26. Cloud-scale computing abstractions
Introduction to Distributed Systems

Azar Bestavros
Computer Science Department
Boston University

© Copyright Azar Bestavros / All rights reserved.

Distributed Systems

- Typical characteristics:
  - Entities in a Distributed System are “Autonomous”
  - No Inherent Synchronization (e.g. global clock)
  - No Shared Resources (e.g. memory)
  - Heterogeneous Platforms (e.g. Unix, Windows, etc.)
  - Communication via Messages
  - Wires have state (e.g. packets in transit)
  - Unreliability is Inherent
  - “Byzantine” Failures are possible
  - …

- Client-Server Systems
  - Distributed File systems (e.g., NFS, AFS, etc.)
  - Web Applications (e.g. HTTP server/browser, etc.)

- Peer-Peer Systems
  - Distributed Apps (e.g., DDBMS, DHT, Caching, etc.)
  - Internet Infrastructure (e.g. routing, DNS, etc.)

To be able to get anything useful, we need to rely on some basic abstractions

- Communication is reliable; no messages are lost
- Message delivery delays can be arbitrary, but not infinite
- Messages are delivered in order

And often, we need to build/rely on some key capabilities (typically related to synchronization)

- Nodes in a distributed system can elect a leader
- Nodes in a distributed system can agree on event ordering
- Nodes in a distributed system can agree on time

Abstractions vs Expressiveness

- How much should we expose to programmers?
  - The more we expose, the more efficient we could be
  - The less we expose, the more programmers we serve

- We need a “spectrum” of paradigms
  - Distributed Systems for Dummies
  - Distributed Systems for CS-552

Distributed Computing Paradigms

- Remote Procedure Calls (RPC)
  - Most natural progression – from threads on a single CPU to threads on multiple CPUs
  - Allow function calls to cross the boundaries of a single host, with possibility of no shared memory…
  - Must deal with all the synchronization and management overhead
Distributed Computing Paradigms

- **Client-Server Computing**
  - Application specific paradigm (e.g., HTTP, SMTP, ...)
  - Blurred client and server roles gave rise to the P2P paradigm (e.g., Gnutella, Bittorrent, ...)

- **MapReduce**
  - Designed (reinvented) by Google for making a subset of distributed problems easier to code
  - Automates data distribution & result aggregation
  - Restricts the ways data can interact to eliminate locks (no shared state = no locks!)

A Spectrum of Paradigms

- Paradigms differ in what they abstract
  - Less granular abstractions (e.g., RPC) are more expressive but not for consumption by amateurs
  - Granular abstractions (e.g., client-server) are less expressive, but safe for consumption by amateurs
  - The "right" granularity (e.g., MapReduce) is somewhere in the middle and depends on targeted application

- Easier if paradigm does not have to worry about the communication infrastructure – how messages between threads are exchanged?
  - Need an abstraction for that too!

Sockets and Ports

- A socket is the basic network interface
  - Provides a two-way "pipe" abstraction between two applications

- On server, a port identifies a "listening" program
  - Allows multiple clients to connect to a server at once

Example: Web Server

1. Server creates a socket attached to port 80 and listens in on that port for client requests

2. Client creates a local socket and connects to port 80 of the server
Example: Web Server

(3) Server accepts connection, creates a dedicated socket, and spawns a service thread. Server continues to listen in on port 80.

Client anonymously connects to server, and server thread is spawned.

Client-Server Interactions

- Regular client-server protocols involve sending data back and forth according to a shared state

Client:
HTTP/1.0 index.html GET
200 OK
Length: 2400
(file data)

Server:
HTTP/1.0 hello.gif GET
200 OK
Length: 81494
...

Remote Procedure Call

- RPC to servers will call arbitrary functions in dll, exe, with arguments passed over the network, and return values back over network

Client: foo.dll,bar(4, 10, "hello")
        "returned_string"
        err: no such function

Server: ...

Possible Interfaces

- RPC can be used with two basic interfaces: 
  - synchronous and asynchronous
    - Synchronous RPC is a "remote function call" – client blocks and waits for return value
    - Asynchronous RPC is a "remote thread spawn" – client does not need to block
Synchronous RPC

Asynchronous RPC

Asynchronous RPC: Callbacks

More Design Considerations

- Numerous protocols: DCOM, CORBA, JRMI...
- Who can call RPC functions? Anybody?
- How to handle multiple versions of a function?
- Need to marshal objects
- How do you handle error conditions?

- Oh! And, then there is safety and security...
- And, what about these "sockets"?

What makes "sockets" work?

- Underneath the socket layer are several more protocols
- Most important are TCP and IP (which are used hand-in-hand so often, they're often spoken of as one protocol: TCP/IP)

IP: The Internet Protocol

- Defines the addressing scheme for computers
- Encapsulates internal data in a "packet"
- Does not provide reliability
- Just includes enough information for the data to tell routers where to send it
TCP: Transmission Control Protocol

- Built on top of IP
- Introduces concept of “connection”
- Provides reliability and ordering

Why is This Necessary?

- Not actually tube-like “underneath the hood”
- Unlike the phone system (circuit switched), the packet switched Internet uses many routes at once

Layering: In Networking

- The set of protocols at the various layers constitute the protocol “stack”.
- Software written for layer i on C1 “believes” that it is interacting with software at layer i on C2.
- The software at layer n at the destination receives exactly the same protocol message sent by layer n at the sender.

Layering: In Networking (IP Stack)

Layering: In Networking

- Layering is an approach for managing complexity
- Systems are designed at one layer with a model of the functionality of the underlying layer
- Details of the implementation details of a lower layer functionality are hidden from the upper layer
- Layering allows compatibility with multiple and/or new implementations
- Any disadvantages?

Layering: In Networking

- Layering allows for services to be shared and/or combined at higher layers.
- Example: TCP/IP Stack
Hadoop / Map Reduce

Azer Bestavros

Slides (with minor adjustments) are courtesy of Andreas Haeberlen

Application: National census

- Suppose we have 10,000 employees, whose job is to collate census forms and to determine how many people are in a given profession
- How would you organize this task?

Abstractive the data flow

Making things more complicated

- Employees take vacations, get sick, work at different rates
- Some forms are incorrectly filled and require corrections or need to be thrown away
- What if the supervisor gets sick?
- How big should the stacks be?
- How do we monitor progress?
- ...

Application: Indexing the web

- Suppose we have 10,000 computers, used to scrape the web daily to compute an index of medical terms for WebMD
- How would you organize this task?

Abstractive the data flow
Making things more complicated

- Computers may go offline, crash, or operate at different speeds
- Some web pages contain misspelled terms and/or are used for spamming/advertisement
- Network or data center control may fail
- How big should the batches per server be?
- How do we monitor progress?
- ...

I don't want to deal with all this!!!

- Wouldn't it be nice if there were some system that took care of all these details for you?
- Ideally, you'd just tell the system what needs to be done...
- That's the Hadoop/MapReduce framework.

What is MapReduce?

- A distributed programming model
- In many circles, considered the key building block for much of Google's data analysis
    - Sawzall has become one of the most widely used programming languages at Google. ... (in one dedicated Workqueue cluster with 5120 Xeon CPUs, there were 32,580 Sawzall jobs launched, using an average of 220 machines each. While running those jobs, 26,624 failures occurred (application failure, network outage, system crash, etc.) that triggered resuming some portion of the job. The jobs read a total of 3.2x10^15 bytes of data (2.8PB) and wrote 9.9x10^12 bytes (9.3TB)
    - Other similar languages: Yahoo's Pig Latin and Pig; Microsoft's Dryad

Why MapReduce?

- Scenario:
  - You have a huge amount of data, e.g., all the Google searches of the last three years
  - You would like to perform a computation on the data, e.g., find out which search terms were the most popular
- How would you do it?
  - The computation isn't necessarily difficult, but parallelizing and distributing it, as well as handling faults, is challenging
- Idea: A programming language!
  - Write a simple program to express the (simple) computation, and let the language runtime do all the hard work

Underlying Abstraction...

- There are two kinds of workers:
  - Those that take input data items and organize them into "stacks" of items for later processing
  - Those that take the stacks and aggregate the results to produce outputs on a per-stack basis
- We'll call these:
  - mapper: takes (item_key, value), produces one or more (stack_key, value') pairs
  - reducer: takes (stack_key, {set of value'}), produces one or more output results – typically (stack_key, agg_value)
- We will refer to this key as the reduce key

The MapReduce programming model

- Simple distributed functional programming primitives modeled after Lisp primitives:
  - map (apply function to all items in a collection) and
  - reduce (apply function to set of items with a common key)
- We start with:
  - A user-defined function to be applied to all data, map: (data key, data value) \rightarrow (key, value)
  - Another user-specified operation, reduce: (key, {set of values}) \rightarrow result
  - A set of n nodes, each with data
- All nodes run map on all of their data, producing new data with keys
  - This data is collected by key, then shuffled, and sent to nodes and reduced
**Simple example: Word count**

**Map**

```java
map(String key, String value) {
    // key: document name, line no
    // value: contents of line for each word w in value:
    emit(w, "1")
}
```

**Reduce**

```java
reduce(String key, Iterator values) {
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    emit(key, result)
}
```

**Goal:** Given a set of documents, count how often each word occurs
- **Input:** Key-value pairs (document:lineNumber, text)
- **Output:** Key-value pairs (word, #occurrences)
- What should be the intermediate key-value pairs?

Each mapper receives some of the KV-pairs as input

The mappers process the KV-pairs one by one

Each KV-pair output by the mapper is sent to the reducer that is responsible for it

The reducers sort their input by key and group it

The reducers process their input one group at a time

**MapReduce dataflow**

Input data → Mapper → Intermediate \( (key, value) \) pairs → Shuffle → Reducer → Output data

What is meant by a 'dataflow'? What makes this so scalable?

**More examples**

- **Distributed grep – all lines matching a pattern**
  - **Map:** filter by pattern
  - **Reduce:** output set

- **Count URL access frequency**
  - **Map:** output each URL as key, with count 1
  - **Reduce:** sum the counts

- **Reverse web-link graph**
  - **Map:** output (target,source) pairs when link to target found in source
  - **Reduce:** concatenates values and emits (target,list(source))

- **Inverted index**
  - **Map:** Emits (word,documentID)
  - **Reduce:** Combines these into (word,list(documentID))

**Recap: MapReduce dataflow**

Input data → Mapper → Intermediate \( (key, value) \) pairs → Shuffle → Reducer → Output data

"The Shuffle"
Designing MapReduce algorithms

- **Key decision:** What should be done by *map*, and what by *reduce*?
  - *map* can do something to each individual key-value pair, but it can’t look at other key-value pairs
  - Example: Filtering out key-value pairs we don’t need
  - *map* can emit more than one intermediate key-value pair for each incoming key-value pair
    - Example: Incoming data is text, *map* produces (word,1) for each word
    - *reduce* can aggregate data; it can look at multiple values, as long as *map* has mapped them to the same (intermediate) key
    - Example: Count the number of words, add up the total cost, ...

- **Need to get the intermediate format right!**
  - If *reduce* needs to look at several values together, *map* must emit them using the same key!

Map Reduce on Movie Ratings

- **Map:**
  
  input → [key=(mID,R); value=1]

- **Reduce:**
  
  [key=(mID,R); {v}] → [key=(mID,R); value = sum({v})]

Can we make the histogram for all movies in database?

Map Reduce on Movie Ratings

- **Map:**
  
  input → [key=(mID,R); value=1]

- **Reduce:**
  
  [key=(mID,R); {v}] → [key=(mID,R); value = sum({v})]

- **Map:**
  
  [key=(mID,R); sum] → [key=R; value=sum]

- **Reduce:**
  
  [key=R; {s}] → [key=R; value = sum({s})]

Can we make the histogram from 1 to 5 stars (instead of an integer from 1-10)?

Map Reduce on Movie Ratings

- **Map:**
  
  input → [key=(mID,R); value=1]

- **Reduce:**
  
  [key=(mID,R); {v}] → [key=(mID,R); value = sum({v})]

- **Map:**
  
  [key=R; {s}] → [key=R; value = sum({s})]

- **Reduce:**
  
  [key=S; {s}] → [key=S; value = sum({s})]

Hadoop and friends

Azer Bestavros
Who uses (or used) Hadoop?

- Hadoop is (was) running search on some of the Internet's largest sites:
  - Amazon Web Services: Elastic MapReduce
  - AOL: Variety of uses, e.g., behavioral analysis & targeting
  - eBay: Search optimization
  - Facebook: Reporting/analytics, machine learning
  - Fox Interactive Media: MySpace, Photobucket, Rotten T.
  - Last.fm: Track statistics and charts
  - IBM: Blue Cloud Computing Clusters
  - LinkedIn: People You May Know
  - Rackspace: Log processing
  - Twitter: Store + process tweets, log files, other data
  - Yahoo: >40,000 nodes; biggest cluster is 4,500 nodes/455PB

What do MR programmers write?

- A mapper
  - Accepts (key,value) pairs from the input
  - Produces intermediate (key,value) pairs to be shuffled
- A reducer
  - Accepts intermediate (key,value) pairs
  - Produces final (key,value) pairs for the output
- A driver
  - Specifies which inputs to use, where to put the outputs

→ Hadoop takes care of the rest!!
  - Storage, scheduling, synchronization, fault tolerance, ...

What is HDFS?

- HDFS is a distributed file system
  - Makes some unique tradeoffs that are good for MapReduce
- What HDFS does well:
  - Very large read-only or append-only files (individual files may contain Gigabytes/Terabytes of data)
  - Sequential access patterns
- What HDFS does not do well:
  - Storing lots of small files
  - Low-latency access
  - Multiple writers
  - Writing to arbitrary offsets in the file

HDFS versus NFS

- Single machine makes part of its file system available to other machines
- Sequential or random access
- PRO: Simplicity, generality, transparency
- CON: Storage capacity and throughput limited by single server
- Single virtual file system spread over many machines
- Optimized for sequential read and local accesses
- PRO: High throughput, high capacity
- "CON": Specialized for particular types of applications

How data is stored in HDFS

- Files are stored as sets of (large) blocks
  - Default block size: 64 MB (ext4 default is 4kB!)
  - Blocks are replicated for durability and availability
  - What are the advantages of this design?
- Namespace is managed by a single name node
  - Actual data transfer is directly between client & data node
  - Pros and cons of this decision?
The Secondary Namenode

- What if the state of the namenode is lost?
  - Data in the file system can no longer be read!

- Solution #1: Metadata backups
  - Namenode can write its metadata to a local disk, and/or to a remote NFS mount

- Solution #2: Secondary Namenode
  - Purpose: Periodically merge the edit log with the fsimage to prevent the log from growing too large
  - Has a copy of the metadata, which can be used to reconstruct the state of the namenode
  - But: State lags behind somewhat, so data loss is likely if the namenode fails

Hadoop (HDFS) was built for MR

Could this be useful for other paradigms?

Apache Spark

- Instead of just “map” and “reduce”, defines a large set of operations
  - Transformations to “set things up” if and when needed
  - Actions to “do computation” if needed
- Operations can be arbitrarily combined in any order
- Supports Java, Scala and Python
- Adds persistent (intermediate) storage

Spark: Example (Python)

```python
# Estimate Pi (compute-intensive task).
# Pick random points in the unit square ((0, 0) to (1,1)),
# See how many fall in unit circle. The fraction = 0.25 Pi
# Note that “parallelize” method creates an RDD

def sample(p):
    x, y = random(), random()
    return 1 if x*x + y*y < 1 else 0

count = spark.parallelize(xrange(0, NUM_SAMPLES)).map(sample)
    .reduce(lambda a, b: a + b)

print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

Source: https://spark.apache.org/docs/latest/quick-start.html

Spark: Libraries

- Spark SQL
  - Queries over structured data
- Spark Streaming
  - Stream processing of live data streams
- Mllib
  - Machine learning
- GraphX
  - Graph manipulation
  - Extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to vertices/edges
## Networking Issues

- If a party to a socket disconnects, how much data did they receive?
- Did they crash? Or did a machine in the middle crash?
- Can someone in the middle intercept/modify the data?
- What is the cheapest, most efficient way to route data? Traffic congestion makes routing topology important for efficient throughput!

## Conclusion

- Distributed systems: Building layered abstractions to facilitate Internet-scale applications
- Designing a large distributed system involves trade-offs at each of these levels
- Appreciating subtle concerns at each level requires diving past the abstractions, but abstractions are still useful