RPC Semantics

Doug Woos
Logistics notes

Tom’s OH canceled this week
Last time

- Go tips and tricks
- RPC intro, using RPCs in Go
- MapReduce discussion
Outline

RPC semantics in detail
Go’s RPC semantics
RPC Warmup

What’s the equivalent of:

- Procedure name?
- Calling convention?
- Return value?
- Return address?
Semantics

semantics: meaning
ok := call(address, "Worker.DoJob", args, &reply)

func (wk *Worker) DoJob(args *DJArgs, reply *DJReply)
Semantics

semantics: meaning

- ok == true: ???
- ok == false: ???
- Possibilities?
Semantics

At least once (NFS, DNS, …):
- true = executed at least once
- false = maybe executed, multiple times

At most once (Go, …):
- true = executed exactly once
- false = maybe executed once

- Exactly once (Lab 2 writes)
  - true = executed exactly exactly once
  - never returns false
At least once

RPC library sends, waits for response
If none arrives, re-send request
After a few retries, give up and return an error
How should applications deal with this?
Example: one-node KV store (Redis)

Client sends PUT k v
Server gets request, reply dropped
Client sends PUT k v again
- What should the server do?

What if instead, op is “deduct $10 from bank acct”
What about TCP?

“Just use TCP”

TCP: reliable byte stream between two endpoints

- Retransmission of dropped packets
- Duplicate detection & removal

What if TCP times out and reconnects?

- User browses to Amazon
- RPC to purchase book
- Wifi spotty during RPC
- Browser reconnects
When does at-least-once work?

No side effects (e.g. MapReduce jobs)
- read-only, idempotent

NFS: readFileBlock, writeFileBlock

Application-level duplicate detection
At most once

Client includes unique id (UID) with each request
  - same UID on re-send

RPC lib on server detects duplicates

```python
if seen[uid] {
    return old[uid]
} else {
    r = Handler()
    old[uid] = r
    seen[uid] = true
    return r
}
```
Some at-most-once issues

How to ensure unique UID?
- large random numbers
- combine UID (e.g. MAC address) w/ sequence #

Can clients use same UID if they crash?

Get UID from server?
When can server discard old?

Option 1
- Never!

Option 2
- Unique client IDs
- per-client sequence number
- client includes “discard <= i” w/ all RPCs

Option 3
- only allow one outstanding RPC per client
- When seq+1 arrives, discard <= seq

Option 4
- Client gives up after n minutes
- Server discards after n minutes
Handling server crashes

Server will lose old on crash

- Does it need to be persisted?
- Does it need to be replicated?
Handling server crashes

Server will lose old on crash
- Does it need to be persisted?
- Does it need to be replicated?

Needs to have same persistence/replication as data
Go RPC revisited

What are the semantics?
Go RPC revisited

At most once
Rely on TCP retry
  - Open connection
  - Write data
  - TCP may retransmit
Return error if no reply after timeout
Go’s at-most once is not enough

Imagine side-effectful MapReduce
Master sends RPC to worker, gets a timeout
What does application do?
- Attempt to figure out if work was done
- Implement better at-most-once
- Lab 2!
Exactly once

Keep retrying forever

Need to survive client and server crashes
  - Client must store pending RPCs on disk
  - Server must store completed RPCs on disk
Takeaways

Failure makes RPCs complicated
Think carefully about semantics
Mechanisms in app vs. RPC vs. transport