UNIT-05

DISTRIBUTED ALGORITHMS

UNIT- 05/LECTURE- 01

Distributed Algorithms

A process (a node in a computer network) is in general not connected directly to every other process by a channel. A node can send packets of information directly only to a subset of the nodes called the neighbours of the node. Routing is the term used to describe the decision procedure by which a node selects one (or, sometimes, more) of its neighbours to forward a packet on its way to an ultimate destination. The objective in designing a routing algorithm is to generate (for each node) a decision-making procedure to perform this function and guarantee delivery of each packet.

It will be clear that some information about the topology of the network must be stored in each node as a working basis for the (local) decision procedure; we shall refer to this information as the routing tables. With the introduction of these tables the routing problem can be algorithmically divided into two parts; the definition of the table structure is of course related to the algorithmically design.

(1) Table computation. The routing tables must be computed when the network is initialized and must be brought up to date if the topology of the network changes.

(2) Packet forwarding. When a packet is to be sent through the network it must be forwarded using the routing tables.

Criteria for “good” routing methods include the following.

(1) Correctness. The algorithm must deliver every packet offered to the network to its ultimate destination.

(2) Efficiency. algorithm must send packets through “good” paths, e.g., paths that suffer

only a small delay and ensure high throughput of the entire network. An algorithm is called optimal if it uses the "best" paths.

(3) Complexity. algorithm for the computation of the tables must use as few messages, time, and storage as possible. Other aspects of complexity are how fast a routing decision can be made, how fast a packet can be made ready for transmission, etc., but these aspects will receive less attention in this chapter.

(4) Robustness. In the case of a topological change (the addition or removal of a channel or node) the algorithm updates the routing tables in order to perform the routing function in the modified network.

(5) Adaptiveness. algorithm balances the load of channels and nodes by adapting the tables in order to avoid paths through channels or nodes that are very busy, preferring channels and nodes with a currently light load.

(6) Fairness. algorithm must provide service to every user in the same degree.

These criteria are sometimes conflicting, and most algorithms perform well only w.r.t. a subset of them.

As usual, a network is represented as a graph, where the nodes of the graph are the nodes of the network, and there is an edge between two nodes if they are neighbours (i.e., they have a communication channel between them). The optimality of an algorithm depends on what is called a "best" path in the graph; there are several notions of what is "best", each with its own class of routing algorithms:

(1) Minimum hop. cost of using a path is measured as the number of hops (traversed channels or steps from node to node) of the path. A minimum-hop routing algorithm uses a path with the smallest possible number of hops.

(2) Shortest path. Each channel is statically assigned a (non-negative) weight, and the cost

of a path is measured as the sum of the weights of the channels in the path. A shortest-path algorithm uses a path with lowest possible cost.

(3) Minimum delay. Each channel is dynamically assigned a weight, depending on the traffic on the channel. A minimum-delay algorithm repeatedly revises the tables in such a way that paths with a (near) minimal total delay are always chosen. As the delays encountered on the channels depend on the actual traffic, the various packets transmitted through the network influence each other.

Other notions of the optimality of paths may be useful in special applications.

Destination-based Routing ([RGPV/June 2014(4)]

The routing decision made when forwarding a packet is usually based only on the destination of the packet (and the contents of the routing tables), and is independent of the original sender (the source) of the packet. Routing can ignore the source and still use optimal paths. The results do not depend on the choice of a particular optimality criterion for paths, but the following assumptions must hold. (Recall that a path is simple if it contains each node at most once, and the path is a cycle if the first node equals the last node.)

(1) The cost of sending a packet via a path P is independent of the actual utilization of the path, in particular, the use of edges of P by other messages. This assumption allows us to regard the cost of using path P as a function of the path; thus denote the cost of P by $C(P) \in \mathbb{R}$.

(2) The cost of the concatenation of two paths equals the sum of the costs of the concatenated paths, i.e., for all i = 0, …, k,

$C(\langle u_0, u_1, \ldots, u_k \rangle) = C(\langle u_0, \ldots, u_i \rangle) + C(\langle u_i, \ldots, u_k \rangle)$.

Consequently, the cost of the empty path (uo) (this is a path from uo to uo) satisfies C((uo))

= 0.

(3) The graph does not contain a cycle of negative cost.

(These criteria are satisfied by minimum-hop and shortest-path cost criteria.) A path from u to υ is called optimal if there exists no path from uto v with lower cost. Observe that an optimal path is not always unique; there may exist different paths with the same (minimal) cost.

Algorithm DESTINATION-BASED FORWARDING (FOR NODE u).

Lemma The forwarding mechanism delivers every packet at its destination if and only if the routing tables are cycle-free.

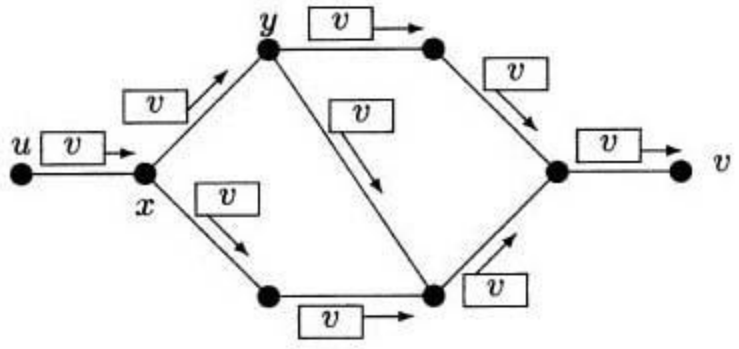
Proof. If the tables contain a cycle for some destination d a packet for d is never delivered if its source is a node in the cycle.

Assume the tables are cycle-free and let a packet with destination d (and source u_0) be forwarded via u_o,u_1,u_2, … If the same node occurs twicein this sequence, say $u_i = u_j$, then the tables contain a cycle, namely $\langle u_i, \ldots, u_j \rangle$, contradicting the assumption that the tables are cycle-free.Thus, each node occurs at most once, which implies that this sequence isfinite, ending, say, in node u_k($k < N$). According to the forwarding procedure the sequence can only end in d, i.e., $u_k = d$ and the packet has reached its destination in at most N — 1 hops.

In some routing algorithms it is the case that the tables are not cycle-free during their computation, but only when the table computation phase has finished. When such an algorithm is used, a packet may traverse a cycle during computation of the tables, but reaches its destination in at most N — 1 hops after completion of the table computation if topological changes cease. If topological changes do not cease, i.e., the network is subject to an infinite sequence of topological changes, packets do not necessarily reach their destination even if tables are cycle-free during updates.

Bifurcated routing for minimum delay

If routing via minimum-delay paths is required, and the delay of a channel depends on its utilization (thus assumption (1) at the beginning of this section is not valid), the cost of using a path cannot simply be assessed as a function of this path alone. In addition, the traffic on the channel must be taken into account. To avoid congestion (and the resulting higher delay) on a path, it is usually necessary to send packets having the same source-destination pair via different paths; the traffic for this pair "splits" at one or more intermediate nodes as depicted in Figure 4.3.Routing methods that use different paths towards the same destination are called multiple-path or bifurcated routing methods.



Traffic for node v splits at nodes x and y.

EXAMPLE OF BIFURCATED ROUTING.

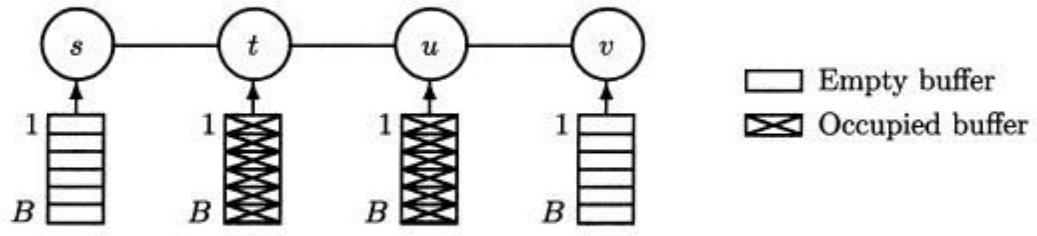
S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain the following (A)destination based routing.	June 2014	4

UNIT- 05/LECTURE- 02

Deadlock free Packet switching ([RGPV/June 2014(4)]

Messages (packets) travelling through a packet-switched communication network must be stored at each node before being forwarded to the next node on the path to their destination. Each node of the network reserves some buffer space for this purpose. As the amount of buffer space is finite in each node, situations may occur where no packet can be forwarded because all buffers in the next node are occupied, as illustrated by Figure. Each of the four nodes has B buffers, each capable of containing exactly one packet. Node B has sent B packets with destination υ to t, and node υ has sent B packets with destination s to u. All buffers in u and υ are now occupied, and consequently none of the packets stored in t and u can be forwarded towards its destination.

Situations where a group of packets can never reach their destination because they are all waiting for the use of a buffer currently occupied by another packet in the group are referred to as store-and-forward deadlocks. (Other types of deadlock will be discussed briefly at the end of this chapter.) An important problem in the design of packet-switching networks is how to deal with store-and-forward deadlocks. In this chapter we shall treat several methods, referred to as controllers, that can be used to avoid the possibility of store-and-forward deadlocks by introducing restrictions on when a packet can be generated or forwarded. Methods of avoiding store-and-forward deadlocks are found in the network layer of the OSI reference model.



AN EXAMPLE OF A STORE-AND-FORWARD DEADLOCK.

Two kinds of method will be discussed, based on structured and unstructured buffer pools. Methods using structured buffer pools will identify for a node and a packet a specific buffer that must be taken if a packet is generated or received. If this buffer is occupied, the packet cannot be accepted. In methods using unstructured buffer pools (Section 5.3) all buffers are equal; the method only prescribes whether or not a packet can be accepted, but does not determine in which buffer it must be placed.

Introduction

As usual, the network is modelled by a graph G = (V, E); the distance between nodes is measured in hops. Each node has B buffers for temporarily storing packets. The set of all buffers is denoted $\mathcal{B}$, and the symbols b, c, b_u, etc., are used to denote buffers.

The handling of packets by the nodes is described by the following three types of moves that can occur in the network.

(1) Generation. A node u “creates” a new packet p (actually by accepting the packet from a higher level protocol) and places it in an empty buffer in u. The node u is called the source of p in this case.

(2) Forwarding. A packet p is forwarded from a node u to an empty buffer in the next node w on its route (the route is determined by the routing algorithm used). As a result of the move the buffer previously occupied by p becomes empty. Although the controllers that we shall define may forbid moves, it is assumed that the network always allows this move, i.e.,

if the controller does not forbid it, it is applicable.

In systems with synchronous message passing this move is easily seen to be a single transition. In systems with asynchronous message passing the move is not a single transition, but it can be implemented, for example, as follows. Node u repeatedly transmits p to w, but does not discard the packet from the buffer as long as no acknowledgement is received. When node w receives the packet it decides whether it will accept the packet in one of its buffers. If so, the packet is placed in the buffer and an acknowledgement is sent to u, otherwise the packet is simply ignored. Of course, more efficient protocols can be designed to implement the move, for example those where u does not transmit p until u knows that w will accept p. In either case the move consists of several transitions of the types , but it will be considered as a single step for the purpose of this chapter.

(3) Consumption. A packet p occupying a buffer in its destination node is removed from the buffer. It is assumed that the network always allows this move.

Denote by $\mathcal{P}$ the collection of all paths followed by the packets. This collection is determined by the routing algorithm ,how it is determined need not concern us here. Let k be the number of hops in the longest path in $\mathcal{P}$. It is not assumed that k equals the diameter G; k may exceed the diameter if the routing algorithm does not select minimum-hop paths, and k may be smaller than the diameter if all communication in G is between nodes at limited distances.

As is seen from the example given at the beginning of this chapter, dead-locks may arise if all moves are allowed to occur unrestrictedly (barring the trivial restriction that u must have an empty buffer if a packet is generated in u and w must have an empty buffer if a packet is forwarded to w). We shall now define a controller as an algorithm that permits or forbids various moves in the network, subject to the following requirements.

(1) The consumption of a packet (at its destination) is always allowed.

(2) The generation of a packet in a node where all buffers are empty is always allowed.

(3) The controller uses only local information, i.e., the decision whether a packet can be accepted in node u depends only on information known to u or contained in the packet.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain the following terms(i) deadlock free packet switching.	June 2014	4

UNIT -05/LECTURE -03

Traversal Algorithms

The wave algorithms in which all events of a wave are totally ordered by the causality relation and in which the last event occurs in the same process as the first event.

Definition A traversal algorithm is an algorithm with the following three properties.

(1) In each computation there is one initiator, which starts the algorithm by sending out exactly one message.

(2) A process, upon receipt of a message, either sends out one message or decides.

The first two properties imply that in each finite computation exactly one process decides. The algorithm is said to terminate in the single process that decides.

(3) The algorithm terminates in the initiator and when this happens, each process has sent a message at least once.

In each reachable configuration of a traversal algorithm there is either exactly one message in transit, or exactly one process that has just received a message and not (yet) sent a message in reply. In a more abstract view the messages of a computation taken together can be seen as a single object (a token) that is handed from process to process and so "visits" all processes.

Election Algorithms ([RGPV/ June 2013 (7), [RGPV/ June 2014 (4)])
The election problem was first posed, who also proposed the first solution. The problem is to start from a configuration where all processes are in the same state, and arrive at a configuration where exactly one process is in state leader and all other processes are in the state lost.
An algorithm for choosing a unique process to play a particular role is called an election

algorithm. For example, in a variant of our central-server algorithm for mutual exclusion, the 'server' is chosen from among the processes that need to use the critical section. An election algorithm is needed for this choice. It is essential that all the processes agree on the choice. Afterwards, if the process that plays the role of server wishes to retire then another election is required to choose a replacement.

An election under the processes must be held if a centralized algorithm is to be executed and there is no a priori candidate to serve as the initiator of this algorithm. For example, this could be the case for an initialization procedure that must be executed initially or after a crash of the system. Because the set of active processes may not be known in advance it is not possible to assign one process once and for all to the role of leader.

Definition An election algorithm is an algorithm that satisfies the following properties.

(1) Each process has the same local algorithm.

(2) The algorithm is decentralized, i.e., a computation can be initialized by an arbitrary non-empty subset of the processes.

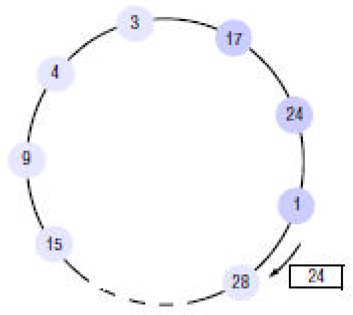
(3) The algorithm reaches a terminal configuration in each computation, and in each reachable terminal configuration there is exactly one process in the state leader and all other processes are in the state lost.

The last property is sometimes weakened to require only that exactly one process is in the state leader. It is then the case that the elected process is aware that it has won the election, but the losers are not (yet) aware of their loss. If an algorithm satisfying this weaker requirement is given, it can easily be extended by a flooding, initiated by the leader, in which all processes are informed of the result of the election. This additional notification is omitted in some algorithms in this chapter.

A ring-based election algorithm ([RGPV/Dec 2012 (7)]

The algorithm of Chang and Roberts is suitable for a collection of processes arranged in a logical ring. Each process pi has a communication channel to the next process in the ring, p(i + 1)mod N , and all messages are sent clockwise around the ring. We assume that no failures occur, and that the system is asynchronous. The goal of this algorithm is to elect a single process called the coordinator, which is the process with the largest identifier.

Initially, every process is marked as a non-participant in an election. Any process can begin an election. It proceeds by marking itself as a participant, placing its identifier in an election message and sending it to its clockwise neighbour. When a process receives an election message, it compares the identifier in the message with its own. If the arrived identifier is greater, then it forwards the message to its neighbour. If the arrived identifier is smaller and the receiver is not a participant, then it substitutes its own identifier in the message and forwards it; but it does not forward the message if it is already a participant. On forwarding an election message in any case, the process marks itself as a participant.



Note: The election was started by process 17. The highest process identifier encountered so far is 24. Participant processes are shown in a darker tint.

If, however, the received identifier is that of the receiver itself, then this process's identifier must be the greatest, and it becomes the coordinator. The coordinator marks itself as a non-participant once more and sends an elected message to its neighbour, announcing its election and enclosing its identity. When a process pi receives an elected message, it marks itself as a nonparticipant, sets its variable elected to the identifier in the message and, unless it is the new coordinator, forwards the message to its neighbour.

It is easy to see that condition E1 is met. All identifiers are compared, since a process must receive its own identifier back before sending an elected message. For any two processes, the one with the larger identifier will not pass on the other's identifier. It is therefore impossible that both should receive their own identifier back. Condition E2 follows immediately from the guaranteed traversals of the ring (there are no failures). Note how the non-participant and participant states are used so that duplicate messages arising when two processes start an election at the same time are extinguished as soon as possible, and always before the 'winning' election result has been announced.

If only a single process starts an election, then the worst-performing case is when its anti-clockwise neighbour has the highest identifier. A total of N – 1 messages are then required to reach this neighbour, which will not announce its election until its identifier has completed another circuit, taking a further N messages. The elected message is then sent N times, making 3N – 1 messages in all. The turnaround time is also 3N – 1 , since these messages are sent sequentially.

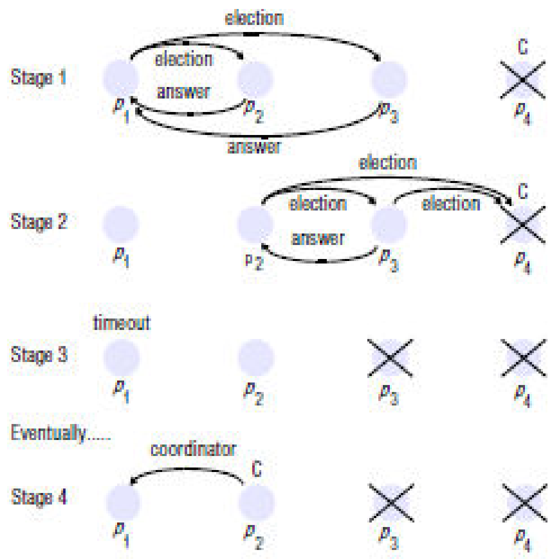
An example of a ring-based election in progress is shown. The election message currently contains 24, but process 28 will replace this with its identifier when the message reaches it.
While the ring-based algorithm is useful for understanding the properties of election algorithms in general, the fact that it tolerates no failures makes it of limited practical value. However, with a reliable failure detector it is in principle possible to reconstitute the ring when a process crashes.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	What is election algorithm? Suppose that two processes detect the demise of the coordinator simultaneously and both decide to hold an election using the bully algorithm. In this situation what happens.	June 2013	7
Q.2	Describe the Ring election algorithm with example. Discuss the complexity of ring election algorithm in terms of message.	Dec 2012	7

UNIT -05/LECTURE -04

The bully algorithm ([RGPV/June 2012(4)]

The bully algorithm allows processes to crash during an election, although it assumes that message delivery between processes is reliable. Unlike the ring-based algorithm, this algorithm assumes that the system is synchronous: it uses timeouts to detect a process failure. Another difference is that the ring-based algorithm assumed that processes have minimal a priori knowledge of one another: each knows only how to communicate with its neighbour, and none knows the identifiers of the other processes. The bully algorithm, on the other hand, assumes that each process knows which processes have higher identifiers, and that it can communicate with all such processes.



The election of coordinator p_2, after the failure of p_4 and then p_3

There are three types of message in this algorithm: an election message is sent to announce an election; an answer message is sent in response to an election message and a coordinator message is sent to announce the identity of the elected process – the new 'coordinator'. A process begins an election when it notices, through timeouts, that the coordinator has failed. Several processes may discover this concurrently. Since the system is synchronous, we can construct a reliable failure detector. There is a maximum message transmission delay, Ttrans , and a maximum delay for processing a message Tprocess . Therefore, we can calculate a time T = 2Ttrans + Tprocess that is an upper bound on the time that can elapse between sending a message to another process and receiving a response. If no response arrives within time T, then the local failure detector can report that the intended recipient of the request has failed.

The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a coordinator message to all processes with lower identifiers. On the other hand, a process with a lower identifier can begin an election by sending an election message to those processes that have a higher identifier and awaiting answer messages in response. If none arrives within time T, the process considers itself the coordinator and sends a coordinator message to all processes with lower identifiers announcing this. Otherwise, the process waits a further period T for a coordinator message to arrive from the new coordinator. If none arrives, it begins another election.

If a process *pi* receives a *coordinator* message, it sets its variable *electedi* to the identifier of the coordinator contained within it and treats that process as the coordinator. If a process receives an *election* message, it sends back an *answer* message and begins another election – unless it has begun one already. When a process is started to replace a crashed process, it begins an election. If it has the highest process identifier, then it will decide that it is the coordinator and announce this to the other processes. Thus it will become the coordinator, even though the current coordinator is functioning. It is for this reason that the algorithm is called the 'bully' algorithm. There are four processes, *p*1 –*p*4 . Process *p*1 detects the failure of the coordinator *p*4 and announces an election. On receiving an *election* message from *p*1 , processes *p*2 and *p*3 send *answer* messages to *p*1 and begin their own elections; *p*3 sends an *answer* message to *p*2 , but *p*3 receives no *answer* message from the failed process *p*4 (stage2). It therefore decides that it is the coordinator. But before it can send out the *coordinator* message, it too fails (stage 3). When *p*1 ’s timeout period *T* expires (which we assume occurs before *p*2 ’s timeout expires), it deduces the absence of a *coordinator* message and begins another election. Eventually, *p*2 is elected coordinator (stage 4).

This algorithm clearly meets the liveness condition E2, by the assumption of reliable message delivery. And if no process is replaced, then the algorithm meets condition E1. It is impossible for two processes to decide that they are the coordinator, since the process with the lower identifier will discover that the other exists and defer to it. But the algorithm is *not* guaranteed to meet the safety condition E1 if processes that have crashed are replaced by processes with the same identifiers. A process that replaces a crashed process *p* may decide that it has the highest identifier just as another process (which has detected *p*’s crash) decides that *it* has the highest identifier. Two processes will therefore announce themselves as the coordinator concurrently. Unfortunately, there are no guarantees on message delivery order, and the recipients of these messages may reach different conclusions on which is the coordinator process. Furthermore, condition E1 may be broken if the assumed timeout values turn out to be inaccurate – that is, if the processes’ failure detector is unreliable.

Taking the example just given, suppose that either p3 had not failed but was just running unusually slowly (that is, that the assumption that the system is synchronous is incorrect), or that p3 had failed but was then replaced. Just as p2 sends its coordinator message, p3 (or its replacement) does the same. p2 receives p3 ‘s coordinator message after it has sent its own and so sets elected2 = p3 . Due to variable message transmission delays, p1 receives p2 ’s coordinator message after p3 ’s and so eventually sets elected1 = p2 . Condition E1 has been broken. With regard to the performance of the algorithm, in the best case the process with the second-highest identifier notices the coordinator’s failure. Then it can immediately elect itself and send N – 2 coordinator messages. The turnaround time is one message. The bully algorithm requires O N2 messages in the worst case – that is, when the process with the lowest identifier first detects the coordinator's failure. For then N – 1 processes altogether begin elections, each sending messages to processes with higher identifiers.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Write short note on bully algorithm.	Dec 2012	4

UNIT- 05/LECTURE- 05

CORBA

We start our study of distributed object-based systems by taking a look at the **Common Object Request Broker Architecture**, simply referred to as **CORBA**. As its name suggests, CORBA is not so much a distributed system but rather the specification of one. These specifications have been drawn up by the **Object Management Group** (**OMG**), a non-profit organization with over 800 members, primarily from industry. An important goal of the OMG with respect to CORBA was to define a distributed system that could overcome many of the interoperability problems with integrating networked applications. The first CORBA specifications became available in the beginning of the 1990s. At present, implementations of CORBA version 2.4 are widely deployed, whereas the first CORBA version 3 systems are becoming available. Like many other systems that are the result of the work of committees,

CORBA has features and facilities in abundance. The core specifications consist of well over 700 pages, and another 1,200 are used to specify the various services that are built on top of that core. And naturally, each CORBA implementation has its own extensions because there is always something that each vendor feels cannot be missed but was not included in the specifications. CORBA illustrates again that making a distributed system that is simple may be a somewhat overwhelmingly difficult exercise.

In the following pages, we will not discuss all the things that CORBA has to offer, but instead concentrate only on the parts that are essential to it as a distributed system and that characterize it with respect to other object-based distributed systems.

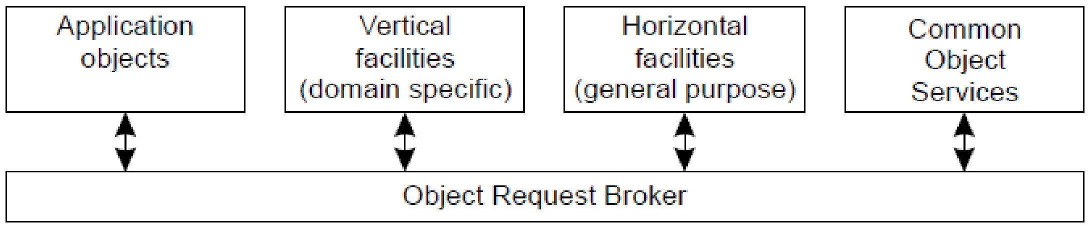
Overview of CORBA

The global architecture of CORBA adheres to a reference model of the OMG that was laid down in (OMG, 1997). This reference model consists of four groups of architectural elements connected to what is called the

Object Request Broker (ORB) ([RGPV/June 2013(5), ([RGPV/June 2014(5)]

The ORB forms the core of any CORBA distributed system; it is responsible for enabling communication between objects and their clients while hiding issues related to distribution and heterogeneity. In many systems, the ORB is implemented as libraries that are linked with a client and server application, and that offers basic communication services. We return to the ORB below when discussing CORBA's object model. Besides objects that are built as part of specific applications, the reference model also distinguishes what are known as **CORBA facilities**. Facilities are constructed as compositions of CORBA services (which we discuss below), and are split into two different groups. **Horizontal facilities** consist of general purpose high-level services that are independent of application domains. Such services currently include those for user interfaces, information management, system management, and task management (which is used to define workflow systems).

Vertical facilities consist of high-level services that are targeted to a specific application domain such as electronic commerce, banking, manufacturing, etc. We will not discuss application objects and CORBA facilities in any detail, but rather concentrate on the basic services and the ORB.


Application objects
Vertical facilities (domain specific)
Horizontal facilities (general purpose)
Common Object Services
Object Request Broker

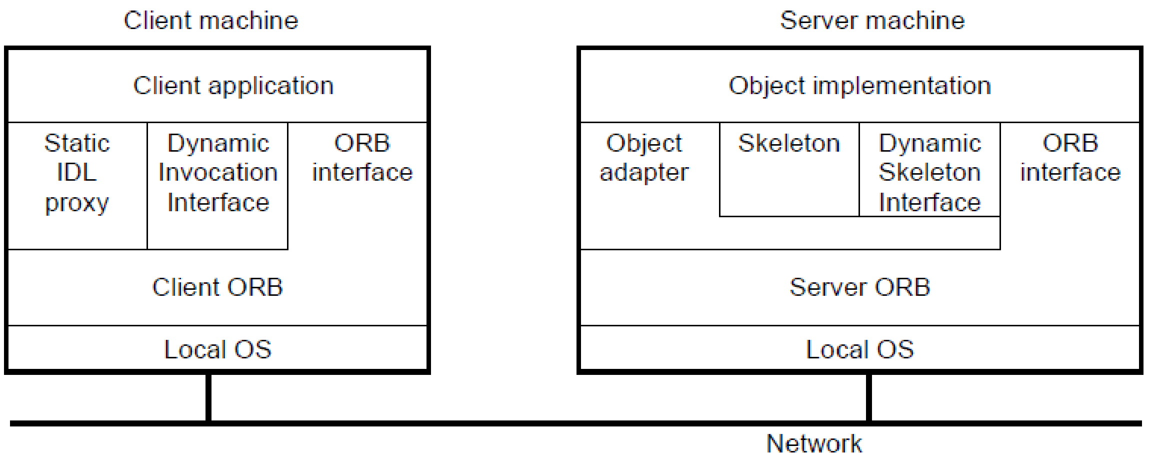
Object Model

In this model, the implementation of an object resides in the address space of a server. It is interesting to note that the CORBA specifications never explicitly state that objects should be implemented only as remote objects. However, virtually all CORBA systems support only this model. In addition, the specifications often suggest that distributed objects in CORBA are actually remote objects. Later, when discussing the Globe object model, we show how a completely different model of an object could, in principle, be equally well supported by CORBA.

Objects and services are specified in the CORBA **Interface Definition Language** (**IDL**). CORBA IDL is similar to other interface definition languages in that it provides a precise syntax for expressing methods and their parameters. It is not possible to describe semantics in CORBA IDL. An interface is a collection of methods, and objects specify which interfaces they implement. Interface specifications can be given only by means of IDL. As we shall see later, in systems such as Distributed COM and Globe, interfaces are specified at a lower level in the form of tables. These so-called **binary interfaces** are by their nature independent of any programming language. In CORBA, however, it is necessary to provide exact rules concerning the mapping of IDL specifications to existing programming languages. At present, such rules have been given for a number of languages, including C, C++, Java, Smalltalk, Ada, and COBOL. Given that CORBA is organized as a collection of clients and object servers, the general organization of a CORBA system is shown.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Write short note on (i) object request broker (ORB) architecture.	June 2013,June 2014	5

UNIT -05/LECTURE -06



Underlying any process in CORBA, be it a client or server, is the ORB. The ORB can best be seen as the runtime system that is responsible for handling the basic communication between a client and an object. This basic communication consists of ensuring that an invocation is sent to the object's server and that the reply is passed back to the client. From the perspective of a process, the ORB offers only a few services itself.

One of these services is manipulating object references. Such references are generally dependent on a particular ORB. An ORB will therefore offer operations to marshal and unmarshal object references so that they can be exchanged between processes, as well as operations for comparing references. Object references are discussed in detail below.

Other operations offered by an ORB deal with initially finding the services that are available to a process. In general, it provides a means to obtain an initial reference to an object implementing a specific CORBA service. For example, in order to make use of a naming service, it is necessary that a process knows how to refer to that service. These initialization aspects apply equally well to other services.

Besides the ORB interface, clients and servers see hardly anything of the ORB. Instead, they generally see only stubs for handling method invocations for specific objects. A client application usually has a proxy available that implements the same interface as each object it is using. A proxy is a client-side stub that merely marshals an invocation request and sends that request to the server. A response from that server is unmarshaled and passed back to the client. Note that the interface between a proxy and the ORB does not have to be standardized. Because CORBA assumes that all interfaces are given in IDL, CORBA implementations offer an IDL compiler to developers that generates the necessary code to handle communication between the client and server ORB.

However, there are occasions in which statically defined interfaces are simply not available to a client. Instead, what it needs is to find out during runtime what the interface to a specific object looks like, and subsequently compose an invocation request for that object. For this purpose, CORBA offers a **Dynamic Invocation Interface** (**DII**) to clients, which allows them to construct an invocation request at runtime. In essence, the DII provides a generic invoke operation, which takes an object reference, a method identifier, and a list of input values as parameters, and returns its result in a list of output variables provided by the caller.

A CORBA system provides an object adapter, which takes care of forwarding incoming requests to the proper object. The actual unmarshaling at the server side is done by means of stubs, called skeletons in CORBA, but it is also possible that the object implementation takes care of= unmarshaling. As in the case of clients, server-side stubs can either be statically compiled from IDL specifications, or be available in the form a generic dynamic skeleton. When using a dynamic skeleton, an object will have to provide the proper implementation of the invoke function as offered to the client. We return to object servers below.

Interface and Implementation Repository

To allow the dynamic construction of invocation requests, it is important that a process can find out during runtime what an interface looks like. CORBA offers an **interface repository**, which stores all interface definitions. In many systems, the interface repository is implemented by means of a separate process offering a standard interface to store and retrieve interface definitions. An interface repository can also be viewed as that part of CORBA that assists in runtime type checking facilities.

Whenever an interface definition is compiled, the IDL compiler assigns a **repository identifier** to that interface. This repository ID is the basic means to retrieve an interface definition from the repository. The identifier is by default derived from the name of the interface and its methods, implying that no guarantees are given with respect to its uniqueness. If uniqueness is required, the default can be overridden. Given that all interface definitions stored in an interface repository adhere to IDL syntax, it becomes possible to organize each definition in a standard way. (In database terminology, this means that the conceptual schema associated with an interface repository is the same for every repository.) As a consequence, the interface repositories in CORBA systems offer the same operations for navigating through interface definitions.

Besides an interface repository, a CORBA system generally offers also an **implementation repository**. Conceptually, an implementation repository contains all that is needed to implement and activate objects. Because such functionality is intimately related to the ORB itself and the underlying operating system, it is difficult to provide a standard implementation. An implementation repository is also tightly coupled to the organization and implementation of object servers. An object adapter has the responsibility for activating an object by ensuring that it is running in the address space of a server so that its methods can be invoked. Given an object reference, an adapter could contact the implementation repository to find out exactly what needs to be done.

For example, the implementation repository could maintain a table specifying that a new server should be started and also to which port number the new server should be listening for the specified object. The repository would furthermore have information about which executable file the server should load and execute. Alternatively, it may not be necessary to start a separate server, but the current one need merely link to a specific library containing the requested method or object. Again, such information would typically be stored in an implementation repository. These two examples illustrate that such a repository is indeed closely tied to an ORB and the platform on which it is running.

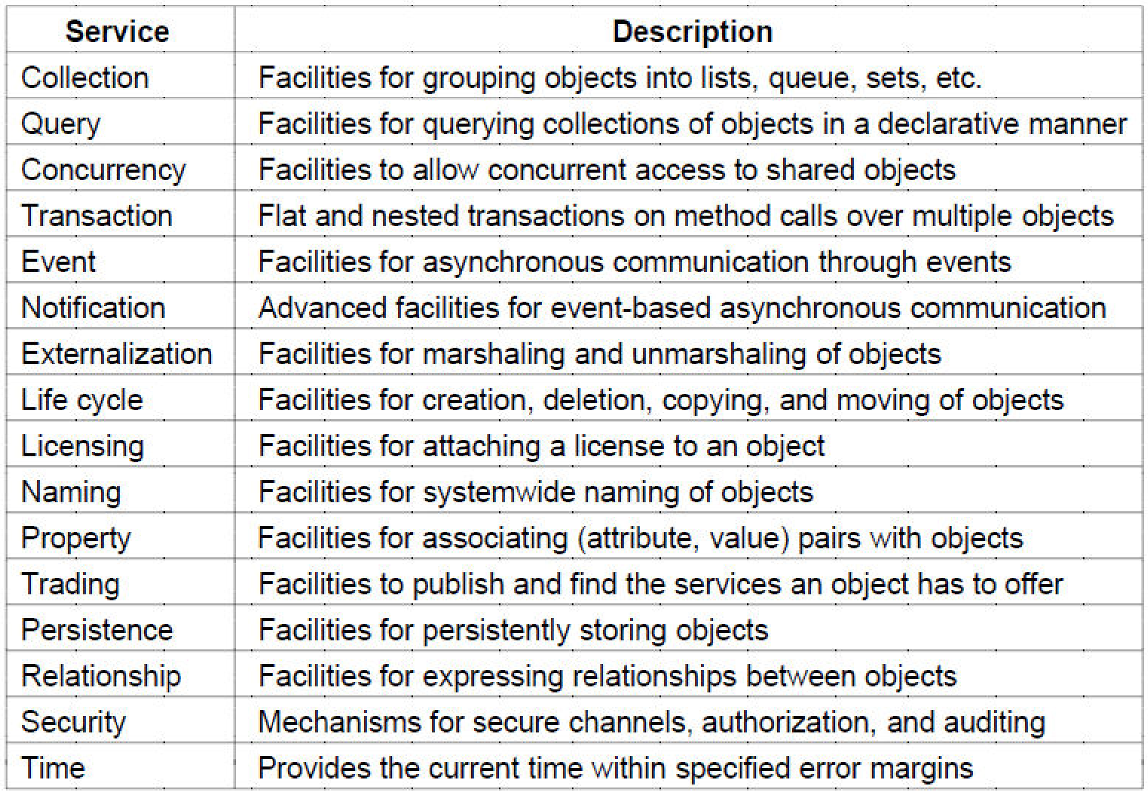
UNIT -05/LECTURE- 07

CORBA Services ([RGPV/June 2013(5), ([RGPV/June 2014(5)]

An important part of CORBA is reference model is formed by the collection of CORBA services. A CORBA service is best thought of as being general purpose and independent of the application for which CORBA is being used. As such, CORBA services strongly resemble the types of services commonly provided by operating systems. There is a whole list of services specified for CORBA. Unfortunately, it is not always possible to draw a clear line between the different services, as they often have overlapping functionality. Let us briefly describe each service so that we can later make a better comparison to services as offered by DCOM and Globe.

The collection service provides the means to group objects into lists, queues, stacks, sets, and so on. Depending on the nature of the group, various access mechanisms are offered. For example, lists can be inspected element wise through what is generally referred to as an iterate. There are also facilities to select objects by specifying a key value. In a sense, the collection service comes close to what is generally offered by class libraries for object-oriented programming languages.

There is also a separate query service that provides the means to construct collections of objects that can be queried using a declarative query language. A query may return a reference to an object or to a collection of objects. The query service augments the collection service with advanced queries. It differs from the collection service in that the latter offers various types of collections. There is also a concurrency control service. It offers advanced locking mechanisms by which clients can access shared objects. This service can be used to implement transactions, which are offered by a separate service. The transaction service allows a client to define a series of method invocations across multiple objects in a single transaction. The service supports flat and nested transactions.



Service	Description
Collection	Facilities for grouping objects into lists, queue, sets, etc.
Query	Facilities for querying collections of objects in a declarative manner
Concurrency	Facilities to allow concurrent access to shared objects
Transaction	Flat and nested transactions on method calls over multiple objects
Event	Facilities for asynchronous communication through events
Notification	Advanced facilities for event-based asynchronous communication
Externalization	Facilities for marshaling and unmarshaling of objects
Life cycle	Facilities for creation, deletion, copying, and moving of objects
Licensing	Facilities for attaching a license to an object
Naming	Facilities for systemwide naming of objects
Property	Facilities for associating (attribute, value) pairs with objects
Trading	Facilities to publish and find the services an object has to offer
Persistence	Facilities for persistently storing objects
Relationship	Facilities for expressing relationships between objects
Security	Mechanisms for secure channels, authorization, and auditing
Time	Provides the current time within specified error margins

Normally, clients invoke methods on objects and wait for the result of that invocation. To support asynchronous communication, CORBA supports an event Service by which clients and objects can be interrupted upon the occurrence of a specified event. Advanced facilities for asynchronous communication are provided by a separate notification service We describe these services in more detail below.

Externalization deals with marshalling objects in such a way that they can be stored on disk or sent across a network. It is comparable to the serialization facilities offered by Java, allowing objects to be written to a data stream as a series of bytes.

The life cycle service provides the means to create, destroy, copy, and move objects. A key concept is that of a factory object , which is a special object used to create other objects .Practice indicates that only the creation of objects needs to be handled by a separate service. However, destroying, copying, and moving objects is often conveniently defined by objects themselves. The reason is that these operations often affect an object's state in an object specific way. The licensing service allows developers of objects to attach a license to their object and enforce a specific licensing policy. A license expresses the rights a client has with respect to using an object. For example, a license attached to an object may enforce that the object can be used by only a single client at a time.

Another license may ensure that an object is automatically disabled after a certain expiration time. CORBA offers a separate naming service by which objects can be given a human-readable name that maps to the object's identifier. The basic facility for describing objects is provided by a separate property service. This service allows clients to associate (attribute, value) pairs with objects. Note that these attributes are not part of the object's state, but instead are used to describe the object.

In other words, they provide information about the object instead of being part of it. Related to these two services is a trading service that allows objects to advertise what they have to offer (by means of their interfaces), and to allow clients to find services using a special language that supports the description of constraints. A separate persistence service offers the facilities for storing information on disk in the form of storage objects. An important issue here is that persistence transparency is provided; a client need not explicitly transfer the data in a storage object between a disk and available main memory.

None of the services so far offer the facilities to explicitly relate two or more objects. These facilities are provided by a relationship service, which essentially provides support for organizing objects according to a conceptual schema like the ones used in databases. Security is provided in a security service. The implementation of this service is comparable to security systems such as SESAME and Kerberos. The CORBA security service provides facilities for authentication, authorization, auditing, secure communication, no repudiation, and administration.

Finally, CORBA offers a time service that returns the current time within specified error ranges.

The CORBA services have been designed with CORBA's object model as their basis. This means that all services are specified in CORBA IDL, and that a separation between interface specification and implementation is made. Another important design principle is that services should be minimal and simple. In the following sections we discuss a number of these services in more detail. From those descriptions, it can be argued to what extent this last principle has been successfully applied.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Write short note on (i) CORBA Services.	June 2013,June 2014	5

UNIT -05/LECTURE- 08

Introduction to Wave & traversal algorithms

In the design of distributed algorithms for various applications several very general problems for process networks appear frequently as subtasks. These elementary tasks include the broadcasting of information (e.g., a start or terminate message), achieving a global synchronization between processes, triggering the execution of some event in each process, or computing a function of which each process holds part of the input. These tasks are always performed by passing messages according to some prescribed, topology-dependent scheme that ensures the participation of all processes. Indeed, as will become more evident to the reader in later chapters, these tasks are so fundamental that solutions to more complicated problems such as election , termination detection , or mutual exclusion can be given in which communication between processes occurs only via these message passing schemes.

The importance of message-passing schemes, called wave algorithms from now on, justifies a separate treatment of them in isolation from a particular application algorithm in which the schemes can be embedded. This chapter formally defines wave algorithms)and proves some general results about them . The observation that the same algorithms can be used for all of the fundamental tasks listed above, i.e., broadcasting, synchronization, and computing global functions, will be made rigorous

The treatment of wave algorithms as a separate issue, even though they are usually employed as subroutines in more involved algorithms, is useful for two reasons. First, the introduction of the concept facilitates the later treatment of more involved algorithms because the properties of their subroutines have already been studied. Second, certain problems in distributed computing can be solved by generic constructions that yield a specific algorithm when parameterized with a specific wave algorithm. The same construction can then be used to give algorithms for different network topologies or for different assumptions about the initial knowledge of processes.

Definition and Use of Wave Algorithms

Unless stated otherwise, it is assumed throughout in this chapter that the network topology is fixed (no topological changes occur), undirected (each channel carries messages in both directions), and connected (there is a path between any two processes). The set of all processes is denoted by $\mathbb{P}$, and the set of channels by E. As in earlier chapters, it is assumed that the system uses asynchronous message passing and that there is no notion of global time or real-time clocks. The algorithms of this chapter can also be used with synchronous message passing (possibly with some small modifications to avoid deadlocks) or with global clocks if these are available. However, some of the more general theorems are not true in these cases

Definition of Wave Algorithms

A distributed algorithm usually allows a large collection of possible computations, due to non-determinism in the processes as well as the communication subsystem. A computation is a collection of events, partially ordered by the causal precedence relation $\preceq$as defined following. The number of events of computation C is denoted |C| and the subset of the events that occur in process p is denoted C_p. It is assumed that there is a special type of internal event called a decide event; in the algorithms in this chapter such an event is simply represented by the statement decide. A wave algorithm exchanges a finite number of messages and then makes a decision, which depends causally on some event in each process.

Definition 6.1 A wave algorithm is a distributed algorithm that satisfies the following three requirements.

(1) **Termination.** Each computation is finite

(2) **Decision.** Each computation contains at least one decide event

(3) **Dependence.** In each computation each decide event is causally preceded by an event in each process

A computation of a wave algorithm is called a wave. As an additional notation, in a computation of an algorithm a distinction is made between initiators, also called starters, and non-initiators, also called followers. A process is an initiator if it starts the execution of its local algorithm spontaneously, i.e., triggered by some condition internal to the process. A non-initiator becomes involved in the algorithm only when a message of the algorithm arrives and triggers the execution of the process algorithm. The first event of an initiator is an internal or send event, the first event of a non-initiator is a receive event.

A variety of wave algorithms exists because algorithms may differ in many respects. As a rationale for the treatment of a large number of algorithms in this chapter and as an aid in selecting one algorithm for a particular purpose a list of aspects in which wave algorithms differ from each other is given here

(1) Centralization. An algorithm is called centralized if there must be exactly one initiator in each computation, and decentralized if the algorithm can be started spontaneously by an arbitrary subset of the processes. Centralized algorithms are also called single-source algorithms, and decentralized algorithms are called multi-source algorithms. Centralization has an important influence on the complexity of wave algorithms.

(2) Topology. An algorithm may be designed for a specific topology, such as a ring, tree, clique, etc.

(3) Initial Knowledge. An algorithm may assume the availability of various types of initial knowledge in the processes Examples of pre required knowledge include the following:

(a) Process identity. Each process initially knows its own unique name.

(b) Neighbour's identities. Each process initially knows the names of its neighbours.

(c) Sense of direction.

(4) Number of Decisions. In all wave algorithms in this chapter at most one decision occurs in each process. The number of processes that execute a decide event may vary; in some algorithms only one process decides, in others all processes decide. The tree algorithm causes a decision in exactly two processes.

(5) Complexity. The complexity measures considered in this chapter are the number of exchanged messages, the number of exchanged bits, and the time needed for one computation.

Each wave algorithm in this chapter will be given with the variables it uses and, if necessary, the information exchanged in its messages. Most of these algorithms send "empty messages", without any actual information: the messages carry causality, not information.

When a wave algorithm is applied there are generally more variables and other information may be included in the message. Many applications rely on the simultaneous or sequential propagation of several waves; in this case information about the wave to which a message belongs must be included in messages. Also a process may keep additional variables to administer the wave or waves in which it is currently active.

An important subclass of wave algorithm is formed by centralized wave algorithms having the following two additional properties: the initiator is the only process that decides; all events are ordered totally by the causal order. Wave algorithms with these properties are called traversal algorithms

UNIT- 05/LECTURE -09

A Collection of Wave Algorithms

A collection of wave and traversal algorithms will be presented in the next three sections. In all cases the algorithm text is given for the process p.

The Ring Algorithm

In this subsection a wave algorithm for a ring network will be given. The same algorithm can be used for Hamiltonian networks in which one fixed Hamiltonian cycle is encoded in the processes. Assume that for each process p a dedicated neighbour $Next_p$ is given such that all channels selected in this way form a Hamiltonian cycle.

The algorithm is centralized; the initiator sends a message ⟨ **tok** ⟩ (called the token) along the cycle, each process passes it on, and when it returns to the initiator the initiator decides

Theorem The ring algorithm is a wave algorithm.

Proof. Call the initiator p_0. As each process sends at most one message the algorithm exchanges at most N messages altogether.

Within a finite number of steps the algorithm reaches a terminal configuration. In this configuration p_0 has already sent the token, i.e., has passed the send statement in its program. Furthermore, no ⟨**tok**⟩ message is in transit in any channel, otherwise it could be received and the configuration would not be terminal. Also no process other then p_0 "holds" the token (i.e., has received, but not sent ⟨**tok**⟩), otherwise this process could send ⟨**tok**⟩ and the configuration is not terminal. Concluding, (1) p_0 has sent the token, (2) for each p that has sent the token, $Next_p$ has received the token, and (3) for each $p \neq p_0$ that has received the token, p has sent the token. From this and the property of Next it follows that each process has sent and received the token. As p_0 has received the token and the configuration is terminal, P_0 has executed the decide statement.

The receipt and sending of ⟨**tok**⟩ by each process $p \neq p_0$ precedes the receipt by p_0, hence the dependence condition is satisfied.

The Tree Algorithm

The same algorithm can be used in an arbitrary network if a spanning tree of the network is available. It is assumed that all leaves of the tree initiate the algorithm. Each process sends exactly one message in the algorithm. If a process has received a message via each of its incident channels except one (this condition is initially true for leaves), the process sends a message via the remaining channel. If a process has received a message via all of its incident channels it decides.

The Echo Algorithm

The echo algorithm is a centralized wave algorithm for networks of arbitrary topology. It was first presented in isolation by Chang and therefore sometimes called Chang's echo algorithm. A slightly more efficient version was given by Segall and this version is presented here.

The algorithm floods ⟨**tok**⟩ messages to all processes, thus defining a spanning. Tokens are "echoed" back via the edges of this tree very much like the flow of messages in the tree algorithm.. The initiator sends messages to all its neighbours. Upon receipt of the first message a non-initiator forwards message to all its neighbours except the one from which the message was received; when a non-initiator has received messages from all its neighbours an echo is sent to the father. When the initiator has received a message from all its neighbours it decides.

The Polling Algorithm

In clique networks a channel exists between each pair of processes. A process can decide if it has received a message from each neighbour. In the polling algorithm the initiator asks each neighbour to reply with a message, and decides after receipt of all messages.

Theorem The polling algorithm is a wave algorithm.

Proof. The algorithm sends two messages via each channel that is incident to the initiator. Each neighbour of the initiator replies once to the original poll, hence the initiator receives N – 1 replies. This is exactly the number it needs to decide, which implies that the initiator will decide, and that its decision is preceded by an event in each process.

Polling can also be used in a star network in which the initiator is the centre.

REFERENCES

BOOK	AUTHOR	PRIORITY
Distributed operating systems; Concepts and design	P K Sinha	1
Distributed systems: Principles and paradigms	Tanenbaum and Steen	2