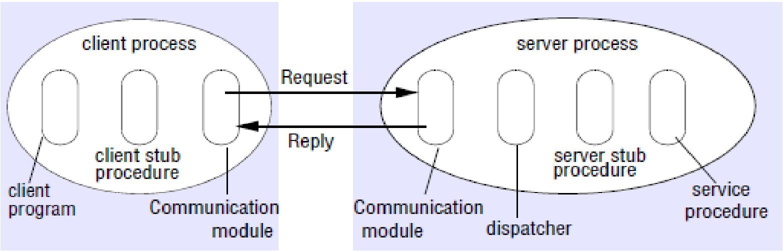
RPC strives to offer at least location and access transparency, hiding the physical location of the (potentially remote) procedure and also accessing local and remote procedures in the same way. Middleware can also offer additional levels of transparency to RPC.

Implementation of RPC

The software components required to implement RPC are shown in Figure 5.10. The client that accesses a service includes one stub procedure for each procedure in the service interface. The stub procedure behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifier and the arguments into a request message, which it sends via its communication module to the server. When the reply message arrives, it unmarshals the results.

The server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in the service interface. The dispatcher selects one of the server stub procedures according to the procedure identifier in the request message. The server stub procedure then unmarshals the arguments in the request message, calls the corresponding service procedure and marshals the return values for the reply message. The service procedures implement the procedures in the service interface. The client and server stub procedures and the dispatcher can be generated automatically by an interface compiler from the interface definition of the service.

Figure 5.10 Role of client and server stub procedures in RPC



Remote method invocation

Remote method invocation (RMI) is similar to RPC but for distributed objects, with added benefits in terms of using object-oriented programming concepts in distributed systems and also extending the concept of an object reference to the global distributed environments, and allowing the use of object references as parameters in remote invocations.

Design issues for RMI

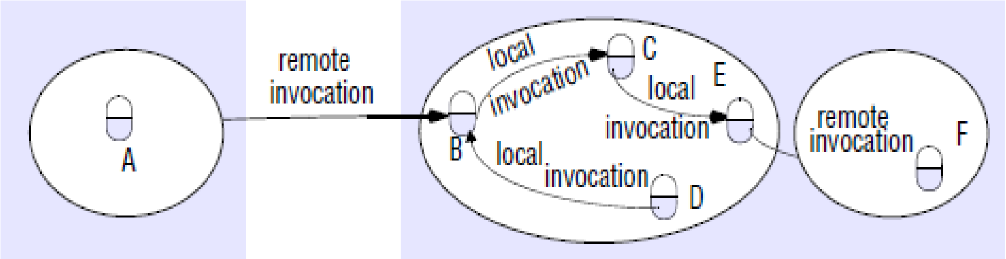
Distributed objects

Distributed object systems may adopt the client-server architecture. In this case, objects are managed by servers and their clients invoke their methods using remote method invocation. The invocation is carried out in RMI by executing a method of the object at the server and the result is returned to the client in another message.

The distributed object model

Each process contains a collection of objects, some of which can receive both local and remote invocations, whereas the other objects can receive only local invocations. Method invocations between objects in different processes, whether in the same computer or not, are known as remote method invocations. Method invocations between objects in the same process are local method invocations. We refer to objects that can receive remote invocations as remote objects. Objects B and F are remote objects.

Figure 5.12 Remote and local method invocations



Remote object references: A remote object reference is an identifier that can be used throughout a distributed system to refer to a particular unique remote object.

Remote interfaces: Objects in other processes can invoke only the methods that belong to its remote interface. Local objects can invoke the methods in the remote interface as well as other methods implemented by a remote object. The CORBA system provides an interface definition language (IDL), which is used for defining remote interfaces. In Java RMI, remote interfaces are defined in the same way as any other Java interface. They acquire their ability to be remote interfaces by extending an interface named Remote. Both CORBA IDL and Java support multiple inheritance of interfaces.

Actions in a distributed object system

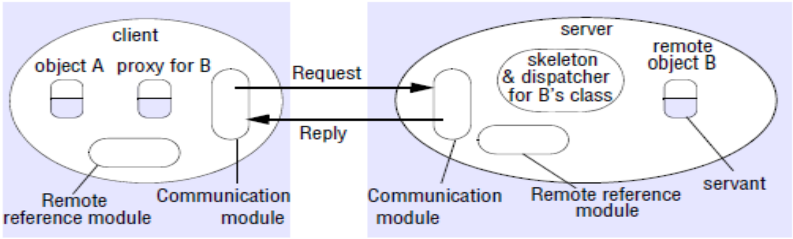
An action is initiated by a method invocation, which may result in further invocations on methods in other objects.

Garbage collection in a distributed-object system: Distributed garbage collection is generally achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection, usually based on reference counting. If garbage collection is not available, then remote objects that are no longer required should be deleted.

Exceptions: Any remote invocation may fail for reasons related to the invoked object being in a different process or computer from the invoker. Therefore, remote method invocation should be able to raise exceptions such as timeouts that are due to distribution as well as those raised during the execution of the method invoked.

Implementation of RMI

Figure 5.15 The role of proxy and skeleton in remote method invocation



Communication module

The two cooperating communication modules carry out the request-reply protocol, which transmits request and reply messages between the client and server. The communication module in the server selects the dispatcher for the class of the object to be invoked, passing on its local reference, which it gets from the remote reference module in return for the remote object identifier in the request message.

Remote reference module

The remote reference module in each process has a remote object table that records the correspondence between local object references in that process and remote object references (which are system-wide).

Servants

A servant is an instance of a class that provides the body of a remote object. It is the servant that eventually handles the remote requests passed on by the corresponding skeleton. Servants live within a server process. They are created when remote objects are instantiated and remain in use until they are no longer needed, finally being garbage collected or deleted.

The RMI software

This consists of a layer of software between the application-level objects and the communication and remote reference modules. The roles of the middleware objects shown in Figure 5.15 are as follows:

- **Proxy**: The role of a proxy is to make remote method invocation transparent to clients by behaving like a local object to the invoker; but instead of executing an invocation, it forwards it in a message to a remote object. It hides the details of the remote object reference, the marshalling of arguments, unmarshalling of results and sending and receiving of messages from the client. There is one proxy for each remote object.
- **Dispatcher**: A server has one dispatcher and one skeleton for each class representing a remote object. In our example, the server has a dispatcher and a skeleton for the class of remote object B. The dispatcher receives request messages from the communication module. It uses the operationId to select the appropriate method in the skeleton, passing on the request message.
- **Skeleton**: A skeleton method unmarshals the arguments in the request message and invokes the corresponding method in the servant. It waits for the invocation to complete and then marshals the result, together with any exceptions, in a reply message to the sending proxy’s method.

Group communication

Group communication offers a service whereby a message is sent to a group and then this message is delivered to all members of the group. In this action, the sender is not aware of the identities of the receivers.

Implementation issues

a.Reliability and ordering in multicast

In group communication, all members of a group must receive copies of the messages sent to the group, generally with delivery guarantees.

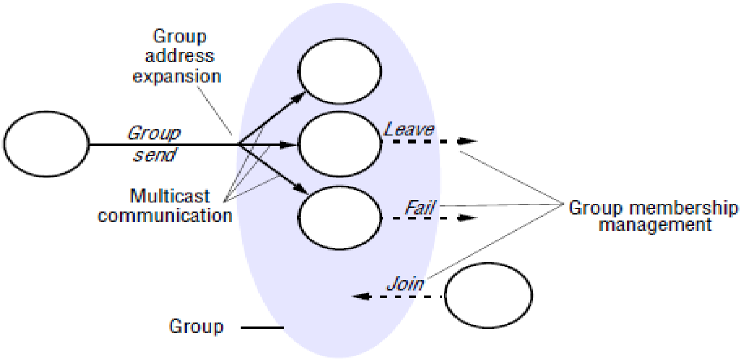
Reliability in one-to-one communication in terms of two properties: integrity (the message received is the same as the one sent, and no messages are delivered twice) and validity (any outgoing message is eventually delivered). To extend the semantics to cover delivery to multiple receivers, a third property is added – that of agreement, stating that if the message is delivered to one process, then it is delivered to all processes in the group.

Ordering is not guaranteed by underlying interprocess communication primitives.

b.Group membership management

The key elements of group communication management are summarized in Figure 6.3, which shows an open group.

Figure 6.3 The role of group membership management



A group membership service has four main tasks:

Providing an interface for group membership changes: The membership service provides operations to create and destroy process groups and to add or withdraw a process to or from a group.

Failure detection: The service monitors the group members not only in case they should crash, but also in case they should become unreachable because of a communication failure.

Notifying members of group membership changes: The service notifies the group's members when a process is added, or when a process is excluded.

Performing group address expansion: When a process multicasts a message, it supplies the group identifier rather than a list of processes in the group. The membership management service expands the identifier into the current group membership for delivery. The service can coordinate multicast delivery with membership changes by controlling address expansion. That is, it can decide consistently where to deliver any given message, even though the membership may be changing during delivery.

Publish-subscribe systems

A publish-subscribe system is a system where publishers publish structured events to an event service and subscribers express interest in particular events through subscriptions which can be arbitrary patterns over the structured events.

Characteristics of publish-subscribe systems

Heterogeneity

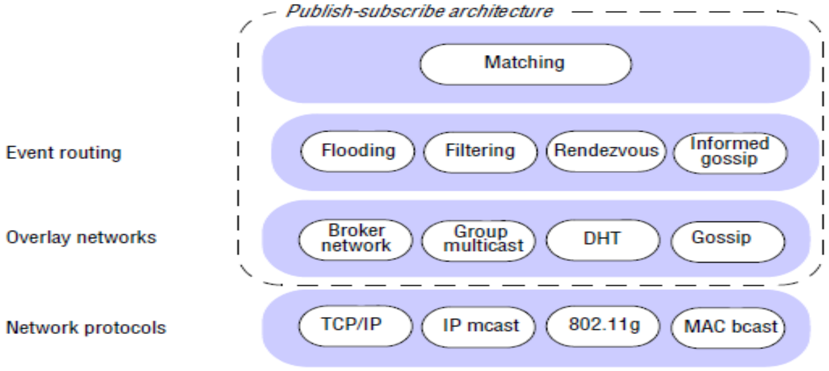
When event notifications are used as a means of communication, components in a distributed system that were not designed to interoperate can be made to work together. All that is required is that event-generating objects publish the types of events they offer, and provide an interface for receiving and dealing with the resultant notifications.

Asynchronicity

Notifications are sent asynchronously by event-generating publishers to all the subscribers that have expressed an interest in them.

Overall systems architecture

Figure 6.10 The architecture of publish-subscribe systems



In the bottom layer, publish-subscribe systems make use of a range of interprocess communication services, such as TCP/IP, IP multicast (where available) or more specialized services, as offered for example by wireless networks. The heart of the architecture is provided by the event routing layer supported by a network overlay infrastructure. Event routing performs the task of ensuring that event notifications are routed as efficiently as possible to appropriate subscribers, whereas the overlay infrastructure supports this by setting up appropriate networks of brokers or peer-to-peer structures. The top layer implements matching – that is, ensuring that events match a given subscription.

Message queues

Whereas groups and publish-subscribe provide a one-to-many style of communication, message queues provide a point-to-point service using the concept of a message queue as an indirection, thus achieving the desired properties of space and time uncoupling.

The programming model

It offers an approach to communication in distributed systems through queues. In particular, producer processes can send messages to a specific queue and other (consumer) processes can then receive messages from this queue. Three styles of receive are generally supported:

- a **blocking receive**, which will block until an appropriate message is available;
- a **non-blocking receive** (a polling operation), which will check the status of the queue and return a message if available, or a not available indication otherwise;
- a **notify operation**, which will issue an event notification when a message is available in the associated queue.

A number of processes can send messages to the same queue, and likewise a number of receivers can remove messages from a queue. The queuing policy is normally first-in-first-out (FIFO), but most message queue implementations also support the concept of priority, with higher-priority messages delivered first. Consumer processes can also select messages from the queue based on properties of a message.

A message consists of a destination (that is, a unique identifier designating the destination queue), metadata associated with the message, including fields such as the priority of the message and the delivery mode, and also the body of the message.

One crucial property of message queue systems is that messages are persistent – that is, message queues will store the messages indefinitely (until they are consumed) and will also commit the messages to disk to enable reliable delivery.

Implementation issues

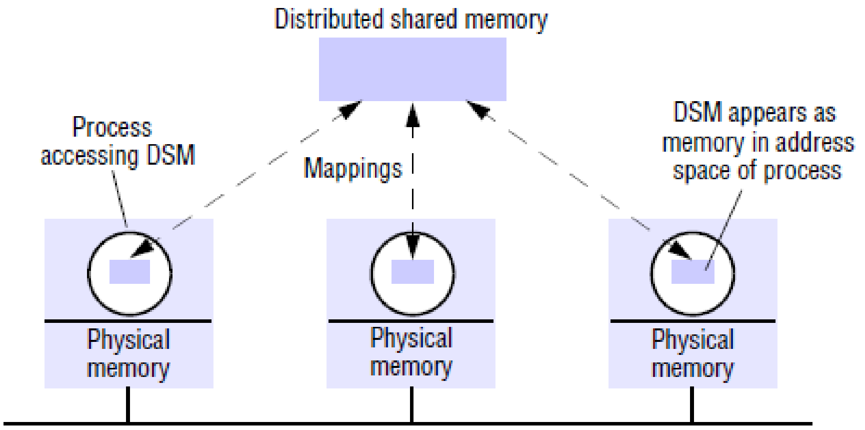
The key implementation issue the choice between centralized and distributed implementations. Some implementations are centralized, with one or more message queues managed by a queue manager located at a given node. The advantage is simplicity, but such managers have the potential to become a bottleneck. As a result, more distributed implementations have been proposed as adopted in WebSphere MQ.

Shared memory approaches

Distributed shared memory

Distributed shared memory (DSM) is used for sharing data between computers that do not share physical memory. Processes access DSM by reads and updates to what appears to be ordinary memory within their address space.

Figure 6.19 The distributed shared memory abstraction


Distributed shared memory
Process accessing DSM
Mappings
DSM appears as memory in address space of process
Physical memory
Physical memory
Physical memory

DSM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly. Message passing cannot be avoided altogether in a distributed system: in the absence of physically shared memory, the DSM runtime support has to send updates in messages between computers. DSM systems manage replicated data: each computer has a local copy of recently accessed data items stored in DSM, for speed of access.

Tuple space communication

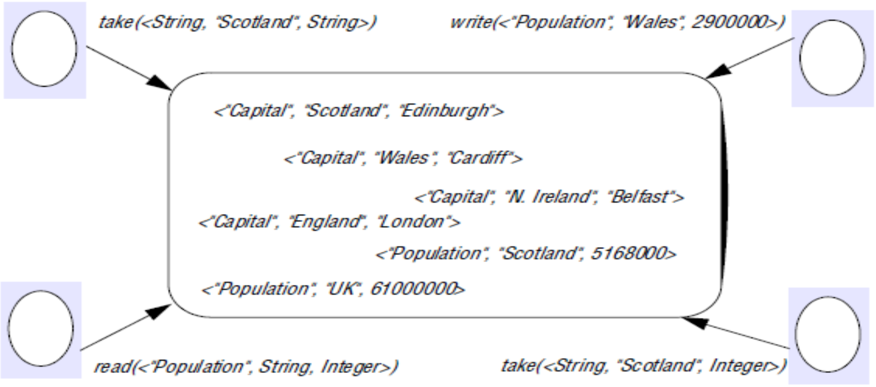
In this approach, processes communicate indirectly by placing tuples in a tuple space, from which other processes can read or remove them. Tuples do not have an address but are accessed by pattern matching on content.

The programming model

In the tuple space programming model, processes communicate through a tuple space – a shared collection of tuples. Tuples in turn consist of a sequence of one or more typed data fields such as <"fred", 1958>, <"sid", 1964> and <4, 9.8, "Yes">. Any combination of types of tuples may exist in the same tuple space.

Illustration of the tuple space paradigm

Figure 6.20 The tuple space abstraction


take(<String, "Scotland", String>)
write(<"Population", "Wales", 2900000>)
<"Capital", "Scotland", "Edinburgh">
<"Capital", "Wales", "Cardiff">
<"Capital", "N. Ireland", "Belfast">
<"Capital", "England", "London">
<"Population", "Scotland", 5168000>
<"Population", "UK", 61000000>
read(<"Population", String, Integer>)
take(<String, "Scotland", Integer>)

Properties associated with tuple spaces

- **Space uncoupling**: A tuple placed in tuple space may originate from any number of sender processes and may be delivered to any one of a number of potential recipients. This property is also referred to as distributed naming in Linda.
- **Time uncoupling**: A tuple placed in tuple space will remain in that tuple space until removed (potentially indefinitely), and hence the sender and receiver do not need to overlap in time.

Implementation issues

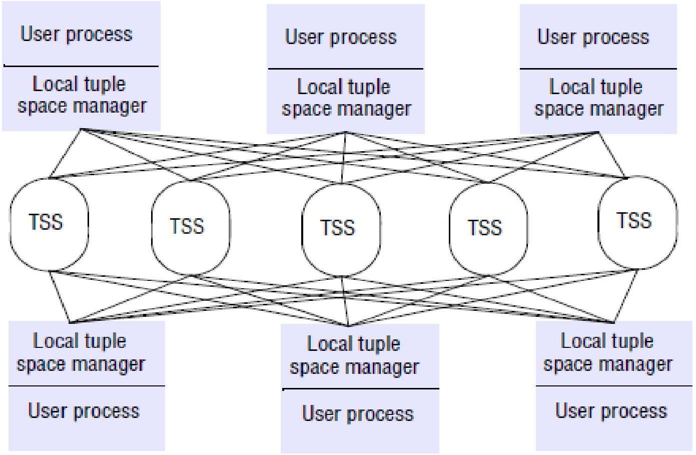
Many of the implementations of tuple spaces adopt a centralized solution where the tuple space resource is managed by a single server. This has advantages in terms of simplicity, but such solutions are clearly not fault tolerant and also will not scale. Because of this, distributed solutions have been proposed.

Replication: Several systems have proposed the use of replication to overcome the problems identified above.

The **state machine approach** assumes that a tuple space behaves like a state machine, maintaining state and changing this state in response to events received from other replicas or from the environment. To ensure consistency the replicas (i) must start in the same state (an empty tuple space), (ii) must execute events in the same order and (iii) must react deterministically to each event.

The **Linda Kernel** developed at the University of York adopts an approach in which tuples are partitioned across a range of available tuple space servers (TSSs). There is no replication of tuples; that is, there is only one copy of each tuple. The motivation is to increase performance of the tuple space, especially for highly parallel computation. When a tuple is placed in tuple space, a hashing algorithm is used to select one of the tuple space servers to be used.

Figure 6.22 Partitioning in the York Linda Kernel


User process
Local tuple space manager
User process
Local tuple space manager
User process
Local tuple space manager
TSS
TSS
TSS
TSS
TSS
Local tuple space manager
User process
Local tuple space manager
User process
Local tuple space manager
User process

Distributed objects (Refer RMI Objects also)

Distributed object middleware offers a programming abstraction based on object oriented principles. Leading examples of distributed object middleware include Java RMI and CORBA. While Java RMI and CORBA share a lot in common, there is one important difference: the use of Java RMI is restricted to Java-based development, whereas CORBA is a multi-language solution allowing objects written in a variety of languages to interoperate.

Key differences between objects and distributed objects

- **Class** is a fundamental concept in object-oriented languages but does not feature so prominently in distributed object middleware.
- Distributed object middleware offers **interface inheritance**, which is a relationship between interfaces whereby the new interface inherits the method signatures of the original interface and can add extra ones.

Added complexities

- **Inter-object communication**: A distributed object middleware framework must offer one or more mechanisms for objects to communicate in the distributed environment. This is normally provided by remote method invocation.
- **Lifecycle management**: Lifecycle management is concerned with the creation, migration and deletion of objects, with each step having to deal with the distributed nature of the underlying environment.
- **Activation and deactivation**: In non-distributed implementations, it can often be assumed that objects are active all the time while the process that contains them runs. In distributed systems, however, this cannot be assumed as the numbers of objects may be very large, and hence it would be wasteful of resources to have all objects available at any time. **Activation** is the process of making an object active in the distributed environment by providing the necessary resources for it to process incoming invocations – effectively, locating the object in virtual memory and giving it the necessary threads to execute. **Deactivation** is then the opposite process, rendering an object temporarily unable to process invocations.
- **Persistence**: Objects typically have state, and it is important to maintain this state across possible cycles of activation and deactivation and indeed system failures. Distributed object middleware must therefore offer persistency management for stateful objects.
- **Additional services**: A comprehensive distributed object middleware framework must also provide support for the range of distributed system services, including naming, security and transaction services.