Distributed Software Development

Problem Solving I

Chris Brooks

Department of Computer Science
University of San Francisco

Distributed Problem Solving

- The preliminary portion of the course focused on techniques for achieving properties or states in distributed systems.
  - Causal delivery, mutual exclusion, etc.
- Now, we turn to the question of how to solve problems in a distributed fashion, assuming that we have implemented some of these properties.

Problem environments

- One dimension along which we can characterize distributed problem solving is according to the degree of autonomy or self-interestedness of the participants.
- How much can a protocol assume about the behavior and motives of the participants?

Centrally controlled environments

- At one extreme, all processes in a system are controlled by a single individual or organization.
  - Beowulf cluster
  - Parallel computer
  - Intranet
- This allows us to make fairly restrictive assumptions about the behavior of system processes.
  - NPS, parallel computation (e.g. conjugate gradient)

Cooperative processes

- We'll also think about processes that are controlled by separate individuals, but assumed to be cooperative.
  - SETI@Home, distributed.net
  - Meeting scheduling
  - TCP (originally)
- In this case, we can assume that processes will act benevolently, but that they will be heterogeneous.

Non-cooperative processes

- We'll also need to think about non-cooperative systems, in which each process is self-interested.
  - Not necessarily malevolent, just concerned only about its own performance.
- This will require a different set of assumptions about how our protocol should work.
  - Resource allocation, auctions, some scheduling problems, file-sharing

TCP: an illustration

- TCP is an example of a protocol that was designed to work in a cooperative environment.
- Recall that TCP is built on top of UDP
  - UDP provides packet-oriented delivery.
- TCP provides reliable in-order delivery on top of UDP.
- Sender A sends a packet to receiver B.
- B returns an acknowledgment that the packet was received.
- If A does not receive an ACK before a timer expires, the packet is resent.

TCP: an illustration

- To improve transmission efficiency, TCP uses a concept called sliding windows.
  - The sender has a "window" of size n. It sends all packets within that window.
  - As the lowest-numbered packet in the window is acknowleged, the window "slides" upward, and more packets are sent.
- This improves transmission rates – the goal is for the network to be completely saturated.

TCP: an illustration

- The problem is how to deal with congestion.
  - Packets may be dropped by the receiver, or by intermediate hosts.
  - When should the sender resend?
  - Too slow → inefficiency
  - Too quickly → oversaturation is worsened.
- TCP uses an adaptive retransmission policy.
  - As connection performance changes, so does timeout duration.

TCP: an illustration

- The TCP congestion algorithm does the following (loosely):
  - When a packet is lost, halve the window size and double timeout.
  - If all packets in a window are transmitted successfully, increase window size by 1.
- There are lots of details in the implementation of this that I'm glossing over.
- The key point is this: This protocol works wonderfully, *as long as everyone else also uses it.*
  - Designed to minimize congestion over the entire Internet.

TCP: an illustration

- In the early days if the Internet, this was not a problem.
  - Small number of users, fewer bandwidth-saturating apps.
- Parallel download of images from web pages was the first concern.
- Later, non-TCP protocols (RTSP, proprietary schemes) implemented their own congestion control algorithms.
- These applications are not necessarily tuned to any sort of global optimum.

Tragedy of the Commons

- This is an example of a problem known as *tragedy of the commons.*
  - Cost of using a resource is not borne equally by the beneficiaries of that resource.
- Leads to overuse.
- Shared resources, such as networks, tend to be vulnerable to this problem.
- Game theory provides some ideas for dealing with this dilemma.

Distributing a Problem

- We'll also need to think about how well a problem can be partitioned.
- Typically, a problem is distributed by dividing it into subproblems.
- Each node or process works on its own subproblem.
- Processes may need to communicate with each other.
- A center or coordinator is responsible for doling out subproblems and collecting results.

Problem Coupling

- We can characterize distributed problems by the degree of interaction that is required between nodes.
- Tightly coupled: nodes must communicate frequently in order to solve subproblems.
- Loosely coupled: Subproblems are relatively independent of each other.
- "Medium coupled": Some interaction must take place.

Tightly Coupled Problems

- Tightly coupled problems require each node to communicate with other nodes very frequently in order to solve its subproblem.
- Fast, low-latency communication is essential.
- These are the sorts of problems you studied in Prof. Pacheco's Parallel and Distributed Computing class.
  - Inverting a matrix.
  - Solving a system of linear equations
  - Fourier transform

Tightly Coupled Problems

- Tightly coupled problems typically have a great deal of data dependency between subproblems.
  - Nodes must frequently share partial results in order to proceed.
- This means that tightly coupled problems are best solved in a parallel computer or a LAN.
- All nodes should have roughly the same computing power.
  - A slow process can act as a bottleneck.

Loosely coupled problems

- At the other end of the spectrum are loosely coupled problems.
- The center can divide up a problem and allow processes to work independently on subproblems.
- Nice for settings in which communication is slow, or nodes may run at different speeds
- Examples:
  - distributed.net
  - SETI@Home

distributed.net

- A distributed project set up to test the security of symmetric-key encryption algorithms.
- A problem is chosen to solve
- Each node is assigned a subset of the keyspace.
- Node try their subset of the keys and return results to a central server.
- No interaction with other nodes is required.