



Distributed Software Development

Introduction

Chris Brooks

Department of Computer Science
University of San Francisco

Class structure

- Combination of lectures and labs
 - Labs (many Tuesdays): hands-on exposure to a piece of Web technology.
 - Lectures (Thursdays): problems, principles and algorithms related to large-scale distributed systems.

Class structure

- Work:
 - 8 labs - 1 week turnaround.
 - Midterm and final
 - Two projects:
 - P2P client
 - Extension of your choosing.

Course Policy

- Class participation is very important.
 - Attendance is required
 - Active participation is encouraged (and rewarded!)

Course Policy

- Texts:
 - Readings from a number of texts are available online through the USF library
 - I will also assign readings from O'Reilly's technical collection.
 - This is available for free to all USF students at proquest.safari.com
 - I will expect you to do the reading before class.

On Reading

- “There’s so much reading! Do we really have to read all of this?”
- “Can’t we just use the slides?”

On Reading

- “There’s so much reading! Do we really have to read all of this?”
 - **Yes.**
- “Can’t we just use the slides?”
 - **No.**

On Reading

- One thing I want you to learn in this class is how to synthesize large amounts of printed information.
- Wrong: Front-to-back, sequential.
- Right: Skim first. Identify important sections, or material you are weak on. Return to that.

Course Policy

- Languages:
- You may use any language that “makes sense” for the lab/project.
 - I will present examples in Python and/or Java.
 - If you want to use something esoteric (not Python, Java, C/C++, Ruby, Perl, or C#) please talk to me first.
 - I recommend choosing a language that has good support for XML parsing.

On Plagiarism

- You are all expected to do your own work. No exceptions.
- There may be cases in which it is acceptable to use a third-party library (with appropriate credit) to assist on an assignment. Please ask first.
- I will check your assignments against each other, databases of code repositories, and google.
- Plagiarism is simply not worth the risk.

Assignments

- You will have 8 labs, plus two larger programming assignments.
- Labs will provide you with some components that can potentially be used in larger projects.
 - Good SE practices are important here.
- All labs **must** be available 24/7 on a publically available server.
- Pace is fairly quick; it's very important to start early and keep up.

Course Policy

- Late assignments:
 - Labs may be turned in up to 24 hours late for a maximum of 75% credit
 - Projects may not be turned in late. You will be expected to demo your work the day it is due.
- In general, getting in the habit of turning things in late is a very bad idea.

Aphorisms

- “Success is 1% inspiration and 99% perspiration” - Thomas Edison
- “90% of life is showing up.” - Woody Allen
- “Teachers open the door. Students must walk through on their own.” Chinese proverb.
- “Just keep swimming.” Finding Nemo

What is a distributed system?

- (Coulouris) “A distributed system is one in which hardware or software components communicate or coordinate their actions only by passing messages.”
- This covers everything from a parallel computer to the Internet.

What is a distributed system?

- So how is this class different from Pacheco's or Benson's?
- Different set of challenges:
 - Heterogeneity
 - Openness
 - Scalability
 - Failure models
 - Degree of parallelism

Tightly-coupled systems

- Parallel computing typically works with the following sorts of systems:
 - Shared clock
 - Low latency/fast communication
 - Single owner
 - Homogenous systems
 - Tightly coupled
- Prof. Pacheco's class addresses systems with these characteristics.

Middle ground

- Prof. Benson's class relaxes these assumptions somewhat.
 - Systems may not be homogenous
 - Communication still fast/low latency
 - Limited autonomy
 - Less tightly coupled.

This class

- This class will look at systems on the other end of the spectrum
 - Widely distributed - high latency communication
 - Autonomous, heterogenous components
 - No shared clock
 - Very large scale
 - Discovery may be a problem
- The Web is a classic example of this sort of system (but not the only one)

Issues in distributed systems

- The definition Couloris provides leads to the following problems that must be considered:
 - Concurrency - work is happening on multiple computers simultaneously and must be coordinated.
 - No global clock.
 - Independent failures, both network and computer.
- We'll spend a lot of time discussing these problems.

An example: DNS

- To illustrate some of the issues involved in distributed computing, we'll look at a well-known distributed system: DNS.
- DNS stands for domain name system.
 - This is the system that maps *symbolic* hostnames (such as stargate.cs.usfca.edu) to IP addresses (such as 138.202.171.14).
- Symbolic names are much easier for humans to work with.
- Computers, on the other hand, do better with IP addresses.
- How can we look up the correct IP address for a hostname?

DNS from the client side

- From a web browser's perspective, resolving a hostname looks like this:
 1. The hostname is extracted from the URL.
 2. The browser sends a query to a *DNS server*.
 3. The server eventually returns a reply, which contains the corresponding IP address.
 4. The browser then opens a TCP connection to that IP address.
- From the client side, this looks pretty simple.

Other things DNS does

- Along with translating hostname/IP address pairs, DNS can do the following:
 - Host aliasing - for example, `nexus.cs.usfca.edu` is also `www.cs.usfca.edu`
 - `nexus.cs` is the *canonical hostname*
 - Mail server aliasing - the CS mail server is `nexus.cs.usfca.edu`, yet I can send email to `brooks@cs.usfca.edu` successfully.
 - Load distribution. We can map multiple IP addresses to a single hostname.
 - The DNS server will return all IP addresses, but permute the order.

Service-oriented computing

- This description of DNS is an example of a *service-oriented* description.
- We specify what parameters must be provided to a service, and what information is returned.
- Client does not need to know how this information is generated.
- Only interface is specified.
- Functionality can be engaged remotely.

A naive solution

- One way to build a DNS server would be the following:
- Get a fast Internet connection and set up a single high-speed computer that will do all DNS resolutions via a huge database.
- What are some problems with this approach?

A naive solution

- Single point of failure
- Not scalable
- Not “close to” all clients - unacceptable delay.
- Difficult to maintain.

How DNS actually works

- DNS is a distributed, hierarchical database.
- Large number of servers worldwide.
- No database contains all DNS entries.
- DNS servers are divided into three classes:
 - Root servers
 - top-level domain servers
 - authoritative servers.

DNS from the client side, redux

- Returning to our from-the-client-perspective:
 1. The client contacts a root server.
 2. This returns the addresses of TLD servers
 3. The root server contacts a TLD server (for example, for the .com domain)
 4. This returns the address of an authoritative server at cs.usfca.edu (for example).
 5. The client then queries this server to find out the IP address of nexus.cs.usfca.edu

Local DNS servers

- Often, the authoritative DNS server will not have the addresses for all hosts in an organization.
- Instead, local DNS servers will manage the names for subsections of the network.
- This is typically the program that acts as a proxy for the client.
- DNS servers also typically cache much of this information so as to avoid resending requests for common lookups.

DNS as a distributed system

- What issues does this particular design of DNS solve?

DNS as a distributed system

- What issues does this particular design of DNS solve?
 - Heterogeneity: It will work with computers using different operating systems, languages, hardware, network interfaces or platforms.

DNS as a distributed system

- What issues does this particular design of DNS solve?
 - Openness - Can new additions, changes, or improvements be made to the system?
 - Are the system and its interfaces publically described?
 - DNS (and most internet systems and protocols) are publically described in a set of RFCs.

DNS as a distributed system

- What issues does this particular design of DNS solve?
 - Openness - Can new additions, changes, or improvements be made to the system?
- Open systems can typically be constructed from heterogeneous components, as long as the vendors/implementors conform to a published standard.
- This makes it easier to extend a service or add new services on top of existing ones.
 - For example, TCP was added on top of IP.

DNS as a distributed system

- What issues does this particular design of DNS solve?
 - Redundancy - In DNS, every address is stored in at least two servers. If one server fails, others can be queried.

DNS as a distributed system

- What issues does this particular design of DNS solve?
 - Concurrency: Many requests can happen simultaneously.
 - No bottleneck waiting for queries to resolve.
 - The resource (lookup tables) is distributed across a large number of hosts.
 - The system is able to operate consistently in a concurrent environment.
 - This is easier said than done.

DNS as a distributed system

- What issues does this particular design of DNS solve?
 - Transparency - The separation of components is hidden from the user.
 - No need for a user to know about root or TLD servers.
 - As in OO design, transparency makes development easier for clients.

DNS as a distributed system

- What issues does this particular design of DNS solve?
 - Scalability - As we add machines, resources, or users to the system, how quickly does performance degrade?
 - How do costs change as more resources or machines are added?
 - Will resources ever run out?
 - Are there bottlenecks in the system?
- DNS provides a scalable way to do hostname resolution.

DNS as a distributed system

- What issues does this particular design of DNS solve?
 - Handling failure. This includes both detecting and recovering from failure.
 - This might also include proceeding in the face of failures that are suspected, but can't be detected.
- DNS is able to recover from or tolerate failure in the sense that, if one local DNS server fails, hosts on other parts of the Internet can still be resolved.

Summary

- DNS is a nice example of a system that was designed to be distributed from the beginning.
 - Heterogeneous, fault-tolerant, scalable, open, concurrent, transparent ...
- We'll spend quite a bit of time in the next few weeks talking about how to handle these challenges and ensure different properties of distributed systems.

Summary

- For example:
 - If our system does not have a global clock, how can we guarantee that operations are seen in the same order everywhere?
 - What sorts of failures can occur, and how can they be detected?
 - How do we deal with concurrency in a distributed system?
 - How can systems be made to scale effectively?