CSE 535 Asynchronous Systems
Elementary Graph Algorithms

YoungMin Kwon

Contents

- Terminologies
- Distributed spanning-tree algorithms

Centralized vs Distributed

- Centralized Algorithm
  - Predominant amount of work is performed by one or few processes.
  - E.g. Client-Server configuration
  - Potential bottleneck for processing and bandwidth
  - Single point of failure

- Distributed Algorithm
  - Each process plays an equal role w.r.t. message overhead, time overhead, space overhead…
  - It is difficult to design a purely distributed algorithm (e.g. Chandy-Lamport algorithm has an initiator)
  - Peer-to-peer network, ubiquitous and ad-hoc networks, mobile systems will require distributed solutions

Symmetric vs Asymmetric

- Symmetric algorithm
  - All processes execute the same logical functions
- Asymmetric algorithm
  - Different processes execute logically different functions
  - E.g. centralized algorithms, client-server configuration

Deterministic vs Nondeterministic

- Deterministic receive
  - Specifies the source from which it receives a message
- Nondeterministic receive
  - Can receive a message from any source

- Deterministic execution
  - An execution of a distributed program without any nondeterministic receives
  - Re-execution can reproduce the same partial order of events
- Nondeterministic execution
  - An execution of a distributed program that has at least one nondeterministic receive
  - Re-execution may result in different outcomes

Synchronous vs Asynchronous

- Synchronous system
  - Known upper bound on communication delay
  - Known bounded drift rate for the local clock
  - Known upper bound on time to execute a logical step

- Asynchronous system
  - A system that does not satisfy any of the three properties of Synchronous systems

Online vs Offline

- On-line algorithm
  - An algorithm that executes as the data is being generated
  - On-line scheduling can dynamically handle new requests
  - On-line debugging can detect errors as they occur instead of collecting the entire traces

- Off-line algorithm
  - An algorithm that requires all the date to be available before the execution begins

Anonymous Algorithm, Uniform Algorithm

- Anonymous system
  - A system where neither process ids nor processor ids are used to make any execution decisions
- Anonymous algorithm
  - An algorithm that runs on anonymous systems
  - Hence, it does not use any process ids or processor ids in the code
- Uniform algorithm
  - An algorithm that does not use the number of processes in the system as a parameter in its code.
  - Promotes scalability
  - A process can leave or join the system

Adaptive algorithm

- Given a problem X, if the complexity of an algorithm can be expressed in terms of the number of participating processes, the algorithm is called an adaptive algorithm.

- E.g. A mutual exclusion algorithm whose complexity depends on the number of processes contending for a critical section.

Execution Inhibition

- Protocol that requires processes to suspend until some event are performed
- A protocol is non-inhibitory if
  - No event is disabled in the execution of the protocol
- A disabled event is locally delayed if
  - The event is enabled after a while
  - There is no intervening receive event
- A disabled event is globally delayed if
  - The event is disabled until receiving a message from other processes

Execution Inhibition

- A protocol is
  - Send inhibitory: if the delayed events are send events
  - Receive inhibitory: if the delayed events are receive events
  - Internal event inhibitory: if the delayed events are internal events

Process Failure Models

- Fail-Stop
  - Failure by stopping execution
  - Other processes can learn that the process has failed

- Crash
  - Failure by stopping execution
  - Other processes do not learn this crash

Process Failure Models

- Receive omission
  - Failure by intermittently receiving only some of the messages sent to it.
- Send omission
  - Failure by intermittently sending only some of the messages it is supposed to send
- Byzantine or malicious failure
  - A process exhibits any arbitrary behavior

Communication Failure Models

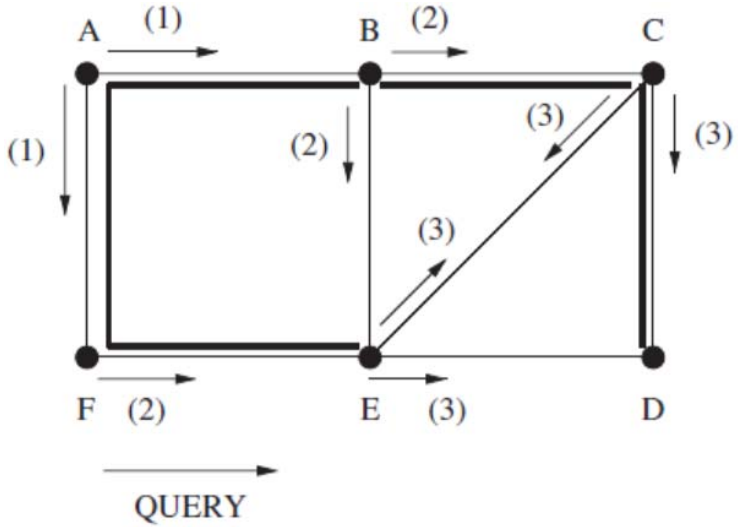
- Crash failure
  - A link stops carrying messages
- Omission failure
  - A link carries some messages but not the others
- Byzantine failure
  - A link exhibits arbitrary behaviors including creating spurious messages and modifying messages

Wait-free algorithm

- An algorithm that is resilient to n-1 process failures
- Any process must complete in a bounded number of steps irrespective of the failure of all other processes
- E.g. mutual exclusion synchronization
  - Enable a process to access its critical section even if the process in the critical section fails or not exiting the critical section

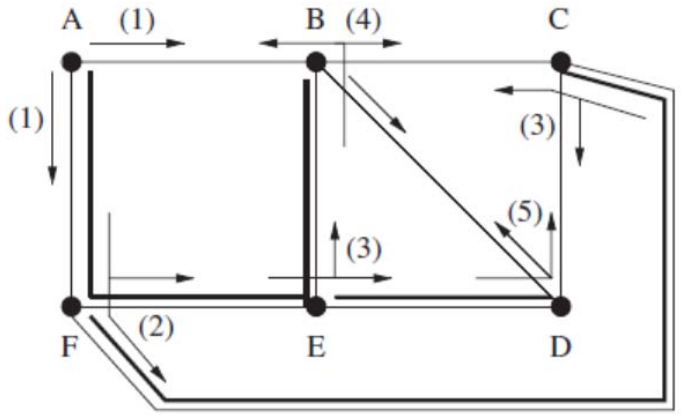
Spanning-Tree Algorithm: Synchronous Single-Initiator

```
(local variables)
int visited, depth ⟵ 0
int parent ⟵ ⊥
set of int Neighbors ⟵ set of neighbors
(message types)
QUERY

(1)  if i = root then
(2)      visited ⟵ 1;
(3)      depth ⟵ 0;
(4)      send QUERY to Neighbors;
(5)  for round = 1 to diameter do
(6)      if visited = 0 then
(7)          if any QUERY messages arrive then
(8)              parent ⟵ randomly select a node from which
                    QUERY was received;
(9)              visited ⟵ 1;
(10)             depth ⟵ round;
(11)             send QUERY to Neighbors \ {senders of
                    QUERYs received in this round};
(12)     delete any QUERY messages that arrived in this round.
```



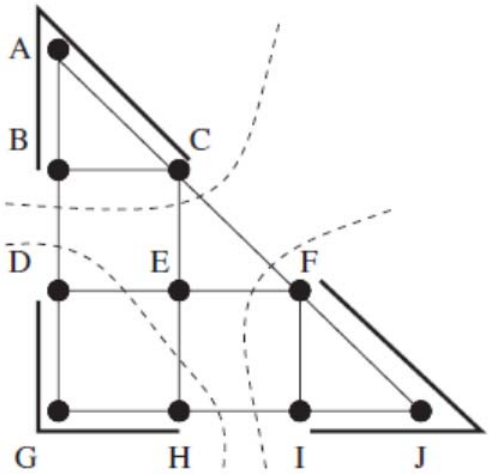
Spanning-Tree Algorithm: Asynchronous Single-Initiator

```
(local variables)
int parent ⟵ ⊥
set of int Children, Unrelated ⟵ ∅
set of int Neighbors ⟵ set of neighbors
(message types)
QUERY, ACCEPT, REJECT

(1)   When the predesignated root node wants to initiate the algorithm:
(1a)  if (i = root and parent = ⊥) then
(1b)      send QUERY to all neighbors;
(1c)      parent ⟵ i.

(2)   When QUERY arrives from j:
(2a)  if parent = ⊥ then
(2b)      parent ⟵ j;
(2c)      send ACCEPT to j;
(2d)      send QUERY to all neighbors except j;
(2e)      if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(2f)          terminate.
(2g)  else send REJECT to j.

(3)   When ACCEPT arrives from j:
(3a)  Children ⟵ Children ∪ {j};
(3b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(3c)      terminate.

(4)   When REJECT arrives from j:
(4a)  Unrelated ⟵ Unrelated ∪ {j};
(4b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(4c)      terminate.
```



Spanning-Tree Algorithm: Asynchronous Concurrent-Initiator Using Flooding

```
(local variables)
int parent, myroot ⟵ ⊥
set of int Children, Unrelated ⟵ ∅
set of int Neighbors ⟵ set of neighbors
(message types)
QUERY, ACCEPT, REJECT
(1)   When the node wants to initiate the algorithm as a root:
(1a)  if (parent = ⊥) then
(1b)      send QUERY(i) to all neighbors;
(1c)      parent, myroot ⟵ i.

(2)   When QUERY(newroot) arrives from j:
(2a)  if myroot < newroot then    // discard earlier partial execution due
                                  // to its lower priority
(2b)      parent ⟵ j; myroot ⟵ newroot; Children, Unrelated ⟵ ∅;
(2c)      send QUERY(newroot) to all neighbors except j;
(2d)      if Neighbors = {j} then
(2e)          send ACCEPT(myroot) to j; terminate.    // leaf node
(2f)  else send REJECT(newroot) to j.
          // if newroot = myroot then parent is already identified.
          // if newroot < myroot ignore the QUERY. j will update its root
          // when it receives QUERY(myroot).
```



(3) When ACCEPT(*newroot*) arrives from *j*:
(3a) **if** *newroot* = *myroot* **then**
(3b) *Children* ⟵ *Children* ∪ {*j*};
(3c) **if** (*Children* ∪ *Unrelated*) = (*Neighbors*/{*parent*}) **then**
(3d) **if** *i* = *myroot* **then**
(3e) **terminate**.
(3f) **else send** ACCEPT(*myroot*) to *parent*.
// if *newroot* < *myroot* then ignore the message. *newroot* > *myroot*
// will never occur.

(4) When REJECT(*newroot*) arrives from *j*:
(4a) **if** *newroot* = *myroot* **then**
(4b) *Unrelated* ⟵ *Unrelated* ∪ {*j*};
(4c) **if** (*Children* ∪ *Unrelated*) = (*Neighbors*/{*parent*}) **then**
(4d) **if** *i* = *myroot* **then**
(4e) **terminate**.
(4f) **else send** ACCEPT(*myroot*) to *parent*.
// if *newroot* < *myroot* then ignore the message. *newroot* > *myroot*
// will never occur.

Spanning-Tree Algorithm: Asynchronous Concurrent-Initiator Using Depth-First-Search

```
(local variables)
int parent, myroot ⟵ ⊥
set of int Children ⟵ ∅
set of int Neighbors, Unknown ⟵ set of neighbors
(message types)
QUERY, ACCEPT, REJECT

(1)   When the node wants to initiate the algorithm as a root:
(1a)  if (parent = ⊥) then
(1b)        send QUERY(i) to i (itself).

(2)   When QUERY(newroot) arrives from j:
(2a)  if myroot < newroot then
(2b)        parent ⟵ j; myroot ⟵ newroot; Unknown ⟵ set of
              neighbors;
(2c)        Unknown ← Unknown/{j};
(2d)        if Unknown ≠ ∅ then
(2e)              delete some x from Unknown;
(2f)              send QUERY(myroot) to x;
(2g)        else send ACCEPT(myroot) to j;
(2h)  else if myroot = newroot then
(2i)        send REJECT to j.    // if newroot < myroot ignore the query.
      // j will update its root to a higher root identifier when it receives its
      // QUERY.
```

(3) When ACCEPT(*newroot*) or REJECT(*newroot*) arrives from *j*:
(3a) **if** *newroot* = *myroot* **then**
(3b) **if** ACCEPT message arrived **then**
(3c) *Children* ⟵ *Children* ∪ {*j*};
(3d) **if** *Unknown* = ∅ **then**
(3e) **if** *parent* ≠ *i* **then**
(3f) **send** ACCEPT(*myroot*) to *parent*;
(3g) **else** set *i* as the root; **terminate**.
(3h) **else**
(3i) **delete** some *x* from *Unknown*;
(3j) **send** QUERY(*myroot*) to *x*.
// if *newroot* < *myroot* ignore the query. Since sending QUERY to *j*, *i*
// has updated its *myroot*.
// *j* will update its *myroot* to a higher root identifier when it receives a
// QUERY initiated by it.
// *newroot* > *myroot* will never occur.

Broadcast and Convergecast

- Broadcast
  - The root node sends messages to its children
  - Each intermediate node forwards the message received from its parent to its children

- Convergecast
  - Each leaf node sends its message to its parent
  - On receiving messages from all of its children, intermediate nodes send the collective messages to their parents