**UNIT-II**

**Inter process communication:**

Interprocess communication in the Internet provides both datagram and stream communication. The application program interface to UDP provides a *message passing* abstraction – the simplest form of interprocess communication. This enables a sending process to transmit a single message to a receiving process. The independent packets containing these messages are called *datagrams*.

The application program interface to TCP provides the abstraction of a two-way *stream* between pairs of processes. The information communicated consists of a stream of data items with no message boundaries.

**The API for the Internet protocols:**

**1. The characteristics of interprocess communication:**

Message passing between a pair of processes can be supported by two message communication operations, *send* and *receive*, defined in terms of destinations and messages. To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message. This activity involves the communication of data from the sending process to the receiving process and may involve the synchronization of the two processes.

**Synchronous and asynchronous communication •** In the *synchronous* form of communication, the sending and receiving processes synchronize at every message. In this case, both *send* and *receive* are *blocking* operations.

In the *asynchronous* form of communication, the use of the *send* operation is *nonblocking* in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process. The *receive* operation can have blocking and non-blocking variants. In the non-blocking variant, the receiving process proceeds with its program after issuing a *receive* operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt.

**Message destinations •** A local port is a message destination within a computer, specified as an integer. A port has exactly one receiver but can have many senders. Processes may use multiple ports to receive messages. Any process that knows the number of a port can send a message to it. Servers generally publicize their port numbers for use by clients. If the client uses a fixed Internet address to refer to a service, then that service must always run on the same computer for its address to remain valid. This can be avoided by using the following approach to providing location transparency:

**•** Client programs refer to services by name and use a name server or binder to translate their names into server locations at runtime. This allows services to be relocated but not to migrate – that is, to be moved while the system is running.

**Reliability •** reliable communication can be defined in terms of validity and integrity. As far as the validity property is concerned, a point-to-point message service can be described as reliable if messages are guaranteed to be delivered despite a ‘reasonable’ number of packets being dropped or lost. In contrast, a point-to-point message service can be described as unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost. For integrity, messages must arrive uncorrupted and without duplication.

**Ordering •** Some applications require that messages be delivered in *sender order* – that is, the order in which they were transmitted by the sender. The delivery of messages out of sender order is regarded as a failure by such applications.

**2. Sockets:**

Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process. 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.

Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number. Processes may use the same socket for sending and receiving messages. Each socket is associated with a particular protocol – either UDP or TCP.

**Java API for Internet addresses •** As the IP packets underlying UDP and TCP are sent to Internet addresses, Java provides a class, *InetAddress*, that represents Internet addresses. Users of this class refer to computers by Domain Name System (DNS) hostnames. The method uses the DNS to get the corresponding Internet address. For example, to get an object representing the Internet address of the host whose DNS name is *bruno.dcs.qmul.ac.uk*, use:

***InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk");***

This method can throw an *UnknownHostException*. Note that the user of the class does not need to state the explicit value of an Internet address. In fact, the class encapsulates the details of the representation of Internet addresses. Thus the interface for this class is not dependent on the number of bytes needed to represent Internet addresses – 4 bytes in IPv4 and 16 bytes in IPv6.

**3. 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. To send or receive messages a process must first create a socket bound to an Internet address of the local host and a local port. A server will bind its socket to a *server port* – one that it makes known to clients so that they can send messages to it. A client binds its socket to any free local port. The *receive* method returns the Internet address and port of the sender, in addition to the message, allowing the recipient to send a reply.

The following are some issues relating to datagram communication:

* ***Message size*:** The receiving process needs to specify an array of bytes of a particular size in which to receive a message. If the message is too big for the array, it is truncated on arrival. The underlying IP protocol allows packet lengths of up to 216 bytes, which includes the headers as well as the message.
* ***Blocking*:** Sockets normally provide non-blocking *sends* and blocking *receives* for datagram communication.
* ***Timeouts*:** The *receive* that blocks forever is suitable for use by a server that is waiting to receive requests from its clients. But in some programs, it is not appropriate that a process that has invoked a *receive* operation should wait indefinitely in situations where the sending process may have crashed or the expected message may have been lost. To allow for such requirements, timeouts can be set on sockets. Choosing an appropriate timeout interval is difficult, but it should be fairly large in comparison with the time required to transmit a message.
* ***Receive from any*:** The *receive* method does not specify an origin for messages. Instead, an invocation of *receive* gets a message addressed to its socket from any origin.

**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.

**Use of UDP •** UDP datagrams are sometimes an attractive choice because they do not suffer from the overheads associated with guaranteed message delivery. There are three main sources of overhead:

**•** The need to store state information at the source and destination;

**•** The transmission of extra messages;

**•** Latency for the sender.

**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:

*Datagram packet*

**

An instance of *DatagramPacket* may be transmitted between processes when one process *sends* it and another *receives* it.

A received message is put in the *DatagramPacket* together with its length and the Internet address and port of the sending socket. The message can be retrieved from the *DatagramPacket* by means of the method *getData.* The methods *getPort* and *getAddress* access the port and Internet address.

***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. It also provides a no-argument constructor that allows the system to choose a free local port. These constructors can throw a *SocketException* if the chosen port is already in use or if a reserved is specified when running over UNIX.

The class *DatagramSocket* provides methods that include the following:

* ***Send* and *receive*:** These methods are for transmitting datagrams between a pair of sockets. The methods *send* and *receive* can throw *IOExceptions*.
* ***setSoTimeout*:** This method allows a timeout to be set. With a timeout set, the *receive* method will block for the time specified and then throw an *InterruptedIOException*.
* ***Connect*:** This method is used for connecting to a particular remote port and Internet address.

**4. TCP stream communication:**

The following characteristics of the network are hidden by the stream abstraction:

* ***Message sizes*:** The application can choose how much data it writes to a stream or reads from it. It may deal in very small or very large sets of data.
* ***Lost messages*:** The TCP protocol uses an acknowledgement scheme.
* ***Flow control*:** The TCP protocol attempts to match the speeds of the processes that read from and write to a stream.
* ***Message duplication and ordering*:** Message identifiers are associated with each IP packet, which enables the recipient to detect and reject duplicates, or to reorder messages that do not arrive in sender order.
* ***Message destinations*:** A pair of communicating processes establishes a connection before they can communicate over a stream.

The pair of sockets in the client and server is connected by a pair of streams, one in each direction. Thus each socket has an input stream and an output stream. One of the pair of processes can send information to the other by writing to its output stream, and the other process obtains the information by reading from its input stream.

The following are some outstanding issues related to stream communication:

* ***Matching of data items*:** Two communicating processes need to agree as to the contents of the data transmitted over a stream.
* ***Blocking*:** The data written to a stream is kept in a queue at the destination socket. When a process attempts to read data from an input channel, it will get data from the queue or it will block until data becomes available.
* ***Threads*:** When a server accepts a connection, it generally creates a new thread in which to communicate with the new client.

**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.

When a connection is broken, a process using it will be notified if it attempts to read or write. This has the following effects:

**•** The processes using the connection cannot distinguish between network failure and failure of the process at the other end of the connection.

**•** The communicating processes cannot tell whether the messages they have sent recently have been received or not.

**Use of TCP •** Many frequently used services run over TCP connections, with reserved port numbers. These include the following:

* ***HTTP*:** The Hypertext Transfer Protocol is used for communication between web browsers and web servers;
* ***FTP*:** The File Transfer Protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.
* ***Telnet*:** Telnet provides access by means of a terminal session to a remote computer.
* ***SMTP*:** The Simple Mail Transfer Protocol is used to send mail between computers.

**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 *connects* requests from clients. Its *accept* method gets a *connect* request from the queue or, if the queue is empty, blocks until one arrives. The result of executing *accepts* is an instance of *Socket* – a socket to use for communicating with the client.

***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. This constructor not only creates a socket associated with a local port but also *connects* it to the specified remote computer and port number. It can throw an *UnknownHostException* if the hostname is wrong or an *IOException* if an IO error occurs.

The *Socket* class provides the methods *getInputStream* and *getOutputStream* for accessing the two streams associated with a socket. The return types of these methods are *InputStream* and *OutputStream*, respectively – abstract classes that define methods for reading and writing bytes. The return values can be used as the arguments of constructors for suitable input and output streams.

**External data representation and marshalling:**

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 structures must be flattened before transmission and rebuilt on arrival. There are two variants for the ordering of integers: the so-called *big-endian* order, in which the most significant byte comes first; and *little-endian* order, in which it comes last.

One of the following methods can be used to enable any two computers to exchange binary data values:

• The values are converted to an agreed external format before transmission and converted to the local form on receipt; if the two computers are known to be the same type, the conversion to external format can be omitted.

• The values are transmitted in the sender’s format, together with an indication of the format used, and the recipient converts the values if necessary.

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

***Marshalling***is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. ***Unmarshalling***is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination. Three alternative approaches to external data representation and marshalling are discussed:

**•** CORBA’s common data representation, which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA.

**•** Java’s object serialization, which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.

**•** XML (Extensible Markup Language), which defines a textual format for representing structured data.

In the first two cases, the marshalling and unmarshalling activities are intended to be carried out by a middleware layer without any involvement on the part of the application programmer. In the first two approaches, the primitive data types are marshalled into a binary form. In the third approach (XML), the primitive data types are represented textually. The textual representation of a data value will generally be longer than the equivalent binary representation.

Two other techniques for external data representation are worthy of mention. Google uses an approach called *protocol buffers* to capture representations of both stored and transmitted data. There is also considerable interest in JSON (JavaScript Object Notation) as an approach to external data representation. Protocol buffers and JSON represent a step towards more lightweight approaches to data representation.

**1. CORBA’s Common Data Representation (CDR):**

CORBA CDR is the external data representation defined with CORBA 2.0. CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA. These consist of 15 primitive types, which include *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit), and *any* (which can represent any basic or constructed type); together with a range of composite types. Each argument or result in a remote invocation is represented by a sequence of bytes in the invocation or result message.

***Primitive types*:** CDR defines a representation for both big-endian and little-endian orderings. Values are transmitted in the sender’s ordering, which is specified in each message. The recipient translates if it requires a different ordering. Characters are represented by a code set agreed between client and server.

***Constructed types*:** The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order.



**Marshalling in CORBA •** Marshalling operations can be generated automatically from the specification of the types of data items to be transmitted in a message. The types of the data structures and the types of the basic data items are described in CORBA IDL which provides a notation for describing the types of the arguments and results of RMI methods. For example, we might use CORBA IDL to describe the data structure in the message as follows:

***struct Person{***

***string name;***

***string place;***

***unsigned long year;***

***};***

The CORBA interface compiler generates appropriate marshalling and unmarshalling operations for the arguments and results of remote methods from the definitions of the types of their parameters and results.

**2. Java objects serialization:**

In Java RMI, both objects and primitive data values may be passed as arguments and results of method invocations. An object is an instance of a Java class. For example, the Java class equivalent to the *Person struct* defined in CORBA IDL might be:

*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*

*}*

The above class states that it implements the *Serializable* interface, which has no methods. Stating that a class implements the *Serializable* interface has the effect of allowing its instances to be serialized.

The process that deserializes an object can check that it has the correct version of the class. Java objects can contain references to other objects. When an object is serialized, all the objects that it references are serialized together with it to ensure that when the object is reconstructed, all of its references can be fulfilled at the destination. References are serialized as *handles*.

The contents of the instance variables that are primitive types, such as integers, chars, booleans, bytes and longs, are written in a portable binary format using methods of the *ObjectOutputStream* class. Strings and characters are written by its *writeUTF* method using the Universal Transfer Format (UTF-8), which enables ASCII characters to be represented unchanged (in one byte), whereas Unicode characters are represented by multiple bytes. Strings are preceded by the number of bytes they occupy in the stream.

As an example, consider the serialization of the following object:

*Person p = new Person("Smith", "London", 1984);*

To make use of Java serialization, for example to serialize the *Person* object, creates an instance of the class *ObjectOutputStream* and invokes its *writeObject* method, passing the *Person* object as its argument. To deserialize an object from a stream of data, open an *ObjectInputStream* on the stream and use its *readObject* method to reconstruct the original object. The use of this pair of classes is similar to the use of *DataOutputStream* and *DataInputStream*,

**The use of reflection •** The Java language supports *reflection* – the ability to enquire about the properties of a class, such as the names and types of its instance variables and methods. It also enables classes to be created from their names, and a constructor with given argument types to be created for a given class. Reflection makes it possible to do serialization and deserialization in a completely generic manner.

**3. Extensible Markup Language (XML):**

XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web. In general, the term *markup language* refers to a textual encoding that represents both a text and details as to its structure or its appearance. Both XML and HTML were derived from SGML (Standardized Generalized Markup Language), a very complex markup language. XML was designed for writing structured documents for the Web.

XML data items are tagged with ‘markup’ strings. The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures. XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services.

XML is *extensible* in the sense that users can define their own tags, in contrast to HTML, which uses a fixed set of tags. However, if an XML document is intended to be used by more than one application, then the names of the tags must be agreed between them.

**XML elements and attributes •** 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. For example, one of the elements consists of the data *Smith* contained within the <*name*> ... </*name*> tag pair. Note that the element with the <*name*> tag is enclosed in the element with the <*person id="123456789"*> ... </*person* > tag pair. The ability of an element to enclose another element allows hierarchic data to be represented – a very important aspect of XML. An empty tag has no content and is terminated with */>* instead of *>*. For example, the empty tag <*european*/> could be included within the <*person*> ...</*person*> tag.

***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.

***Names:*** The names of tags and attributes in XML generally start with a letter, but can also start with an underline or a colon. The names continue with letters, digits, hyphens, underscores, colons or full stops. Letters are case-sensitive. Names that start with *xml* are reserved.

***Binary data:*** All of the information in XML elements must be expressed as character data.

**Parsing and well-formed documents •** An XML document must be well formed – that is, it must conform to rules about its structure. A basic rule is that every start tag has a matching end tag. Another basic rule is that all tags are correctly nested. Finally, every XML document must have a single root element that encloses all the other elements. These rules make it very simple to implement parsers for XML documents. When a parser reads an XML document that is not well formed, it will report a fatal error.

***CDATA:*** XML parsers normally parse the contents of elements because they may contain further nested structures. If text needs to contain an angle bracket or a quote, it may be represented in a special way: for example, *&lt* represents the opening angle bracket. However, if a section should not be parsed – for example, because it contains special characters – it can be denoted as *CDATA*. For example, if a place name is to include an apostrophe, then it could be specified in either of the two following ways:

*<place> King&apos Cross </place >*

*<place> <![CDATA [King's Cross]]></place >*

***XML prolog:*** Every XML document must have a *prolog* as its first line. The prolog must at least specify the version of XML in use.

For example: *<?XML version = "1.0" encoding = "UTF-8" standalone = "yes"?>*

The prolog may specify the encoding. The term *encoding* refers to the set of codes used to represent characters – ASCII being the best-known example. Note that in the XML prolog, ASCII is specified as *usascii*.

**XML namespaces •** Traditionally, namespaces provide a means for scoping names. An XML namespace is a set of names for a collection of element types and attributes that is referenced by a URL. Any other XML document can use an XML namespace by referring to its 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. For example:

*xmlns:pers =* [*http://www.cdk5.net/person*](http://www.cdk5.net/person)

The name after *xmlns,* in this case *pers* can be used as a prefix to refer to the elements in a particular namespace.

**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 intention is that a single schema definition may be shared by many different documents. An XML document that is defined to conform to a particular schema may also be validated by means of that schema.

***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. The syntax of DTDs is different from the rest of XML and it is quite limited in what it can specify;

**4. Remote object references**

When a client invokes a method in a remote object, an invocation message is sent to the server process that hosts the remote object. This message needs to specify which particular object is to have its method invoked. A *remote object reference* is an identifier for a remote object that is valid throughout a distributed system. A remote object reference is passed in the invocation message to specify which object is to be invoked.

Remote object references must be generated in a manner that ensures uniqueness over space and time. In general, there may be many processes hosting remote objects, so remote object references must be unique among all of the processes in the various computers in a distributed system. Even after the remote object associated with a given remote object reference is deleted, it is important that the remote object reference is not reused, because its potential invokers may retain obsolete remote object references. Any attempt to invoke a deleted object should produce an error rather than allow access to a different object.

There are several ways to ensure that a remote object reference is unique. One way is to construct a remote object reference by concatenating the Internet address of its host computer and the port number of the process that created it with the time of its creation and a local object number. The local object number is incremented each time an object is created in that process.

. In the simplest implementations of RMI, remote objects live only in the process that created them and survive only as long as that process continues to run. In such cases, the remote object reference can be used as the address of the remote object. In other words, invocation messages are sent to the Internet address in the remote reference and to the process on that computer using the given port number.



*Representation of a remote object reference*

To allow remote objects to be relocated into a different process on a different computer, the remote object reference should not be used as the address of the remote object.

**Client-Server Communication:**

**1. Request-reply protocols:**

This form of communication is designed to support the roles and message exchanges in typical client-server interactions. In the normal case, 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.

The client-server exchanges are described in the following paragraphs in terms of the *send* and *receive* operations in the Java API for UDP datagrams, although many current implementations use TCP streams. A protocol built over datagrams avoids unnecessary overheads associated with the TCP stream protocol.

In particular:

**•** Acknowledgements are redundant, since requests are followed by replies.

**•** Establishing a connection involves two extra pairs of messages in addition to the pair required for a request and a reply.

**•** Flow control is redundant for the majority of invocations, which pass only small arguments and results.

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

The *doOperation* method is 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. 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 message or a reply message is shown in Figure 5.4



The first field indicates whether the message is a *Request* or a *Reply* message. The 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. The third field is a remote reference. The fourth field is an identifier for the operation to be invoked. if the client and server use a common language that supports reflection, a representation of the operation itself may be put in this field.

**Message identifiers •** Any scheme that involves the management of messages to provide additional properties such as reliable message delivery or request-reply communication requires that each message have a unique message identifier by which it may be referenced. A message identifier consists of two parts:

1. a *requestId*, which is taken from an increasing sequence of integers by the sending process;

2. an identifier for the sender process, for example, its port and Internet address.

**Failure model of the request-reply protocol •** If the three primitives *doOperation*, *getRequest* and *sendReply* are implemented over UDP datagrams, then they suffer from the same communication failures. That is:

• They suffer from omission failures.

• Messages are not guaranteed to be delivered in sender order.

**Timeouts •** There are various options as to what *doOperation* can do after a timeout. The simplest option is to return immediately from *doOperation* with an indication to the client that the *doOperation* has failed. To compensate for the possibility of lost messages, *doOperation* sends the request message repeatedly until either it gets a reply or it is reasonably sure that the delay is due to lack of response from the server rather than to lost messages. Eventually, when *doOperation* returns, it will indicate to the client by an exception that no result was received.

**Lost reply messages •** If the server has already sent the reply when it receives a duplicate request it will need to execute the operation again to obtain the result, unless it has stored the result of the original execution. Some servers can execute their operations more than once and obtain the same results each time. An *idempotent operation* is an operation that can be performed repeatedly with the same effect as if it had been performed exactly once.

**History •** For servers that require retransmission of replies without re-execution of operations, a history may be used. The term ‘history’ is used to refer to a structure that contains a record of (reply) messages that have been transmitted. An entry in a history contains a request identifier, a message and an identifier of the client to which it was sent.

**Styles of exchange protocols •** Three protocols, that produce differing behaviors in the presence of communication failures are used for implementing various types of request behaviour. They are

• the *request (R)* protocol;

• the *request-reply (RR)* protocol;

• the *request-reply-acknowledge reply (RRA)* protocol.

In the R protocol, a single *Request* message is sent by the client to the server. The R protocol may be used when there is no value to be returned from the remote operation and the client requires no confirmation that the operation has been executed.

The RR protocol is useful for most client-server exchanges because it is based on the request-reply protocol. Special acknowledgement messages are not required, because a server’s reply message is regarded as an acknowledgement of the client’s request message.

The RRA protocol is based on the exchange of three messages: request-reply acknowledge reply.

**Use of TCP streams to implement the request-reply protocol • I**t is often difficult to decide on an appropriate size for the buffer in which to receive datagrams. In the request-reply protocol, this applies to the buffers used by the server to receive request messages and by the client to receive replies. The limited length of datagrams (usually 8 kilobytes) may not be regarded as adequate for use in transparent RMI or RPC systems, since the arguments or results of procedures may be of any size.

The desire to avoid implementing multipacket protocols is one of the reasons for choosing to implement request-reply protocols over TCP streams, allowing arguments and results of any size to be transmitted.

However, if the application does not require all of the facilities offered by TCP, a more efficient, specially tailored protocol can be implemented over UDP.

**HTTP: An example of a request-reply protocol •** web servers manage resources implemented in different ways:

**•** as data – for example the text of an HTML page, an image or the class of an applet;

**•** as a program – for example, servlets, or PHP or Python programs that run on the web server.

Client requests specify a URL that includes the DNS hostname of a web server and an optional port number on the web server as well as the identifier of a resource on that server.

HTTP is a protocol that specifies the messages involved in a request-reply exchange, the methods, arguments and results, and the rules for representing (marshalling) them in the messages. In addition to invoking methods on web resources, the protocol allows for content negotiation and password-style authentication:

* ***Content negotiation*:** Clients’ requests can include information as to what data representations they can accept, enabling the server to choose the representation that is the most appropriate for the user.
* ***Authentication*:** Credentials and challenges are used to support password-style authentication.

HTTP is implemented over TCP. In the original version of the protocol, each client-server interaction consisted of the following steps:

**•** The client requests and the server accept a connection at the default server port or at a port specified in the URL.

**•** The client sends a request message to the server.

**•** The server sends a reply message to the client.

**•** The connection is closed.

Data resources are supplied as MIME-like structures in arguments and results. Multipurpose Internet Mail Extensions (MIME) is a standard for sending multipart data containing, text, images and sound in email messages.

**HTTP methods •** The methods include the following:

* ***GET*:** Requests the resource whose URL is given as its argument. If the URL refers to data, then the web server replies by returning the data identified by that URL. If the URL refers to a program, then the web server runs the program and returns its output to the client.
* ***HEAD*:** This request is identical to *GET*, but it does not return any data. It does return all the information about the data, such as the time of last modification, its type or its size.
* ***POST*:** Specifies the URL of a resource that can deal with the data supplied in the body of the request. It is designed to deal with:
* providing a block of data to a data-handling process such as a servlet posting a message to a mailing list or updating details of members of the list;
* Extending a database with an append operation.
* ***PUT*:** Requests that the data supplied in the request is stored with the given URL as its identifier, either as a modification of an existing resource or as a new resource.
* ***DELETE*:** The server deletes the resource identified by the given URL.
* ***OPTIONS*:** The server supplies the client with a list of methods it allows to be applied to the given URL and its special requirements.
* ***TRACE*:** The server sends back the request message. Used for diagnostic purposes.

**Group Communication:**

The pair-wise exchange of messages is not the best model for communication from one process to a group of other processes, which may be necessary. 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.

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

**1. *Fault tolerance based on replicated services*:** A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation.

**2. *Discovering services in spontaneous networking*:** Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.

**3. *Better performance through replicated data*:** Data are replicated to increase the performance of a service – in some cases replicas of the data are placed in users’ computers. Each time the data changes, the new values are multicast to the processes managing the replicas.

**4. *Propagation of event notifications*:** Multicast to a group may be used to notify processes when something happens.

**1. IP multicast – An implementation of multicast communication:**

This section discusses IP multicast and presents Java’s API to it via the *MulticastSocket* class.

**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. A *multicast group* is specified by a Class D Internet address – that is, an address whose first 4 bits are 1110 in IPv4.

When a multicast message arrives at a computer, copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number. The following details are specific to IPv4:

* ***Multicast routers*:** IP packets can be multicast both on a local network and on the wider Internet. Local multicasts use the multicast capability of the local network. Internet multicasts make use of multicast routers, which forward single datagrams to routers on other networks, where they are again multicast to local members.
* ***Multicast address allocation*:** Class D addresses (that is, addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast traffic and managed globally by the Internet Assigned Numbers Authority (IANA). This document defines a partitioning of this address space into a number of blocks, including:
* Local Network Control Block (224.0.0.0 to 224.0.0.225), for multicast traffic within a given local network.
* Internet Control Block (224.0.1.0 to 224.0.1.225).
* Ad Hoc Control Block (224.0.2.0 to 224.0.255.0), for traffic that does not fit any other block.
* Administratively Scoped Block (239.0.0.0 to 239.255.255.255), which is used to implement a scoping mechanism for multicast traffic

**Failure model for multicast datagrams •** Datagrams multicast over IP multicast have the same failure characteristics as UDP datagrams – that is, they suffer from omission failures. The effect on a multicast is that messages are not guaranteed to be delivered to any particular group member in the face of even a single omission failure. That is, some but not all of the members of the group may receive it. This can be called *unreliable* multicast, because it does not guarantee that a message will be delivered to any member of a group.

**Java API to IP multicast •** The Java API provides a datagram interface to IP multicast through the class *MulticastSocket*, which is a subclass of *DatagramSocket* with the additional capability of being able to join multicast groups. The class *MulticastSocket* provides two alternative constructors, allowing sockets to be created to use either a specified local port or any free local port.

The Java API allows the TTL to be set for a multicast socket by means of the *setTimeToLive* method. The default is 1, allowing the multicast to propagate only on the local network. An application implemented over IP multicast may use more than one port.

**2. Reliability and ordering of multicast:**

A datagram sent from one multicast router to another may be lost, thus preventing all recipients beyond that router from receiving the message.

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.

**Some examples of the effects of reliability and ordering •**

* *Fault tolerance based on replicated services*
* *Discovering services in spontaneous networking*
* *Better performance through replicated data*
* *Propagation of event notifications*

**Distributed objects and Remote Invocation:**

Programming models for distributed programs/applications: applications composed of cooperating programs running in several different processes.

Such programs need to invoke operations in other processes.

* RPC – client programs call procedures in server programs, running in separate and remote computers (e.g., Unix RPC)
* RMI – extensions of object-oriented programming models to allow a local method (of a local object) to make a remote invocation of objects in a remote process (e.g., Java RMI)
* EBP (event-based programming) model – allows objects anywhere to receive notification of events that occur at other objects in which they have registered interest (e.g., Jini EBP)

**Middleware:** software that allows a level of programming beyond processes and message passing

– Uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocation and events

– Supports location transparency

– Usually uses an interface definition language (IDL) to define interfaces



**Interfaces in Programming Languages**

* Current PL allows programs to be developed as a set of modules that communicate with each other. Permitted interactions between modules are defined by interfaces
* A specified interface can be implemented by different modules without the need to modify other modules using the interface

**Interfaces in Distributed Systems**

* When modules are in different processes or on different hosts there are limitations on the interactions that can occur. Only actions with parameters that are fully specified and understood can communicate effectively to request or provide services to modules in another process.
* A service interface allows a client to request and a server to provide particular services
* A remote interface allows objects to be passed as arguments to and results from distributed modules

**Object Interfaces**

An interface defines the signatures of a set of methods, including arguments, argument types, return values and exceptions. Implementation details are not included in an interface. A class may implement an interface by specifying behavior for each method in the interface. Interfaces do not have constructors.

**Communication between Distributed Objects:**

The object model

1. **Objects** (in classes) encapsulate methods and data variables, with some variables being directly accessible; and communication via passing arguments and receiving results from (locally) invoked objects.
2. **Object references:** objects can be accessed via references. Accessing target/receiver objects requires – reference.methodname(args); and references can be passed as args, too.
3. **Interfaces:** provides a definition of the signatures of a set of object methods – arg type, return values, and exceptions. A class may implement several ‘interfaces,’ and an interface may be implemented by any class
4. **Actions:** effect of method invocation – state of receiver maybe changed; new object maybe instantiated; further invocation may take place
5. **Exceptions:** Provide a clean way to deal with error conditions without complicating the code. thrown and catch
6. **Garbage collection:** reclaiming freed object spaces – Java (automatic), C++ (user supplied)

**Distributed objects**

* State of an object: current values of its variables
* State of program: partitioned into separate parts, each of which is associated with an object – locally partitioned
* As a natural extension, objects are physically distributed into different processes or computers in a distributed system. Therefore, the object model is very appropriate for distributed systems.
* For C-S architecture, objects are managed by servers, clients invoke their methods using remote method invocation
* In RMI, request is sent in a message to the server, the server execute it, and send result back to the client via a message
* Distributed objects in different processes enforces encapsulation: the state of an object can be accessed only by the methods of the object
* Only accept authorized methods to act on the state
* Possibility to handle concurrent access to distributed objects
* Allows heterogeneity: different data formats may be used at different sites

**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*. The following two fundamental concepts are at the heart of the distributed object model:

* ***Remote object references*:** Other objects can invoke the methods of a remote object if they have access to its *remote object reference*
* ***Remote interfaces*:** Every remote object has a *remote interface* that specifies which of its methods can be invoked remotely.

***Remote object references***: The notion of object reference is extended to allow any object that can receive an RMI to have a remote object reference. A remote object reference is an identifier that can be used throughout a distributed system to refer to a particular unique remote object. Remote object references are analogous to local ones in that:

1. The remote object to receive a remote method invocation is specified by the invoker as a remote object reference.

2. Remote object references may be passed as arguments and results of remote method invocations.

***Remote interfaces:*** The class of a remote object implements the methods of its remote interface. 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.

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 inheritances of interfaces. That is, an interface is allowed to extend one or more other interfaces.

**Actions in a distributed object system •** As in the non-distributed case, an action is initiated by a method invocation, which may result in further invocations on methods in other objects. But in the distributed case, the objects involved in a chain of related invocations may be located in different processes or different computers. When an invocation crosses the boundary of a process or computer, RMI is used, and the remote reference of the object must be available to the invoker.

When an action leads to the instantiation of a new object, that object will normally live within the process where instantiation is requested. Distributed applications may provide remote objects with methods for instantiating objects that can be accessed by RMI, thus effectively providing the effect of remote instantiation of 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.
* ***Exceptions:*** Any remote invocation may fail for reasons related to the invoked object being in a different process or computer from the invoker. 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:**

Several separate objects and modules are involved in achieving a remote method invocation. We explore the following related topics: the generation of proxies, the binding of names to their remote object references, the activation and passivation of objects and the location of objects from their remote object references.

**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 uses only the first three items, which specify the message type, its *requestId* and the remote reference of the object to be invoked.

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 •** A remote reference module is responsible for translating between local and remote object references and for creating remote object references. To support its responsibilities, 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). The table includes:

**•** An entry for all the remote objects held by the process.

**•** An entry for each local proxy.

The actions of the remote reference module are as follows:

**•** When a remote object is to be passed as an argument or a result for the first time, the remote reference module is asked to create a remote object reference, which it adds to its table.

**•** When a remote object reference arrives in a *request* or *reply* message, the remote reference module is asked for the corresponding local object reference, which may refer either to a proxy or to a remote object. In the case that the remote object reference is not in the table, the RMI software creates a new proxy and asks the remote reference module to add it to the table.

**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 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.
* ***Dispatcher*:** A server has one dispatcher and one skeleton for each class representing a remote object. The dispatcher receives *request* messages from the communication module.
* ***Skeleton*:** The class of a remote object has a *skeleton*, which implements the methods in the remote interface. A skeleton method unmarshals the arguments in the *request* message and invokes the corresponding method in the servant.

**Generation of the classes for proxies, dispatchers and skeletons •** The classes for the proxy, dispatcher and skeleton used in RMI are generated automatically by an interface compiler. For Java RMI, the set of methods offered by a remote object is defined as a Java interface that is implemented within the class of the remote object. The Java RMI compiler generates the proxy, dispatcher and skeleton classes from the class of the remote object.

**Dynamic invocation: An alternative to proxies •** *Dynamic invocation* gives the client access to a generic representation of a remote invocation like the *doOperation* method used, which is available as part of the infrastructure for RMI. The client will supply the remote object reference, the name of the method and the arguments to *doOperation* and then wait to receive the results.

The dynamic invocation interface is not as convenient to use as a proxy, but it is useful in applications where some of the interfaces of the remote objects cannot be predicted at design time.

**Server and client programs •** The server program contains the classes for the dispatchers and skeletons, together with the implementations of the classes of all of the servants that it supports. The client program will contain the classes of the proxies for all of the remote objects that it will invoke. It can use a binder to look up remote object references.

***Factory methods:*** The term *factory method* is sometimes used to refer to a method that creates servants, and a *factory object* is an object with factory methods. Any remote object that needs to be able to create new remote objects on demand for clients must provide methods in its remote interface for this purpose. Such methods are called factory methods, although they are really just normal methods.

**The binder •** Client programs generally require a means of obtaining a remote object reference for at least one of the remote objects held by a server. A *binder* in a distributed system is a separate service that maintains a table containing mappings from textual names to remote object references.

**Activation of remote objects •** A remote object is described as *active* when it is available for invocation within a running process, whereas it is called *passive* if is not currently active but can be made active. A passive object consists of two parts:

1. The implementation of its methods;

2. Its state in the marshalled form.

***Activation***consists of creating an active object from the corresponding passive object by creating a new instance of its class and initializing its instance variables from the stored state. Passive objects can be activated on demand, for example when they need to be invoked by other objects.

An *activator* is responsible for:

**•** registering passive objects that are available for activation

**•** starting named server processes and activating remote objects in them;

**•** keeping track of the locations of the servers for remote objects that it has already activated.

**Persistent object stores •** An object that is guaranteed to live between activations of processes is called a *persistent object*. Persistent objects are generally managed by persistent object stores, which store their state in a marshalled form on disk.

There are two approaches to deciding whether an object is persistent or not:

**•** The persistent object store maintains some persistent roots, and any object that is reachable from a persistent root is defined to be persistent.

**•** The persistent object store provides some classes on which persistence is based – persistent objects belong to their subclasses.

**Object location •** Clients making invocations require both a remote object reference and an address to which to send invocations.

A *location service* helps clients to locate remote objects from their remote object references. It uses a database that maps remote object references to their probable current locations.

**Remote procedure call:**

**Design issues for RPC**

Before looking at the implementation of RPC systems, we look at three issues that are important in understanding this concept:

**•** The style of programming promoted by RPC – programming with interfaces;

**•** The call semantics associated with RPC;

**•** The key issue of transparency and how it relates to remote procedure calls.

**Programming with interfaces •** Communication between modules can be by means of procedure calls between modules or by direct access to the variables in another module. In order to control the possible interactions between modules, an explicit *interface* is defined for each module. The interface of a module specifies the procedures and the variables that can be accessed from other modules.

***Interfaces in distributed systems:*** In a distributed program, the modules can run in separate processes. 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.

There are a number of benefits to programming with interfaces in distributed systems, stemming from the important separation between interface and implementation:

**•** As with any form of modular programming, programmers are concerned only with the abstraction offered by the service interface and need not be aware of implementation details.

**•** Extrapolating to (potentially heterogeneous) distributed systems, programmers also do not need to know the programming language or underlying platform used to implement the service

**•** This approach provides natural support for software evolution in that implementations can change as long as long as the interface (the external view) remains the same.

The definition of service interfaces is influenced by the distributed nature of the underlying infrastructure:

**•** It is not possible for a client module running in one process to access the variables in a module in another process. Therefore the service interface cannot specify direct access to variables.

**•** The parameter-passing mechanisms used in local procedure calls – for example, call by value and call by reference, are not suitable when the caller and procedure are in different processes.

**•** Another difference between local and remote modules is that addresses in one process are not valid in another remote one.

***Interface definition languages:*** An RPC mechanism can be integrated with a particular programming language if it includes an adequate notation for defining interfaces, allowing input and output parameters to be mapped onto the language’s normal use of parameters. This approach is useful when all the parts of a distributed application can be written in the same language. It is also convenient because it allows the programmer to use a single language, for example, Java, for local and remote invocation.

*Interface definition languages* (IDLs) are designed to allow procedures implemented in different languages to invoke one another. An IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified.

**RPC call semantics •** InRequest-reply protocols *doOperation* can be implemented in different ways to provide different delivery guarantees. The main choices are:

* ***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.

The choices of RPC invocation semantics are defined as follows.

* ***Maybe semantics:*** With *maybe* semantics, the remote procedure call may be executed once or not at all. Maybe semantics arises when no fault-tolerance measures are applied and can suffer from the following types of failure:

**•** Omission failures if the request or result message is lost;

**•** Crash failures when the server containing the remote operation fails.

* ***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-least-once* semantics can be achieved by the retransmission of request messages, which masks the omission failures of the request or result message. *At-least-once* semantics can suffer from the following types of failure:

**•** Crash failures when the server containing the remote procedure fails;

**•** Arbitrary failures – in cases when the request message is retransmitted, the remote server may receive it and execute the procedure more than once, possibly causing wrong values to be stored or returned.

* ***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.

**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. Although request messages are retransmitted after a timeout, this is transparent to the caller to make the semantics of remote procedure calls like that of local procedure calls.

**Implementation of RPC**

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.

**Case study: Java RMI**

Java RMI extends the Java object model to provide support for distributed objects in the Java language.

**Remote interfaces in Java RMI •** Remote interfaces are defined by extending an interface called *Remote* provided in the *java.rmi* package. The methods must throw *RemoteException*, but application-specific exceptions may also be thrown.

**Parameter and result passing •** In Java RMI, the parameters of a method are assumed to be *input* parameters and the result of a method is a single *output* parameter. Any object that is serializable – that is, that implements the *Serializable* interface – can be passed as an argument or result in Java RMI. All primitive types and remote objects are serializable. Classes for arguments and result values are downloaded to the recipient by the RMI system where necessary.

* ***Passing remote objects*:** When the type of a parameter or result value is defined as a remote interface, the corresponding argument or result is always passed as a remote object reference.
* ***Passing non-remote objects*:** All serializable non-remote objects are copied and passed by value.
* **Downloading of classes •** Java is designed to allow classes to be downloaded from one virtual machine to another. This is particularly relevant to distributed objects that communicate by means of remote invocation. This has two advantages:

1. There is no need for every user to keep the same set of classes in their working environment.
2. Both client and server programs can make transparent use of instances of new classes whenever they are added.

**RMIregistry •** The RMIregistry is the binder for Java RMI. An instance of RMIregistry should normally run on every server computer that hosts remote objects. It maintains a table mapping textual, URL-style names to references to remote objects hosted on that computer.

**Design and implementation of Java RMI:**

But in Java 1.2, the reflection facilities were used to make a generic dispatcher and to avoid the need for skeletons. Prior to J2SE 5.0, the client proxies were generated by a compiler called *rmic* from the compiled server classes.

**Use of reflection •** Reflection is used to pass information in request messages about the method to be invoked. This is achieved with the help of the class *Method* in the reflection package. Each instance of *Method* represents the characteristics of a particular method, including its class and the types of its arguments, return value and exceptions.

**Java classes supporting RMI •** The only class that the programmer need be aware of is *UnicastRemoteObject*, which every simple servant class needs to extend. The class *UnicastRemoteObject* extends an abstract class called *RemoteServer*, which provides abstract versions of the methods required by remote servers. *UnicastRemoteObject* was the first example of *RemoteServer* to be provided. Another called *Activatable* is available for providing activatable objects. Further alternatives might provide for replicated objects. The class *RemoteServer* is a subclass of *RemoteObject* that has an instance variable holding the remote object reference and provides the following methods:

***equals*** This method compares remote object references.

***toString*:** This method gives the contents of the remote object reference as a *String*.

***readObject, writeObject*:** These methods deserialize/serialize remote objects.

**NAME SERVICES:**

In a distributed system, names are used to refer to a wide variety of resources such as computers, services, remote objects and files, as well as to users. Naming is an issue that is easily overlooked but is nonetheless fundamental in distributed system design. Names facilitate communication and resource sharing. A name is needed to request a computer system to act upon a specific resource chosen out of many;

**1. Names, addresses and other attributes:**

Any process that requires access to a specific resource must possess a name or an identifier for it. The term *identifier* is sometimes used to refer to names that are interpreted only by programs.

Pure names are simply uninterpreted bit patterns. Non-pure names contain information about the object that they name; in particular, they may contain information about the location of the object. The association between a name and an object is called a *binding*. In general, names are bound to *attributes* of the named objects, rather than the implementation of the objects themselves. An attribute is the value of a property associated with an object. A key attribute of an entity that is usually relevant in a distributed system is its address.

**Names and services •** Many of the names used in a distributed system are specific to some particular service; a client may use a service-specific name when requesting a service to perform an operation upon a named object or resource that it manages.

Names are also sometimes needed to refer to entities in distributed systems that are beyond the scope of any single service. The major examples of these entities are users (, computers and services themselves. In object-based middleware, names refer to remote objects that provide services or applications.

**Uniform Resource Identifiers •** *Uniform Resource Identifiers* (URIs) came about from the need to identify resources on the Web, and other Internet resources such as electronic mailboxes. URIs are ‘uniform’ in that their syntax incorporates that of indefinitely many individual types of resource identifiers (that is, URI *schemes*), and there are procedures for managing the global namespace of schemes. The advantage of uniformity is that it eases the process of introducing new types of identifier, as well as using existing types of identifier in new contexts, without disrupting existing usage.

***Uniform Resource Locators:*** Some URIs contains information that can be used to locate and access a resource; others are pure resource names. The familiar terms Uniform Resource Locator (URL) is often used for URIs that provides location information and specifies the method for accessing the resource, including the ‘http’.

***Uniform Resource Names:*** *Uniform Resource Names* (URNs) are URIs that is used as pure resource names rather than locators. For example, the URI:

*mid:0E4FC272-5C02-11D9-B115-000A95B55BC8@hpl.hp.com*

is a URN that identifies the email message containing it in its ‘Message-Id’ field. The URI distinguishes that message from any other email message. But it does not provide the message’s address in any store, so a lookup operation is needed to find it.

**Name services and the Domain Name System:**

A *name service* stores information about a collection of textual names, in the form of bindings between the names and the attributes of the entities they denote, such as users, computers, services and objects. The major operation that a name service supports is to resolve a name – that is, to look up attributes from a given name. Operations are also required for creating new bindings, deleting bindings and listing bound names, and adding and deleting contexts.

Name management is separated from other services largely because of the openness of distributed systems, which brings the following motivations:

* ***Unification*:** It is often convenient for resources managed by different services to use the same naming scheme.
* ***Integration*:** It is not always possible to predict the scope of sharing in a distributed system.

**General name service requirements •** Name services were originally quite simple, since they were designed only to meet the need to bind names to addresses in a single management domain, corresponding to a single LAN or WAN. The interconnection of networks and the increased scale of distributed systems have produced a much larger name-mapping problem.

The Global Name Service, developed at the Digital Equipment Corporation Systems Research Center is a descendant of Grapevine with ambitious goals, including:

* *To handle an essentially arbitrary number of names and to serve an arbitrary number of administrative organizations*: For example, the system should be capable of handling the names of all the documents in the world.
* *A long lifetime*: Many changes will occur in the organization of the set of names and in the components that implement the service during its lifetime.
* *High availability*: Most other systems depend upon the name service; they can’t work when it is broken.
* *Fault isolation*: Local failures should not cause the entire service to fail.
* *Tolerance of mistrust*: A large open system cannot have any component that is trusted by all of the clients in the system.

**13.2.1 Name spaces**

A *name space* is the collection of all valid names recognized by a particular service. The service will attempt to look up a valid name, even though that name may prove not to correspond to any object – i.e., to be *unbound*. Name spaces require a syntactic definition to separate valid names from invalid names.

Names may have an internal structure that represents their position in a hierarchic name space such as pathnames in a file system or in an organizational hierarchy such as Internet domain names; or they may be chosen from a flat set of numeric or symbolic identifiers.

DNS names are strings called *domain names*. Some examples are *net*, *com* and *ac.uk* (the latter three are domains). The DNS name space has a hierarchic structure: a domain name consists of one or more strings called *name components* or *labels*, separated by the delimiter ‘.’. There is no delimiter at the beginning or end of a domain name, although the root of the DNS name space is sometimes referred to as ‘.’ for administrative purposes. The name components are non-null printable strings that do not contain ‘.’. In general, a *prefix* of a name is an initial section of the name that contains only zero or more entire components. DNS names are not case-sensitive.DNS servers do not recognize relative names: all names are referred to the global root.

**Aliases •** An *alias* is a name defined to denote the same information as another name, similar to a symbolic link between file path names. Aliases allow more convenient names to be substituted for relatively complicated ones, and allow alternative names to be used by different people for the same entity.

**Naming domains •** A *naming domain* is a name space for which there exists a single overall administrative authority responsible for assigning names within it. This authority is in overall control of which names may be bound within the domain, but it is free to delegate this task. Domains in DNS are collections of domain names; syntactically, a domain’s name is the common suffix of the domain names within it, but otherwise it cannot be distinguished from, for example, a computer name.

**Combining and customizing name spaces •** The DNS provides a global and homogeneous name space in which a given name refers to the same entity, no matter which process on which computer looks up the name. By contrast, some name services allow distinct name spaces – sometimes heterogeneous name spaces – to be embedded into them; and some name services allow the name space to be customized to suit the needs of individual groups, users or even processes.

***Merging:*** The moral is that we can always merge name spaces by creating a higher-level root context, but this may raise a problem of backward-compatibility. Fixing the compatibility problem, in turn, leaves us with hybrid name spaces and the inconvenience of having to translate old names between the users of the two computers.

***Heterogeneity:*** The Distributed Computing Environment (DCE) name space allows heterogeneous name spaces to be embedded within it. DCE names may contain *junctions*, which are similar to mount points in NFS and UNIX except that they allow heterogeneous name spaces to be mounted.

***Customization:*** File system mounting enables users to import files that are stored on servers and shared, while the other names continue to refer to local, unshared files and can be administered autonomously. But the same files accessed from two different computers may be mounted at different points and thus have different names. Not sharing the entire name space means users must translate names between computers.

The spring naming service [Radia *et al.* 1993] provides the ability to construct name spaces dynamically and to share individual naming contexts selectively. Even two different processes on the same computer can have different naming contexts. Spring naming contexts are first-class objects that can be shared around a distributed system.

**13.2.2 Name resolution**

A naming context either maps a given name onto a set of primitive attributes directly, or maps it onto a further naming context and a derived name to be presented to that context. To resolve a name, it is first presented to some initial naming context; resolution iterates as long as further contexts and derived names are output.

**Name servers and navigation •** Any name service, such as DNS, that stores a very large database and is used by a large population will not store all of its naming information on a single server computer. Such a server would be a bottleneck and a critical point of failure. Any heavily used name services should use replication to achieve high availability.

The partitioning of data implies that the local name server cannot answer all enquiries without the help of other name servers. The process of locating naming data from more than one name server in order to resolve a name is called *navigation*. The client name resolution software carries out navigation on behalf of the client. It communicates with name servers as necessary to resolve a name.

One navigation model that DNS supports is known as *iterative navigation.* To resolve a name, a client presents the name to the local name server, which attempts to resolve it. If the local name server has the name, it returns the result immediately. If it does not, it will suggest another server that will be able to help. Resolution proceeds at the new server, with further navigation as necessary until the name is located or is discovered to be unbound.

NFS also employs iterative navigation in the resolution of a file name, on a component-by component basis. This is because the file service may encounter a symbolic link when resolving a name. A symbolic link must be interpreted in the client’s file system name space because it may point to a file in a directory stored at another server. The client computer must determine which server this is, because only the client knows its mount points.

In *multicast navigation*, client multicasts the name to be resolved and the required object type to the group of name servers. Only the server that holds the named attributes responds to the request.

**Caching •** In DNS and other name services, client name resolution software and servers maintain a cache of the results of previous name resolutions. When a client requests a name lookup, the name resolution software consults its cache. If it holds a recent result from a previous lookup for the name, it returns it to the client; otherwise, it sets about finding it from a server. That server, in turn, may return data cached from other servers.

Caching is key to a name service’s performance and assists in maintaining the availability of both the name service and other services in spite of name server crashes. Its role in enhancing response times by saving communication with name servers is clear. Caching can be used to eliminate high-level name servers – the root server, in particular – from the navigation path, allowing resolution to proceed despite some server failures.

**13.2.3 The Domain Name System**

The Domain Name System is a name service design whose main naming database is used across the Internet. DNS replaced the original Internet naming scheme, in which all host names and addresses were held in a single central master file and downloaded by FTP to all computers that required them.

This original scheme was soon seen to suffer from three major shortcomings:

**•** It did not scale to large numbers of computers.

**•** Local organizations wished to administer their own naming systems.

**•** A general name service was needed – not one that serves only for looking up computer addresses.

**Domain names •** The DNS is designed for use in multiple implementations, each of which may have its own name space. In practice, however, only one is in widespread use, and that is the one used for naming across the Internet. The Internet DNS name space is partitioned both organizationally and according to geography. The names are written with the highest-level domain on the right. The original top-level organizational domains (also called *generic domains*) in use across the Internet were:

*com* – Commercial organizations

*edu* – Universities and other educational institutions

*gov* – US governmental agencies

*mil* – US military organizations

*net* – Major network support centres

*org* – Organizations not mentioned above

*int* – International organizations

New top-level domains such as *biz* and *mobi* have been added since the early 2000s. A full list of current generic domain names is available from the Internet Assigned Numbers Authority.

In addition, every country has its own domains:

*us* – United States

*uk* – United Kingdom

*fr* – France

**DNS queries •** The Internet DNS is primarily used for simple host name resolution and for looking up electronic mail hosts, as follows:

* ***Host name resolution*:** In general, applications use the DNS to resolve host names into IP addresses.
* ***Mail host location*:** Electronic mail software uses the DNS to resolve domain names into the IP addresses of mail hosts – i.e., computers that will accept mail for those domains.
* ***Reverse resolution*:** Some software requires a domain name to be returned given an IP address. This is just the reverse of the normal host name query, but the name server receiving the query replies only if the IP address is in its own domain.
* ***Host information*:** The DNS can store the machine architecture type and operating system with the domain names of hosts

**DNS name servers •** Each server records the domain names and addresses of other name servers, so that queries pertaining to objects outside the domain can be satisfied.

The DNS naming data are divided into *zones*. A zone contains the following data:

**•** Attribute data for names in a domain, less any sub domains administered by lower level authorities.

**•** The names and addresses of at least two name servers that provide *authoritative* data for the zone. These are versions of zone data that can be relied upon as being reasonably up-to-date.

**•** The names of name servers that hold authoritative data for delegated sub domains; and ‘glue’ data giving the IP addresses of these servers.

**•** Zone-management parameters, such as those governing the caching and replication of zone data.

A server may hold authoritative data for zero or more zones. So that naming data are available even when a single server fails, the DNS architecture specifies that each zone must be replicated authoritatively in at least two servers.

There are two types of server that are considered to provide authoritative data. A *primary* or *master server* reads zone data directly from a local master file. *Secondary servers* download zone data from a primary server.

**Navigation and query processing •** A DNS client is called a *resolver*. It is normally implemented as library software. It accepts queries, formats them into messages in the form expected under the DNS protocol and communicates with one or more name servers in order to satisfy the queries. The resolver times out and resends its query if necessary.

The DNS architecture allows for recursive navigation as well as iterative navigation. The resolver specifies which type of navigation is required when contacting a name server.

**Resource records •** Zone data are stored by name servers in files in one of several fixed types of

resource record.



***Load sharing by name servers:*** At some sites, heavily used services such as the Web and FTP are supported by a group of computers on the same network. When a domain name is shared by several computers, there is one record for each computer in the group, giving its IP address. Successive clients are given access to different servers so that the servers can share the workload. Caching has a potential for spoiling this scheme, for once a non-authoritative name server or a client has the server’s address in its cache it will continue to use it. To counteract this effect, the records are given a short time to live.

**The BIND implementation of the DNS •** The Berkeley Internet Name Domain (BIND) is an implementation of the DNS for computers running UNIX. Client programs link in library software as the resolver. DNS name server computers run the named daemon.

BIND allows for three categories of name server: primary servers, secondary servers and caching-only servers. The named program implements just one of these types, according to the contents of a configuration file. Caching-only servers read in from a configuration file sufficient names and addresses of authoritative servers to resolve any name. Thereafter, they only store this data and data that they learn by resolving names for clients.

A typical organization has one primary server, with one or more secondary servers that provide name serving on different local area networks at the site.

**Directory services:**

A service that stores collections of bindings between names and attributes and that looks up entries that match attribute-based specifications is called a *directory service*. Examples are Microsoft’s Active Directory Services, X.500 and LDAP.

Directory services are sometimes called *yellow pages services*, and conventional name services are correspondingly called *white pages services*, in an analogy with the traditional types of telephone directory. Directory services are also sometimes known as *attribute-based name services*.

A directory service returns the sets of attributes of any objects found to match some specified attributes. The Universal Directory and Discovery Service (UDDI), provides both white pages and yellow pages services to provide information about organizations and the web services they offer.

One core difference between a discovery service and other directory services is that the address of a directory service is normally well known and preconfigured in clients, whereas a device entering a spontaneous networking environment has to resort to multicast navigation, at least the first time it accesses the local discovery service.

**Case study: The X.500 Directory Service**

X.500 is a directory service used in the same way as a conventional name service, but it is primarily used to satisfy descriptive queries and is designed to discover the names and attributes of other users or system resources. They range from enquiries that are directly analogous to the use of telephone directories, such as a simple ‘white pages’ access to obtain a user’s electronic mail address or a ‘yellow pages’ query aimed.

Individuals and organizations can use a directory service to make available a wide range of information about themselves and the resources that they wish to offer for use in the network. Users can search the directory for specific information with only partial knowledge of its name, structure or content.

The ITU and ISO standards organizations defined the *X.500 Directory Service* as a network service intended to meet these requirements. The data stored in X.500 servers is organized in a tree structure with named nodes, as in the case of the other name servers, but in X.500 a wide range of attributes are stored at each node in the tree, and access is possible not just by name but also by searching for entries with any required combination of attributes.

The X.500 name tree is called the *Directory Information Tree* (DIT), and the entire directory structure including the data associated with the nodes, is called the *Directory Information Base* (DIB). There is intended to be single integrated DIB containing information provided by organizations throughout the world, with portions of the DIB located in individual X.500 servers. In the terminology of the X.500 standard, servers are *Directory Service Agents* (DSAs), and their clients are termed *Directory User Agents* (DUAs).

Each entry in the DIB consists of a name and a set of attributes. As in other name servers, the full name of an entry corresponds to a path through the DIT from the root of the tree to the entry. In addition to full or *absolute* names, a DUA can establish a context, which includes a base node, and then use shorter relative names that give the path from the base node to the named entry.

Now we can consider the methods by which the directory is accessed. There are two main types of access request:

* ***Read*:** An absolute or relative name for an entry is given, together with a list of attributes to be read. The DSA locates the named entry by navigating in the DIT, passing requests to other DSA servers where it does not hold relevant parts of the tree. It retrieves the required attributes and returns them to the client.
* ***Search*:** This is an attribute-based access request. A base name and a filter expression are supplied as arguments. The base name specifies the node in the DIT from which the search is to commence; the filter expression is a boolean expression that is to be evaluated for every node below the base node. The filter specifies a search criterion: a logical combination of tests on the values of any of the attributes in an entry. The *search* command returns a list of names (domain names) for all of the entries below the base node for which the filter evaluates to *TRUE*.

**Administration and updating of the DIB •** The DSA interface includes operations for adding, deleting and modifying entries. Access control is provided for both queries and updating operations, so access to parts of the DIT may be restricted to certain users or classes of user.

The DIB is partitioned, with the expectation that each organization will provide at least one server holding the details of the entities in that organization. Portions of the DIB may be replicated in several servers.

**Lightweight Directory Access Protocol •** A group at the University of Michigan proposed a more lightweight approach called the *Lightweight Directory Access Protocol* (LDAP), in which a DUA accesses X.500 directory services directly over TCP/IP instead of the upper layers of the ISO protocol stack.

An implementation may use any other directory server that obeys the simpler LDAP specification, as opposed to the X.500 specification. For example, Microsoft’s Active Directory Services provides an LDAP interface. Unlike X.500, LDAP has been widely adopted, particularly for intranet directory services. It provides secure access to directory data through authentication.