Remote Procedure Call

Tom Anderson
Why Are Distributed Systems Hard?

• Asynchrony
  – Different nodes run at different speeds
  – Messages can be unpredictably, arbitrarily delayed

• Failures (partial and ambiguous)
  – Parts of the system can crash
  – Can’t tell crash from slowness

• Concurrency and consistency
  – Replicated state, cached on multiple nodes
  – How to keep many copies of data consistent?
Why Are Distributed Systems Hard?

• Performance
  – Have to efficiently coordinate many machines
  – Performance is variable and unpredictable
  – Tail latency: only as fast as slowest machine

• Testing and verification
  – Almost impossible to test all failure cases
  – Proofs (emerging field) are really hard

• Security
  – Need to assume adversarial nodes
Three-tier Web Architecture

• Scalable number of front-end web servers
  – Stateless ("RESTful"): if crash can reconnect the user to another server

• Scalable number of cache servers
  – Lower latency (better for front end)
  – Reduce load (better for database)
  – Q: how do we keep the cache layer consistent?

• Scalable number of back-end database servers
  – Run carefully designed distributed systems code
And Beyond

• Worldwide distribution of users
  – Cross continent Internet delay ~ half a second
  – Amazon: reduction in sales if latency > 100ms

• Many data centers
  – One near every user
  – Smaller data centers just have web and cache layer
  – Larger data centers include storage layer as well
  – Q: how do we coordinate updates across DCs?
MapReduce Computational Model

For each key $k$ with value $v$, compute a new set of key-value pairs:

$$\text{map } (k,v) \rightarrow \text{list}(k',v')$$

For each key $k'$ and list of values $v'$, compute a new (hopefully smaller) list of values:

$$\text{reduce } (k',\text{list}(v')) \rightarrow \text{list}(v'')$$

User writes map and reduce functions.
Framework takes care of parallelism, distribution, and fault tolerance.
MapReduce Example: grep
find lines that match text pattern

1. Master splits file into M almost equal chunks at line boundaries
2. Master hands each partition to mapper
3. map phase: for each partition, call map on each line of text
   – search line for word
   – output line number, line of text if word shows up, nil if not
4. Partition results among R reducers
   – map writes each output record into a file, hashed on key
Example: grep

5. Reduce phase: each reduce job collects 1/R output from each Map job
   – all map jobs have completed!
   – Reduce function is identity: v1 in, v1 out

6. merge phase: master merges R outputs
MapReduce (or ML or ...) Architecture

• Scheduler accepts MapReduce jobs
  – finds a MapReduce master and set of avail workers
• For each job, MapReduce master <array>
  – farms tasks to workers; restarts failed jobs; syncs task completion
• Worker <array>
  – executes Map and Reduce tasks
• Storage <array>
  – stores initial data set, intermediate files, end results
Remote Procedure Call (RPC)

A request from the client to execute a function on the server.

– To the client, looks like a procedure call
– To the server, looks like an implementation of a procedure call
Remote Procedure Call (RPC)

A request from the client to execute a function on the server.

- **On client**
  - Ex: result = DoMap(worker, i)
  - Parameters marshalled into a message (can be arbitrary types)
  - Message sent to server (can be multiple pkts)
  - Wait for reply

- **On server**
  - message is parsed
  - operation DoMap(i) invoked
  - Result marshalled into a message (can be multiple pkts)
  - Message sent to client
RPC implementation

DoMap(worker, i)
- Serialize args
- Open connection
- Write data
- TCP/IP write
- Transport

Map(worker, i)
- Serialize reply
- Read data
- Deserialize reply
- TCP/IP read
- Transport

RPC library

CSE 461
RPC vs. Procedure Call

• What is equivalent of:
  – The name of the procedure?
  – The calling convention?
  – The return value?
  – The return address?
**RPC vs. Procedure Call**

**Binding**
- Client needs a connection to server
- Server must implement the required function
- What if the server is running a different version of the code?

**Performance**
- procedure call: maybe 10 cycles = \( \sim 3 \) ns
- RPC in data center: 10 microseconds \( \Rightarrow \) \( \sim 1K \) slower
- RPC in the wide area: millions of times slower
RPC vs. Procedure Call

Failures

– What happens if messages get dropped?
– What if client crashes?
– What if server crashes?
– What if server crashes after performing op but before replying?
– What if server appears to crash but is slow?
– What if network partitions?
Semantics

• Semantics = meaning

• reply == ok => ???
• reply != ok => ???
Semantics

• At least once (NFS, DNS, lab 1b)
  – true: executed at least once
  – false: maybe executed, maybe multiple times

• At most once (lab 1c)
  – true: executed once
  – false: maybe executed, but never more than once

• Exactly once
  – true: executed once
  – false: never returns false
At Least Once

RPC library waits for response for a while
If none arrives, re-send the request
Do this a few times
Still no response -- return an error to the application
Non-replicated key/value server

Client sends Put k v

Server gets request, but network drops reply

Client sends Put k v again
  – should server respond "yes"?
  – or "no"?

What if op is “append”?
Does TCP Fix This?

• TCP: reliable bi-directional byte stream between two endpoints
  – Retransmission of lost packets
  – Duplicate detection

• But what if TCP times out and client reconnects?
  – Browser connects to Amazon
  – RPC to purchase book
  – Wifi times out during RPC
  – Browser reconnects
When does at-least-once work?

- If no side effects
  - read-only operations (or idempotent ops)
- Example: MapReduce
- Example: NFS
  - readFileBlock
  - writeFileBlock
At Most Once

Client includes unique ID (UID) with each request
  – use same UID for re-send

Server RPC code detects duplicate requests
  – return previous reply instead of re-running handler

```python
if seen[uid] {
    r = old[uid]
} else {
    r = handler()
    old[uid] = r
    seen[uid] = true
}
```
Some At-Most-Once Issues

How do we ensure UID is unique?

– Big random number?
– Combine unique client ID (IP address?) with seq #?
– What if client crashes and restarts? Can it reuse the same UID?
– In labs, nodes never restart
– Equivalent to: every node gets new ID on start
When Can Server to Discard Old RPCs?

Option 1: Never?

Option 2: unique client IDs per-client RPC sequence numbers client includes "seen all replies <= X" with every RPC

Option 3: only allow client one outstanding RPC at a time arrival of seq+1 allows server to discard all <= seq

Labs use Option 3
What if Server Crashes?

If at-most-once list of recent RPC results is stored in memory, server will forget and accept duplicate requests when it reboots

– Does server need to write the recent RPC results to disk?

– If replicated, does replica also need to store recent RPC results?

In Labs, server gets new address on restart

– Client messages aren’t delivered to restarted server