RGPVNOTES.IN

Subject Name: **Distributed System**

Subject Code: **IT-6005**

Semester: **6th**

UNIT 5

Distributed Algorithms

A distributed algorithm is an algorithm designed to run on computer hardware constructed from interconnected processors. Distributed algorithms are used in many varied application areas of distributed computing, such as telecommunications, scientific computing, distributed information processing, and real-time process control. Standard problems solved by distributed algorithms include leader election, consensus, distributed search, spanning tree generation, mutual exclusion, and resource allocation.

Distributed algorithms are a sub-type of parallel algorithm, typically executed concurrently, with separate parts of the algorithm being run simultaneously on independent processors, and having limited information about what the other parts of the algorithm are doing. One of the major challenges in developing and implementing distributed algorithms is successfully coordinating the behavior of the independent parts of the algorithm in the face of processor failures and unreliable communications links. The choice of an appropriate distributed algorithm to solve a given problem depends on both the characteristics of the problem, and characteristics of the system the algorithm will run on such as the type and probability of processor or link failures, the kind of inter-process communication that can be performed, and the level of timing synchronization between separate processes.

Destination-Based Routing

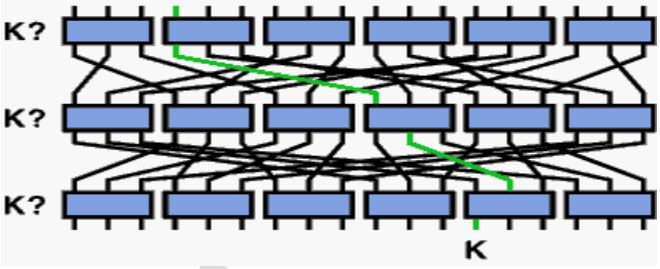

Fig 5.1 Destination-Based Routing

A packet enters the second switching module in the top row. The module looks up the destination address, "K", in its routing table and determines that the packet should exit out of left most port. The packet travels to the third module in the second row. This module looks up "K" in its routing table and determines that the packet should exit out the middle port. The packet then travels to the fifth module in the last row. The module looks up "K" in its routing table and determines that the packet should exit out of the left port.

The disadvantages of destination-based routing are that requires a table lookup at each switching module and that global information is required to compile the table.

The advantages of destination-based routing is that it can handle almost any topology and and that it can also handle fault induced change in the topology.

Assignment problem in parallel in distributed system:

Refer the given link for the APP:

1. chrome-extension://ngpampappnmepgilojfohadhhmbhlaek/captured.html?back=1
2. http://www.jatit.org/volumes/research-papers/Vol4No7/8.pdf

Deadlock free Packet switching

Deadlock states have been observed in existing computer networks, emphasizing the need for carefully designed flow control procedures (controllers) to avoid deadlocks. Such a deadlock-free controller is readily found if we allow it global information about the overall network state. Generally, this assumption is not realistic, and we must resort to deadlock-free local controllers using only packet and node information.

Buffer-size = 5

Node s sending 5 packets to v through t

Node v sending 5 packets to s through u

Shown in fig 5.2

Store and forward deadlock

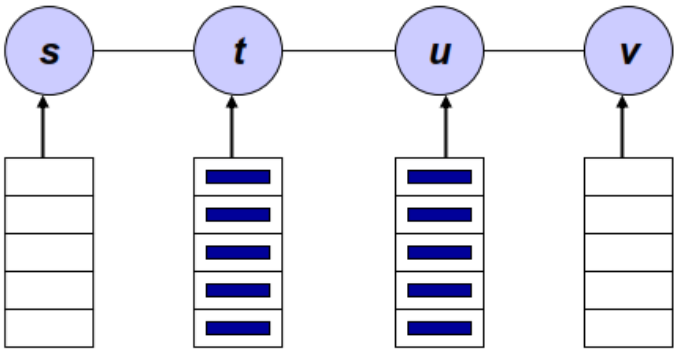

Fig 5.2 stored and Forward Deadlock

Model

- The network is a graph G = (V, E)
- Each node has B buffers

Moves

- Generation:
  A node u creates a new packet p and places it in an empty buffer in u. Node u is the source of p.
- Forwarding:
  A packet p is forwarded from a node u to an empty buffer in the next node w on its route.
- Consumption:
  A packet p occupying a buffer in its destination node is removed from the buffer.

Requirements

The packet switching controller has 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, that is, whether a packet can be accepted in a node u depends only on information known to u or contained in the packet

Solutions

Structured solutions

✓ Buffer graph based schemes

- The destination scheme
- The hops so far scheme
- Acyclic orientation based scheme

Unstructured solutions

- Forward count and backward count schemes
- Forward state and backward state schemes

Buffer Graph

A buffer graph (for, G, B) is a directed graph BG on the buffers of the network, such that

1. BG is acyclic (contains no directed cycle);
2. bc is an edge of BG if b and c are buffers in the same node, or buffers in two nodes connected by a channel in G; and
3. for each path $\pi \in P$ there exists a path in BG whose image is π
   - P is the collection of all paths followed by the packets- this collection is determined by the routing algorithm.

Suitable buffer and guaranteed path:

Let p be a packet in node u with destination v.

- A buffer b in u is suitable for p if there is a path in BG from b to a buffer c in v, whose image is a path, that p can follow in G.
- One such path in BG will be designated as the guaranteed path and nb(p, b) denotes the next buffer on the guaranteed path.
- For each newly generated packet p in u there exists a designated suitable buffer, fb(p) in u.

The buffer graph controller

1. The generation of a packet p in u is allowed iff the buffer fb (p) is free. If the packet is generated it is placed in this buffer.
2. The forwarding of a packet p from a buffer in u to a buffer in w is allowed iff nb(p, b) (in w) is free. If the forwarding takes place p is placed in nb(p, b).

The buffer graph controller is a deadlock free controller.

The Destination Scheme

Uses N buffers in each node u, with a buffer $b_u[v]$ for each possible destination v

- It is assumed that the routing algorithm forwards all packets with destination v via a directed treeT_y

The buffer graph is defined by BG = (B, E), where $b_u[v_1]b_w[v_2] \in E$ iff $v_1 = v_2$ and uw is an edge of T_{v1}.

There exists a deadlock free controller for arbitrary connected networks that uses N buffers in each node and allows packets to be routed via arbitrarily chosen sink trees.

The Hops-so-far Scheme

- Node u contains k +1 buffers $b_u[0]$…. $B_u[k]$.
- It is assumed that each packet contains a hop count indicating how many hops the packet has made from its source

The buffer graph is defined by BG = (B, E), where $b_u[i]b_w[j] \in E$ iff $i+1=j$ and uw is an edge of the network.

There exists a deadlock free controller for arbitrary connected networks that uses D+1 buffers in each node (where D is the diameter of the network), and requires packets to be sent via minimum-hop paths.

Acyclic Orientation based Scheme

Goal: To use only a few buffers per node

- An acyclic orientation of G is a directed acyclic graph obtained by directing all edges of G
- A sequence G … G_B of acyclic orientations of G is an acyclic orientation cover of size B for the collection P of paths if each path, Π ∈ P can be written as a concatenation of B paths π_1 … π_B. where π_1 is a path in G_i.
- A packet is always generated in node u in buffer $b_u[1]$
- A packet in buffer $b_u[i]$ that must be forwarded to node w is placed in buffer $b_w[i]$ if the edge between u and w is directed towards w in G_i, and to $b_w[i+1]$ if the edge is directed towards u in G_i

If an acyclic orientation cover for P of size B exists, then there exists a deadlock free controller using only B buffers in each node.

Forward and Backward-count Controllers

Forward count Controller:

- For a packet p, let S be the number of hops it still has to make to its destination ($0 \le s_p \le k$}
- For a node u, f_u denotes the number of free buffers in u ($0 \le f_u \le B$)

The controller accepts a packet p in node u Iff $s_p < f_u$

If B > k then the above controller is a deadlock free controller

Backward count Controller:

- For a packet p, let t_p be the number of hops it has made from its source

The controller accepts a packet p in node u iff $t_p > k-f_u$

Forward and Backward state Controllers

Forward state Controller:

For a node u define (as a function of the state of u) the state vector as (j_0 … j_k), where j_s is the number of packets p in u with $s_p = s$.

The controller accepts a packet p in node u with state (j_0 .., j_k) iff :

∀ I,0 ≤ I ≤ s_p : i< B -$\sum_{s=i}^{k} j_s$

If B > k then the above controller is a deadlock free controller

Backward-state Controller

Define the state vector as (i_0 ,…., i_k), where i_t is the number of packets in node u that have made t hope.

The controller accepts a packet p in node u with state (i_0 … , i_k) iff:

∀ j,t_p ≤ j ≤ k : j> $\sum_{t=0}^{j} i_t - B + k$

Forward state versus Forward count

Forward state controller is more liberal than the forward-count controller

Every move allowed by the forward count controller is also allowed by the forward state controller

Wave Algorithms

- A wave algorithm wave algorithm is a distributed algorithm that is a distributed algorithm that satisfies the following three requirements: satisfies the following three requirements:

– Termination: Termination: Each computation is finite each computation is finite

– Decision: Decision: Each computation contains at least one each computation contains at least one decide event decide event

– Dependence: Dependence: In each computation each decide event is In each computation each decide event is causally preceded by an event in each process causally preceded by an event in each process.

Traversal Algorithms

- A traversal algorithm traversal algorithm is an algorithm with the is an algorithm with the following three properties: following three properties: –

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

– A process, upon receipt of a message, either sends out A process, upon receipt of a message, either sends out one message or decides one message or decides

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

Election Algorithms

Election Algorithms

- The **coordinator election problem** is to choose a process from among a group of processes on different processors in a distributed system to act as the central coordinator.
- An **election algorithm** is an algorithm for solving the coordinator election problem. By the nature of the coordinator election problem, any election algorithm must be a distributed algorithm.

-a group of processes on different machines need to choose a coordinator

-peer to peer communication: every process can send messages to every other process.

-Assume that processes have unique IDs, such that one is highest

-Assume that the priority of process P_i is *i*

(a) Bully Algorithm

Background: any process P_i sends a message to the current coordinator; if no response in *T* time units, P_i tries to elect itself as leader. Details follow:

Algorithm for process P_i that detected the lack of coordinator

1. Process P_i sends an "Election" message to every process with higher priority.
2. If no other process responds, process P_i starts the coordinator code running and sends a message to all processes with lower priorities saying "Elected P_i"
3. Else, P_i waits for *T'* time units to hear from the new coordinator, and if there is no response start from step (1) again.

Algorithm for other processes (also called P_i)

If P_i is not the coordinator then P_i may receive either of these messages from P_j

if Pi sends "Elected P_j"; [this message is only received if $i < j$]

Pi updates its records to say that P_j is the coordinator.

Else if P_j sends "election" message (i > j)

Pi sends a response to P_j saying it is alive

Pi starts an election.

Example:

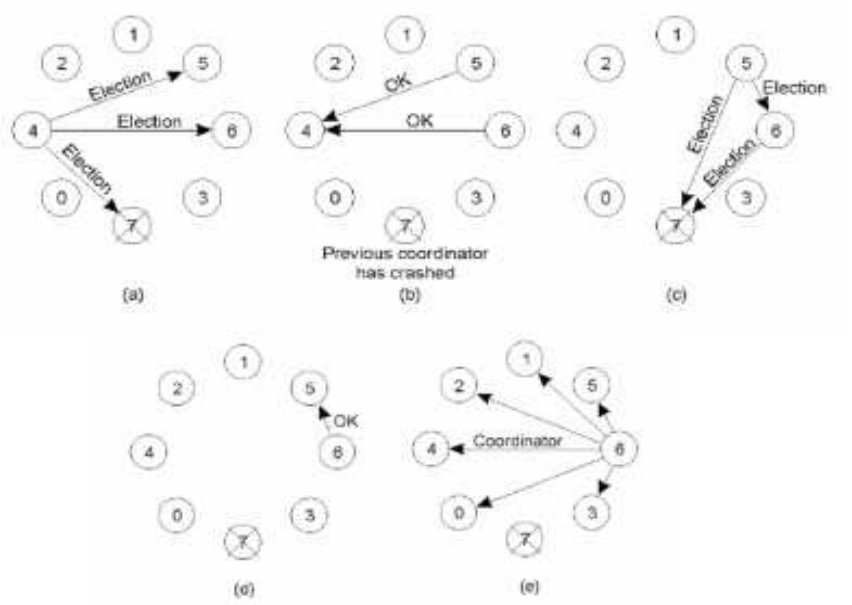

Fig 5.3 Example of Bully Algorithm

(b) Election In A Ring => Ring Algorithm.

-assume that processes form a ring: each process only sends messages to the next process in the ring

- Active list: its info on all other active processes

- assumption: message continues around the ring even if a process along the way has crashed.

Background: any process P_i sends a message to the current coordinator; if no response in T time units, P_i initiates an election

1. initialize active list to empty.
2. Send an "Elect(i)" message to the right. + add i to active list.

If a process receives an "Elect(j)" message

(a) this is the first message sent or seen

initialize its active list to [i,j]; send "Elect(i)" + send "Elect(j)"

(b) if i != j, add i to active list + forward "Elect(j)" message to active list

(c) otherwise (i = j), so process i has complete set of active processes in its active list.

=> choose highest process ID + send "Elected (x)" message to neighbor

If a process receives "Elected(x)" message,

set coordinator to x

Example:

Processes are arranged in a logical ring, each process knows the structure of the ring shown in fig 3.4

- A process initiates an election if it just recovered from failure or it notices that the coordinator has failed
- Initiator sends *Election* message to closest downstream node that is alive
  – *Election* message is forwarded around the ring
  – Each process adds its own ID to the *Election* message
- When *Election* message comes back, initiator picks node with highest ID and sends a Coordinator message specifying the winner of the election

– Coordinator message is removed when it has circulated once.

- Multiple elections can be in progress

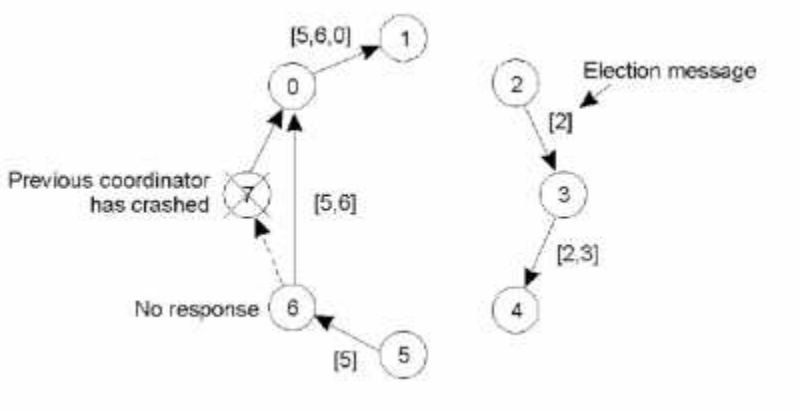

Fig. 3.4 Election algorithm using a ring.

What is CORBA?

The Common Object Request Broker Architecture (CORBA) is a standard developed by the Object Management Group (OMG) to provide interoperability among distributed objects. CORBA is the world's leading middleware solution enabling the exchange of information, independent of hardware platforms, programming languages, and operating systems. CORBA is essentially a design specification for an Object Request Broker (ORB), where an ORB provides the mechanism required for distributed objects to communicate with one another, whether locally or on remote devices, written in different languages, or at different locations on a network.

The CORBA Interface Definition Language, or IDL, allows the development of language and location-independent interfaces to distributed objects. Using CORBA, application components can communicate with one another no matter where they are located, or who has designed them. CORBA provides the location transparency to be able to execute these applications.

CORBA is often described as a "software bus" because it is a software-based communications interface through which objects are located and accessed. The illustration below identifies the primary components seen within a CORBA implementation.

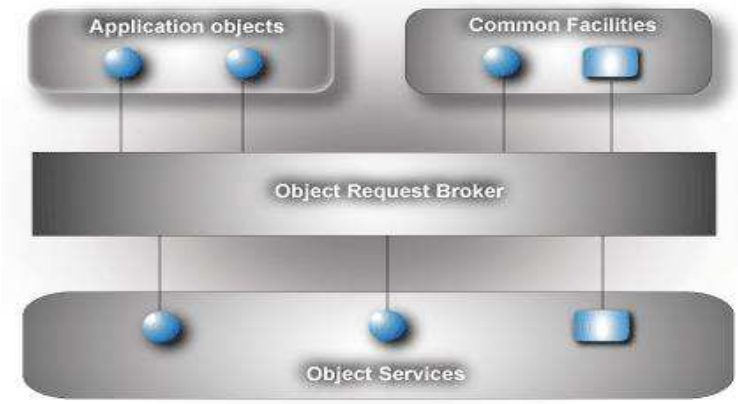

Fig 5.5 CORBA architecture

Data communication from client to server is accomplished through a well-defined object-oriented interface. The Object Request Broker (ORB) determines the location of the target object, sends a request to that object, and returns any response back to the caller. Through this object-oriented technology, developers can take advantage of features such as inheritance, encapsulation, polymorphism, and runtime dynamic binding. These features allow applications to be changed, modified and re-used with minimal changes to the parent interface. The illustration below identifies how a client sends a request to a server through the ORB:

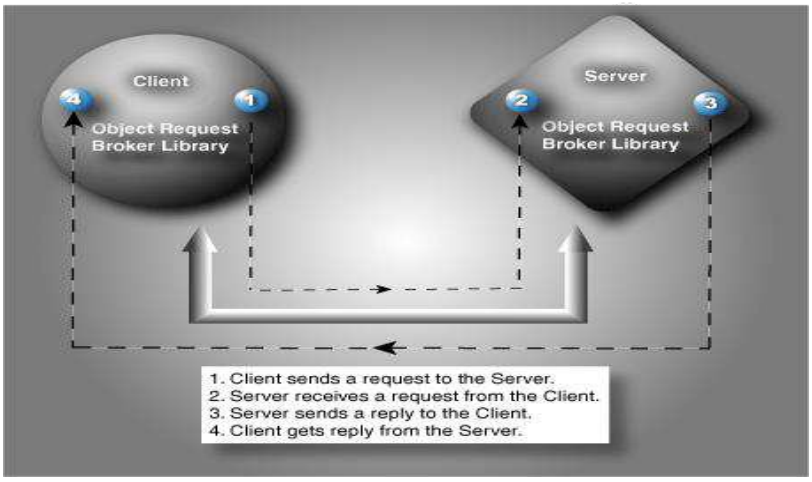

Fig 5.6 Client server communication

CORBA services

The CORBA standards define some standard services that may be provided to support distributed object oriented applications. You can think of CORBA services as those facilities that are likely to be required by many distributed systems. The standards define approximately 15 common services.

Some examples of these generic services are:

- Naming and trading services that allow objects to refer to and discover other objects on the network. The naming service is a directory service that allows objects to be named and discovered by other objects. This is like the white pages of a phone directory. The trading services are like the yellow pages. Objects can find out what other objects have registered with the trader service and can access the specification of these objects.
- Notification services that allow objects to notify other objects that some event has occurred. Objects may register their interest in a particular event with the service and, when that event occurs, they are automatically notified. For example, say the system includes a print spooler that queues documents to be printed and a number of printer objects. The print spooler registers that it is interested in an 'end of printing' event from a printer object. The notification service informs it when printing is complete. It can then schedule the next document on that printer.
- Transaction services that support atomic transactions and rollback on failure. Transactions are a fault-tolerance facility that supports recovery from errors during an update operation. If an object update operation fails then the object state can be rolled back to its state before the update was started.

Interface Definition Language (IDL)

A cornerstone of the CORBA standards is the Interface Definition Language. IDL is the OMG standard for defining language-neutral APIs and provides the platform-independent delineation of the interfaces of distributed objects. The ability of the CORBA environments to provide consistency between clients and servers in heterogeneous environments begins with a standardized definition of the data and operations constituting the client/server interface. This standardization mechanism is the IDL, and is used by CORBA to describe the interfaces of objects.

IDL defines the modules, interfaces and operations for the applications and is not considered a programming language. The various programming languages, such as Ada, C++, or Java, supply the implementation of the interface via standardized IDL mappings.

Application Development Using ORB*express*

The basic steps for CORBA development can be seen in the illustration below. This illustration provides an overview of how the IDL is translated to the corresponding language (in this example, C++), mapped to the source code, compiled, and then linked with the ORB library, resulting in the client and server implementation.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in