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.
  - NFS, 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 heterogenous.

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.

mmetric-key encryption: a brief digressio

- Symmetric key encryption (or secret-key encryption) uses one key to encrypt and decrypt a message.
  - As opposed to public-key encryption, which uses pairs of keys.
- A series of bit shifts and ANDs with a key are used to conceal a message.
- Secret-key encryption is "more secure" than public key encryption in the sense that a shorter key is needed to provide the same level of security.

Symmetric-key encryption: a brief digressio

- Two well-known algorithms: DES, RC5.
  - DES was developed by the government in the 50s
  - RC5 was developed at RSA labs in the 90s.
- The only *known* way to defeat them is through exhaustive search of all keys.
- DES keyspace is $2^{keysize}$
- 56-bit secret-key algorithm has a keyspace of $2^{56} = 72$ quadrillion keys.

distributed.net

- History:
  - 1997: RC5-56 is cracked: 212 days, 34 quadrillion keys searched. (47% of keyspace)
  - 2002: RC5-64 is cracked: 1757 days, over 1.16×10^{19} keys (63%) of keyspace searched. (270 GKeys/sec at completion)
  - RC5-72 is ongoing. (how long will this take at current speeds?)
- Other problems:
  - DES
  - Factoring
  - Golomb rulers

How does it work?

- The keyspace is broken into set of blocks.
- A master keyserver tracks all blocks:
  - Which are unprocessed
  - Which are currently being processed
  - Which are done.
- It communicates with a set of *proxy keyservers*

How does it work?

- Proxies serve as a layer between clients and servers.
- Proxies request a block of keys, which are then handed out to clients on demand.
  - Avoids server bottleneck.
  - Round-robin DNS provides fault-tolerance; if one proxy fails, client uses the next available.
- When a client is done processing a block, it returns it to the server.
- Blocks that are unreturned after 90 days are reassigned.

irid computing vs. Public resource cor

- distributed.net is an example of what's sometimes referred to as *public resource computing*
  - Unused computing resources are added into an application dynamically
  - Focus is on easily integrating large numbers of clients into an application.
- *Grid computing* is a similar concept
  - Applications can discover and use resources dynamically as they become available.
  - Difference is one of degree: grid computing tends to focus more on interactions between national-lab-level computing facilities.
  - Focus is on access control and rights, management of identity, automated service description, etc.

SETI@Home

- SETI stands for Search for Extraterrestrial Intelligence
- Radio telescopes listen for transmissions from outer space
  - SETI@Home uses signals captured by a telescope in Puerto Rico
  - Either intended or unintended transmissions
- Radio telescopes produce a vast amount of continuously-occurring data.
  - Approximately 35GB/day
- Standard SETI programs can only examine the data superficially
- By dividing the data into small pieces, it can be distributed to clients worldwide for processing.

SETI@Home

- Data is captured by the telescope onto 35 GB magnetic tapes, then mailed to Berkeley.
- It is then broken into 0.25 MB chunks.
- Each chunk represents about 107 seconds of data in a 10kHz range of the electromagnetic spectrum.
- As with distributed.net, each chunk can be processed completely independently of the others.

SETI@Home

- FFTs are used to extract signals at specific frequencies
- Doppler effects are removed. (this is the computationally intensive part)
- Looks for signals with a Gaussian shape (weaker, then strong, then weak again)
  - Since the telescope is fixed and the Earth rotates, a signal will 'move across it' in about 12 seconds.
  - Earth-based transmissions will have a constant amplitude.
- Also looks for pulsed signals.

The SETI@Home architecture

- Once data arrives at Berkeley, a *splitter* program preprocesses it and divides it into workunits (or chunks).
- These are then stored in a database.
- Clients interact with a data server that distributes workunits.
- The client may then disconnect and work on the data for as long as necessary.
- Results are then returned from the client to the server.

The SETI@Home architecture

- Data is distributed redundantly (the same block is sent to several clients).
  - This provides fault tolerance.
- Results are returned to the server, where they are written to a file, then processed and entered into a database.
- Once a workunit has enough results, it is considered complete and the results are aggregated.

The distributed search problem

- SETI@Home and distributed.net are both examples of *distributed search*
  - Exhaustively examine a huge search space.
- This sort of problem has many characteristics that make it appealing for large-scale distributed computing
  - All compute nodes are independent of each other.
    - No bottlenecks at client
    - No need for client-client communication
  - Failure of a compute node is easily tolerated.
  - Redundant computation of results is not a problem.
  - Clients can be stopped and restarted without problem.

Folding@Home

- Folding@Home is a similar project that aims to simulate the protein folding process.
  - Goal: understand how individual proteins bind together into larger structures.
- Simulating a process that may take microseconds can take months
- Folding@Home allows many clients to work on the problem in parallel.
- Archutecturally very similar to SETI@Home.

Folding@Home

- Unlike SETI@Home, a single simulation of a protein fold can't be easily subdivided.
  - Simulating a series of dependent events
- Folding@Home takes advantage of the statistical nature of protein folding
  - Results are not deterministic
  - Processes change from state to state periodically
- Insight: run many simulations simultaneously. When one changes state, change all the others.
- Only infrequent communication with clients is needed.

BOINC

- BOINC (Berkeley Open Infrastructure for Network Computing) is a generalization of the SETI@Home project
  - Rosetta@Home - protein structure
  - Einstein@Home - search for pulsars
  - Climateprediction.net - Eco-simulations
  - Predictor@Home - studies protein-related diseases.
- All of these problems can be expressed in the framework used by SETI@Home.
  - Problem is decomposed into work units.
  - Scheduling servers hand out work units to cleints and handle reports of completed results.
  - Data servers handle upload of actual data.

BOINC

- Given this basic model for doing distributed computing, the problem becomes one of translating your problem into the BOINC framework.
  - Parallelize your algorithm if possible
    - Identify existing serial implementations for pieces of your problem
  - Identify appropriately-sized workunits
    - Can (should) multiple clients work on the same workunit?
    - Do clients need to generate data?

"Medium-coupled" problems

- The BOINC model works great for highly parallel problems
- A large class of problems exist between the extrema of matrix inversion and SETI@home/BOINC.
- Typically, the problem can be somewhat decomposed, but some communication or synchronization between computing nodes is needed.
  - Scheduling problems
  - Dynamic programming problems
  - Planning problems
- Next week, we'll look at a particular well-studied example: distributed constraint satisfaction.

Summary

- Distributed problem solving requires an awareness of:
  - Distribution of control
  - How a problem can be decomposed
- Tightly-coupled problems are best attacked in environments with synchronous, low-latency communication and homogenous processors.
- Loosely-coupled problems (distributed.net, SETI@home) are appropriate for heterogenous environments with asynchronous, sproadic communication.