CS 677: Big Data

Networking and Serialization

Lecture 5
Today’s Schedule

- General Announcements
- Thought Experiment: Computation Framework
- Project 1 Hints
- The Network
- Serialization
Today’s Schedule

- General Announcements
- Thought Experiment: Computation Framework
- Project 1 Hints
- The Network
- Serialization
Announcements

- HDFS install doc has been updated with a new starter configuration
  - Has additional hints for choosing your hostnames
  - Also: environment variables
- emacs is now installed on the cluster for your editing enjoyment
- Review 1 is due tomorrow at 11:59pm!
Paper Discussions

- Our paper discussion days will be split into three phases:
  - Individual questionnaire
    - This will include topics we’ve covered in lecture as well
  - Group discussion
    - (including discussing the questionnaire)
- Afterwards, we’ll discuss as a class
Today’s Schedule

- General Announcements
- Thought Experiment: Computation Framework
- Project 1 Hints
- The Network
- Serialization
Let’s say we are going to make a simple system that executes programs on multiple machines in parallel.

The user gives us a list of *tasks*, or jobs to run on the systems:

- `cd /home/me/somewhere; ./my_prog –n 1 > output.txt`
- `cd /home/me/somewhere; ./my_prog –n 2 > output.txt`
- ...

What components/messages do we need?
Components

- Coordinator
  - Holds the list of tasks to run
  - Assigns tasks to machines

- Task Server
  - Retrieves tasks from the coordinator
  - Executes tasks
  - Tells the coordinator when finished

- Client
  - Provides tasks to the controller
Messages/Events

- Task specification
  - Coordinator → Task Server
- Task request
  - Task Server → Coordinator
- Task completion notification
  - Task Server → Coordinator
- New task import request
  - Client → Coordinator
- New task import response
  - Coordinator → Client
Things to Think About

- What about fault tolerance?
- Should the Task Servers be continuously connected to the Controller?
  - Why or why not?
- Could we add administrative messages that tell us how many tasks are complete, number of errors, etc?
Today’s Schedule

- General Announcements
- Thought Experiment: Computation Framework
- Project 1 Hints
- The Network
- Serialization
For our first project, we’ll design our own distributed file system
  ▪ Decentralized for file retrieval
  ▪ The system will allow you to store and distribute files over a set of machines

Restrictions:
  ▪ Java
  ▪ No outside libraries
Each file will be broken up into multiple chunks.
Chunks are distributed to the Storage Nodes.
When you want to retrieve a file, you’ll:
- Contact any Storage Node to find out where the chunks for the file are.
- Retrieve the chunks in parallel from the storage nodes.
- Reconstruct the file on the client side.
The system will also be fault tolerant

Each chunk is replicated twice for a total of three chunks

If a storage node goes down, you can locate and use other chunks in the system seamlessly

- You will also maintain the replication level by creating more copies
- This part can be facilitated by the Coordinator if you’d like
To detect failures, storage nodes will periodically send **heartbeat** messages.

Heartbeats include amount of free space + number of requests processed.
- Coordinator reply: either an OK message or an updated list of nodes in the system.

Finally, to communicate we’ll be using Google **Protocol Buffers**.
- Since many of you are already comfortable with protobufs, you may also use **manual serialization** by writing raw bytes (more on this in a bit).
- **Do not** use Java serialization, though!
Logistics (1/2)

- I will continue to update the spec with clarifications/hints as things come up
  - But I will not modify the requirements mid-project
- There is also a list of **milestones**
  - Suggested times for completing functionality
- Two graded **checkpoints**
- We'll have in-class lab sessions to discuss and work on the projects
Logistics (2/2)

- This is a large project:
  - 20% of your grade
  - Lots of coding

- Start early and ask questions!
  - We’ll use Piazza for online discussions/help
  - Come by office hours or email
Today’s Schedule

- General Announcements
- Thought Experiment: Computation Framework
- Project 1 Hints
- The Network
- Serialization
For the most part, we can rely on the network to do its job and live at a higher level of abstraction.

Many networking concerns still creep up in distributed systems:
- For instance, do we use TCP or UDP?

We still need to think about:
- Bandwidth
- Latency
Bandwidth

- Also known as *throughput*

- How many bits we can push through the network
  - Trending upward over time… slowly
    - 1000 Mbps networks are common in data centers
    - 10 Gbps is gaining some traction
    - Many home internet services are still in the range of 25-100 Mbps

- If we’re going to use plumbing metaphors, it’s the size of the pipe
Latency

- How long it takes your bits to get from one point to another
  - Latency has been trending downward overall, but not by leaps and bounds
  - We are limited by the laws of physics here
- How long the pipe is and how fast data can travel through it
  - Some communication mediums are more prone to latency: Ethernet vs WiFi
Another factor to consider is the round-trip time (RTTs).

Another way of looking at latency:

Communication rarely goes one way:

- Even if you’re uploading a file, you’d like to confirm that it actually made it over in one piece.

Testing RTTs: pinging a host.
Ping Round Trip Times

- PING alpha.lan (10.0.0.1): 56 data bytes
  64 bytes from 10.0.0.1: icmp_seq=0 ttl=64 **time=1.133 ms**
  (From my couch to the closet)

  64 bytes from 138.202.168.21: icmp_seq=0 ttl=53 **time=15.640 ms**
  (Down the road to USF)

- PING ruby.cs.colostate.edu (129.82.45.204): 56 data bytes
  64 bytes from 129.82.45.204: icmp_seq=0 ttl=50 **time=70.991 ms**
  (Fort Collins, Colorado)

- PING speedtest.shg1.linode.com (139.162.65.37): 56 data bytes
  64 bytes from 139.162.65.37: icmp_seq=0 ttl=50 **time=124.017 ms**
  (Singapore)
Interesting: Physical Distances

- From my router to USF: 2 miles / 3.2 km
- From USF to Colorado: ~950 miles / 1609 km
- From USF to Singapore: ~8,434 miles / 13,573 km
- [https://www.submarinecablemap.com](https://www.submarinecablemap.com)
Source: https://www.submarinecablemap.com
Link capacity (bps) * Round-trip delay time (s)

This measures how many bits are “in flight”
  - How much data is sent before the receiver gets anything

A network with a large bandwidth-delay product is called a **long fat network**, or **LFN** (*elephen*)

Satellite networking has a large bandwidth-delay product
Important because of its interplay with TCP
- TCP dynamically tunes its window size
- If the window is too small, link capacity is wasted
- If the window is too large, the other end gets overwhelmed!
  - Queuing delays

Thinking back to our ping example:
- My local network (300 Mbit/sec, 1.1 ms) = **0.14 MB**
- Singapore (50 Mbit/sec, 124.0 ms) = **0.78 MB**
Most big data applications operate on top of enough abstraction to almost forget the network exists.

This is fine until we hit a situation where one of our nodes experiences packet loss or congestion.

- Distributed computations often wait on all participating nodes’ replies.
- You’re only as fast as your slowest worker.

In the cloud, this can become a major issue.

- Where are your nodes? What is the network like?
Today’s Schedule

- General Announcements
- Thought Experiment: Computation Framework
- Project 1 Hints
- The Network
- Serialization
Previously, we talked a bit about network design.

The messages you send between components and the network design you choose are closely related.

For instance, recall our ring overlay: we can get by with just a single message type.
If you want to send one well-defined message between components, all you need is a fixed-size buffer:

- `byte[] buffer = new byte[25];`
- Remember: sockets are `byte` streams, *not* "message streams"
- A much more common format:
Once you’ve unpacked the message payload, it can contain more fields:

- This allows for a layered approach:
  - Network code
  - Object creation code
  - Pass through a chain of handlers
If you don’t need advanced features, size prefixed messages work well

Exceptions:

- You’d like to avoid reading the entire message before you start processing it
- You don’t even need to process the whole message (perhaps you are forwarding it somewhere else)

Distributed systems’ wire formats have a huge range of features and complexity
Serialization transforms an object, structure, or application state into a format for transmission (and often storage to disk).

Most common: **binary** formats

- Better performance

When you receive a serialized message, transforming it back into its original representation is called **deserialization**.
Java Serialization (1/2)

- Another option is Java’s built-in serialization
  - My advice: don’t use it for anything but prototyping
  - Python has similar functionality in the pickling module

- These types of serialization are language-specific, brittle, and can lead to application errors
  - Memory leaks
  - Broken messages between versions
Automated serialization is often not very performant
- May produce large object graphs
- Class members may be serialized that you don’t need

What’s the big deal though? Aren’t there more important things to worry about?
- Distributed systems, whether they’re storing data or processing it, have to communicate frequently
- In some applications you’ll speed ~50-70% of your CPU time serializing / deserializing messages
One is Protocol Buffers, of course

Another is manual serialization
  - In Java: `DataOutputStream`
    - Write an int for the message size, followed by an int for the message type, then a string, etc...
    - This is tougher to get right

Apache Hive SerDes require manual work but automate some of the processes
Apache Thrift is quite similar to Protocol Buffers

- Designed by Facebook
- Lets you define your messages using a DSL, generate code for different languages, etc.
- Also includes an remote procedure call framework

- XML 😞
- JSON
  - Not highly performant but user friendly, widely supported, and easy to debug
The **JVM Serializers** project by Eishay Smith makes it easy to compare serialization technologies

- See: [https://github.com/eishay/jvm-serializers](https://github.com/eishay/jvm-serializers)

Results shown here were collected on my laptop (more or less similar specs to a commodity server... at least back when it was new!)

- macOS, Quad Core I7-4770HQ, 16 GB RAM
Automatic Serialization + Deserialization (ns)
Size (bytes)
Manual Optimization (ns)