Csc72010

Parallel and Distributed Computation and Advanced Operating Systems

Lecture 1

January 27, 2005

Business

Introduce myself & research interests
Security seminar
Class roll

Go over syllabus & calendar
- Every Thursday except March 24 and April 29, until May 12
- Midterm exam on March 17;
- Project plan on April 7;
- Project due on May 19

The course is preparation for one of the sections of the first exam (qualifying exam?)

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: 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. Make assumptions about environment; decide on algorithm requirements; define and model one or more algorithms to solve problem. Do some complexity analysis (number of messages, time).

Language is I/O automata – why?

Step 2. Observe how similar or same problems are solved in Internet; consider if and why the Internet solution is different.

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

Synchronous until midterm (not realistic, much easier to analyze, often a good approximation)

Asynchronous (realistic, much harder to model and analyze)

Environmental Assumptions

How does communication take place?
- Message-passing

Timing?
- Synchronous: one message per process per round
- 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.

Requirements

First two apply to the theoretical algorithms we will study

Functional correctness

Atomicity
Resource allocation
Message delivery

Reliability

Guaranteed message delivery
No duplicates
Server uptime

The rest may apply to the Internet algorithms/protocols

Availability

Uptime/downtime

Maintainability

Network management
Network configuration
Network monitoring

Performance

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

Synchronous Model – chapter 2

Reading: Chapters 1, 2, 3.1-3.3

Assumptions

Rounds
In a round, a process does each of the following tasks:

Sends messages to its neighbors
Receives messages from its neighbors
Takes a transition (changes state)

We allow concurrent activity (all processes active in all rounds)
We allow failures

We use a directed graph to describe the network – processes are nodes of the graph, they communicate with their neighbors in the graph

I/O Automaton Definition

Notation G=(V, E)

    n = |V| is the size of the digraph (the number of nodes or processes)

For each i, there is a process (node):

    out-nbrs_i
    in-nbrs_i
    distance(i,j)
    $\text{diam}(G) = \max_{i,j}\{\ \text{distance}(i,j)\ \}$

The "program" running at process i is defined by:

    states_i
    start_i
    outputs:
        $\text{msgs}_i : \text{states}_i \times \text{out-nbrs}_i \rightarrow M \cup \{\text{null}\}$
    state change:
        $\text{trans}_i : \text{states}_i \times (\text{vectors over } M \cup \{\text{null}\} \text{ indexed by in-nbrs}_i) \rightarrow \text{states}_i$

There's a message alphabet M, we assume $\text{null} \notin M$ (doesn't have to be finite)

In each round, the processes:

    Apply the output function to generate messages to neighbors
    Collect incoming messages from neighbors
    Apply transition function
    Repeat

There are no restrictions on the computation – so if you want to say that a process computes a NP-hard problem in one round, that's ok
No halting states – not used

Inputs and outputs end up being encoded as variables in the states

Example

Let's look at a very simple network, just for example.
At the first round, each process will send its input and process id to the next process.
At each subsequent round, each process will send any messages received to its out-nbrs.
(it just makes a vector of messages received)

Lets start by defining process states. Each process needs an ID and an input:
states_i
u, which is i's unique ID
input, which is i's input

received, which is the set of messages received in the previous round, initially $\varnothing$

start$_i$ is defined by the "initially" clause

Let's consider what the set M contains:
All sets of pairs (u,i) where u is a unique ID and i is an input. Let U be the set of unique ID's and IN be the set of inputs, then $M = \wp(U \times IN)$

msgs$_i$
send received $\cup$ (u,input) to all $j \in$ out-nbrs$_i$

trans$_i$
received := set of messages received from in-nbrs$_i$

Trace a run with a fully-connected graph with 3 nodes; all nodes end up sending all (id,input) pairs in each round. For transferring information, each node could just not forward any message that contains its (id,input).

Proving properties

Does an algorithm satisfy requirements?

Executions

A *state assignment* is an assignment of a state to each process.
A *message assignment* is an assignment of a message to each channel.
An *execution* is a sequence of state assignments and message assignments:

$C_0, M_1, N_1, C_1, M_2, N_2, \ldots,$

Where C_r is a state assignment and M_r and N_r are message assignments.

M_r is messages sent; N_r is messages received.

Proof techniques

α and α' are indistinguishable to process 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 (uid, input) pairs.

Is this true? Not in all topologies
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 these networks?
Claim: after diam(G) rounds, it's true.
Invariant assertion: at the end of round r, $received_i$ contains (uid, input) for every process j such that distance(i,j) <= r
Set up the induction

Complexity

Time complexity: number of rounds

    In the preceding example: diam

Communication complexity: number of messages

    Diam*edges until the all nodes have seen all inputs