Multi-Process Web Development

CS-554 – WEB PROGRAMMING

Terms

IPC: Inter-Process-Communication; the act communicating between processes on the same or multiple machines

Web Role: A role that handles access to your data through an API or UI

Worker Role: A role that performs computation and does not handle any display related functionality

Redis: A key/value store that also has pub/sub capabilities

Pub/Sub: Publishing / Subscription; the ability to listen for events across messages

MQ: A messaging queue

IPC

What is IPC?

Inter-Process Communication is the act of communicating data between processes.

- Could be on the same machine
- Could be on the same network
- Could be across the world

IPC allows us to create incredibly large and stable applications.

- Allows us to scale out extensively
- Allows us to dramatically separate our concerns
- Allows many moving parts to work together

Why?

Separate parts of the application that do not need to be associated with each other. When building out a web app that lists recipes, does your web server really need to contain all the logic needed to generate the caloric content of those recipes?

We can have many, many moving parts with a high degree of performance. Our apps don't need to just exist on a web site; instead, we can start building out application that have things such as mailing lists that generate 10,000 emails an hour and have them running on an entire different machine to ensure that we don't have many resources wasted that make the web server slow.

We can have many more technologies this way; with IPC, we can write our web application in one technology that is good for just rendering web applications. We can then write others in things that are super-optimized for other tasks. Why generate PDFs in node, when we can generate LaTeX in one process and have another process compile it?

Running Multiple Processes

There are many ways you can run multiple processes, but traditionally you:

- Create a single virtual machine instance for each process you need to run; each VM has dependent processes and programs installed.
- Link the processes together through some sort of IPC software across a network
- Each machine

For our course, we will simply:

- Run multiple node applications and have them communicate
- Have them linked together and share communication through Redis Pub/Sub

Our multi-process interactions will therefore be highly event based.

What are web roles?

Web roles are processes that are entirely devoted to handling requests and serving responses, and do not manipulate data of their own.

Web roles tend to have read-access only to a database, and send off data through some sort of messaging queue or other IPC method to another process to perform computations, data storage, etc.

Web roles are intended to be:

- Lightweight
- Scalable
- Very fast

We try to minimize the work our web roles have to do, and make sure that they focus only on responding to requests.

What are worker roles?

Worker roles are processes that do heavy computation and other time sensititve

- They would perform intense calculations
- They would perform the expensive DB and external API calls
- They would perform any potentially locking processes

In Web Development, the primary concern of our worker roles are:

- Make sure the worker role handles tasks as efficiently as possible
- Make sure the server does as little as possible

You use multiple, specialized worker roles to make sure that your heavy computation does not interfere with other tasks. Say we are building an email-alert app based on news that averages 100,000 alerts / hour. You would have:

- Many workers to constantly pull news into our system from many data sources
- Many workers to take news articles and parse them for anything relevant to alerts
- Many workers that handle sending out emails to subscribed users

Benefits to the Programmer

Our code, as usual, becomes even more modular; we can begin to break our code up into packages that only pertain to small amounts of our functionality. We can therefore make our tasks even more self-contained.

We can also optimize our **hardware** for our different apps! Imagine a simple website that has to calculate large Fibonacci Sequence numbers.

Our server needs:

- High RAM on the web server to cache result
- Only a medium amount of CPU to server requests with responses

Our worker role needs:

- Extremely high CPU power to compute results quickly
- Relatively low RAM; it calculates results, then throws them back to the web server

Headaches of the Programmer

We have now introduced a concept of **concurrency!**

Concurrency means that we are running many tasks at the same time, and we have to make sure that there are no state-wide factors that will make our results change based on any other factors.

It is harder to know when a task is completed, and we have to write a larger amount of code.

We have to keep cautious track of the order that things occur, which becomes very hard with this asynchronous programming.

We will try to minimize these issues with simple examples; to see a great deal of how to become a better concurrent programmer, take **CS-511**.

Implementing IPC

Our Strategy: MQ with Redis

We will setup an MQ using Redis.

- Redis is **lightning fast**
- We will learn more about Redis in the coming weeks
- It allows us to focus on application development, rather than the internals of how MQs work and need to be setup.
- http://redis.io/topics/pubsub
  - If you're interested, many more details there; **not required to know**

We will also simply run many processes from multiple terminal windows:

- Run one process for our lightweight web server
- Run one process for each worker role
- We will communicate over Redis
- We will have each process have different DB user roles

Communication between the processes:

Our API will:

- Perform serialization
- Perform error checking
- Respond to successful requests with valid data
- Respond to failed requests with descriptions of errors

Our worker will:

- Listen for messages
- Respond to actions
- Send a response back to the web server

Our data package will:

- Perform error checking on data
- Actually read / write to database

Production level optimizations

In production, we would do the following optimizations:

- Create read-only DB users, where our mongodb connection on the web role would have read-only access
- Create read-write DB users, where our mongodb connection on the worker would have read-write access
- Create DB clusters
- Create Redis clusters
- Create many instances of each worker role
- Create many instances of each web role, and have them hidden behind a load balancer

With a proper IPC setup, you will never have to worry how many instances you are running, as you will simply. This allows for an amazingly easy horizontal scaleout setup.

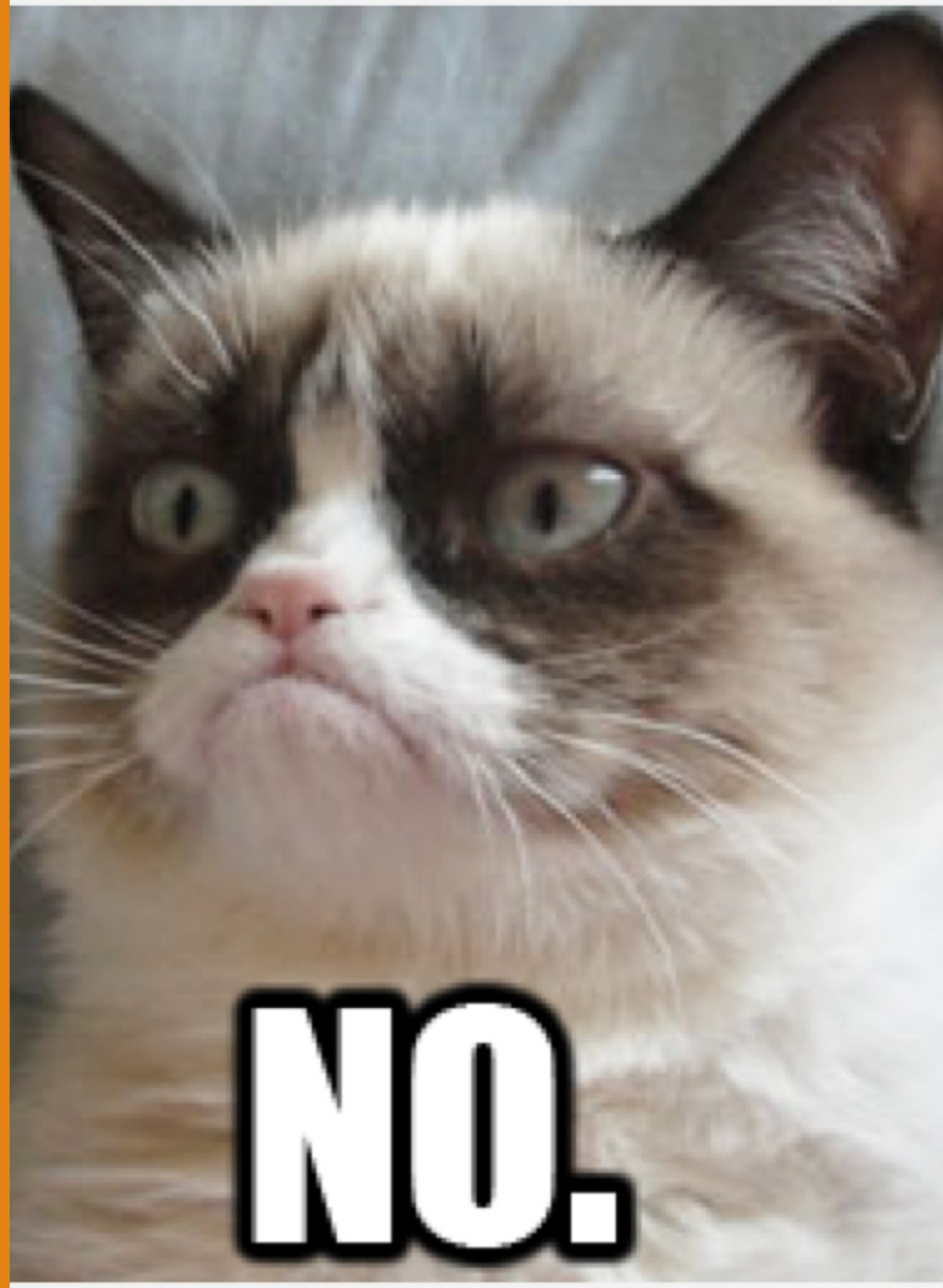
Scaling Out

Didn’t we just cover this?

Actually, so far, we’ve covered *multi-process* web development. This is when the processing of your development life cycle is split across multiple processes (usually on separate virtual machines).

There are many, many things you will setup as an advanced web developer that will cause you to have more machines (virtual, or real) than you will care to manage.

This looks like it was taken at a poor resolution and photoshopped to be larger; I can tell from some of the pixels and from seeing quite a few shops in my time.



Partitioning

Partitioning is when you split up large databases into several smaller databases.

This allows you to:

- Have parts of your program only connect to relevant databases, rather than a master database
- Improve performance by only having to scale out the databases that are frequently used.

Clustering

Clustering is the process of using multiple virtual machines to access the same part of your program. This allows for a number of gains:

- Performance gains from running multiple copies of the same code, and distributing load across those machines
- Backup gains, from having multiple copies of your data across different machines
- Stability gains from removing single points of failure

Clustering allows us to shard, and replicate, as well.

Usually, you would interact with a cluster as if it were a single machine and the particular technology would figure out how to route the request to the proper machine inside the cluster.

Sharding

Sharding is the process of splitting your whole data set across many other machines.

For example, say you have a 10GB database that should be stored in memory but wanted to improve its speed by holding the entire data-set in RAM.

- You would make 5 machines, each that have 6GB of usable RAM (assuming 2GB for OS) for a total of 30GB usable RAM.
- Each of these 5 machines would hold 2 GB of the data set, which is also fully contained in memory
- Your data set can now grow to be 30GB before you have to write data to disk

Sharding allows you to have many smaller, cheaper, machines with relative ease.

Replication

Replication is the process of copying your entire data-set across multiple machines.

- Usually for backup
- Offers performance gains in that you can load balance your requests
- Can access data roughly n-times as fast, where n is the number of machines in your replica.
- **Extremely** useful for reliability. You will ideally never lose data quality if you have a good replica setup.

Sharding and Replicating Together

While their definitions make them sound fundamentally different, you can shard and replicate at the same time to achieve high performance and super reliability.

Say you have a 10GB database that should be stored in memory and want to improve its speed by holding the entire data-set in RAM with 3 replicated backups of each piece of data.

- We therefore need: 10GB of sharded data, and 30GB of replicated data (40GB total)
- Make 10 machines, each that have 6GB of usable RAM (2GB for OS) for a total of 60GB usable RAM.
- Each of these 10 machines would hold 1 GB of the data set, which is also fully contained in memory
- Each of these 10 machines would hold 3 GB of replicated data, which is fully contained in memory.
- Each machine now is using 4GB of data, with 2 GB to spare.
- Machine #8 goes down, and it's 4 GB are spread out over 9 other machines, meaning each other machine uses 4.5GB/6GB of RAM
- You receive more funding and buy 2 more machines, and the load is spread to 3.6GB across each

What gets clustered?

Usually, you'll have a cluster running for each of the following:

- Web server
- Load balancer
- Per database server type (ie, redis gets one; mongo gets one)
- Message queue
- Worker role

A good rule of thumb is "at least 3 machines" in the cluster.

- In a 50 / 50 split with 2 machines, if one machine goes down, then you can't function
- In a 50 / 50 split with 3 machines, where each machine has more than 50% of the total data / processing power / etc, then when 1 goes down you can survive as if nothing was going wrong until you figure out what went wrong.
- If more than 1 machine goes down and you only have 3 machines, you usually have a bigger catastrophe on your hands.

IPC Examples

Hello -> World

We’re going to start incredibly simple: IPC without a reply, and without a server

Let’s load up the *simple* directory into **two** terminal windows.

- In terminal one, run: **node receiver.js**
- After that, in terminal 2, run: **node sender.js**

This will fire off messages from *sender.js* to *receiver.js*

POST Hello -> World

We’re going to start a little less simply, this time with a server.

Let’s load up the *api* directory and run **npm-start**

Let’s load up the **simple-worker** directory and **npm-start** as well.

Now, let’s POST a message to */send-message*.

- You’ll notice that the simple worker logs it.

Abstracting Bi-Directional Communication into a Promise

We're going to start a little less simply, this time with a server.

Let's load up the **api** directory and run **npm-start**

Let's load up the **worker** directory and **npm-start** as well.

Now, let's POST a message to http://localhost:3000/send-message-with-reply