Distributed Systems

Arvind Krishnamurthy
Fall 2003

Concurrent Systems

- Collection of individual computing devices/processes that can communicate with each other
  - General definition encompasses a wide range of computer systems
    - VLSI chip with multiple functional units
    - Tightly coupled shared memory multiprocessor
    - Processes in an operating system
    - Local area cluster of workstations
    - Internet
  - Distributed systems is the study of systems that are loosely coupled
    - Each processor has its own semi-independent agenda
    - Coordinate their actions for various reasons:
      Sharing of resources, availability, fault-tolerance, etc.

Challenges of Distributed Computing

- Three major issues:
  - Asynchrony
    - Different processor speeds
    - Unpredictable message delivery costs
  - Limited local knowledge
  - Failures
- Secondary issues:
  - Heterogeneous hardware
  - Lack of adherence to standards
  - Selfish entities (need for incentives)
  - Byzantine entities (rogue elements)

This Course

- Both theory and practice of distributed systems
- Reasoning about distributed systems is hard
  - Need to identify and abstract fundamental problems
  - State them precisely and design algorithms
  - Prove correctness and analyze complexity
  - Prove impossibility results and lower bounds
- Building distributed systems is hard
  - Communication protocols exhibit race conditions
  - Have to deal with issues of scale
  - Performance bottlenecks are hard to identify
  - No substitute for hands-on experience in building distributed systems

Course Overview

- Introduce two basic communication models
  - Message passing
  - Shared memory
- And two basic timing models:
  - Synchronous
  - Asynchronous

	MP	SM
synch	yes	no
asynch	yes	yes

Overview

- Theory part of the course:
  - Basic graph algorithms
  - Leader election
  - Time in distributed systems (causality and snapshots)
  - Fault-tolerant consensus
  - Advanced graph algorithms (finding minimum spanning tree, “part-time consensus”)
  - Shared memory model: mutual exclusion
  - “Simulation” of one model on top of another model
- Material from Attiya and Welsh
- In some cases, we will look at the original papers covering the same material

Overview (contd.)

- We will also study some real systems:
  - Peer-to-peer file sharing systems
  - Distributed hash tables
  - Overlay networks
  - Routing algorithms (including routing for mobile networks)
  - Study of the Internet
- Material mostly from technical papers
- Programming assignments deal with building a simple peer-to-peer file-sharing system
- Course project will let you explore any topic of your choice

Course Work

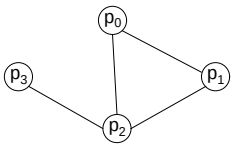
- Two programming assignments (worth 20-25% of the points)
  - Groups of two
  - Design followed by actual coding
- Few non-programming homeworks (worth 15-20%)
  - Individual work
- Course project (for the second half of the semester)
  - Worth about 40-50% of the grade
  - Groups of two
- Two short exams (worth 15-20% of the grade)

Course Logistics

- Course homepage:
  http://lambda.cs.yale.edu/cs425
- Getting in touch with me:
  - Email: arvind@cs.yale.edu
  - No specific office hours. Drop by any afternoon.
- Both assignments and the course project will include design review meetings.
- Monitor the course webpage for announcements. Refinements/changes to the assignments will also be posted on it.

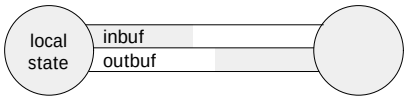
Message-Passing Model

- Processors are p_0, p_1, … , p_{n-1}
  - Represented as nodes of a graph
  - Number of nodes: n
- Bidirectional point-to-point channels
  - Represented as undirected edges of a graph
  - Total number of such edges: m



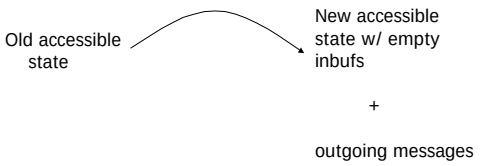
Processors, channels and configurations

- Processor is a state machine
  - Maintains local state
  - Next state determined by current state and incoming messages
  - Message in sent to outbuf when initiated; eventually reaches inbuf of target processor
  - Configuration: vector of processor states (current snapshot)
  - "Accessible" state includes local state and what is in the inbuf



Events

- There are two kinds of events
  - Deliver event: moves a message from sender's outbuf to receiver's inbuf; message will be processed the next time the receiver does a computation event
  - Computation event: a step by a processor



Asynchronous Execution

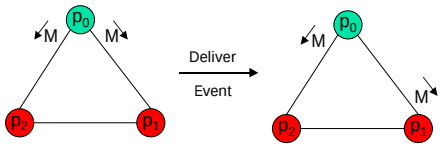
- Execution:
  Series of events
  config, event, config, event, ...
- In first (or initial configuration):
  - Each processor is in initial state specified by the algorithm and all inbufs are empty
- For each (config, event, config) triple:
  - If delivery event: specified message is transferred from sender's outbuf to receiver's inbuf
  - If computation event: specified processor's state (including outbufs) change according to its transition function

"Admissibility"

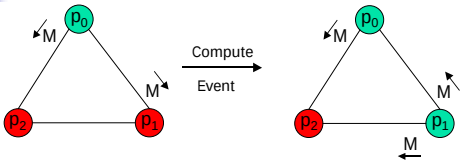
- Admissible executions are the ones that "do the right thing."
- Correctness conditions are usually classified into "safety" and "liveness" conditions
  - Safety conditions: things that must be true in every finite prefix of an execution (or in other words: "nothing bad ever happens")
  - Liveness conditions: things that must happen a certain number of times, perhaps infinitely often (or in other words: "eventually something good happens")
- For the asynchronous model, admissible means:
  - Every message in an outbuf is eventually delivered
  - Every processor takes an infinite number of steps
- There is no constraint on "when" these events must take place
  - Arbitrary message delays and relative processor speeds are allowed

Example: flooding

- A simple flooding algorithm as a collection of state machines. A message "M" needs to be propagated.
- A processor p has:
  - Local state variable: "color" that holds green or red
  - Initial states: for p_0, color is green and all outbufs contain M; for others, color is red and outbufs are empty
  - Transitions: if M is in an outbuf and color is red, then change color to green and send M on all outbufs



Flooding (contd.)



- To model algorithm termination, identify terminated states of processors – states which, once entered, are never left
- An execution has terminated when all processors are terminated and no messages are in transit (in inbufs or outbufs)

Complexity Measures

- These are "worst-case" over all possible executions and all possible initial configurations
- Message complexity: maximum number of messages sent in any admissible execution
- Time complexity: maximum time until termination in any admissible execution
- But how is time measured?
- Produce a **timed execution** from an execution by assigning non-decreasing real times to the events such that *the time between the sending and receiving of any message is at most 1*
- Essentially normalizes so that the greatest message delay in any execution is one time unit, yet still allows arbitrary interleavings

Complexity measures of flooding algorithm

- Terminated state: when all the processors are in "green"
- Number of messages sent in every admissible execution is 2*m, where m is the number of edges
  - One message is sent over each edge in each direction
  - Could be optimized by remembering the node from which the message was received
- Maximum time until termination is "diam + 1" where diam is the diameter of the graph

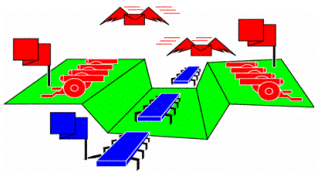
Synchronous Message Passing Systems

- In the synchronous model, processors operate in lock-step rounds
- To model this, define an execution to be a series of rounds
- A round looks like this:
  deliver deliver ... deliver comp comp ... comp
- First deliver all messages in outbufs
- Then each processor takes one computation step each
- An admissible execution is an execution that is infinite:
  - Every message sent is delivered
  - No processor fails during execution
- Time is measured in synchronous execution as the number of rounds until termination

Questions:

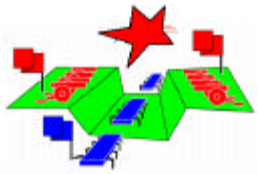
- Are the asynchronous and synchronous models reasonable?
- Are there aspects missing from these models that do not reflect reality?
- Are the semantics of the model completely specified?

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