Unit – 6
Coordination

6.1 Clock Synchronization

- In a centralized system, time is unambiguous (System clock keeps time, all entities use this for time). When a process wants to know the time, it makes a system call and the kernel tells it.
- If process *A* asks for the time and then a little later process *B* asks for the time, the value that *B* gets will be higher than (or possibly equal to) the value *A* got. It will certainly not be lower.
- In a distributed system, each node has own system clock.

Clock Synchronization

Communication between processes in a distributed system can have unpredictable delays, processes can fail, messages may be lost .

Synchronization in distributed systems is harder than in centralized systems because the need for distributed algorithms.

Synchronization in distributed system is more complicated than in the centralized system because of 4 major reasons.

1. *Information is scatter among multiple machine.*
2. *Processes make decision based on only on local information.*
3. *A Single point of failure in system should be avoided*
4. ***No common clock or other precise global time source exists***

- =>Points number 1 to 3 explain that it is unpredictable to collect all the information in a single place for processing because for resource allocation all requests need to send into a single manager, which examines and grant or denies request based on resource allocation table. For heavy system it is great burden on particular machine or process only. Moreover, if a system is failed then entire system become non predictable or unreliable
- => **Clock is crucial issue because it is ambiguous. If a process of a system wants to talk to its kernel. It means the time of asking and reply should hold the principles HAPPEN BEFORE.**

Clock Synchronization

It is the process of setting all the cooperating systems of distributed network to the same logical or physical clock.

Logical and Physical Clocks

- Clock synchronization is dramatic and it is fitting in process execution. But, Is it possible to synchronize all the clocks in the distributed environment into single clock.

- There are different concepts and implementations regarding the clock synchronization by using logical and physical clocks.

•logical clocks - to provide consistent event ordering
•physical clocks - clocks whose values must not deviate from the real time by more than a certain amount.

Logical and Physical Clocks

- **Logical Clocks.**
  - For many applications:
    - it is sufficient that all machines agree on the same time.
    - it is not essential that this time also agree with the real time
  - E.g. make example - it is adequate that all machines agree that it is 10:00 even if it is really 10:02.
  - Meaning: it is the *internal consistency of the clocks that matters, not* whether they are particularly close to the real time.
  - For these algorithms it is conventional to speak of the clocks as ***logical clocks.***

- **Physical Clocks**
- when the additional constraint(limitation or restriction) is present that the clocks
  - must not only be the same,
  - but also must not deviate from the real time by more than a certain amount,
- Then the clocks are called ***physical clocks.***

Problems with Different Clocks

- There are several problems that occur as a repercussion(an unintended consequence of an event or action, especially an unwelcome one) of clock rate differences
- Besides the incorrectness of the time itself, there are problems associated with clock skew(neither parallel nor intersecting) that take on more complexity in a distributed system in which several computers will need to realize the same global time.
- **Example:** in Unix systems the make command is used to compile new or modified code without the need to recompile unchanged code. The make command uses the clock of the machine it runs on to determine which source files need to be recompiled. If the sources reside on a separate file server and the two machines have unsynchronized clocks, the make program might not produce the correct results
- Hence there is a need of **Clock Synchronization**

- The physical clocks are used to adjust the time of nodes. Each node in the system can share its local time with other nodes in the system. The time is set based on UTC (Universal Time Coordination). UTC is used as a reference time clock for the nodes in the system.
- The clock synchronization can be achieved by 2 ways: External and Internal Clock Synchronization.

1. **External clock synchronization** is the one in which an external reference clock is present. It is used as a reference and the nodes in the system can set and adjust their time accordingly.
2. **Internal clock synchronization** is the one in which each node shares its time with other nodes and all the nodes set and adjust their times accordingly

- The physical clocks are used to adjust the time of nodes. Each node in the system can share its local time with other nodes in the system. The time is set based on UTC (Universal Time Coordination). UTC is used as a reference time clock for the nodes in the system.
- The clock synchronization can be achieved by 2 ways: External and Internal Clock Synchronization.

1. **External clock synchronization** is the one in which an external reference clock is present. It is used as a reference and the nodes in the system can set and adjust their time accordingly.
2. **Internal clock synchronization** is the one in which each node shares its time with other nodes and all the nodes set and adjust their times accordingly

- There are 2 types of clock synchronization algorithms: Centralized and Distributed.

1. **Centralized** is the one in which a time server is used as a reference. The single time server propagates its time to the nodes and all the nodes adjust the time accordingly. It is dependent on single time server so if that node fails, the whole system will lose synchronization. Examples of centralized are-Berkeley Algorithm, Passive Time Server, Active Time Server etc.
2. **Distributed** is the one in which there is no centralized time server present. Instead the nodes adjust their time by using their local time and then, taking the average of the differences of time with other nodes. Distributed algorithms overcome the issue of centralized algorithms like the scalability and single point failure. Examples of Distributed algorithms are – Global Averaging Algorithm, Localized Averaging Algorithm, NTP(Network time protocol) etc.

6.2 Logical Clock

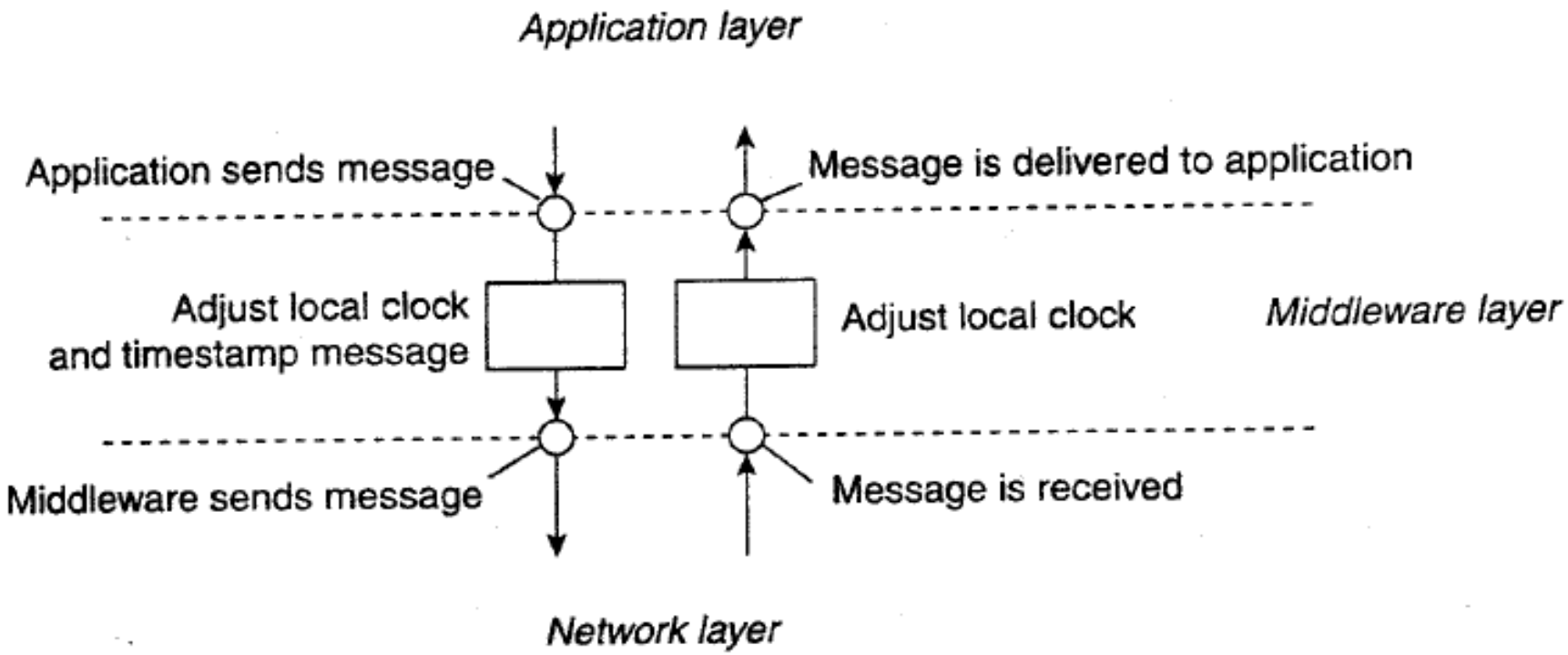
- Lamport's Logical Clocks
- To synchronize logical clocks, Lamport defined a relation called happens-before.
- The expression *a* ~ *b* is read "*a* happens before *b*" and means that all processes agree that first event *a* occurs, then afterward, event *b* occurs.
- The happens-before relation can be observed directly in two situations:

1. If *a* and *b* are events in the same process, and *a* occurs before *b,* then *a* ~ *b* is true.

2. If *a* is the event of a message being sent by one process, and *b* is the event of the message being received by another process, then *a* ~ *b* is also true.

- A message cannot be received before it is sent, or even at the same time it is sent, since it takes a finite, nonzero amount of time to arrive.

- Happens-before is a transitive relation, so if *a* ~ band *b* ~ *c,* then *a* ~ *c.*
- If two events, *x* and *y,* happen in different processes that do not exchange messages (not even indirectly via third parties), then *x* ~ *y* is not true, but neither is *y* ~ *x.*
- These events are said to be **concurrent,** which simply means that nothing can be said (or need be said) about when the events happened or which event happened first.


Application layer
Application sends message
Message is delivered to application
Adjust local clock
and timestamp message
Adjust local clock
Middleware layer
Middleware sends message
Message is received
Network layer

6.3 Mutual Exclusion

- Fundamental to distributed systems is the concurrency and collaboration among multiple processes.
- In many cases, this also means that processes will need to simultaneously access the same resources.
- To prevent that such concurrent accesses corrupt the resource, or make it inconsistent, solutions are needed to grant mutual exclusive access by processes

- Distributed mutual exclusion algorithms can be classified into two different categories.
- In **token-based** solutions mutual exclusion is achieved by passing a special message between the processes, known as a token.
- There is only one token available and who ever has that token is allowed to access the shared resource.
- When finished, the token is passed on to a next process.
- If a process having the token is not interested in accessing the resource, it simply passes it on.

- **Token-based solutions have a few important properties.**
- First, depending on the how the processes are organized, they can fairly easily ensure that every process will get a chance at accessing the resource. In other words, they avoid starvation(dead of process).
- Second, deadlocks by which several processes are waiting for each other to proceed, can easily be avoided, contributing to their simplicity.
- Unfortunately, the main drawback of token-based solutions is a rather serious one: when the token is lost (e.g., because the process holding it crashed), an intricate distributed procedure needs to be started to ensure that a new token is created, but above all, that it is also the only token.

- As an alternative, many distributed mutual exclusion algorithms follow a permission-based approach.
- In this case. a process wanting to access the resource first requires the permission of other processes.
- There are many different ways toward granting such a permission and in the sections that follow we will consider a few of them.

- **A Centralized Algorithm**
- The most straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator.
- Whenever a process wants to access a shared resource, it sends a request message to the coordinator stating which resource it wants to access and asking for permission.
- If no other process is currently accessing that resource, the coordinator sends back a reply granting permission, as shown in Fig.6(a).
- When the reply arrives, the requesting process can go ahead.

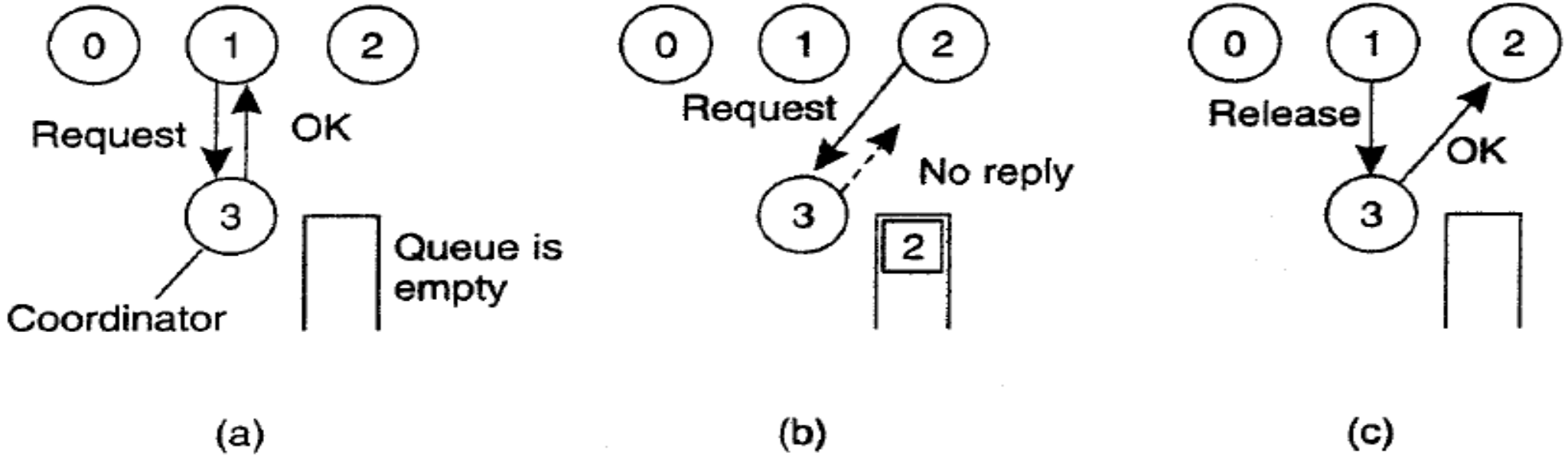


Figure 6-14. (a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply. (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

- Now suppose that another process, 2 in Fig. 6-14(b), asks for permission to access the resource.
- The coordinator knows that a different process is already at the resource, so it cannot grant permission. The exact method used to deny permission is system dependent.
- In Fig. 6-14(b), the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply. Alternatively, it could send a reply saying "permission denied." Either way, it queues the request from 2 for the time being and waits for more messages.

- When process 1 is finished with the resource, it sends a message to the coordinator releasing its exclusive access, as shown in Fig.6-14(c).
- The coordinator takes the first item off the queue of deferred requests and sends that process a grant message.
- If the process was still blocked (i.e., this is the first message to it), it unblocks and accesses the resource. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later.
- Either way, when it sees the grant, it can go ahead as well.

- It is easy to see that the algorithm guarantees mutual exclusion: the coordinator only lets one process at a time to the resource. It is also fair, since requests are granted in the order in which they are received.
- No process ever waits forever (no starvation). The scheme is easy to implement, too, and requires only three messages per use of resource (request, grant, release).
- It's simplicity makes an attractive solution for many practical situations.
- The centralized approach also has shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down.
- If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back.

- **A Decentralized Algorithm**
- Having a single coordinator is often a poor approach. Let us take a look at fully decentralized solution. Lin et al. (2004) propose to use a voting algorithm that can be executed using a DHT-based system. In essence, their solution extends the central coordinator in the following way.
- Each resource is assumed to be replicated *n* times. Every replica has its own coordinator for controlling the access by concurrent processes.
- However, whenever a process wants to access the resource, it will simply need to get a majority vote from 111 > *nl2* coordinators.
- Unlike in the centralized scheme discussed before, we assume that when a coordinator does not give permission to access a resource (which it will do when it had granted permission to another process), it will tell the requester.

- This scheme essentially makes the original centralized solution less vulnerable to failures of a single coordinator.
- The assumption is that when a coordinator crashes, it recovers quickly but will have forgotten any vote it gave before it crashed.

6.4 Election Algorithm

- Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform some special role. In general, it does not matter which process takes on this special responsibility, but one of them has to do it.
- In this section we will look at algorithms for electing a coordinator (using this as a generic name for the special process).
- If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special.
- Consequently, we will assume that each process has a unique number, for example, its network address (for simplicity, we will assume one process per machine).
- In general, election algorithms attempt to locate the process with the highest process number and designate it as coordinator.
- The algorithms differ in the way they do the location.

- Furthermore, we also assume that every process knows the process number of every other process.
- What the processes do not know is which ones are currently up and which ones are currently down.
- The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be.

Traditional Election Algorithms

- **The Bully Algorithm**
- As a first example, consider the bully algorithm devised by Garcia-Molina (1982). When any process notices that the coordinator is no longer responding to requests, it initiates an election. A process, *P,* holds an election as follows:

1. P sends an *ELECTION* message to all processes with higher numbers.

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

3. If one of the higher-ups answers, it takes over. *P's* job is done.

- At any moment, a process can get an *ELECTION* message from one of its lower-numbered colleagues.
- When such a message arrives, the receiver sends an *OK* message back to the sender to indicate that he is alive and will take over.
- The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator.
- It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.
- If a process that was previously down comes back up, it holds an election.
- If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job.
- Thus the biggest guy in town always wins, hence the name "bully algorithm."

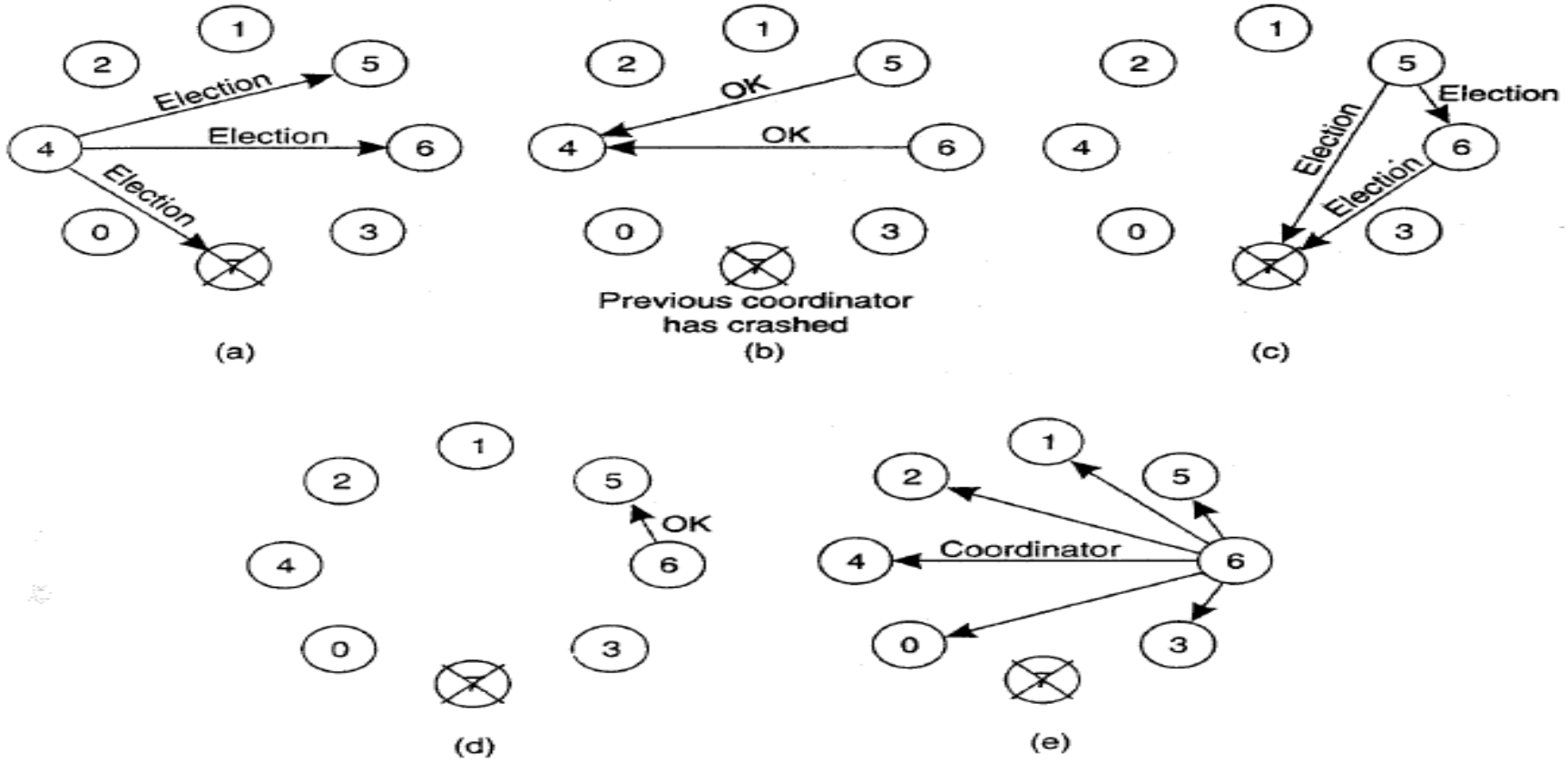


Figure 6·20. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

- In Fig. 6-20 we see an example of how the bully algorithm works. The group consists of eight processes, numbered from 0 to 7.
- Previously process 7 was the coordinator, but it has just crashed. Process 4 is the first one to notice this, so it sends *ELECTION* messages to all the processes higher than it, namely 5, 6, and 7.as shown in Fig. 6-20(a).
- Processes 5 and 6 both respond with *OK,* as shown in Fig. 6-20(b). Upon getting the first of these responses, 4 knows that its job is over.
- It knows that one of these bigwigs will take over and become coordinator.
- It just sits back and waits to see who the winner will be (although at this point it can make a pretty good guess).

- In Fig. 6-20(c), both 5 and 6 hold elections, each one only sending messages to those processes higher than itself.
- In Fig. 6-20(d) process 6 tells 5 that it will take over. At this point 6 knows that 7 is dead and that it (6) is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed.
- When it is ready to take over, 6 announces this by sending a *COORDINATOR* message to all running processes.
- When 4 gets this message, it can now continue with the operation it was trying to do when it discovered that 7 was dead, but using 6 as the coordinator this time.
- In this way the failure of 7 is handled and the work can continue.
- If process 7 is ever restarted, it will just send an the others a *COORDINATOR* message and bully them into submission.

- **A Ring Algorithm**
- Another election algorithm is based on the use of a ring. Unlike some ring algorithms, this one does not use a token.
- We assume that the processes are physically or logically ordered, so that each process knows who its successor is.
- When any process notices that the coordinator is not functioning, it builds an *ELECTION* message containing its own process number and sends the message to its successor.
- If the successor is down, the sender skips over the successor and goes to the next member along the ring or the one after that, until a running process is located.
- At each step along the way, the sender adds its own process number to the list in the message effectively making itself a candidate to be elected as coordinator.

- Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message containing its own process number.
- At that point, the message type is changed to *COORDINATOR* and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest number) and who the members of the new ring are.
- When this message has circulated once, it is removed and everyone goes back to work.

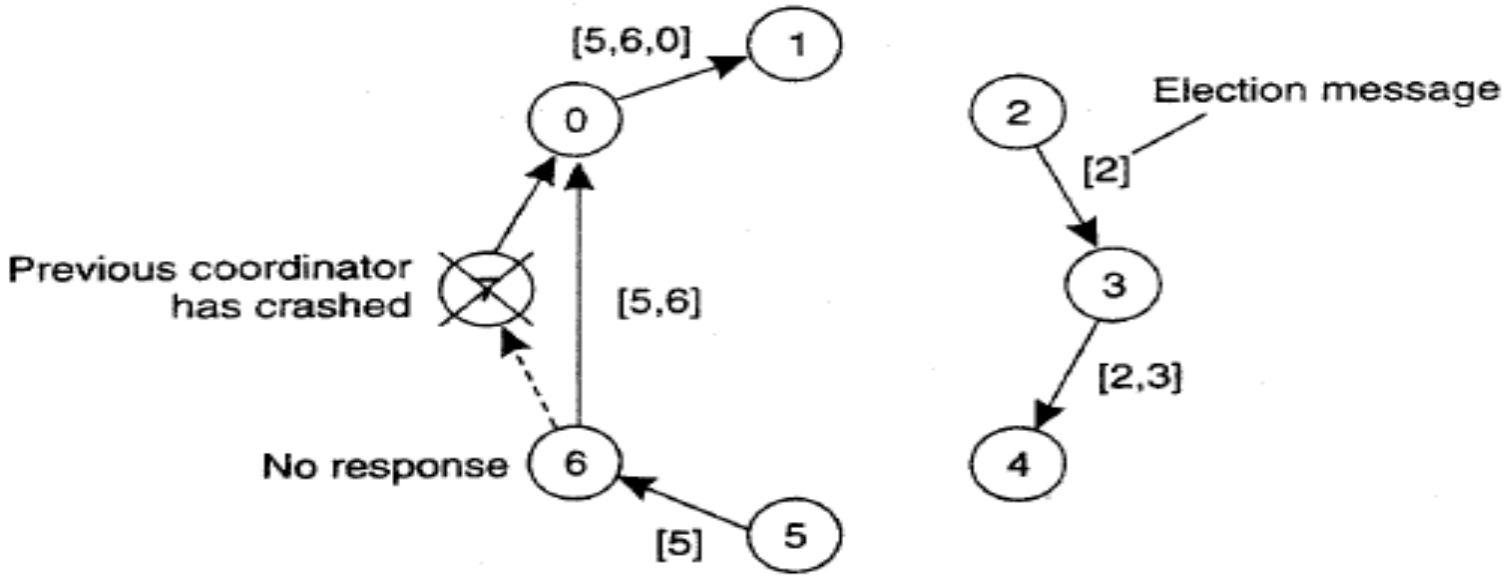


Figure 6-21. Election algorithm using a ring.

In Fig. 6-21 we see what happens if two processes, 2 and 5, discover simultaneously that the previous coordinator, process 7, has crashed. Each of these builds an *ELECTION* message and and each of them starts circulating its message, independent of the other one. Eventually, both messages will go all the way around, and both 2 and 5 will convert them into *COORDINATOR* messages, with exactly the same members and in the same order. When both have gone around again, both will be removed. It does no harm to have extra messages circulating; at worst it consumes a little bandwidth, but this not considered wasteful.

- In Fig. 6-21 we see what happens if two processes, 2 and 5, discover simultaneously that the previous coordinator, process 7, has crashed.
- Each of these builds an *ELECTION* message and each of them starts circulating its message, independent of the other one.
- Eventually, both messages will go all the way around, and both 2 and 5 will convert them into *COORDINATOR* messages, with exactly the same members and in the same order.
- When both have gone around again, both will be removed. It does no harm to have extra messages circulating;at worst it consumes a little bandwidth, but this not considered wasteful.

- **Elections in Wireless Environments**
- Traditional election algorithms are generally based on assumptions that are not realistic in wireless environments. For example, they assume that message passing is reliable and that the topology of the network does not change. These assumptions are false in most wireless environments, especially those for mobile ad hoc networks.
- Only few protocols for elections have been developed that work in ad hoc networks.
- Vasudevan et al. (2004) propose a solution that can handle failing nodes and partitioning networks. An important property of their solution is that the *best* leader can be elected rather than just a random as was more or less the case in the previously discussed solutions.

- Consider a wireless ad hoc network.
- To elect a leader, any node in the network, called the source, can initiate an election by sending an *ELECTION* message to its immediate neighbors (i.e., the nodes in its range).
- When a node receives an *ELECTION* for the first time, it designates the sender as its parent, and subsequently sends out an *ELECTION* message to all its immediate neighbors, except for the parent.
- When a node receives an *ELECTION* message from a node other than its parent, it merely acknowledges the receipt.

- This process is illustrated in Fig. 6-22. Nodes have been labeled *a* to *j,* along with their capacity. Node *a* initiates an election by broadcasting an *ELECTION* message to nodes band *j,* as shown in Fig. 6-22(b).
- After that step, *ELECTION* messages are propagated to all nodes, ending with the situation shown in Fig. 6- 22(e), where we have omitted the last broadcast by nodes *f* and *i:*
- From there on, each node reports to its parent the node with the best capacity, as shown in Fig.6-22(f).
- For example, when node *g* receives the acknowledgments from its children *e* and *h,* it will notice that *h* is the best node, propagating *[h,* 8] to its own parent, node *b.* In the end, the source will note that *h* is the best leader and will broadcast this information to all other nodes.

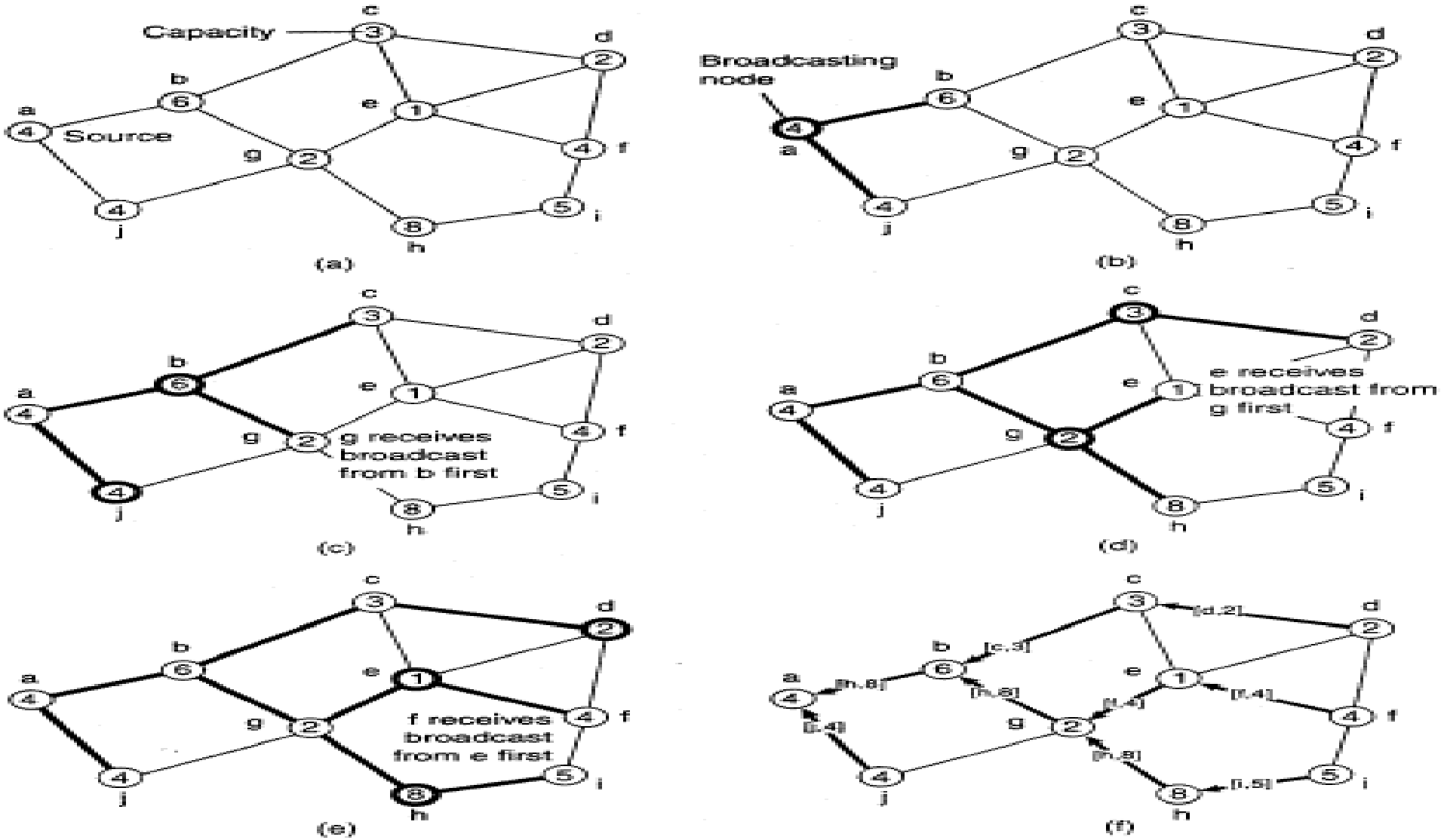


Figure 6-22. Election algorithm in a wireless network, with node *a* as the source. (a) Initial network. (b)-(e) The build-tree phase (last broadcast step by nodes f and i not shown). (f) Reporting of best node to source.

- When multiple elections are initiated, each node will decide to join only one election.
- To this end, each source tags its *ELECTION* message with a unique identifier.
- Nodes will participate only in the election with the highest identifier, stopping any running participation in other elections.
- With some minor adjustments, the protocol can be shown to operate also when the network partitions, and when nodes join and leave.

- **Elections in Large-Scale Systems**
- The algorithms we have been discussing so far generally apply· to relatively small distributed systems.
- Moreover, the algorithms concentrate on the selection of only a single node. There are situations when several nodes should actually be selected, such as in the case of superpeers in peer-to-peer networks.
- In this section, we concentrate specifically on the problem of selecting superpeers.
- Lo et al. (2005) identified the following requirements that need to be met for superpeer selection:

1. Normal nodes should have low-latency access to superpeers.

2. Superpeers should be evenly distributed across the overlay network.

3. There should be a predefined portion of superpeers relative to the total number of nodes in the overlay network.

4. Each superpeer should not need to serve more than a fixed number of normal nodes.

6.5 Location System

- A real-time location system (RTLS) is one of a number of technologies that detects the current geolocation of a target, which may be anything from a vehicle to an item in a manufacturing plant to a person.
- RTLS-capable products are used in an ever-increasing number of sectors including supply chain management (SCM), health care, the military, retail, recreation, and postal and courier services.

- RTLS is typically embedded in a product, such as a mobile phone or a navigational system.
- Most such systems consist of wireless nodes -- typically tags or badges -- that emit signals and readers that receive those signals. Current real-time location systems are based on wireless technologies, such as Wi-Fi, Bluetooth, ultrawideband, RFID, and GPS.

RTLS applications include:

- **Fleet tracking:** Fleet-tracking RTLS systems make it possible for an enterprise to track vehicle location and speed, optimize routes, schedule jobs, aid navigation and analyze driver efficiency.
- **Navigation**: The most basic navigation services provide directions for how to get from Point A to Point B. Incorporating GPS, mapping and mobile cellular technology will enable more complex navigation services.

•**Inventory and asset tracking:** RFID(Radio Frequency Identification) technologies are widely used for asset and inventory tracking. RFID tags communicate wirelessly with RFID readers throughout the enterprise.

•**Personnel tracking:** Different technologies are used for on-site personnel and workers in the field. Systems that track field workers are typically GPS-enabled mobile phones. On-site personnel tracking systems often use RFID technology, such as RFID-enabled badges.

•**Network security**: Wi-Fi Protected Access (WPA) can limit the physical area from which a user can connect to restrict access based on the user's location.

- Advances in electronic location technology and the coming of age of mobile computing have opened the door for location-aware applications to permeate all aspects of everyday life.
- Location is at the core of a large number of high-value applications ranging from the life-and-death context of emergency response to serendipitous social meet-ups.
- For example, the market for GPS products and services alone is expected to grow to US$200 billion by 2015.
- Unfortunately, there is no single location technology that is good for every situation and exhibits high accuracy, low cost, and universal coverage.
- In fact, high accuracy and good coverage seldom coexist, and when they do, it comes at an extreme cost.
- Instead, the modern localization landscape is a kaleidoscope of location systems based on a multitude of different technologies including satellite, mobile telephony, 802.11, ultrasound, and infrared among others.

6.6 Distributed Event Matching

- Today's distributed systems running at Internet scale are heterogeneous, asynchronous, and loosely coupled.
- A variety of computing devices can be connected into the system, ranging from traditional workstations, mainframes to intelligent embedded devices, wireless PDAs and mobile phones.
- The system must support applications that continuously monitor or react to changes in the environment, inform other components asynchronously about transitions in their internal state, request services from and provide services to other components.

- **Publish/subscribe**(pub/sub for short) stands for a type of communication paradigm.
- It allows clients to publish events(useful information), subscribe to a pattern of events and asynchronously get notified of interested events once they become available.
- Unlike traditional RPCs where communication is based on knowing the other party's address, clients in pub/sub can send and receive events without knowing each other's identity.
- As such, it is considered as a natural fit to the loosely coupled nature of modern distributed applications.
- Although pub/sub is not a new invention, its use in large-scale wide-area communication has vastly increased its popularity in recent years.

- **Event forwarding** is the primary purpose of a pub/sub system. In a distributed network, event forwarding has both local and global aspects.
- Locally an event is matched against a set of registered subscriptions based on its content, which we call a “content-match”.
- Content-match is a hard combinatorial problem, as the matching now is based on an event’s content, rather than on a fixed destination address.
- It is accepted that content-match should be done as few times as possible.

Event Routing

- The core mechanism behind a distributed pub/sub system is event routing.
- Informally, event routing is the process of delivering an event to all the subscribers that issued a matching subscription before the publication.
- This involves a visit of the nodes in the Notification Service in order to find, for any published event, all the clients whose registered subscription is present in the system at publication time.
- The impossibility of defining a global temporal ordering between a subscription and a publication that occurred at two different nodes makes this definition of routing rather ambiguous.

- The main issue with an event routing algorithm is scalability. That is, an increase of the number of brokers, subscriptions and publications should not cause a serious (e.g., exponential) degradation of performance.
- This requires on one hand controlling the publication process, in order to possibly involve in propagation of events only those brokers hosting matching subscriptions.
- On the other, reducing the amount of routing information to be maintained at brokers, in order to support and flexibly allow subscription changes.
- These two aspects are evidently conflicting and reaching a balance between them is the main aim of a pub/sub system's designer.

Matching

- Matching is the process of checking an event against a subscription. Matching is performed by the pub/sub system in order to determine whether dispatching the event to a subscriber or not.
- As the context of interest is that of large-scale systems, we expect on one side the overall number of subscriptions in the system to be very high, and on the other a high rate of events to be processed.
- Then, in general the matching operation has to be performed often and on massive data sizes.
- While obviously this poses no problems in a topic-based system, where matching reduces to a simple table lookup, it is a fundamental issue for the overall performance of a content-based system.
- The trivial solution of testing sequentially each subscription against the event to be matched often results in poor performance.

- Globally an event needs to traverse many routers before reaching interested subscribers.
- This is because we no longer rely on a single router to find out all the matching subscriptions.
- During the traversal of an event, similar content-match will be performed at multiple pub/sub routers.
- This will further aggravate(worse) the situation.
- As users are sensitive to the end-to-end delay reflected in event forwarding, our first challenge is to conquer the complexity inherent in content-based event forwarding to shorten the delivery time

6.7 Gossip Based coordination

- In basic gossip-based protocols, each node contacts one or a few nodes in each round (usually chosen at random), and exchanges information with these nodes.
- The dynamics of information spread resembles the spread of an epidemic and lead to high robustness, reliability and self-stabilization .
- Being randomized, rather than deterministic, these protocols are simple and do not require to maintain any event routing data structure at each node trying to timely track churn and subscriptions changes.
- The drawback is a moderate redundancy in message overhead compared to deterministic solution.

- Gossiping is therefore a probabilistic and fully distributed approach to event routing and the basic algorithm achieves high stability under high network dynamics, and scales gracefully to a huge number of nodes .

- In gossip protocols, the random choice of the nodes to contact can be sometimes driven by local information, acquired by a node during its execution, describing the state either of the network or of the subscription distribution or both .

- Groups are built and organized in hierarchies according to the physical proximity of nodes.
- Each process maintains in its view the identities and the subscriptions of its neighbors in a group.
- Special members in a group, namely delegates, maintain an aggregation of the subscriptions within a group and have access to the delegates view of nodes at adjacent levels of the tree.
- Events are gossiped throughout the tree. The membership information allows to exclude from gossiping the nodes that are not interested in an event.