Distributed Graph Algorithms 5

Abstract

A distributed system consists of a number of computational nodes connected by a communication network. The nodes of a distributed system cooperate and communicate to achieve a common goal. We first describe the type of distributed systems, the communication and synchronization methods used in these systems. We then investigate few fundamental distributed algorithms including spanning tree construction, broadcast and convergecast operations over a spanning tree, and leader election.

5.1 Introduction

A distributed computing system or a distributed system as more commonly termed consists of a number of computational nodes connected by a communication network. Computing nodes are autonomous and the network can be a wired medium; a wireless communication channel or both. The nodes of a distributed system cooperate and communicate to achieve a common goal. It is evident that synchronization among computations at the nodes of such system is needed to provide this coordination.

A distributed system appears to users as a single computing system. In that respect, a cloud is a distributed system since there are numerous computing elements and databases in a cloud, yet it appears as a single system to a user. Distributed systems are needed because they provide convenient access to remote resources for users and applications. In many cases, the application itself is inherently distributed. For example, an airline reservation system is used by many users and provides all necessary communication and synchronization. Distributed systems provide fault tolerance in which case failure of a node or a link does not harm the operation of the system as these are replaced by other nodes or links. Distributed systems are

K. Erciyes, *Guide to Graph Algorithms*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-73235-0_5

commonly dynamic in which nodes and links may be inserted to or deleted from the network due to failures or movement of the nodes as in the case of a mobile network. A rescue operation consisting of moving nodes is an example of a mobile network.

A distributed system can be conveniently modeled by a graph in which vertices of the graph represent the computational nodes and an edge between two nodes represents a communication facility between them. The algorithms running at the nodes of a graph representing the distributed system are commonly termed *distributed graph* or *network algorithms*. Note that distributed memory-employing algorithms in a parallel processing environment are also called distributed algorithms in the literature but in the context of this book, we will use distributed (graph) algorithms to mean algorithms running in a network represented by a graph. We will see designing a distributed version of a sequential graph algorithm is not a trivial task. We start this chapter by describing common distributed system platforms. We then investigate distributed graph algorithms, classify them, and show the operation of some basic distributed graph algorithms.

5.2 Types of Distributed Systems

Distributed system applications vary from clusters of computers to networks of embedded systems. We can classify distributed systems as distributed computing systems, distributed information systems, and distributed pervasive systems [5]. Distributed computing systems typically consist of a cluster of homogenous computers connected by a local area network. The *Grid* is also a distributed computing system, which consists of numerous heterogenous computing systems with many different users that cooperate to achieve a common goal [3]. *Cloud computing* is more general than grid computing and provides users with various resources such as storage, data management, web site hosting, and computation [4]. Fault tolerance due to failing nodes and links, and load balancing are the main issues to be handled in both Grid and a cloud.

Distributed information systems commonly involve large database applications such as a transaction processing system. An online banking system with millions of users is an example of such system. Distributed pervasive systems typically consist of small and sometimes mobile computers that communicate using wireless medium. We will take a closer look at these systems since unlike Grid or a cloud, these can be modeled conveniently by a graph. The Internet is the largest network in the world connecting personal, infrastructured, wireless, or any other type of network.

Wireless networks communicate using wireless communication and networking medium. They can be broadly classified as *infrastructured* and *ad hoc*. An infrastructured wireless network has a fixed wired backbone consisting of routers and access points to provide communication among hosts of the network such as a cellular network. In contrast, ad hoc wireless networks do not have this structure and each node in such a network acts as a router for the transfer of messages. Ad hoc networks are widely used due to easiness and speed in their deployment. Two types of wireless networks have gained importance recently; *mobile ad hoc networks* and *wireless sensor networks*.

5.2.1 Mobile Ad hoc Networks

A mobile ad hoc network (MANET) is an infrastructure-less wireless network consisting of nodes that move dynamically. Vehicular ad hoc networks (VANETs) that provide communication between moving vehicles, military MANETs used by military, and MANETs used in rescue operations are examples of such systems. Each node in a MANET acts as a router for *multi-hop communications* between hosts in which a message is transferred between a number of host pairs before it reaches its destination.

One of the main challenges in a MANET is *routing*, which is the process of transferring a message between a sender and a receiver in the most efficient way. Nodes are mobile which means routes have to be computed dynamically requiring efficient routing algorithms. Staying connected in a MANET is also another problem that needs to be solved. A robot network is another example of a MANET and keeping the network connected at all times is needed for the coordinated operations of robots in such a network.

5.2.2 Wireless Sensor Networks

A wireless sensor network (WSN) consists of a network of sensors with radio transceivers and controllers. These networks of physically tiny nodes in most cases, have many applications including environmental control, e-health, and intelligent buildings. A sensor node has a very limited power and sensors are typically controlled by a central node called the *sink* with more computational capabilities. Data recorded by sensor nodes is collected at the sink for further processing. Routing of data messages to the sink efficiently using network protocols as well as keeping the network connected are the main issues to be addressed in WSNs. Sensor networks are mostly stationary and require low-power operation, which is more critical than managing power in MANETs.

A MANET or a WSN can be conveniently modeled by a graph and the problems such as routing, connectivity can then be transferred to graph domain to be solved with methods developed for graphs. For example, efficient routing problem can be solved with the aid of the method of finding the shortest distance between two nodes of a weighted graph. However, these problems should now be solved in a distributed manner without any global knowledge, which makes the problem harder than an ordinary graph problem. A node in a graph representing a WSN can only communicate with its neighbors, but we need to have a global decision using the

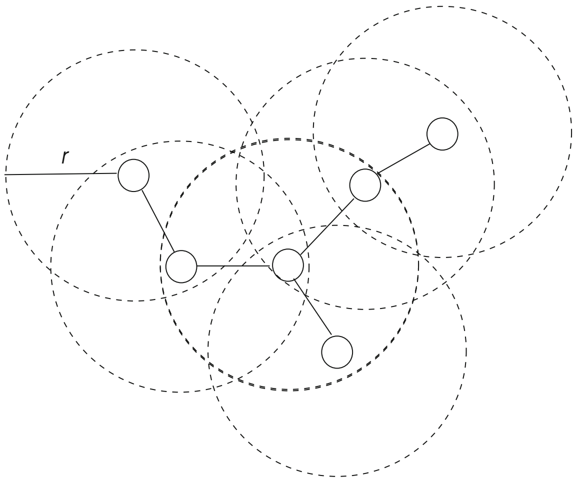


Fig. 5.1 The graph representation of a wireless network. Transmission range of a node is shown by dashed circles centered at that node

collected data from all of the sensors. Figure 5.1 displays a wireless network with nodes that can transmit and receive radio signals within a radius of r meters. We can then connect the nodes that are within transmission ranges of each other by an edge and obtain the graph shown.

5.3 Models

Messages are crucial for the correct operation of a distributed algorithm. We can define the widely accepted *message passing model* of a distributed system formally as follows [1,6]:

- A process p_i at a node i communicates with other processes by exchanging messages only.
- Each process p_i has a state $s_i \in S$, where S is the set of all possible states that a process p_i can be.
- A configuration of a system consists of a vector of states as $C = [s_1, \ldots, s_n]$
- The configuration of a system may change by either a *message delivery event* or a *computation event.*
- A distributed system continuously goes through executions as $C_0, \phi_1, C_1, \phi_2, \ldots$ where ϕ_i is either a computation or a message delivery event.

A *finite-state machine* (FSM) or *finite-state automaton* is a mathematical model to represent a complex system. An FSM consists of states, inputs, and outputs. It may change its state based on its current state and the input it receives. FSMs are widely used to design algorithms, network protocols, and sequence analysis in bioinformatics. Formally, a deterministic FSM is a quintuple (I, S, S_0, δ, F) where

- I is a set of input signals.
- S is a finite nonempty set of states.
- $s_0 \in S$ is the initial start state.
- δ is the state transition function such that $\delta : S \times I \rightarrow S$.
- $O \in S$ is the set of output states.

The next state of an FSM is determined by its current state and the input it receives. The same input may cause different actions in different states. As an everyday example, let us consider students in a school who for simplicity can have only two states: *in_class* or *out_class* meaning they can be either in the class or out of the class. When the bell rings in *in_class* state, it means they can go out and the bell ringing in *out_class* state means they should go in the class. An *FSM diagram* or a *state transition diagram* is a visual aid to understand the behavior of an FSM. The circles in such a diagram denote its states and transitions between states are shown by directed arcs which are labeled as *a*/*b* where *a* is the set of inputs received and *b* is the set of outputs produced when these inputs are received. A double circle denotes the accept state.

A *state table* provides an alternative way of representing a FSM. It has states of the FSM as rows and inputs as columns and the elements of the table can be the next FSM state and actions to be taken when the input is received. The output of a *Moore Machine* type of FSM is the next state, whereas the output in a *Mealy Machine* type of FSM contains outputs as well as the next state.

Example 5.1 We will design a simple FSM for an elevator that can only go to floors 0, 1, and 2. There are two buttons in the elevator: *up* and *down* which take the elevator up and down respectively. We can associate the current state of the elevator with the floor it currently stays; therefore we have three states 0, 1, and 2. At each state, the *up* or *down* button can be pressed represented by two inputs *up* by 0 and *down* by 1. The FSM diagram for this example is shown in Fig. 5.2 which shows all state transitions, considering there will be two inputs at each state. We cannot go down from 0 state and also going up from second floor is not allowed shown by loops at these states.

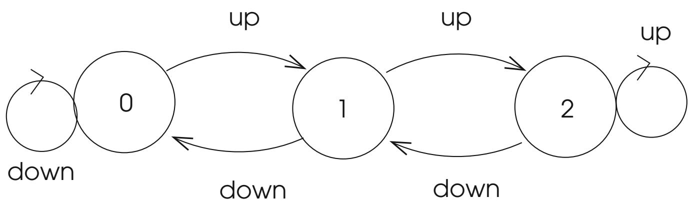


Fig. 5.2 FSM of the elevator example

Table 5.1 Elevator state table

State	0(Up)	1(Down)
0	1	0
1	2	0
2	2	1

We can now form the state table for this FSM with entries showing the next state of the FSM when the input shown in columns is received at state shown in rows as shown in Table 5.1. This way of expressing an FSM provides a very convenient way of writing its algorithm. We can form a 2-D array with each element being a function pointer. We then define functions to be performed for each table entry; for example, receiving "0" (up) at "1" (first floor) state should cause a transition to state 2 (elevator should move to second floor) which is realized by changing the current state to "2". The running of the algorithm is then straightforward; every time an input is received, we activate the function shown by the FSM table entry as shown by the C programming language code below.

```
#include <stdio.h>
    # define UP     0
    # define DOWN   1

    void *fsm_tab[3][2]();
    int  input;

    void act00(){curr_state=1;}
    void act01(){curr_state=0;}
    void act10(){curr_state=2;}
    void act11(){curr_state=0;}
    void act20(){curr_state=2;}
    void act21(){curr_state=1;}

    main()
     {  curr_state=0;          // initialize curr_state
        fsm_tab[0][0]=act00;   // initialize FSM table
        fsm_tab[0][1]=act01;
        fsm_tab[0][2]=act02;
        fsm_tab[1][0]=act10;
        fsm_tab[1][1]=act11;
        fsm_tab[1][2]=act12;

        while (true)
```

```
             { printf("Type 0 for up, and 1 for down \n");
               scanf("%d", &input);
               *fsm_tab[curr_state][input];
               printf("now at floor \%d", curr_state)
             }
      }
```

5.4 Communication and Synchronization

The algorithms that run at the nodes of a distributed system need to synchronize to accomplish a common goal. This process can be performed at various levels. Let us see how synchronization can be handled locally at three main levels of hierarchy; the hardware, the operating system, and the application. At the lowest level, hardware may provide synchronization at a certain number of clock ticks periodically. At a higher level, one of the main tasks of local operating systems at each node is the synchronization of the processes residing at that node. Moreover, this function can be extended to processes running at the nodes of the distributed system at the application level.

However, we need a mechanism to provide synchronization among the nodes which should be translated to local synchronization mechanisms described above. A very commonly used method in a distributed system is synchronization via messages. In this so-called *message passing* model, each local operating system or middleware provides two basic primitives; *send* and *receive* for sending and receiving messages. These procedures can be executed in *blocking* or *non-blocking* fashion. A blocking *send* stops the caller until an acknowledgment from the receiver is received. A blocking *receive* means the receiver should wait until a message is received. The blocking *receive* maybe *selective* in which a message from a particular sender is waited and execution is resumed only after this happens. It is common practice to employ a non-blocking *send* with a blocking *receive* since the sent message is assumed to be delivered correctly while the actions of a receiver depend on whether the message is received and also its contents and thus a blocking *receive* is frequently used.

Sending and receiving are commonly employed indirectly using data structures called *ports* or *mailboxes*. These are depository places for messages, and placement or removal of messages can be performed asynchronously from these structures. In a distributed system, the locally executed *send* procedure typically deposits the message in the mailbox of the network process which appends the necessary network headers, and transfers the message through lower network layer software to the network. The receiving network process removes network headers and deposits the message in the mailbox of the receiver which takes it from there as shown in simplified form in Fig. 5.3. There are three main software modules at each node of a distributed system: system(OS), network protocol stack (N/W), and the application (APP) as shown in this figure.

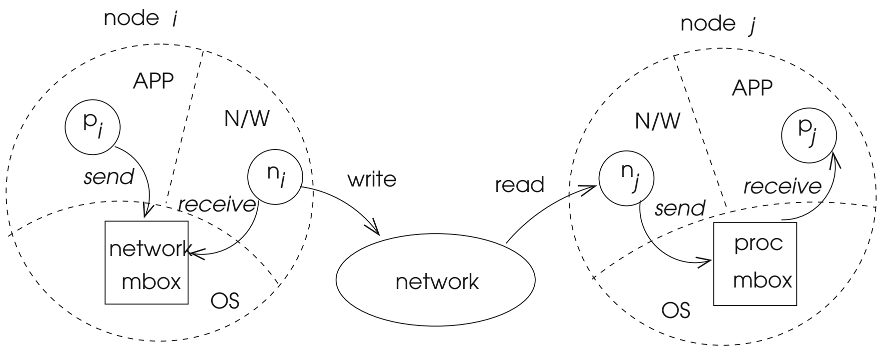


Fig. 5.3 Distributed communication via mailboxes. Process p_i at node i sends a message to process p_j at node j using mailboxes via network processes n_i and n_j

In summary, operating system and network processes provide synchrony between two processes p_i and p_j which execute distributed algorithms at nodes i and j of the network. At a higher level of abstraction at application, synchronization among distributed algorithms at different nodes may be achieved by *rounds* which are executed in lock-step fashion. In this case, a special process at a node starts each round and each process waits for all other processes to finish execution before the next round starts. Synchronization at the beginning of a round can be achieved by broadcasting a special message and end of a round can be identified by the convergence of messages which are two basic communication operations as we will see shortly. Commonly, a process p_i of a distributed system performs the following steps at each round:

1. *send* message.
2. *receive* message.
3. do some computation.

We assume here that a process sends the results of its computation from round k-1 in kth round. This order is not strict however, we could have *compute-send-receive* sequence which would mean each process now computes new results in round k based on what it has received in the previous round and sends the new results in the current round. Distributed algorithms that work asynchronously and do not have this synchronously executing rounds are called *asynchronous algorithms*. Detecting the termination of distributed algorithms is needed to stop the algorithm when a certain condition is met and this is not a trivial task. Although starting and ending a round cause overheads in terms of needed extra messages, designing synchronous distributed algorithms is more straightforward than asynchronous algorithms in general. The asynchronous algorithms require more complex control logic and detection of termination in such algorithms is also more difficult.

Yet another distinction is whether a single initiator starts the distributed algorithm or there are more than one initiators. A single initiator that also controls the overall running of the algorithm means a single point of supervision which is easier to manage than individually controlled processes. We can, hence, classify distributed algorithms based on synchronization at application or algorithmic level as follows.

- Synchronous Single Initiator (SSI) algorithms: There is a single initiator which starts the algorithm, synchronizing start and end of each round, and initiating termination. These algorithms are simpler to design than others since there is a single process that controls the operation.
- Asynchronous Single Initiator (ASI) algorithms: This type of algorithms have a single initiator but activity at each node is performed asynchronously from the other nodes. Synchronization and termination detection are more difficult for such an algorithm than a synchronous algorithm.
- Synchronous Concurrent Initiator (SCI) algorithms: These algorithms execute synchronously but may be started by concurrent initiators.
- Asynchronous Concurrent Initiator (ACI) algorithms: There are more than one initiators in this case and the activities are asynchronous. This mode of operation is the most general case but may require complex control.

In the case of an SSI algorithm, a previously built spanning tree to transfer control messages can be conveniently used. Based on foregoing, a possible SSI algorithm template is sketched in Algorithm 5.1. All processes start the kth round when they receive the *start* message over the spanning tree T, which is basically a broadcast operation over T as we will see shortly. The three actions in the round are sending results of the previous round to all neighbors, receiving results of the previous round from neighbors, and prepare new results for the next round. When a process finishes executing a round, it waits for all of its children in T to finish before it can send the *stop* message to its parent. When the root of the spanning tree T receives *stop* message from all of its children, the round k is over and the root can now start the round $k + 1$. We will use this structure frequently while designing distributed graph algorithms.

Algorithm 5.1 *SSI_template*

```
1: boolean $finished, round\_over \leftarrow false$
2: message type $start, result, stop$
3: while $\neg round\_over$ do
4:     receive msg(j)
5:     case msg(j).type of
6:         $\underline{start}$ : send $result(i)$ to all neighbors
7:                  receive $result(j)$, from each neighbor $j$
8:                  compute $result(i)$, $finished \leftarrow true$
9:         $\underline{stop}$ : if $stop$ received from all children and $finished$ then
10:                    send $stop$ to $parent$
11:                    $round\_over \leftarrow true$, $finished \leftarrow false$
12: end while
```

5.5 Performance Criteria

The performance of a distributed algorithm is evaluated in terms of time, message, space, and bit complexities outlined below:

- *Time Complexity*: Time complexity is the number of steps required for the distributed algorithm to finish as in a sequential algorithm. For synchronous distributed algorithms, we would be mostly interested in the number of rounds as time complexity.
- *Message Complexity*: This parameter is commonly considered as the dominant cost of a distributed algorithm since it directly shows the utilization of the network and indicates synchronization costs among the nodes of the network. Transferring a message over a network is magnitudes of orders more costly than doing local computations.
- *Bit Complexity*: The length of a message may also affect the performance of a distributed algorithm, especially if message length increases as the message traverses the network. For a large network modeled by a graph with many vertices and edges, bit complexity may be significant which directly affects the network performance.
- *Space Complexity*: This is the required storage at a node of the distributed system for the algorithm under consideration.

5.6 Distributed Graph Algorithm Examples

We are now ready to design and implement simple distributed graph algorithms. We will describe sample basic algorithms which follow a logical sequence. The first algorithm uses a technique named *flooding* to send a message from a node of the graph representing the network to all other nodes. We then make use of this algorithm to build a spanning tree of the network which can be used for efficient broadcast and convergecast of messages in the network as described next.

5.6.1 Flooding Algorithm

Our aim is to send a message from a single node to all nodes in the graph. This operation is called *broadcast* and has many applications in real networks, for example to inform all nodes of an alarm condition that occurs at a node. In the simplest case, we can have the following rules as a first attempt to solve this problem:

1. The initiator i sends a message $msg(i)$ to all of its neighbors.
2. Any node that receives message msg sends it to its neighbors except the one it received it from.

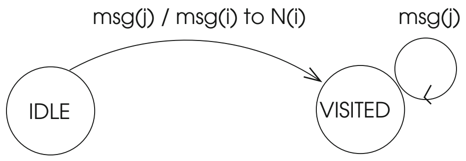


Fig. 5.4 FSM of the *Flooding* algorithm

This algorithm works fine and all nodes will receive message msg sent by p_i eventually. However, we can obtain a more efficient algorithm with less messages transferred between the nodes by a simple modification: A node sends msg to its neighbors only when it receives it for the first time. This way, duplicate transmission along an edge of the graph in the same direction is prevented. We now need a way to detect whether a message is received first time or not which can be implemented simply using a variable such as *visited* which is false initially and becomes true when msg arrives for the first time. Nevertheless, this modified algorithm is simple to implement by an FSM having two states as shown in Fig. 5.4, which will also aid us to understand the use of FSMs in distributed algorithms.

We can implement this algorithm based on the FSM as shown in Algorithm 5.2. When the message msg arrives for the first time, the VISITED state is entered and any further receptions of msg are ignored.

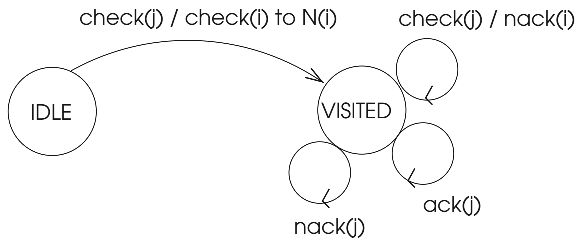
Algorithm 5.2 *Flooding*

```
1: { code for process $i$, message received from process $j$ }
2: $currstate \leftarrow$ IDLE                                ▷ start with IDLE state
3: if $i = initiator$ then
4:     send $msg(i)$ to $N(i)$
5:     $currstate \leftarrow$ VISITED
6: end if
7:
8: while $true$ do                                          ▷ all nodes execute this part
9:     receive($msg(j)$)
10:    case $currstate$ of
11:        IDLE: send $msg(i)$ to $N(i) \setminus j$
12:              $currstate \leftarrow$ VISITED
13:        VISITED:                                         ▷ do nothing
14: end while
```

We can have some improvements to this algorithm as follows.

- Instead of waiting forever at line 8, we can have a terminating condition. A process i can wait certain times such as the diameter $diam(G)$ of the graph. This is logical since $diam(G)$ is the longest path that can be traversed by the message msg. Once the message msg is received $diam(G)$ times, process i terminates.

Fig. 5.5 FSM of spanning tree construction using flooding



- We can have a counter commonly named *time-to-live* (TTL) contained in the message. Each time it is received, TTL is decremented and a message with 0 TTL value is not forwarded to neighbors.

Analysis

A careful look at this algorithm reveals that each edge of the graph will be traversed at most twice, once in each direction when both nodes at the ends of an edge start sending the message *msg* concurrently. Therefore, message complexity is $O(m)$. Assuming there is at least one message transfer at each time unit, time taken by this algorithm is the longest distance between any two vertices of the graph which is its diameter and thus, time complexity is $\Theta(diam(G))$.

5.6.2 Spanning Tree Construction Using Flooding

We can design a spanning tree construction of a network using the *Flooding* algorithm with few modifications. Building a spanning tree in a network environment means each node knows its parent and its children in the general sense. We will not attempt to store all of the tree structure at a special node or at each node of the graph since parent/children relationship at each node is adequate for transferring messages over the spanning tree. We have a single initiator as in the *Flooding* algorithm and this node becomes the root of the spanning tree to be formed. The first modification we have is to assign the sender j of the message $msg(j)$ as the *parent* of the receiver i if $msg(j)$ is received for the first time. Since we also require the parent to be aware of its children, node i should send an acknowledgment message $ack(i)$ to j to inform j of this situation. Otherwise, if node i already has a parent, meaning it has been visited before, it sends back a negative acknowledgment message $nack(i)$ to node j. We have, therefore, three types of messages; *check*, *ack*, and *nack*. Determining the types of messages is crucial in the design of distributed graph algorithms, moreover, determination of states is performed by messages if we are to use a FSM. Let us modify the FSM of Fig. 5.4 to reflect what we have been discussing. We can see that the states may remain the same since a node can be either in IDLE or VISITED state as before. Based on its state and the type of the message, we may need to take different actions. The modified FSM is shown in Fig. 5.5 with the VISITED state having all possible message types as input now.

This FSM can be directly translated to a distributed algorithm as shown in Algorithm 5.3 where additionally, a termination condition for a node is also specified. The activity of any node is finished when it has received *ack* or *nack* messages from all of its neighbors except the sender of the message it has received for the first time.

Algorithm 5.3 *Flooding2*

```
1: int $parent \leftarrow \emptyset$
2: set of int $childs \leftarrow \{\emptyset\}$ , $others \leftarrow \{\emptyset\}$
3: message types $check$, $ack$, $nack$
4:
5: if $i = initiator$ then
6:     send $check$ to $N(i)$
7:     $currstate \leftarrow$ VISITED
8: end if
9:
10: while $(childs \cup others) \neq (N(i) \backslash \{parent\})$ do      ▷ all nodes execute this part
11:     receive($msg(j)$)
12:     case $currstate \wedge msg(j).type$ of
13:         IDLE $\wedge$ $check$:      $parent \leftarrow j$
14:                                send $check$ to $N(i) \backslash j$
15:                                send $ack$ to $j$
16:                                $currstate \leftarrow$VISITED
17:         VISITED $\wedge$ $check$:   send $nack$ to $j$
18:         VISITED $\wedge$ $ack$:     $childs \leftarrow childs \cup \{j\}$      ▷ $j$ is now a child
19:         VISITED $\wedge$ $nack$:    $others \leftarrow childs \cup \{j\}$      ▷ $j$ is not a child
20: end while
```

The operation of this algorithm is shown in Fig. 5.6.

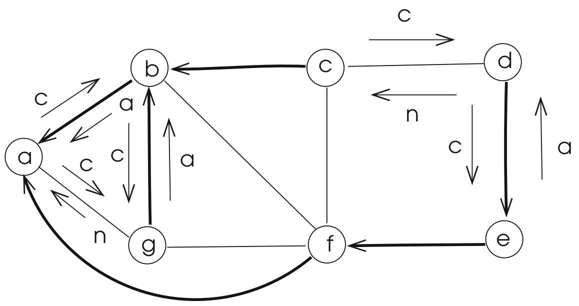


Fig. 5.6 A spanning tree constructed in a graph using flooding. The branch (g, b) is on tree but (g, a) is not since *check* message (c) from node a arrives at g later than c from b, which is replied by a *nack* (n) message. A similar situation is depicted for branch (e, d) where message c from node d is replied by an *ack* (a) message and (d, e) is included in the tree

We could have easily implemented this algorithm without using an FSM, a node having a parent or not basically shows its state as IDLE or VISITED. With this in mind, this algorithm is shown in Algorithm 5.4 as in [2]. However, for complicated distributed algorithms, using FSMs would ease the design and implementation.

Algorithm 5.4 *Flooding3*

```
1: int $parent \leftarrow \perp$
2: set of int $childs \leftarrow \{\emptyset\}$ , $others \leftarrow \{\emptyset\}$
3: message types $check$, $ack$, $nack$
4:
5: if $i = initiator$ then                                    ▷ root initiates tree construction
6:     send $check$ to $N(i)$
7:     $parent \leftarrow i$
8: end if
9:
10: while $(childs \cup others) \neq (N(i) \setminus \{parent\})$ do
11:     receive $msg(j)$
12:     case $msg(j).type$ of
13:         $\underline{check}$ : if $parent = \emptyset$ then              ▷ $check$ received first time
14:                 $parent \leftarrow j$
15:                 send $ack$ to $j$
16:                 send $msg(i).check$ to $N(i) \setminus \{j\}$
17:             else                                             ▷ $check$ received before
18:                 send $msg(i).reject$ to $j$
19:         $\underline{ack}$ : $childs \leftarrow childs \cup \{j\}$                       ▷ $j$ is a child
20:         $\underline{nack}$ : $others \leftarrow others \cup \{j\}$                     ▷ $j$ is not a child
21: end while
```

Analysis

Each edge of the graph will be traversed at least twice by *check/ack* or *check/nack* message pairs and at most four times when two nodes start to send *check* messages to each other simultaneously. Therefore, message complexity of this algorithm is $O(m)$. The depth of the tree constructed will be at most $n - 1$, assuming a linear network is built. If there is at least one message transfer per unit time, time complexity is $O(n)$.

5.6.3 Basic Communications

There are a number of basic communication operations performed in a distributed system. One such process is the *broadcast* which is initiated by a node by sending a message and all of the nodes in the distributed system have a copy of the message at the end of the broadcast operation. Another fundamental primitive is the *convergecast* where data from each node is collected at a special node in the system. We will look into these two operations in this section. One other activity is the multicast sending of messages in which a message is delivered to only a specified subset of processes.

Broadcast over a Spanning Tree

For the broadcast operation, we will assume a graph represents the network of the distributed system and a spanning tree T is already built by an algorithm similar to what we have discussed. The broadcast is initiated by a node by sending msg to all of its children. Any node on the tree T that has children simply forwards msg to all of its children. Since msg is transferred only over tree edges, the number of messages will be $n - 1$ for a graph with n vertices. Time taken will be the depth of T, assuming concurrent sending of messages at each level. Depth of T can be a maximum of $n - 1$ assuming a linear network.

Convergecast over a Spanning Tree

In certain networks, data from all nodes are to be collected at a node with more capabilities and this special node can then analyze and evaluate all of the data, provide reports containing statistics which can be transferred to more advanced computation centers or users for further processing. This situation is commonly encountered in wireless sensor networks where data sensed needs to go through these steps of operation. Collecting data is very much simplified when a spanning tree constructed beforehand is used. In this case, the leaves of the tree send their data to their parents, the parents combine their own data with those of leaves, and send these to their parents. An intermediate node may in fact perform some simple operation on data such as taking average or finding extreme values. This way, data sent upwards in the tree does not have to get much larger at each level. This process of gathering called *convergecast* continues until all data is collected at the special node, commonly called the *sink* in sensor networks. Algorithm 5.5 shows the pseudocode for the convergecast process over a spanning tree. Leaves of the tree start the algorithm and any intermediate node in the tree should wait until data from all of its children are received before combining these data with its own to be sent to its parent as realized at line 8 of the algorithm. The termination condition for the root of the tree is met when it receives the convergecast messages from all of its children at line 12. For all others, termination is on line 17 when they send their data to their parents.

Message and time complexities for this algorithm are the same as the *Broadcast* algorithm using similar reasoning. Figure 5.7 shows the operation of the $Convergecast$ algorithm using the spanning tree built in Fig. 5.6. The messages are labeled with pair (a, b); a showing the time frame and b is the duration of the message. We can see the highest level of tree finishes convergecast in 5 time units as this is the longest duration, followed by 6 units at level 2 and 2 units at level 1 for a total of 13 time units.

Algorithm 5.5 *Convergecast*

```
1: int parent
2: set of int childs, received ← {Ø} , data ← {Ø}
3: message types convcast
4:
5: if childs = {Ø} then                          ▷ leaf nodes start convergecast
6:     send convcast to parent
7: else                                          ▷ any intermediate node or root
8:     while childs ≠ received do     ▷ wait for convergecast messages from all children
9:         receive convcast(j)
10:        received ← received ∪ {j}
11:        data ← data∪ convcast(j)
12:    end while
13: end if
14: if i ≠ root then
15:     combine data into convcast
16:     send convcast to parent
17: end if
```

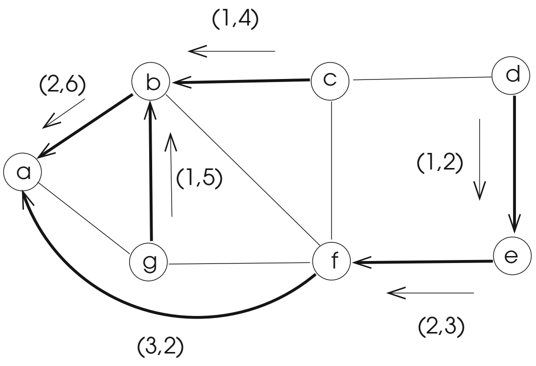


Fig. 5.7 Convergecast over the spanning tree of Fig. 5.6. Message label values show concurrent transfer of siblings and the duration of messages

5.6.4 Leader Election in a Ring

Leader or coordinator election is needed in distributed systems as this special node can initiate an algorithm and supervise the overall execution of the algorithm as in a SSI algorithm. The leader may also take remedy actions when failures are encountered in the execution of an algorithm. Nodes and communication links may physically fail and although we can initially assign a node as the leader of the network, we need to elect a new leader when failure happens. *Election algorithms* provide ways of assigning a new leader in the network when the current leader fails.

There are many leader election algorithms in literature for distributed systems. As an introductory distributed algorithm example, we will consider leader election in a ring with nodes having unique identifiers. The transfer of messages is in one direction only. This example can be conveniently described by a simple FSM with the following states:

- LEAD: The nodes have a leader in this stable state.
- ELECT: Election is going on when a node is in this state.

The main idea of this algorithm is that any node detecting the failure of the current leader initiates the algorithm by sending an *election* message containing its identifier to its neighbor at its right assuming a clockwise unidirectional ring. A node that receives this message changes its state to ELECT. If the identifier in the message is greater than its own identifier, it simply passes the election message to its next neighbor. Otherwise, it inserts its identifier which is greater than the identifier in the incoming message and sends it to the neighbor. We have two messages in this example:

- *election*: Sent by any node that detects leader failure. This message may be sent by more than one initiator.
- *leader*: The new leader broadcasts this message to notify all nodes that election is over.

When a process with identifier i receives a message with an identifier j in it, it checks and does one of the following:

- $i > j$: Process i replaces j with i in message and passes it to the next node.
- $i < j$: Process i simply passes message to next node.
- $i = j$: Process i becomes the leader and sends the *leader* message to its next neighbor.

In the last case, the election message originating from node i has returned to itself meaning it has the highest identifier among all active processes. Basically, the highest identifier is transferred between all functioning nodes and when the originator receives its own message, it determines it is the leader and sends the *leader* message to its neighbor which is then broadcast to all nodes by neighbor transfers. The FSM for this algorithm is depicted in Fig. 5.8.

Analysis

The worst case happens when the nodes are ordered from smallest to largest with respect to their identifiers in clockwise direction and start election concurrently in anticlockwise direction. The largest identifier message travels through all nodes n times, the second largest identifier is transferred $n - 1$ times and in total there will

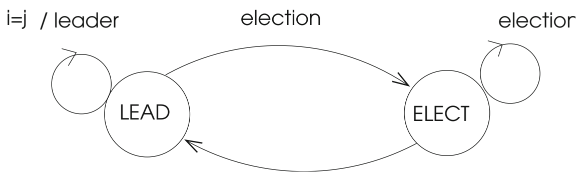


Fig. 5.8 FSM of the ring leader election algorithm

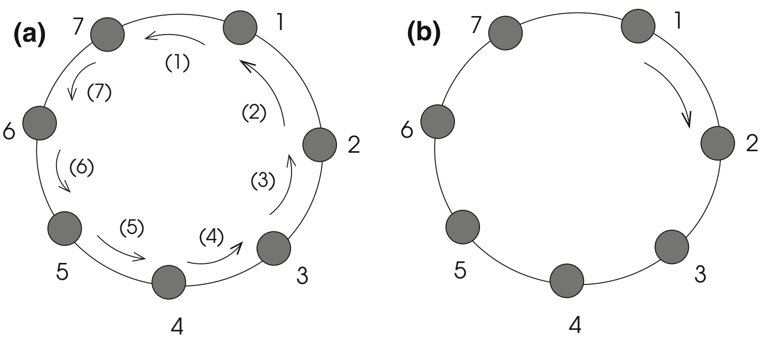


Fig. 5.9 Ring leader election algorithm: worst and best scenarios. In **a**, each message by the originator is tagged with the number of links it travels. For example message originating at node 7 is tagged with 7 since it goes through 7 edges back to node 7. The best case is depicted in **b**

be $\sum_{i=1}^{n} = n(n+1)/2$ messages as shown in Fig. 5.9a. The best case occurs for a total of $2n$-1 messages when messages are transmitted in clockwise direction as in Fig. 5.9b. In this case, even if all nodes start election concurrently, their messages will be purged by the next nodes for $n-1$ times and only the message of the highest identifier node, which is 7 in this case, will traverse the ring all the way back to the originator at n step. Total number of steps will then be $2n-1$, excluding the declaration message sent by the leader.

5.7 Chapter Notes

We have described distributed systems and the fundamental problems in designing algorithms for such systems in this chapter. Distributed systems are needed since they provide sharing of resources, fault tolerance and in various implementations, the application is inherently distributed such as a factory control system. The common platforms to implement distributed algorithms are the Internet, mobile ad hoc networks, wireless sensor networks, the Grid, and the Cloud. In many cases, we can model these networks suitably by graphs with vertices of a graph representing computational nodes and edges showing the communication links between them.

Distributed systems require distributed algorithms that run at the nodes of such a system. Synchronization and communication are two basic requirements in efficient design of such algorithms. Synchronization may be realized at various levels; hardware, operating system/middleware, and at application level. We see synchronization at application level using messages is commonly used due to versatility and easiness in implementation which may then be transferred to local synchronization primitives. In this so-called *message passing* distributed systems, the main communication and synchronization are achieved by messages only. The receiver of a message decides on what to do next mainly by the type of the arriving message. A synchronous distributed algorithm typically runs in rounds and the next round is not started until all nodes finish executing the current round. The synchronization at the beginning and end of round are commonly realized by special messages sent by a special node.

Distributed algorithms can be modeled by FSMs which are mathematical models which include states and transitions between states as we have outlined. We can design a distributed algorithm without a FSM but for complicated algorithms, FSMs provide a neater algorithm with visual aid and less error-prone than algorithms which otherwise could involve many decision- making statements.

We then described some sample distributed graph algorithms which include building a spanning tree of the graph, broadcast and convergecast operations over a spanning tree, and a leader election algorithm to find the new coordinator of nodes in a ring when leader fails. We need to prove that a distributed algorithm correctly achieves what it is intended for; and time, message, bit, and space complexities of a distributed algorithm are used to evaluate its performance. In general, message complexity is considered as the dominant cost of a distributed algorithm.

Exercises

1. The elevator algorithm of Example 5.1 is to be modified so that a green light showing moving up and a yellow light for downwards movement are added. Provide the necessary modifications to the FSM diagram, FSM table, and C code to incorporate these two outputs.
2. A binary bit string *S* has even parity if the number of bits in *S* is an even number and has odd parity otherwise. Provide the FSM diagram, FSM table, and the C code of an algorithm that reads a binary string bit-by-bit and decides to be in either even or odd state after each read. Use the programming style shown in the C code of Example 5.1.
3. We need to modify the broadcast algorithm over a spanning tree so that the initiator becomes aware that each node has received the broadcast message. This can be realized simply by each node deferring to send an acknowledgment to the sender of the message until it receives acknowledgments from all of its children, similar to the convergecast operation which should be started by the

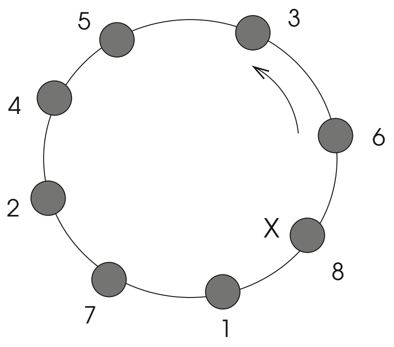


Fig. 5.10 Ring structure for Exercises 4

leaves of the spanning tree once they receive the broadcast message. Write the pseudocode for this algorithm with comments and work out its time and message complexities.

4. Show the execution of the ring election algorithm for the nodes shown in Fig. 5.10. Assume nodes 2 and 5 find concurrently that the leader is not working and decide to run an election.
5. In a fully connected graph with each node having unique identifiers, *bully algorithm* may be used to elect a new leader. A node u that finds leader is not functioning may start this algorithm by sending an election message to all nodes that have higher identifiers than itself. Any node v that receives this message sends back and *ack* message to the node u which then leaves election. The node v now starts election and this process continues until there is one winner which is the active node with highest identifier. The new leader broadcasts it is winning by a special message to all nodes. Write the pseudocode for this algorithm and find its time and message complexities. Show its operation in a complete graph of 8 nodes where nodes 4 and 6 find simultaneously the leader 8 is down.

References

1. Attiya H, Welch J (2004) Distributed computing: fundamentals, simulations, and advanced topics, 2nd edn. Wiley, New York
2. Erciyes K (2013) Distributed graph algorithms for computer networks. Springer computer communications and networks series. Springer, Berlin. ISBN- 10:1447151720 (May 16, 2013)
3. Foster I, Kesselman C (2004) The grid: blueprint for a new computing infrastructure. Morgan Kaufmann, San Mateo
4. Mell P, Grance T (2011) The NIST definition of cloud computing. National institute of standards and technology, US department of commerce, special publication, 800145
5. Tanenbaum AS, Steen MV (2007) Distributed systems, principles and paradigms, 2nd edn. Pearson-Prentice Hall, Upper Saddle River. ISBN 0-13-239227-5
6. Tel G (2000) Introduction to distributed algorithms, 2nd edn. Cambridge University Press, Cambridge