# UNIT V

## Introduction to Distributed Algorithms

Distributed algorithms have been the subject of intense development over the last twenty years. The second edition of this successful textbook provides an up-to-date introduction both to the topic, and to the theory behind the algorithms. The clear presentation makes the book suitable for advanced undergraduate or graduate courses, whilst the coverage is sufficiently deep to make it useful for practising engineers and researchers. The author concentrates on algorithms for the point-to-point message passing model, and includes algorithms for the implementation of computer communication networks. Other key areas discussed are algorithms for the control of distributed applications (wave, broadcast, election, termination detection, randomized algorithms for anonymous networks, snapshots, deadlock detection, synchronous systems), and fault-tolerance achievable by distributed algorithms. The two new chapters on sense of direction and failure detectors are state-of-the-art and will provide an entry to research in these still-developing topics.

- Comprehensive overview of distributed algorithms and related theory and methods
- Use of uniform model of computing and uniform programming notation throughout with treatment of wave algorithms as an algorithmic building block
- Many exercises with teachers' solutions available, plus clear and rigorous presentation makes it ideal for courses on distributed systems and algorithms

## Introduction to Communication Protocols

Here you will learn about computer network protocols, tcp/ip introduction, ftp, http overview, x.25, ftp, smtp and snmp. The word protocol is derived from the Greek word “protocollon” which means a leaf of paper glued to manuscript volume. In computer protocols means a set of rules, a communication language or set of standards between two or more computing devices. Protocols exist at the several levels of the OSI (open system interconnectivity) layers model.

In the telecommunication system, there are one more protocols at each layer of the telephone exchange. On the internet, there is a suite of the protocols known as TCP/IP protocols that are consisting of transmission control protocol, internet protocol, file transfer protocol, dynamic host configuration protocol, Border gateway protocol and a number of other protocols.

In the telecommunication, a protocol is set of rules for data representation, authentication, and error detection. The communication protocols in the computer networking are intended for the secure, fast and error free data delivery between two communication devices. Communication protocols follow certain rules for the transmission of the data.

**Protocols Properties**

Different protocols perform different functions so it is difficult to generalize the properties of the protocols. There are some basic properties of most of the protocols.

• Detection of the physical (wired or wireless connection)
• Handshaking
• How to format a message.
• How to send and receive a message.
• Negotiation of the various connections
• Correction of the corrupted or improperly formatted messages.
• Termination of the session.

The widespread use of the communication protocols is a prerequisite to the internet. The term TCP/IP refers to the protocols suite and a pair of the TCP and IP protocols are the most important internet communication protocols. Most protocols in communication are layered together where the various tasks listed above are divided. Protocols stacks refer to the combination of the different protocols. The OSI reference model is the conceptual model that is used to represent the stacks. There are different network protocols that

Perform different functions. Following is the description of the some of the most commonly used protocols.

**HTTP (Hyper Text Transfer Protocol)**

Hypertext transfer protocol is a method of transmitting the information on the web. HTTP basically publishes and retrieves the HTTP pages on the World Wide Web. HTTP is a language that is used to communicate between the browser and web server. The information that is transferred using HTTP can be plain text, audio, video, images, and hypertext. HTTP is a request/response protocol between the client and server. Many proxies, tunnels, and gateways can be existing between the web browser (client) and server (web server). An HTTP client initializes a request by establishing a TCP connection to a particular port on the remote host (typically 80 or 8080). An HTTP server listens to that port and receives a request message from the client. Upon receiving the request, server sends back 200 OK messages, its own message, an error message or other message.

**POP3 (Post Office Protocol)**

In computing, e-mail clients such as (MS outlook, outlook express and thunderbird) use Post office Protocol to retrieve emails from the remote server over the TCP/IP connection. Nearly all the users of the Internet service providers use POP 3 in the email clients to retrieve the emails from the email servers. Most email applications use POP protocol.

**SMTP (Simple Mail Transfer Protocol)**

Simple Mail Transfer Protocol is a protocol that is used to send the email messages between the servers. Most email systems and email clients use the SMTP protocol to send messages to one server to another. In configuring an email application, you need to configure POP, SMTP and IMAP protocols in your email software. SMTP is a simple, text based protocol and one or more recipient of the message is specified and then the message is transferred. SMTP connection is easily tested by the Telnet utility.

**FTP (File Transfer Protocol)**

FTP or file transfer protocol is used to transfer (upload/download) data from one computer to another over the internet or through or computer network. FTP is a most commonly communication protocol for transferring the files over the internet. Typically, there are two computers are involved in the transferring the files a server and a client. The client computer that is running FTP client software such as Cutest and Accept etc initiates a connection with the remote computer (server). After successfully connected with the server, the client computer can perform a number of the operations like downloading the files, uploading, renaming and deleting the files, creating the new folders etc. Virtually operating system supports FTP protocols.

**IP (Internet Protocol)**

An Internet protocol (IP) is a unique address or identifier of each computer or communication devices on the network and internet. Any participating computer networking device such as routers, computers, printers, internet fax machines and switches may have their own unique IP address. Personal information about someone can be found by the IP address. Every domain on the internet must have a unique or shared IP address.

**DHCP (Dynamic Host Configuration Protocol).**

The DHCP or Dynamic Host Configuration Protocol is a set of rules used by a communication device such as router, computer or network adapter to allow the device to request and obtain and IP address from a server which has a list of the larger number of addresses. DHCP is a protocol that is used by the network computers to obtain the IP addresses and other settings such as gateway, DNS, subnet mask from the DHCP server. DHCP ensures that all the IP addresses are unique and the IP address management is done by the server and not by the human. The assignment of the IP addresses is expires after the predetermined period of time. DHCP works in four phases known as DORA such as Discover, Observe, Request and Authorize

**IMAP (Internet Message Access Protocol)**

The Internet Message Access Protocol known as IMAP is an application layer protocol that is used to access to access the emails on the remote servers. POP3 and IMAP are the two most commonly used email retrieval protocols. Most of the email clients such as outlook express, thunderbird and MS outlooks support POP3 and IMAP. The email messages are generally stored on the email server and the users generally retrieve these messages whether by the web browser or email clients. IMAP is generally used in the large networks. IMAP allows users to access their messages instantly on their systems.

**ARCNET**

ARCNET is a local area network technology that uses token bus scheme for managing line sharing among the workstations. When a device on a network wants to send a message, it inserts a token that is set to 1 and when a destination device reads the message it resets the token to 0 so that the frame can be used by another device.

**FDDI**

Fiber distributed data interface (FDDI) provides a standard for data transmission in a local area network that can extend a range of 200 kilometers. The FDDI uses token ring protocol as its basis. FDDI local area network can support a large number of users and can cover a large geographical area. FDDI uses fiber optic as a standard communication medium. FDDI uses dual attached token ring topology. A FDDI network contains two token rings and the primary ring offers the capacity of 100 Mbits/s. FDDI is an ANSI standard network and it can support 500 stations in 2 kilometers.

**UDP**

The user datagram protocol is a most important protocol of the TCP/IP suite and is used to send the short messages known as datagram. Common network applications that uses UDP are DNS, online games, IPTV, TFTP and VOIP. UDP is very fast and light weight. UDP is an unreliable connectionless protocol that operates on the transport layer and it is sometimes called Universal Datagram Protocol.

**X.25**

X.25 is a standard protocol suite for wide area networks using a phone line or ISDN system. The X.25 standard was approved by CCITT now ITU in 1976.

**TFTP**

Trivial File Transfer Protocol (TFTP) is a very simple file transfer protocol with the very basic features of the FTP. TFTP can be implemented in a very small amount of memory. TFTP is useful for booting computers such as routers. TFTP is also used to transfer the files over the network. TFPT uses UDP and provides no security features.

## SNMP

The simple network management protocol (SNMP) forms the TCP/IP suite. SNMP is used to manage the network attached devices of the complex network.

## PPTP

The point to point tunneling protocol is used in the virtual private networks. PPP works by sending regular PPP session. PPTP is a method of implementing VPN networks.

## Sliding window protocol

A **sliding window protocol** is a feature of packet-based data transmission protocols. Sliding window protocols are used where reliable in-order delivery of packets is required, such as in the Data Link Layer (OSI model) as well as in the Transmission Control Protocol (TCP).

Conceptually, each portion of the transmission (packets in most data link layers, but bytes in TCP) is assigned a unique consecutive sequence number, and the receiver uses the numbers to place received packets in the correct order, discarding duplicate packets and identifying missing ones. The problem with this is that there is no limit of the size of the sequence numbers that can be required.

By placing limits on the number of packets that can be transmitted or received at any given time, a sliding window protocol allows an unlimited number of packets to be communicated using fixed-size sequence numbers.

For the highest possible throughput, it is important that the transmitter is not forced to stop sending by the sliding window protocol earlier than one round-trip delay time (RTT). The limit on the amount of data that it can send before stopping to wait for an acknowledgment should be larger than the bandwidth-delay product of the communications link. If it is not, the protocol will limit the effective bandwidth of the link.

Motivation

In any communication protocol based on automatic repeat request for error control, the receiver must acknowledge received packets. If the transmitter does not receive an acknowledgment within a reasonable time, it re-sends the data.

A transmitter that does not hear an acknowledgment cannot know if the receiver actually received the packet; it may be that the packet was lost in transmission (or damaged; if error detection finds an error, the packet is ignored), or it may be that an acknowledgment was sent, but it was lost. In the latter case, the receiver must acknowledge the retransmission, but must otherwise ignore it.

## Protocol operation

The transmitter and receiver each have a current sequence number $n_t$ and $n_r$, respectively. They each also have a window size $w_t$ and $w_r$. The window sizes may vary, but in simpler implementations they are fixed. The window size must be greater than zero for any progress to be made.

As typically implemented, $n_t$ is the next packet to be transmitted, i.e. the sequence number of the first packet not yet transmitted. Likewise, $n_r$ is the first packet not yet received. Both numbers are monotonically increasing with time; they only ever increase.

The receiver may also keep track of the highest sequence number not yet received; the variable $n_s$ is one more than the sequence number of the highest sequence number received. For simple receivers that only accept packets in order ($w_r$=1), this is the same as $n_r$, but can be greater if $w_r$>1. Note the distinction: all packets below $n_r$ have been received, no packets above $n_s$ have been received, and between $n_r$ and $n_s$, some packets have been received.

When the receiver receives a packet, it updates its variables appropriately and transmits an acknowledgment with the new $n_r$. The transmitter keeps track of the highest acknowledgment it has received $n_a$. The transmitter knows that all packets up to, but not including $n_a$ have been received, but is uncertain about packets between $n_a$ and $n_s$; i.e. $n_a \le n_r \le n_s$.

The sequence numbers always obey the rule that $n_a \le n_r \le n_s \le n_t \le n_a + w_t$. That is:

- $n_a \le n_r$: The highest acknowledgement received by the transmitter cannot be higher than the highest $n_r$ acknowledged by the receiver.
- $n_r \le n_s$: The span of fully-received packets cannot extend beyond the end of the partially-received packets.
- $n_s \le n_t$: The highest packet received cannot be higher than the highest packet sent.
- $n_t \le n_a + w_t$: The highest packet sent is limited by the highest acknowledgement received and the transmit window size.

## Transmitter operation

Whenever the transmitter has data to send, it may transmit up to $w_t$ packets ahead of the latest acknowledgment $n_a$. That is, it may transmit packet number $n_t$ as long as $n_t < n_a+w_t$.

In the absence of a communication error, the transmitter soon receives an acknowledgment for all the packets it has sent, leaving $n_a$ equal to $n_t$. If this does not happen after a reasonable delay, the transmitter must retransmit the packets between $n_a$ and $n_t$.

Techniques for defining "reasonable delay" can be extremely elaborate, but they only affect efficiency; the basic reliability of the sliding window protocol does not depend on the details.

## Receiver operation

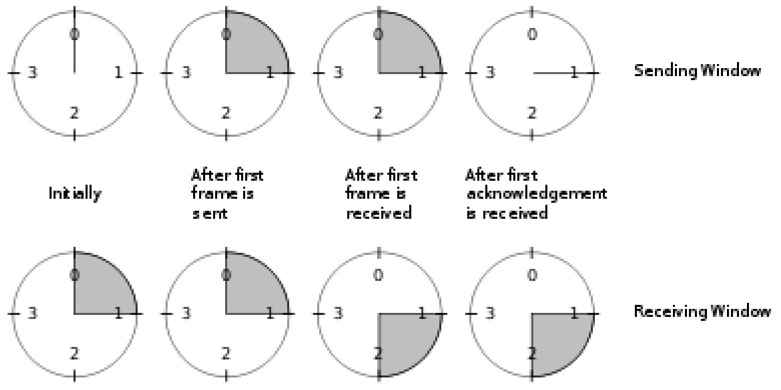
Every time a packet numbered x is received, the receiver checks to see if it falls in the receive window, $n_r \le x < n_s+w_r$. (The simplest receivers only have to keep track of one value $n_r=n_s$.) If it falls within the window, the receiver accepts it. If it is numbered $n_r$, the receive sequence number is increased by 1, and possibly more if further consecutive packets were previously received and stored. If $x > n_r$, the packet is stored until all preceding packets have been received.[1] If $x \ge n_s$, the latter is updated to $n_s=x+1$.

If the packet's number is not within the receive window, the receiver discards it and does not modify $n_r$ or $n_s$.

Whether the packet was accepted or not, the receiver transmits an acknowledgment containing the current $n_r$. (The acknowledgment may also include information about additional packets received between $n_r$ or $n_s$, but that only helps efficiency.)

Note that there is no point having the receive window $w_r$ larger than the transmit window $w_t$, because there is no need to worry about receiving a packet that will never be transmitted; the useful range is $1 \leq w_r \leq w_t$.

## Sequence number range required


Sending Window
Initially
After first frame is sent
After first frame is received
After first acknowledgement is received
Receiving Window
0
1
2
3

A sliding window with a 2-bit sequence, of size 1

Sequence numbers modulo 4, with $w_r$=1. Initially, $n_t$=$n_r$=0

So far, the protocol has been described as if sequence numbers are of unlimited size, ever-increasing. However, rather than transmitting the full sequence number x in messages, it is possible to transmit only x mod N, for some finite N. (N is usually a power of 2.)

For example, the transmitter will only receive acknowledgments in the range $n_a$ to $n_t$, inclusive. Since it guarantees that $n_t - n_a \leq w_t$, there are at most $w_t+1$ possible sequence numbers that could arrive at any given time. Thus, the transmitter can unambiguously decode the sequence number as long as $N > w_t$.

A stronger constraint is imposed by the receiver. The operation of the protocol depends on the receiver being able to reliably distinguish new packets (which should be accepted and processed) from retransmissions of old packets (which should be discarded, and the last acknowledgment retransmitted). This can be done given knowledge of the transmitter's window size. After receiving a packet numbered x, the receiver knows that $x < n_a+w_t$, so $n_a > x-w_t$. Thus, packets numbered $x-w_t$ will never again be retransmitted.

The lowest sequence number we will ever receive in future is $n_s-w_t$

The receiver also knows that the transmitter's $n_a$ cannot be higher than the highest acknowledgment ever sent, which is $n_r$. So the highest sequence number we could possibly see is $n_r+w_t \leq n_s+w_t$.

Thus, there are $2w_t$ different sequence numbers that the receiver can receive at any one time. It might therefore seem that we must have $N \geq 2w_t$. However, the actual limit is lower.

The additional insight is that the receiver does not need to distinguish between sequence numbers that are too low (less than $n_r$) or that are too high (greater than or equal to $n_s+w_r$). In either case, the receiver ignores the packet except to retransmit an acknowledgment. Thus, it is only necessary that $N \geq w_t+w_r$. As it is common to have $w_r<w_t$ (e.g. see Go-Back-N below), this can permit larger $w_t$ within a fixed N.

## Routing

Routing or routing is the process of selecting paths in a network along which to send network traffic. Routing is performed for many kinds of networks, including the telephone network (Circuit switching) , electronic data networks (such as the Internet), and transportation networks. This article is concerned primarily with routing in electronic data networks using packet switching technology.

In packet switching networks, routing directs packet forwarding, the transit of logically addressed packets from their source toward their ultimate destination through intermediate nodes, typically hardware devices called routers, bridges, gateways, firewalls, or switches.

General-purpose computers can also forward packets and perform routing, though they are not specialized hardware and may suffer from limited performance. The routing process usually directs forwarding on the basis of routing tables which maintain a record of the routes to various network destinations. Thus, constructing routing tables, which are held in the router's memory, is very important for efficient routing. Most routing algorithms use only one network path at a time, but multipath routing techniques enable the use of multiple alternative paths.

Routing, in a more narrow sense of the term, is often contrasted with bridging in its assumption that network addresses are structured and that similar addresses imply proximity within the network. Because structured addresses allow a single routing table entry to represent the route to a group of devices, structured addressing (routing, in the narrow sense) outperforms unstructured addressing (bridging) in large networks, and has become the dominant form of addressing on the Internet, though bridging is still widely used within localized environments.

## Delivery semantics

Routing schemes differ in their delivery semantics:

- unicast delivers a message to a single specified node;
- broadcast delivers a message to all nodes in the network;
- multicast delivers a message to a group of nodes that have expressed interest in receiving the message;
- anycast delivers a message to any one out of a group of nodes, typically the one nearest to the source.
- geocast delivers a message to a geographic area

## Topology distribution

In a practice known as static routing (or non-adaptive routing), small networks may use manually configured routing tables. Larger networks have complex topologies that can change rapidly, making the manual construction of routing tables unfeasible. Nevertheless, most of the public switched telephone network (PSTN) uses pre-computed routing tables, with fallback routes if the most direct route becomes blocked (see routing in the PSTN). Adaptive routing, or dynamic routing, attempts to solve this problem by constructing routing tables automatically, based on information carried by routing protocols, and allowing the network to act nearly autonomously in avoiding network failures and blockages.

Examples of adaptive-routing algorithms are the Routing Information Protocol (RIP) and the Open-Shortest-Path-First protocol (OSPF). Adaptive routing dominates the Internet. However, the configuration of the routing protocols often requires a skilled touch; networking technology has not developed to the point of the complete automation of routing.

## Distance vector algorithms

Distance vector algorithms use the Bellman-Ford algorithm. This approach assigns a number, the cost, to each of the links between each node in the network. Nodes will send information from point A to point B via the path that results in the lowest total cost (i.e. the sum of the costs of the links between the nodes used).

The algorithm operates in a very simple manner. When a node first starts, it only knows of its immediate neighbors, and the direct cost involved in reaching them. (This information, the list of destinations, the total cost to each, and the next hop to send data to get there, makes up the routing table, or distance table.) Each node, on a regular basis, sends to each neighbor its own current idea of the total cost to get to all the destinations it knows of. The neighboring node(s) examine this information, and compare it to what they already 'know'; anything which represents an improvement on what they already have, they insert in their own routing table(s). Over time, all the nodes in the network will discover the best next hop for all destinations, and the best total cost.

When one of the nodes involved goes down, those nodes which used it as their next hop for certain destinations discard those entries, and create new routing-table information. They then pass this information to all adjacent nodes, which then repeat the process. Eventually all the nodes in the network receive the updated information, and will then discover new paths to all the destinations which they can still "reach".

## Link-state algorithms

When applying link-state algorithms, each node uses as its fundamental data a map of the network in the form of a graph. To produce this, each node floods the entire network with information about what other nodes it can connect to, and each node then independently assembles this information into a map. Using this map, each router then independently determines the least-cost path from itself to every other node using a standard shortest paths algorithm such as Dijkstra's algorithm. The result is a tree rooted at the current node such that the path through the tree from the root to any other node is the least-cost path to that node. This tree then serves to construct the routing table, which specifies the best next hop to get from the current node to any other node.

## Optimized Link State Routing algorithm

A link-state routing algorithm optimized for mobile ad-hoc networks is the Optimized Link State Routing Protocol (OLSR).[1] OLSR is proactive; it uses Hello and Topology Control (TC) messages to discover and disseminate link state information through the mobile ad-hoc network. Using Hello messages, each node discovers 2-hop neighbor information and elects a set of multipoint relays (MPRs). MPRs distinguish OLSR from other link state routing protocols.

## Path vector protocol

Distance vector and link state routing are both intra-domain routing protocols. They are used inside an autonomous system, but not between autonomous systems. Both of these routing protocols become intractable in large networks and cannot be used in Inter-domain routing. Distance vector routing is subject to instability if there are more than a few hops in the domain. Link state routing needs huge amount of resources to calculate routing tables. It also creates heavy traffic because of flooding.

Path vector routing is used for inter-domain routing. It is similar to distance vector routing. In path vector routing we assume there is one node (there can be many) in each autonomous system which acts on behalf of the entire autonomous system. This node is called the speaker node. The speaker node creates a routing table and advertises it to neighboring speaker nodes in neighboring autonomous systems. The idea is the same as distance vector routing except that only speaker nodes in each autonomous system can communicate with each other. The speaker node advertises the path, not the metric of the nodes, in its autonomous system or other autonomous systems. Path vector routing is discussed in RFC 1322; the path vector routing algorithm is somewhat similar to the distance vector algorithm in the sense that each border router advertises the destinations it can reach to its neighboring router. However, instead of advertising networks in terms of a destination and the distance to that destination, networks are advertised as destination addresses and path descriptions to reach those destinations. A route is defined as a pairing between a destination and the attributes of the path to that destination, thus the name, path vector routing, where the routers receive a vector that contains paths to a set of destinations. The path, expressed in terms of the domains (or confederations) traversed so far, is carried in a special path attribute that records the sequence of routing domains through which the reach ability information has passed.

### Comparison of routing algorithms

Distance-vector routing protocols are simple and efficient in small networks and require little, if any, management. However, distance-vector algorithms do not scale well (due to the count-to-infinity problem), have poor convergence properties, and are based on a 'hop count' metric rather than a 'link-state' metric. They thus ignore bandwidth (a major drawback) when calculating the best path.

This has led to the development of more complex but more scalable algorithms for use in large networks. Interior routing mostly uses link-state routing protocols such as OSPF and IS-IS.

A more recent development is loop-free distance-vector protocols (e.g., EIGRP). Loop-free distance-vector protocols are as robust and manageable as distance-vector protocols but avoid counting to infinity, so they have good worst-case convergence times.

## Path selection

Path selection involves applying a routing metric to multiple routes, in order to select (or predict) the best route.

In the case of computer networking, the metric is computed by a routing algorithm, and can cover such information as bandwidth, network delay, hop count, path cost, load, MTU, reliability, and communication cost (see e.g. this survey for a list of proposed routing metrics). The routing table stores only the best possible routes, while link-state or topological databases may store all other information as well.

Because a routing metric is specific to a given routing protocol, multi-protocol routers must use some external heuristic in order to select between routes learned from different routing protocols. Cisco's routers, for example, attribute a value known as the administrative distance to each route, where smaller administrative distances indicate routes learned from a supposedly more reliable protocol.

A local network administrator, in special cases, can set up host-specific routes to a particular machine which provides more control over network usage, permits testing and better overall security. This can come in handy when required to debug network connections or routing tables.

## Multiple agents

In some networks, routing is complicated by the fact that no single entity is responsible for selecting paths: instead, multiple entities are involved in selecting paths or even parts of a single path. Complications or inefficiency can result if these entities choose paths to optimize their own objectives, which may conflict with the objectives of other participants.

A classic example involves traffic in a road system, in which each driver picks a path which minimizes their own travel time. With such routing, the equilibrium routes can be longer than optimal for all drivers. In particular, Brass paradox shows that adding a new road can lengthen travel times for all drivers.

In another model, for example used for routing automated guided vehicles (AGVs) on a terminal, reservations are made for each vehicle to prevent simultaneous use of the same part of an infrastructure. This approach is also referred to as context-aware routing.[3]

The Internet is partitioned into autonomous systems (ASs) such as internet service providers (ISPs), each of which has control over routes involving its network, at multiple levels. First, AS-level paths are selected via the BGP protocol, which produces a sequence of ASs through which packets will flow. Each AS may have multiple paths, offered by neighboring ASs, from which to choose. Its decision often involves business relationships with these neighboring ASs,[4] which may be unrelated to path quality or latency. Second, once an AS-level path has been selected, there are often multiple corresponding router-level paths, in part because two ISPs may be connected in multiple locations. In choosing the single router-level path, it is common practice for each ISP to employ hot-potato routing: sending traffic along the path that minimizes the distance through the ISP's own network—even if that path lengthens the total distance to the destination.

Consider two ISPs, A and B, which each have a presence in New York, connected by a fast link with latency 5 ms; and which each have a presence in London connected by a 5 ms link. Suppose both ISPs have trans-Atlantic links connecting their two networks, but A's link has latency 100 ms and B's has latency 120 ms. When routing a message from a source in A's London network to a destination in B's New York network, A may choose to immediately send the message to B in London. This saves A the work of sending it along an expensive trans-Atlantic link, but causes the message to experience latency 125 ms when the other route would have been 20 ms faster.

A 2003 measurement study of Internet routes found that, between pairs of neighboring ISPs, more than 30% of paths have inflated latency due to hot-potato routing, with 5% of paths being delayed by at least 12 ms. Inflation due to AS-level path selection, while substantial, was attributed primarily to BGP's lack of a mechanism to directly optimize for latency, rather than to selfish routing policies. It was also suggested that, were an appropriate mechanism in place, ISPs would be willing to cooperate to reduce latency rather than use hot-potato routing.[5]

## Route Analytics

As the Internet and IP networks become mission critical business tools, there has been increased interest in techniques and methods to monitor the routing posture of networks.

Incorrect routing or routing issues cause undesirable performance degradation, flapping and/or downtime. Monitoring routing in a network is achieved using Route analytics tools and techniques

**Routing Algorithms**

• Adaptive algorithm:

  - ❖ Reflect change in topology
  - ❖ Get information locally from adjacent routers

• Non Adaptive Algorithm

  - ❖ Static routers

**Principle of Optimality:**

If router I on optimal path from router I to K then optimal path from J to K also on same route!

Computer Networks Prof.

Routing Algorithms (Static)

• Set of all optimal routes from: Source to a given destination

  - ❖ A sink tree!

• Goal of routing algorithm find sink trees that are there!

• Shortest Path Routing:

  - ❖ Dijkstra
  - ❖ Uses topology
  - ❖ Greedy approach
  - ❖ Possible shorter path of equal length – need not be unique

**Static Routing Algorithms**

• Shortest path routing

  - ❖ To send a packet from one node to another find the shortest path between the pair of nodes

• Multipath Routing

  - ❖ Multiple paths from Node a to node b.
  - ❖ Randomly choose one of the paths

**I Multipath Routing**

• Forward traffic based on – a random number

• Example: Path from a to d

- ❖ via b: 0.0 - 0.65
- ❖ via f: 0.65 -1.0

• Packet for d from a:

- ❖ Generate a random number r:
- ❖ If $0 < r \leq 0.65$, choose b
- ❖ otherwise choose f

**Multipath Routing**

• Advantages:

- ❖ Reliability
- ❖ disjoint entries
- ❖ multiple routes possible

**Static Routing**

• Disadvantages:

- ❖ SSSP and Multipath:

• Require complete knowledge of Network topology to make a good decision.

• Hot potato routing

- ❖ Forward on to shortest Queue (defined by hopcount)
- ❖ Use hot potato with static routing
- ❖ rank = Shortest Queue + shortest path

**Distance Vector Routing**

• Distance Vector Routing:

• (Distributed Bellman Ford, Fulkerson)

- ❖ Each router maintain a table:
- ❖ destination, estimated cost, link, hop count, time delay in ms, queue length, …
- ❖ Updated by exchanging information between router - ICMP

**Dynamic Routing**

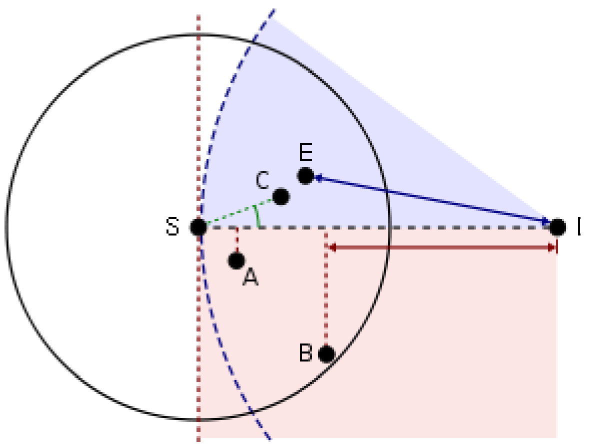
• Distributed Routing:

- ❖ Dynamic routing
- ❖ Changing topology of the network
- ❖ Need to recomputed route continuously

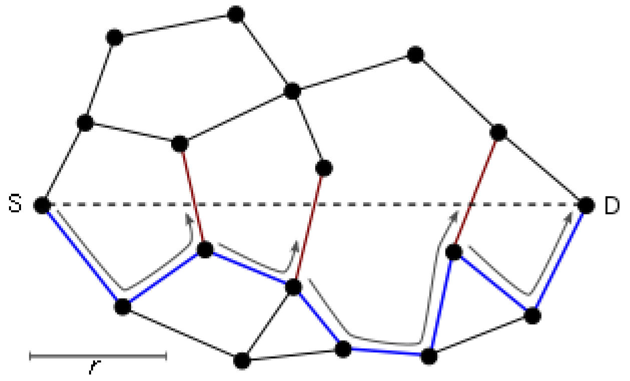
## Geographic routing

Geographic routing (also called georouting or position-based routing) is a routing principle that relies on geographic position information. It is mainly proposed for wireless networks and based on the idea that the source sends a message to the geographic location of the destination instead of using the network address. The idea of using position information for routing was first proposed in the 1980s in the area of packet radio networks [1] and interconnection networks Geographic routing requires that each node can determine its own location and that the source is aware of the location of the destination. With this information a message can be routed to the destination without knowledge of the network topology or a prior route discovery.

There are various approaches, such as single-path, multi-path and flooding-based strategies (see  for a survey). Most single-path strategies rely on two techniques: greedy forwarding and face routing. Greedy forwarding tries to bring the message closer to the destination in each step using only local information. Thus, each node forwards the message to the neighbor that is most suitable from a local point of view. The most suitable neighbor can be the one who minimizes the distance to the destination in each step (Greedy). Alternatively, one can consider another notion of progress, namely the projected distance on the source-destination-line (MFR, NFP), or the minimum angle between neighbor and destination (Compass Routing). Not all of these strategies are loop-free, i.e. a message can circulate among nodes in a certain constellation. It is known that the basic greedy strategy and MFR are loop free, while NFP and Compass Routing are not .[4]



Greedy forwarding variants: The source node (S) has different choices to find a relay node for further forwarding a message to the destination (D). A = Nearest with Forwarding Progress (NFP), B = Most Forwarding progress within Radius (MFR), C = Compass Routing, E = Greedy



Face routing: A message is routed along the interior of the faces of the communication graph, with face changes at the edges crossing the S-D-line (red). The final routing path is shown in blue.

Greedy forwarding can lead into a dead end, where there is no neighbor closer to the destination. Then, face routing helps to recover from that situation and find a path to another node, where greedy forwarding can be resumed. A recovery strategy such as face routing is necessary to assure that a message can be delivered to the destination. The combination of greedy forwarding and face routing was first proposed in 1999 under the name GFG (Greedy-Face-Greedy).[5] It guarantees delivery in the so-called unit disk graph network model. Various variants, which were proposed later, also for non-unit disk graphs, are based on the principles of GFG

**Election & Mutual Exclusion Algorithms**

**Election Algorithms**

• Many Distributed Systems require a single process to act as coordinator (for various reasons).

- ❖ Time server in the Berkley's algorithm
- ❖ Coordinator in the two-phase commit protocol

- ❖ Master process in distributed computations
- ❖ Master database server

• Coordinator may fail →  the distributed group of processes must execute anelection algorithm to determine a new coordinator process For simplicity, we assume the following:

• Processes each have a unique, positive identifier.

- ❖ Processor ID
- ❖ IP number

• All processes know all other process identifiers.

• The process with the highest valued identifier is duly elected coordinator.

**Goal of Election Algorithms**

The overriding goal of all election algorithms is to have all the processes in a group agree on a coordinator. **Bully**: “the biggest guy in town wins”.

**The “Bully” Election Algorithm (1)**

Assumes:

• Reliable message delivery (but processes may crash)

• The system is synchronous (timeouts to detect a process failure)

**The “Bully” Election Algorithm (2)**

When any process, P, notices that the coordinator is no longer responding it initiates an election:

• P sends an ELECTION message to all processes with higher id numbers.

• If no one responds, P wins the election and becomes coordinator.

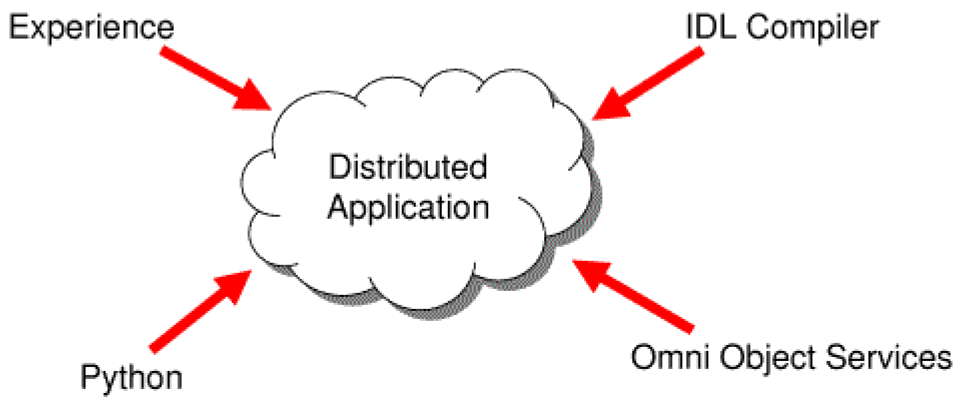
• If a higher process responds, it takes over. Process P’s job is done.

## Background

In our effort to engineer our research visions into real systems, a solid distributed system platform has always been an indispensible component in our toolbox. For over 10 years, we have built several generations of distributed systems middleware. Each middleware platform was built on the knowledge and experience gained from previous generations. Each has been put into use in real systems where robustness and high performance are of paramount importance.

Since 1997, our distributed systems research and deployment has been based upon CORBA, the industry standard specification for object-oriented middleware. We started our work with the development of omni ORB, our own CORBA ORB.

CORBA is a very powerful technology, but with that power comes complexity, meaning that CORBA can be difficult to use. Our recent work has focused on providing tools and other facilities to make CORBA easier to use and deploy. We have a four-pronged approach that helps us design new distributed applications:



## Experience

Our distributed systems experience is both deep and broad because we develop and build distributed systems technologies ***and*** use them. This experience has given us special insights into thorny issues such as scalability, security, naming, and interface evolutions.

Omni ORB is an industrial strength implementation of the CORBA standard, for the C++ language. Not only have we put it into use in our systems, we have also made it publicly available since 1997 under open source (GPL and LGPL) licensing conditions, with great success. It is one of the most standard compliant ORBs, and was one of the first products to pass the Open Group's standard conformance tests.

## Omni ORB for Python

Python is a high-level object-oriented interpreted language, renowned for its clear and concise syntax and its wide range of included library modules.

One of the most awkward aspects of using CORBA is the inconvenience and complexity of the standard language mappings to C++ and Java. Omni ORB ply brings the only complete Python binding to CORBA. The mapping is extremely simple -- the specification is only 16 pages long, compared to 130 for Java and 166 for C++. A simple CORBA interaction that would take 25 lines of C++ might take only 5 lines of Python code, for example.

Using Python dramatically simplifies CORBA programming. It has proved to be the ideal language to learn about CORBA, and it excels at programs used for testing and prototyping applications, as well as "glue" programs that tie other applications together. For some kinds and parts of applications, Python is a fine language for large-scale development too.

## IDL Compiler

The interactions between elements of a CORBA application are specified using an Interface Definition Language, IDL. The IDL definitions are compiled by an IDL compiler into declarations to be used in the application programming language.

One way to provide assistance to CORBA application developers is to make use of the IDL definitions to generate additional output, effectively writing some of their program for them. **omniidl** is a compiler framework that makes it easy to write new code generators, without having to repeatedly implement the inconvenient task of reading in and parsing the IDL.

The code generators we have developed range from simple aids such as generating fill-in-the-blanks object implementations, through automatic test case generation, to some complex additional features provided by the Omni Object Services.

Omni idle is publicly released as part of the Omni ORB distribution, and a number of outside projects make use of it. One notable use is within the ethereal packet analyzer, where Omni idle is used to generate interface-specific packet decoders, so sniffed CORBA network traffic can be decoded.

## Omni Object Services

The Omni Object Services are the final part of our work to make CORBA easier to use. The services stem from the realization that many of the hard problems in distributed systems crop up again and again, in slightly different forms. Attempts to provide generic solutions to these problems lead to services that are extremely awkward to use, which greatly diminishes their worth. The Omni Object Services avoid this problem by extensive use of Omni idle for code generation, so the services can be tailored for specific applications.

## Distributed Java Programming with RMI and CORBA

The Java Remote Method Invocation (RMI) mechanism and the Common Object Request Broker Architecture (CORBA) are the two most important and widely used distributed object systems. Each system has its own features and shortcomings. Both are being used in the industry for various applications ranging from e-commerce to health care. Selecting which of these two distribution mechanisms to use for a project is a tough task. This article presents an overview of RMI and CORBA, and more importantly it shows how to develop a useful application for downloading files from remote hosts. It then:

- Presents a brief overview of distributed object systems
- Provides a brief overview of RMI and CORBA
- Gives you a flavor of the effort involved in developing applications in RMI and CORBA
- Shows how to transfer files from remote machines using RMI and CORBA
- Provides a brief comparison of RMI and CORBA

## The Client/Server Model

The client/server model is a form of distributed computing in which one program (the client) communicates with another program (the server) for the purpose of exchanging information. In this model, both the client and server usually speak the same language -- a protocol that both the client and server understand -- so they are able to communicate.

While the client/server model can be implemented in various ways, it is typically done using low-level sockets. Using sockets to develop client/server systems means that we must design a protocol, which is a set of commands agreed upon by the client and server through which they will be able to communicate. As an example, consider the HTTP protocol that provides a method called GET, which must be implemented by all web servers and used by web clients (browsers) in order to retrieve documents.

## The Distributed Objects Model

A distributed object-based system is a collection of objects that isolates the requesters of services (clients) from the providers of services (servers) by a well-defined encapsulating interface. In other words, clients are isolated from the implementation of services as data representations and executable code. This is one of the main differences that distinguish the distributed object-based model from the pure client/server model.

In the distributed object-based model, a client sends a message to an object, which in turns interprets the message to decide what service to perform. This service, or method, selection could be performed by either the object or a broker. The Java Remote Method Invocation (RMI) and the Common Object Request Broker Architecture (CORBA) are examples of this model.

## RMI

RMI is a distributed object system that enables you to easily develop distributed Java applications. Developing distributed applications in RMI is simpler than developing with sockets since there is no need to design a protocol, which is an error-prone task. In RMI, the developer has the illusion of calling a local method from a local class file, when in fact the arguments are shipped to the remote target and interpreted, and the results are sent back to the callers.

## The Genesis of an RMI Application

Developing a distributed application using RMI involves the following steps:

1. Define a remote interface
2. Implement the remote interface
3. Develop the server
4. Develop a client
5. Generate Stubs and Skeletons, start the RMI registry, server, and client

We will now examine these steps through the development of a file transfer application.

**Example: File Transfer Application**

This application allows a client to transfer (or download) any type of file (plain text or binary) from a remote machine. The first step is to define a remote interface that specifies the signatures of the methods to be provided by the server and invoked by clients.

**Define a remote interface**

The remote interface for the file download application is shown in Code Sample 1. The interface File Interface provides one method download File that takes a String argument (the name of the file) and returns the data of the file as an array of bytes.

**Code Sample 1**: FileInterface.java

```
import java.rmi.Remote;

import java.rmi.RemoteException;



public interface FileInterface extends Remote {

   public byte[] downloadFile(String fileName) throws

   RemoteException;

}
```

Note the following characteristics about the File Interface:

- It must be declared public, in order for clients to be able to load remote objects which implement the remote interface.

- It must extend the Remote interface, to fulfill the requirement for making the object a remote one.
- Each method in the interface must throw a java.rmi.RemoteException.

**Implement the remote interface**

The next step is to implement the interface File Interface. A sample implementation is shown in Code Sample 2. Note that in addition to implementing the FileInterface, the FileImpl class is extending the UnicastRemoteObject. This indicates that the FileImpl class is used to create a single, non-replicated, remote object that uses RMI's default TCP-based transport for communication.

**Code Sample 2**: FileImpl.java

```
import java.io.*;

import java.rmi.*;

import java.rmi.server.UnicastRemoteObject;



public class FileImpl extends UnicastRemoteObject

  implements FileInterface {



   private String name;



   public FileImpl(String s) throws RemoteException{

      super();

      name = s;
```

```
    }



    public byte[] downloadFile(String fileName){

        try {

            File file = new File(fileName);

            byte buffer[] = new byte[(int)file.length()];

            BufferedInputStream input = new

        BufferedInputStream(new FileInputStream(fileName));

            input.read(buffer,0,buffer.length);

            input.close();

            return(buffer);

        } catch(Exception e){

            System.out.println("FileImpl: "+e.getMessage());

            e.printStackTrace();

            return(null);

        }

    }

}
```

**Develop the server**

The third step is to develop a server. There are three things that the server needs to do:

1. Create an instance of the RMISecurityManager and install it
2. Create an instance of the remote object (FileImpl in this case)
3. Register the object created with the RMI registry. A sample implementation is shown in Code Sample 3.

**Code Sample 3**: FileServer.java

```
import java.io.*;

import java.rmi.*;

public class FileServer {

   public static void main(String argv[]) {

      if(System.getSecurityManager() == null) {

         System.setSecurityManager(new RMISecurityManager());

      }

      try {

         FileInterface fi = new FileImpl("FileServer");

         Naming.rebind("//127.0.0.1/FileServer", fi);

      } catch(Exception e) {

         System.out.println("FileServer: "+e.getMessage());

         e.printStackTrace();

      }   }}
```

The statement Naming.rebind("//127.0.0.1/FileServer", fi) assumes that the RMI registry is running on the default port number, which is 1099. However, if you run the RMI registry on a different port number it must be specified in that statement. For example, if the RMI registry is running on port 4500, then the statement becomes:

```
Naming.rebind("//127.0.0.1:4500/FileServer", fi)
```

Also, it is important to note here that we assume the rmi registry and the server will be running on the same machine. If they are not, then simply change the address in the rebind method.

**Develop a client**

The next step is to develop a client. The client remotely invokes any methods specified in the remote interface (FileInterface). To do so however, the client must first obtain a reference to the remote object from the RMI registry. Once a reference is obtained, the downloadFile method is invoked. A client implementation is shown in Code Sample 4. In this implementation, the client accepts two arguments at the command line: the first one is the name of the file to be downloaded and the second one is the address of the machine from which the file is to be downloaded, which is the machine that is running the file server.

**Code Sample 4**: FileClient.java

```
import java.io.*;

import java.rmi.*;

public class FileClient{

  public static void main(String argv[]) {

    if(argv.length != 2) {

      System.out.println("Usage: java FileClient fileName machineName");

      System.exit(0);
    }

    try {

       String name = "//" + argv[1] + "/FileServer";

       FileInterface fi = (FileInterface) Naming.lookup(name);

       byte[] filedata = fi.downloadFile(argv[0]);

       File file = new File(argv[0]);

       BufferedOutputStream output = new

         BufferedOutputStream(new FileOutputStream(file.getName()));

       output.write(filedata,0,filedata.length);

       output.flush();

       output.close();

    } catch(Exception e) {

       System.err.println("FileServer exception: "+ e.getMessage());

       e.printStackTrace();

    }   }}
```

**Running the Application**

In order to run the application, we need to generate stubs and skeletons, compile the server and the client, start the RMI registry, and finally start the server and the client.

To generate stubs and skeletons, use the rmic compiler:

**prompt>** rmic FileImpl

This will generate two files: FileImpl_Stub.class and FileImpl_Skel.class. The stub is a client proxy and the skeleton is a server skeleton.

The next step is to compile the server and the client. Use the javac compiler to do this. Note however, if the server and client are developed on two different machines, in order to compile the client you need a copy of the interface (FileInterface).

Finally, it is time to start the RMI registry and run the server and client. To start the RMI registry on the default port number, use the command rmiregistry or start rmiregistry on Windows. To start the RMI registry on a different port number, provide the port number as an argument to the RMI registry:

```
prompt> rmiregistry portNumber
```

Once the RMI registry is running, you can start the server FileServer. However, since the RMI security manager is being used in the server application, you need a security policy to go with it. Here is a sample security policy:

```
grant {
  permission java.security.AllPermission "", "";
};
```

**Note**: this is just a sample policy. It allows anyone to do anything. For your mission critical applications, you need to specify more constraint security policies.

Now, in order to start the server you need a copy of all the classes (including stubs and skeletons) except the client class (FileClient.class). To start the server use the following command, assuming that the security policy is in a file named policy.txt:

```
prompt> java -Djava.security.policy=policy.txt FileServer
```

To start the client on a different machine, you need a copy of the remote interface (FileInterface.class) and stub (FileImpl_Stub.class). To start the client use the command:

```
prompt> java FileClient fileName machineName
```

where fileName is the file to be downloaded and machineName is the machine where the file is located (the same machine runs the file server). If everything goes ok then the client exists and the file downloaded is on the local machine.

To run the client we mentioned that you need a copy of the interface and stub. A more appropriate way to do this is to use RMI dynamic class loading. The idea is you do not need copies of the interface and the stub. Instead, they can be located in a shared directory for the server and the client, and whenever a stub or a skeleton is needed, it is downloaded automatically by the RMI class loader. To do this you run the client, for example, using the following command: java -Djava.rmi.server.codebase=http://hostname/locationOfClasses FileClient fileName machineName. For more information on this, please see Dynamic Code Loading using RMI.

## CORBA

The Common Object Request Broker Architecture (or CORBA) is an industry standard developed by the Object Management Group (OMG) to aid in distributed objects programming. It is important to note that CORBA is simply a specification. A CORBA implementation is known as an ORB (or Object Request Broker). There are several CORBA implementations available on the market such as VisiBroker, ORBIX, and others. Java IDL is another implementation that comes as a core package with the JDK1.3 or above.

CORBA was designed to be platform and language independent. Therefore, CORBA objects can run on any platform, located anywhere on the network, and can be written in any language that has Interface Definition Language (IDL) mappings.

Similar to RMI, CORBA objects are specified with interfaces. Interfaces in CORBA, however, are specified in IDL. While IDL is similar to C++, it is important to note that IDL is not a programming language. For a detailed introduction to CORBA, please see Distributed Programming with Java: Chapter 11 (Overview of CORBA).

### Define the Interface

When defining a CORBA interface, think about the type of operations that the server will support. In the file transfer application, the client will invoke a method to download a file. Code Sample 5 shows the interface for File Interface. Data is a new type introduced using the type def keyword. A sequence in IDL is similar to an array except that a sequence does not have a fixed size. An octet is an 8-bit quantity that is equivalent to the Java type byte.

Note that the download dFile method takes one parameter of type string that is declared in. IDL defines three parameter-passing modes: in (for input from client to server), out (for output from server to client), and inout (used for both input and output).

**Code Sample 5**: FileInterface.idl

```
interface FileInterface {
  typedef sequence<octet> Data;
  Data downloadFile(in string fileName);
};
```

Once you finish defining the IDL interface, you are ready to compile it. The JDK1.3+ comes with the idlj compiler, which is used to map IDL definitions into Java declarations and statements.

The idlj compiler accepts options that allow you to specify if you wish to generate client stubs, server skeletons, or both. The -f<side> option is used to specify what to generate. The side can be client, server, or all for client stubs and server skeletons. In this example, since the application will be running on two separate machines, the -fserver option is used on the server side, and the -fclient option is used on the client side.

Now, let's compile the FileInterface.idl and generate server-side skeletons. Using the command:

```
prompt> idlj -server FileInterface.idl
```

This command generates several files such as skeletons, holder and helper classes, and others. An important file that gets generated is the _File Interface Impl Base, which will be sub classed by the class that implements the interface.

**Implement the interface**

Now, we provide an implementation to the downloadFile method. This implementation is known as a servant, and as you can see from Code Sample 6, the class FileServant extends the _FileInterfaceImplBase class to specify that this servant is a CORBA object.

**Code Sample 6**: FileServant.java

```
import java.io.*;


public class FileServant extends _FileInterfaceImplBase {

   public byte[] downloadFile(String fileName){

      File file = new File(fileName);

      byte buffer[] = new byte[(int)file.length()];

      try {

         BufferedInputStream input = new

           BufferedInputStream(new FileInputStream(fileName));

         input.read(buffer,0,buffer.length);

         input.close();

      } catch(Exception e) {

         System.out.println("FileServant Error: "+e.getMessage());

         e.printStackTrace();

      }
      return(buffer);

   }

}
```

**Develop the server**

The next step is developing the CORBA server. The FileServer class, shown in Code Sample 7, implements a CORBA server that does the following:

1. Initializes the ORB
2. Creates a FileServant object
3. Registers the object in the CORBA Naming Service (COS Naming)
4. Prints a status message
5. Waits for incoming client requests

**Code Sample 7**: FileServer.java

```
import java.io.*;

import org.omg.CosNaming.*;

import org.omg.CosNaming.NamingContextPackage.*;

import org.omg.CORBA.*;



public class FileServer {

   public static void main(String args[]) {
```

```
  try{

    // create and initialize the ORB

                              ORB orb = ORB.init(args, null);

    // create the servant and register it with the ORB

    FileServant fileRef = new FileServant();

    orb.connect(fileRef);

    // get the root naming context

    org.omg.CORBA.Object objRef =

      orb.resolve_initial_references("NameService");

    NamingContext ncRef = NamingContextHelper.narrow(objRef);

    // Bind the object reference in naming

    NameComponent nc = new NameComponent("FileTransfer", " ");

    NameComponent path[] = {nc};

    ncRef.rebind(path, fileRef);

    System.out.println("Server started....");

    // Wait for invocations from clients

    java.lang.Object sync = new java.lang.Object();

    synchronized(sync){

      sync.wait();

    }
```

```
      } catch(Exception e) {
         System.err.println("ERROR: " + e.getMessage());
         e.printStackTrace(System.out);
      }
   }
}
```

Once the File Server has an ORB, it can register the CORBA service. It uses the COS Naming Service specified by OMG and implemented by Java IDL to do the registration. It starts by getting a reference to the root of the naming service. This returns a generic CORBA object. To use it as a Naming Context object, it must be narrowed down (in other words, casted) to its proper type, and this is done using the statement:

```
NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

The ncRef object is now an org.omg.CosNaming.NamingContext. You can use it to register a CORBA service with the naming service using the rebind method.

**Develop a client**

The next step is to develop a client. An implementation is shown in Code Sample 8. Once a reference to the naming service has been obtained, it can be used to access the naming service and find other services (for example the FileTransfer service). When the FileTransfer service is found, the downloadFile method is invoked.

**Code Sample 8**: FileClient

```
import java.io.*;

import java.util.*;

import org.omg.CosNaming.*;

import org.omg.CORBA.*;


public class FileClient {

   public static void main(String argv[]) {

      try {

         // create and initialize the ORB

         ORB orb = ORB.init(argv, null);

         // get the root naming context

         org.omg.CORBA.Object objRef =

            orb.resolve_initial_references("NameService");

         NamingContext ncRef = NamingContextHelper.narrow(objRef);

         NameComponent nc = new NameComponent("FileTransfer", " ");

         // Resolve the object reference in naming

         NameComponent path[] = {nc};

         FileInterfaceOperations fileRef =

            FileInterfaceHelper.narrow(ncRef.resolve(path));
```

```
    if(argv.length < 1) {
       System.out.println("Usage: java FileClient filename");
    }
    // save the file
    File file = new File(argv[0]);
    byte data[] = fileRef.downloadFile(argv[0]);
    BufferedOutputStream output = new
      BufferedOutputStream(new FileOutputStream(argv[0]));
    output.write(data, 0, data.length);
    output.flush();
    output.close();
  } catch(Exception e) {
    System.out.println("FileClient Error: " + e.getMessage());
    e.printStackTrace();
  }   }}
```

**Running the application**

The final step is to run the application. There are several sub-steps involved:

1. *Running the the CORBA naming service*. This can be done using the command tnameserv. By default, it runs on port 900. If you cannot run the naming service on this port, then you can start it on another port. To start it on port 2500, for example, use the following command: **prompt>** tnameserv -ORBinitialPort 2500

2. *Start the server*. This can be done as follows, assuming that the naming service is running on the default port number:

```
prompt> java FileServer
```

If the naming service is running on a different port number, say 2500, then you need to specify the port using the ORBInitialPort option as follows:

```
prompt> java FileServer -ORBInitialPort 2500
```

3. *Generate Stubs for the client*. Before we can run the client, we need to generate stubs for the client. To do that, get a copy of the FileInterface.idl file and compile it using the idlj compiler specifying that you wish to generate client-side stubs, as follows:

```
prompt> idlj -fclient FileInterface.idl
```

4. *Run the client*. Now you can run the client using the following command, assuming that the naming service is running on port 2500.

```
prompt> java FileClient hello.txt -ORBInitialPort 2500
```

Where hello.txt is the file we wish to download from the server.

**Note**: if the naming service is running on a different host, then use the -ORBInitialHost option to specify where it is running. For example, if the naming service is running on port number 4500 on a host with the name gosling, then you start the client as follows:

```
prompt> java FileClient hello.txt -ORBInitialHost gosling -ORBInitialPort 4500
```

Alternatively, these options can be specified at the code level using properties. So instead of initializing the ORB as:

```
ORB orb = ORB.init(argv, null);
```

It can be initialized specifying that the CORBA server machine (called gosling) and the naming service's port number (to be 2500) as follows:

```
 roperties props = new Properties();
props.put("org.omg.CORBA.ORBInitialHost", "gosling");
props.put("orb.omg.CORBA.ORBInitialPort", "2500");
ORB orb = ORB.init(args, props);
```

Code-wise, it is clear that RMI is simpler to work with since the Java developer does not need to be familiar with the Interface Definition Language (IDL). In general, however, CORBA differs from RMI in the following areas:

- CORBA interfaces are defined in IDL and RMI interfaces are defined in Java. RMI-IIOP allows you to write all interfaces in Java (see RMI-IIOP).
- CORBA supports in and out parameters, while RMI does not since local objects are passed by copy and remote objects are passed by reference.
- CORBA was designed with language independence in mind. This means that some of the objects can be written in Java, for example, and other objects can be written in C++ and yet they all can interoperate. Therefore, CORBA is an ideal mechanism for bridging islands between different programming languages. On the other hand, RMI was designed for a single language where all objects are written in Java. Note however, with RMI-IIOP it is possible to achieve interoperability.
    - CORBA objects are not garbage collected. As we mentioned, CORBA is language independent and some languages (C++ for example) does not support garbage collection. This can be considered a disadvantage since once a CORBA object is created, it continues to exist until you get rid of it, and deciding when to get rid of an object is not a trivial task. On the other hand, RMI objects are garbage collected automatically.