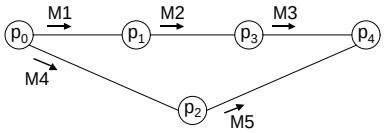
## Algorithms in Message Passing Model

Arvind Krishnamurthy
Fall 2003

## Recap

- Processors communicate over channels
- Asynchronous model:
  - Messages have arbitrary delay (but are reliable)
  - Processors have variable speed of execution
- Two notions of complexity:
  - Message complexity: number of messages in the worst case
  - Time complexity: number of steps in a "timed execution"
    - Timed execution is where each step is associated with some point in time when it happens
    - Subject to the constraint that the worst-case delay in *handling* a message is 1
      - Handling includes message delivery cost + performing a compute step on the destination processor

## Sample Timed Execution




- One possible execution: (at time 0: M1, M4 are initiated)
  D(M1) C(P1) D(M2) C(P3) D(M3) C(P4) D(M4) C(P2) D(M5) C(P5)
- C(P2) should happen before t = 1
- Similarly, C(P4) should happen before time at which C(P2) happens + 1
- Easy to see that in "diam" time the message is propagated across the network

## Broadcast over a known rooted spanning tree

- Suppose each processor has local variables:
  - Parent: to indicate which of its channels lead to its parent
  - Children: list of channels corresponding to its children
- Each processor has a terminated state
- Root initially sends M to its children
- When a processor receives M from its parents:
  - It sends M to its children
  - Enters termination state
- Complexities in both synchronous and asynchronous models:
  - Time: depth of the spanning tree
  - Messages: n-1 since one message is sent over each edge in the tree

## Convergecast over a known rooted spanning tree

- Opposite of the previous broadcast, with same complexities
- Useful for computing some "combined" value of the values stored at different nodes in a tree
  - Such as maximum or sum
- Leaves send message with their info to their parents
- Non-leaf waits to get messages from its children, combines them all, and sends result to its parent
- Notice that the process is initiated by the leaves
- Typically, a broadcast is sent from a root to all nodes to wake them up, and then a convergecast operation takes place

## Finding a spanning tree when a root is known

- Having a spanning tree is very convenient.
- How do you get a spanning tree?
- First, suppose a distinguished processor is known (who can serve as the root)
- Modify the flooding algorithm:
  - Root sends M to all its neighbors
  - When non-root first gets M:
    - It sends its identity back to the sender (and accepts sender as the parent)
    - It sends M to all its neighbors
  - When a processor gets M otherwise (second or later reception)
    - It sends reject to sender
  - Processors wait for replies from all their neighbors before they terminate. At which point, they know their children

## Execution of spanning tree algorithm

- In the synchronous case:
  - It produces a "breadth-first search" (BFS) tree
- In the asynchronous case:
  - Tree could be an arbitrary tree
- In both cases:
  - Message complexity: O(m)
  - Time complexity: O(diam)

## Finding a DFS Spanning Tree

- Analogous to sequential DFS
- When root first takes a step or non-root first receives M:
  - Mark sender as its parent and send an accept message to sender
  - For each neighbor:
    - Send M to it
    - Wait to get "accept" or "reject" in reply
- If processor receives M in any other case (subsequent receives):
  - Send "reject" to sender
- Processors identify children based on accept/reject messages
- Message complexity: O(m)
- Time complexity: O(m)

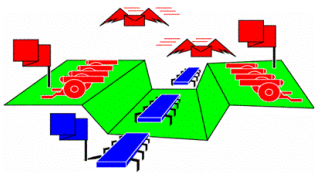
## No distinguished node

- Suppose that there is no specified root
- How to find a spanning tree (and its root)?
  - Need to find a root first
  - Will revisit this issue…

## Realization of message passing models

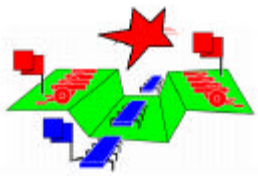
- In practical systems, there are three popular realizations of the message passing model:
  - User Datagram Protocol (UDP)
  - Transmission Control Protocol (TCP)
  - Remote procedure calls (RPC)
- Each has different semantics:
  - UDP: unreliable, can reorder messages (weaker than AW model)
  - TCP: reliable, no reordering, flow-controlled (somewhat stronger than AW model)
  - RPC: depends on implementation
    - Most implementations are reliable (without reordering)
    - No flow control for short messages, but introduces flow control for large messages

## Fault tolerance: two general's problem



- Two armies: red and blue
- Red army on two mountains separated by a valley
- Red army wins if the two generals agree to attack at the same time
- Generals can send messengers across the valley

## Unreliable messages



- Messages might be lost
- Need to devise a protocol that is "fault-tolerant"
- Obvious approach is to use "acknowledgements"
- General 1: "let's attack at 12:30"
- General 2: "Ok got your message"
- General 1: "Did you get my message (which was to attack at 12:30)"
- General 2: "Oh yeah got it, but let me know if you get this one."

## Impossibility result

- No such protocol exists!
- Consider a protocol that sends fewest messages
- It should still "work" if the last message is lost
- So just don't send the last message:
  - Which implies you have a protocol with fewer messages
  - The shorter protocol contradicts our original assumption!
- Distributed computing contains many such impossibility results
- Highly dependent on the properties of the underlying model

## Announcements

- Design document for assignment 1 due next Wednesday
  - Basically, spell out your protocol
  - What language, package you are going to be using?
  - Justify your design

## Leader Election

- Informally, given a set of nodes:
  - Each node eventually decides whether it is the leader or not
  - Exactly one node decides that it is the leader
- Has many uses in a distributed system:
  - Leader could manage/control the system
  - Could run a sequence of leader election rounds to circulate leadership (useful for tasks that need to have mutual exclusion)
- Formally:
  - Each processor has a set of elected states and a set of non-elected states
    - Once an elected state is entered, it cannot exit that state
    - Similarly for non-elected states
  - For every admissible execution:
    - Every processor eventually enters either an elected or a non-elected state (liveness property)
    - Exactly one processor enters an elected state (safety property)

## Rings

- We will study leader election in ring topologies (where nodes are arranged in a circular ring)
- Why study rings?
  - Simple starting point, easy to analyze
  - Abstraction of a token ring network
    - Regenerate lost token in token ring networks
  - Lower bounds for ring topology also apply to arbitrary topologies

## Many variations

- Rings come in different forms:
  - Ring can be unidirectional or bidirectional
  - Processors can be identical or can be somehow distinguished
    - Processors could have some unique processor-ids
    - Processor-ids are chosen from some large space of ids
    - Ids can be manipulated only by certain operations
  - The number of processors (n) may be known or unknown
    - If "n" is not known: considered as "uniform" algorithm
    - Otherwise, it is a "non-uniform" algorithm and can take advantage of the knowledge regarding number of processors
    - Formally: there are different state machines for different ring sizes
  - Communication may be synchronous or asynchronous

## Leader election in anonymous rings

- Theorem: there is no leader election algorithm for anonymous rings, even if the algorithm knows the ring size and the ring is synchronous
- Proof sketch:
  - Each processor begins in the same state with the same messages in transit
  - Every processor receives the same messages and thus makes the same transition and sends the same messages in round 1
  - Every processor receives the same messages and thus makes the same transition and sends the same messages in round 2
  - And so on.
  - Eventually, some processor is supposed to enter an elected state. But then they all would, a contradiction.

## Non-anonymous rings

- Assume that each node has a processor id
- How could you take advantage of unique processor-ids?