CS 677: Big Data

Scaling Out

Lecture 3

Today's Schedule

- Breaking down the log analyzer
  - Designing a better approach
- Scaling out
- Cluster Orchestration

Today’s Schedule

- **Breaking down the log analyzer**
  - Designing a better approach
- Scaling out
- Cluster Orchestration

The Log Analyzer

- We already discussed a high-level approach for the log analyzer assignment last class
  - (or, how we would do it in Java)
- Would someone like to share their approach?
  - ok, let me go first…

Baby's First Log Analyzer

- Time to get out your code review rocket launchers!

```
// Okay, I was too lazy to do the whole assignment. Whoops.

file, _ := os.Open("log.txt")
bytes, _ := io.ReadAll(file)
lines := strings.Split(string(bytes), "\n");

ips := make(map[string]int)
for _, line := range(lines) {
        ip := strings.Split(line, "\t")[2]
        ips[ip] = ips[ip] + 1
}

fmt.Println("There are", len(ips), "unique IP addresses in the file")
```

Problems

- No error checking
  - If we have a billion-line dataset (which is actually ***not*** that huge by the way!) what are the chances a few records are corrupted?
- Reading the entire file into memory
  - This is a **HUGE** problem!
- File is not closed when we're done (minor)
- Hard-coded path

Comparing Approaches

- I tested two versions of this: one that reads the entire file into memory, and another that reads line by line
  - 1.2 GB log file
  - The “all in memory” version took **3.5s** to run
  - The “line by line” version took **2.2s** to run
- On my laptop (with 16 GB of RAM), both programs work
  - On an EC2 VM with 1.5 GB of RAM, the first program crashes!
    - All you get for an error is “killed” on Linux

Your Approach

- How did you tackle this assignment?

Test Dataset

- I mentioned that I wouldn't share a "full size" dataset with you…
- Let's see how fast your implementation is!
- Check out `/bigdata/mmalensek/logs` on `orion02`
  - `ssh` to `USERNAME@stargate.cs.usfca.edu`
  - Then `ssh` to `orion02`
- There are three options: `small.log`, `medium.log`, and `large.log`

Today's Schedule

- Breaking down the log analyzer
  - **Designing a better approach**
- Scaling out
- Cluster Orchestration

Using our Imagination

- If I gave you infinite time **or** resources for this lab, you could come up with a better approach
- Let's hear your ideas!
  - And let's think about what the downsides of these ideas are

The Message

- At this point I think you probably get it
  - We can design a really awesome log analyzer but there's always going to be a way to overwhelm it
- The best we can do is design a ***scalable*** log analyzer
  - At least then we can keep adding more servers as the problem gets bigger
- And to truly scale, we have to distribute the problem to more than one machine
  - Better algorithms and hardware still matter, even in this case

Today's Schedule

- Breaking down the log analyzer
  - Designing a better approach
- **Scaling out**
- Cluster Orchestration

Scalability

- Humanity is storing more and more data at higher and higher resolutions
- The systems we design should be able to handle these growing workloads
- Managing Big Data, Step 1: use software that can actually handle it
  - Mind-blowing insights here, folks
  - Imagine if I came into class and opened up an Excel spreadsheet on day 1...

Scaling up vs. Scaling out

- Scaling up
  - Faster CPUs
  - Larger RAM modules
  - Bigger disks
- Scaling out
  - More cores/CPUs
  - More machines
  - More disks
- Which one do we pick? Is there one answer?

Why we (usually) don't scale up

- We can't just wait for our hardware to get faster
  - In fact, huge leaps in performance are just not happening anymore
  - Making chips run faster and faster consumes too much power and produces too much heat
- Put simply, we can scale out **now**.
- Scaling out also means flexibility: if we use the **cloud** (or the ideas behind it), then we can grow or shrink our resource pool as necessary

Parallel Computing & Storage

- Architecturally, we need parallel systems
- Parallel computing can be summed up with a simple motto:
  - "Divide and conquer"
- Let's take a problem, break it into smaller pieces, and then have multiple cores/processors/machines work on it all at once
- Challenge: getting all these machines to work together

Today's Schedule

- Breaking down the log analyzer
  - Designing a better approach
- Scaling out
- **Cluster Orchestration**

Working Together

- If we want to scale out, then we need to get multiple machines to work together
- We can ***orchestrate*** computations and storage operations over a **cluster** of machines
- How do we do this coordination? The network!

Exchanging State

- Distributed systems do not have **shared memory**
- Instead, we rely on messages for exchanging state between nodes
  - **Message** – packet of information with a well-defined *wire format*
  - **State** – events occur that mutate the system
  - **Node** – one participant (machine) of the distributed system

Sending a Message

1. Information to be shared is constructed in memory on **Node A**
2. The data is encapsulated and serialized for transfer
   - Well-defined *wire format*
3. The message is sent across the network
4. **Node B** receives the data, reconstructs the message, and applies the information/event to its own state space

TCP

- We use the **Internet Protocol (IP)** Suite for a majority of our communications
- For reliable delivery, we use the **Transmission Control Protocol (TCP)**
  - Modeled as a *stream* of bytes
  - Packets will reach their destination (eventually…) and the contents are verified
    - Retransmit when a failure/corruption occurs
  - Packets are received **in order**

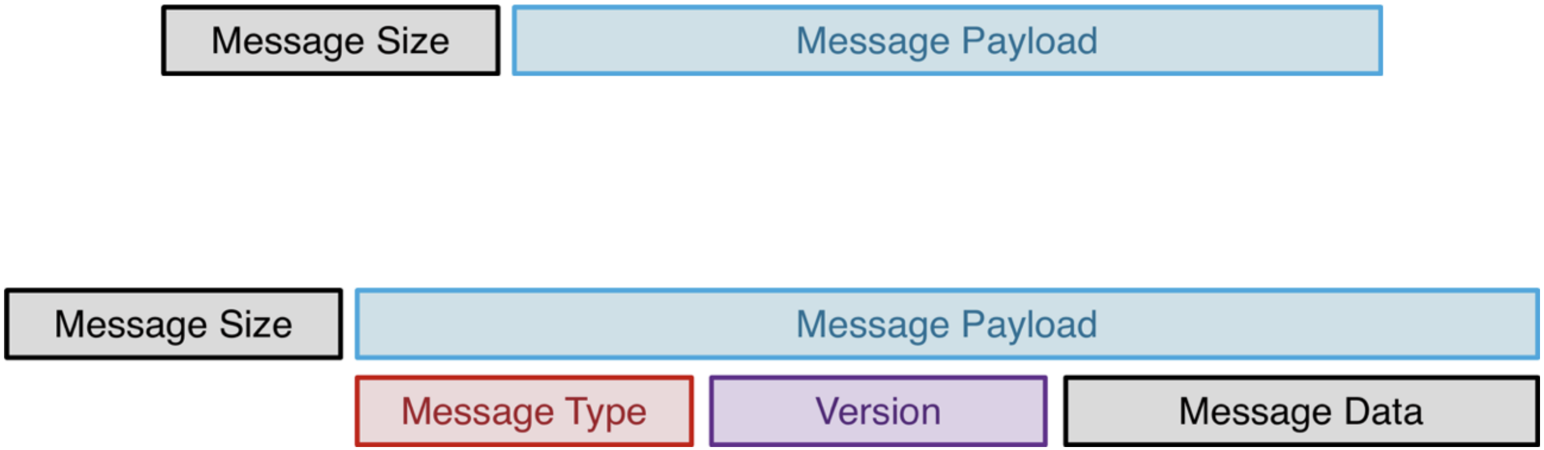
TCP Weirdness

- The first unintuitive thing about (TCP) sockets is there is no concept of a “message”
- Instead, everything gets read/written as streams of bytes
  - Not all the bytes will come in at the same time, although order is guaranteed with TCP
- We generally need to use fixed-size messages or prefix them with a length to know what to expect

Simple Messaging [1/3]

- A common message format:
  - `[ MESSAGE SIZE ][ MESSAGE PAYLOAD ]`
- Once you've unpacked the message payload, it can contain more fields
  - For example: message **type**, version number, flags, etc.
- This allows for a layered approach:
  - Network code
  - Message creation code
  - Pass through a chain of handlers

Simple Messaging [2/3]



Simple Messaging [3/3]

- 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)

Serialization

- **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**

Automated Serialization

- Most languages have built-in serialization functionality (Java Serializable, Python pickling, etc.)
  - My advice: don't use for anything but prototyping
- These types of serialization are language-specific, brittle, and can lead to application errors
  - Memory leaks
  - Broken messages between versions
  - May produce large object graphs
- In some applications you'll speed ~50-70% of your CPU time serializing / deserializing messages

Serialization in Go

- Go provides a built-in serialization format: `gobs`
  - Transforms data types (often used with structs) into bytes
  - Can be written to disk, network, etc.
  - Note: only works with other go-based software
- Another common format: **protocol buffers**
  - Originally designed by Google for internal use
  - Allows broad interoperability
    - Java/Python/etc clients/servers can interact with go seamlessly

Our Approach

- We'll use protocol buffers in this class
  - Decent format, widely used, better compatibility than gobs
- Each message will be prefixed with a size
- You'll send **one** (or maybe a few) types of protobuf messages
  - … **BUT** they will be ***wrappers*** that encapsulate many different sub-types of messages
    - In other words, protobufs will handle encoding the ***message type*** for us

Compiling

- You'll use the `protoc` compiler to generate go code from `.proto` files
- Design your protocol, generate code, and then either `.Marshal()` or `.Unmarshal()` your data
- Recommendation: build helper classes/functions that handle creating these for you
  - They can be kind of… verbose to instantiate inline every time you need them

Sending

```go
// ... a message wrapper has been constructed ... //
serialized, err := proto.Marshal(wrapper)
prefix := make([]byte, 8)
binary.LittleEndian.PutUint64(prefix, uint64(len(serialized)))
util.WriteN(conn, prefix)
util.WriteN(conn, serialized)

// Here, util.WriteN will call conn.Write in a loop
// This ensures *ALL* data is sent!
```

Receiving

```
prefix := make([]byte, 8)
conn.Read(prefix)

payloadSize := binary.LittleEndian.Uint64(prefix)
payload := make([]byte, payloadSize)
util.ReadN(conn, payload)

// util.ReadN reads the data in a loop, similar to WriteN

wrapper := &Wrapper{}
err := proto.Unmarshal(payload, wrapper)

// Ready to determine the type of 'wrapper' and then
// process the message...
```

Determining the Message Type

```
switch msg := wrapper.Msg.(type) {
    case *messages.AwesomeMessage:
        // process ...
    case *messages.NeatMessage:
        // process ...
    case nil:
        log.Println("Received an empty message!")
    default:
        fmt.Errorf("Unexpected message type %T", msg)
}
```

TCP, Messaging, and Protobufs

- In Lab 3, you will put these concepts into practice to create a file transfer suite that is somewhat similar to File Transfer Protocol (FTP).
- You'll use this code to help you implement Project 1
- But first, let's check out an example application that ***also*** uses Protocol Buffers…