Csc72010

Parallel and Distributed Computation and Advanced Operating Systems

Lecture 1

Course Description

What's different about a distributed system?

Bank example

If we think about programming transactions coming into a centralized bank computer, we can assume it would look like this:

A withdraws $100 from 1 (refused)
B withdraws $100 from 2 (accepted)
C deposits $1000 in 1 (accepted)
D withdraws $50 from 2 (refused)

Not true at an ATM – let's assume constant communication between ATM's and banks (not necessarily true):

A begins transaction on account 1	
	B begins transaction on account 2
	B requests withdrawal of $100
C begins transaction on account 1	
C requests deposit of $1000	
	D begins transaction on account 2
	D requests withdrawal of $50
Bank adds $1000 to 1	
A requests withdrawal of $100	
Bank checks that balance > $100	
Bank subtracts $100 from 1	
Bank dispenses $100 cash to A	
	Bank checks that balance > $100
	Bank checks that balance > $50
	Bank subtracts $100 from balance
	Bank dispenses $100 cash to B
	Bank subtracts $50 from balance
	D's ATM fails!!!

Distributed System Problems:

The bank example illustrates all of the problems

Concurrency: multiple people acting on the same object at the same time – order of activities must be controlled
Partial failure: The bank subtracted the total withdrawals requested from account 2, but didn't dispense all of the money
Time: A's request is either accepted or rejected depending on how fast his transaction goes relative to C's. This makes correctness harder to state.

Papers:
Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, July 1978, 21(7):558-565.
Network Time Protocol (Version 3) Specification, Implementation. D. Mills. March 1992.

Global state: Consider spreading the state around, so that the ATM's have the balances and don't have to go to a central site. This makes matters worse – we will learn later that in theory at least there is no guarantee that you can determine "The global state" – instead, there may be many possible global states consistent with a sequence of actions.

Michael J. Fischer, Nancy D. Griffeth, Nancy A. Lynch: Global States of a Distributed System. IEEE Transactions on Software Engineering, 8(3): 198-202 (1982).
K. Mani Chandy, Leslie Lamport: Distributed Snapshots: Determining Global States of Distributed Systems ACM Trans. Comput. Syst. 3(1): 63-75 (1985).

Waldo, Note on Distributed Computing

Solution is centralized – not what we'll be looking at.

Internet: no single node is in control (although we often end up selecting one).

Prototypical distributed problems

Network is a graph, communication links are edges in the graph, network devices are nodes, which we call processes.

Pick a leader (aka leader election): assume all processes identical – how can they select one to be the controlling process?

Broadcast communication: make sure everyone gets a message

*Routing: decide what routes messages should use in the network

Failure recovery/reliability:
*make sure messages reach their destination
one node fails, another takes over its function

Agreement (everybody does the same thing): commitment protocols

Resource allocation: make sure that a resource is given to at most one user, and a user requesting a resource gets one if it is available (fairness?)

Approach

Step 1. Observe how problems are solved by Internet protocols; consider the environment and the requirements, the design goals and the intuition behind the protocol.

Step 2. Model and prove properties of protocol. Make assumptions about environment; decide on algorithm requirements. (Sometimes) do some complexity analysis (number of messages, time).

Language is I/O automata: simple procedural commands, composition, tools for simulation and proof

Go over syllabus again to see what we'll be doing in the course:

- Link layer, IP layer before midterm
- Finish IP layer, TCP, project after midterm

Environmental Assumptions

How does communication take place?

- Message-passing

Timing?

- Asynchronous: any time

Failures

- Processors: stopping or Byzantine (we'll do stopping only)
- Communication: lost messages
  - Duplicate messages
  - Out of order messages
  - Channel failure
  - Network partitions

We'll usually start with simplifying assumptions, solve the problem, then alter the assumptions.

Typical Requirements

Functional correctness

Atomicity
Correct resource allocation
Message delivery

Reliability

Guaranteed message delivery
No duplicates
In-order messages
Server uptime

Availability

Uptime/downtime

Maintainability

Network management
Network configuration
Network monitoring

Performance

Response time
Throughput
Utilization
Congestion
Usual approach: performance modeling, queuing theory

Modeling, Specifications, and Software Engineering

Building software starts with the requirements: what is it that the software has to do? For a network stack, this might be "try to get a message from the sender to the receiver" (IP) or "guarantee delivery of messages from the sender to the receiver, with no reordering or duplication of message" (TCP).

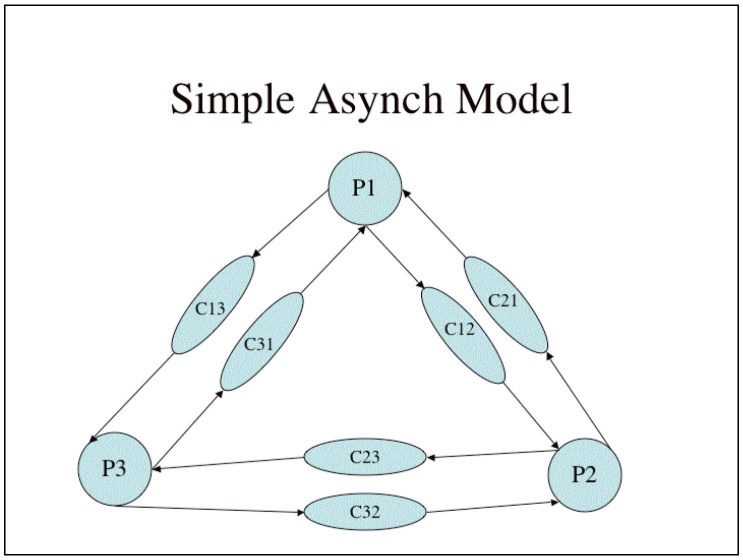
It can be useful to build models for distributed algorithms, supporting simulations and proofs to verify that they meet the stated requirements. Every possible execution in the entire set of executions of the model should be acceptable, in terms of the requirements.

We are going to use a technique of modeling that lets us describe the action of each process in the system and investigate the behavior of any combination of the processes, using either proofs or simulations.

The technique we will use is "Timed I/O Automata Models"

The I/O Automaton Model
I/O automata model
This is a general mathematical model for reactive components. It imposes very little structure – we add structure for various kinds of systems.

For this course, we will model networks as processes communicating via channels



The Pi's and Ci's are reactive components, i.e., modules that interact with their environments using input and output actions (send and receive).

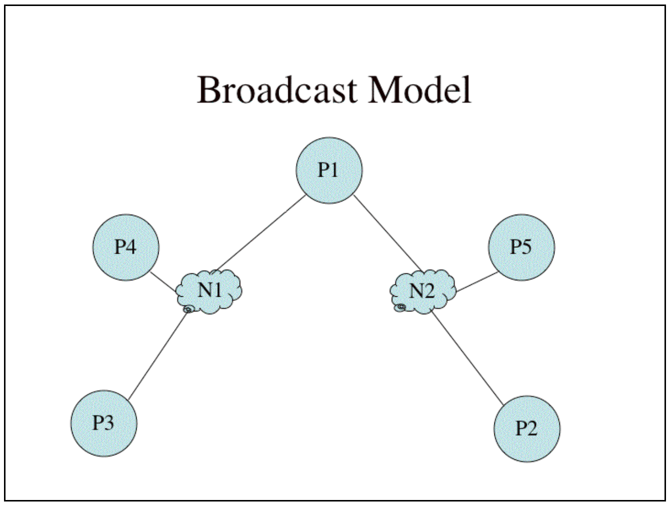
P1, P2, P3 are processes and have input actions receive and output actions send
Cij are channels and have input actions send and output actions receive

The process and channel actions correspond where there is an arrow. So for example P1 has outputs send(m,1,2) and send(m,1,3), while C12 has input send(m,1,2) and C13 has input send(m,1,3).

Also, P1 has inputs receive(m,2,1) and receive(m,3,1) and C21 has output receive(m,2,1) and C31 has output receive(m,3,1).

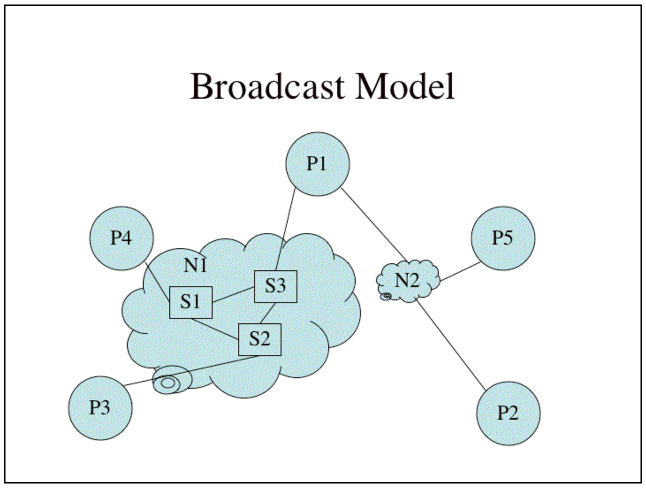
The above diagram is the network we will use for algorithms in the book. However, it is a point-to-point network, and Ethernets are shared (broadcast) networks, so they need to be modeled a little differently. See the diagram below.

A point-to-point network is implicitly specified by giving the endpoints; in the broadcast model, the network needs its own id. So, the actions would be send(n,i,j,m) and receive(n,i,j,m), where n is the network, i is the source, j is the destination, and m is the message. The action send(n,i,j,m) is an ouput of process i and an input of network n. The action receive(n,i,j,m) is an output of network n and an input of process j.



The I/O automaton model is designed to make it easy to organize the description of a system and to prove things about it:

1) We can compose components to form a larger system – we could have started just with P1, P3, and P4, and then added N2, P5, and P2.
2) We can describe systems at different levels of abstraction - e.g., N1 could be a collection of switches running learning bridge and spanning tree algorithms, but all we need to use is that this collection of switches lets P1, P3, and P4 send messages to each other.



3) We have good proof methods
    a. Invariants
    b. Composition and projection
    c. Simulation relations

Definitions

An I/O automaton A consists of

- sig(A), a *signature*, which specifies the input, the outut, and the internal actions of the automaton.
    - in(A) is the set of input actions
    - out(A) is the set of output actions

    - internal(A) is the set of internal actions
    - local(A) = out(A) ∪ internal(A) is the set of locally-controlled actions
    - acts(A) is the set of all actions
- a set of states states(A), which may be infinite
- a set of start states start(A)⊆states(A)
- a set of transitions trans(A)⊆states(A)×acts(A)×states(A)
- an equivalence relation tasks(A) on the local actions of A (i.e., internal and output actions).

There's one restriction on all of this: any input is enabled in any state, i.e., there is a transition involving that input.
For all s∈ states(A) and π∈ in(A), there is a transition <s, π, t> in trans(A).
This is because we don't want an automaton to be able to prevent the environment from doing something. This requires us to model behavior in "bad" environments, which do unexpected things. If we really want to restrict inputs, we can model the "good" environment as another automaton that only passes on the good inputs from the real environment.

In other words, inputs are controlled by the environment and can happen at any time. Input and output are external and can be seen by the environment. Output and internal are locally controlled, i.e., happen under control of the automaton.

States can be infinite, to let us model queues that grow without bound, files, and so on. This may not be realistic sometimes, but usually simplifies the model.

Tasks are groups of locally-controlled actions that should get an opportunity to happen. They are use to model "fair" executions. For example, we may want to say that we're only interested in networks where every process gets to send a message infinitely often (that is, it's not blocked forever from sending). Then for each network that a process is connected to, it would have a task containing all possible send(n,i,j,m), i.e., n and i are fixed but j and m can be any values.

Digression to explain tasks: There are two kinds of properties, safety and liveness. Safety properties say that bad things don't happen; liveness properties say that good things will eventually happen. Usually, we have to know that processes continue to interact with each other in order to guarantee that good things happen. Thus we don't try to prove the "good things" for sequences of events in which one process stops doing anything.

Examples

Channel automaton

This is a reliable FIFO channel, unidirectional between 2 processes.
Fix a message alphabet M.
The signature is:
inputs: { send(m): m∈ M }

output: { receive(m): m∈M }
states: a FIFO queue of messages, call it queue, initially empty.

transitions: these are described by code fragments

In the executable language:

```
automaton Channel(i, j: Int)

  signature
    input send(const i, const j, m: Int)
    output receive(const i, const j, m: Int)

  states
    queue: Seq[Int] := {}

  transitions

    input send(i, j, m)
      eff queue := queue |- m

    output receive(i, j, m)
      pre m = head(queue)
      eff queue := tail(queue)
```

Process automaton

Here is a trivial process:

```
automaton Process(n: Int)
  signature
    input receive(const n-1, const n, x:Int)
    output send(const n, const n+1, x:Int)
  states
    toSend:Seq[Int]:= {}|-n
  transitions
    input receive(i, j, x)
      eff toSend := toSend |- x

    output send(i, j, x)
      pre x = head(toSend)
      eff toSend := tail(toSend)
```

Proving properties

Does an algorithm satisfy the properties representing basic requirements?

Executions

A *state assignment* is an assignment of a state to each process.

An *execution* is a sequence of states and actions:

$s_0, \alpha_1, s_1, \alpha_2, s_2, \alpha_3, s_3, \ldots,$

where each s_i is a state and each α_i is an action.

Proof techniques

α and α' are indistinguishable to automaton i if i has the same sequence of states, the same sequence of outgoing messages, and the same sequence of incoming messages in α and α'
Useful in impossibility proofs.

Invariant assertions: some property holds in every execution. We can often establish this by induction.
Simulations: one algorithm implements another by showing the same input/output behavior. More complicated to use than invariant assertions.

Consider showing that eventually every process has received all messages.
Can we show this in all topologies (no).
Can you identify any topologies for which it is true? Must have a path from every node to every other node
How would you prove it for connected networks?

Example Claim: after diam(G) message sends, all processes have received the message.

Invariant assertion: after "enough" steps, all processes "close enough" to the original sender got the message.