RPC and Threads

Jinyang Li

These slides are based on lecture notes of 6.824
Labs are based on Go language

• Why Golang? (as opposed to the popular alternative: C++)
  – good support for concurrency
  – good support for RPC
  – garbage-collected (no use after freeing problems)
  – good and comprehensive library

• Notable systems built using Go
  – Dropbox’s backend infrastructure
  – CoachroachDB
Threads

• Thread = “thread of execution”
  – allow one program to logically execute many things at once
  – Each thread has its own per-thread state:
    • program counter
    • registers
    • stack
  – All threads share memory (occupy same address space)

Golang refers to threads as goroutines
Why threads and how many?

• Threads are created to exploit concurrency
  – I/O concurrency: while waiting for a response from another machine, process another request
  – Multicore: threads run parallel on many CPU cores

• Go encourages one to create many goroutines
  – many more than # of cores
  – Go runtime schedules threads on available cores

• Goroutines are more efficient than C++ threads
  – but still more expensive than function calls
Threading Challenges: races

- Races arise because of shared memory

```go
var x int
for i := 0; i < 2; i++ {
    go func() {
        x++
        x++
    }()
}
```

Go run –race main.go

Read x=0 into %rax
Incr %rax to 1
Write %rax=1 to x
Threading challenges: races

• Races due to shared memory

```go
var x int
var mu sync.Mutex
for i := 0; i < 2; i++ {
    go func() {
        mu.Lock()
        defer mu.Unlock()
        x++
    }()
}
```

go run –race main.go

• defer is executed when the enclosing function returns
• Try to always put mu.Unlock in defer
Threading challenges: coordination

• Mechanism: Go channel
  – For passing information between goroutines
  – can be unbuffered or have a bounded-size buffer
  – Several threads can send/receive on same channel
    • Go runtime uses locks internally
  – Sending blocks
    • when buffered channel is full
    • when no thread is ready to receive from unbuffered channel
  – Receiving blocks
    • when channel is empty
  – Channel is closed to indicate the end
    • receiving from a closed channel returns error
Threading challenges: coordination

• Mechanism: Waitgroup
  – Used for waiting for a collection of threads to finish
  – Supports 3 methods:
    • Add(int x): add x (threads) to the collection
    • Done(): called when one thread has finished
    • Wait: blocks until all threads in the collection has finished
func main() {

    workChan := make(chan int) // unbuffered channel
    go func() {
        for i := 1; i <= 20; i++ {
            workChan <- i
        }
    }()

    for i := 0; i < 5; i++ {
        go func() {
            for {
                n := <- workChan
                f := computeFactorial(n)
                fmt.Printf("n=%d, f=%d\n", n, f)
            }
        }()
    }
}
func main() {

    workChan := make(chan int, 20) //buffer size 20
    for i := 1; i <= 20; i++ {
        workChan <- i
    }

    var wg sync.WaitGroup
    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for {
                n :=<- workChan

                f := computeFactorial(n)
                fmt.Printf("n=%d, f=%d\n", n, f)
            }
        }()
    }
    wg.Wait()
}
func main() {

    workChan := make(chan int, 20) //buffer size 20
    for i := 1; i <= 20; i++ {
        workChan <- i
    }
    close(workChan)

    var wg sync.WaitGroup
    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for {
                n, ok := <- workChan
                if !ok { //alternative: for n:= range workChan
                    break
                }
                f := computeFactorial(n)
                fmt.Printf("n=%d, f=%d\n", n, f)
            }
        }()
    }
    wg.Wait()
}
RPC

• A key piece of infrastructure when building DS
• RPC’s goals:
  – easier to program than raw sockets
  – hides details of client/server communication
• Ideal RPC interface

Client:

\[ z = \text{fn}(x, y) \]

Server:

\[
\begin{align*}
\text{fn}(x, y) \{ \\
\text{...} \\
\text{return z} \\
\}
\end{align*}
\]
Example: KV service (Server-side)

```go
import "net/rpc"

type PutArgs struct {
    Key string
    Value string
}

type PutReply struct {
    Err error
}

type KV struct {
    mu sync.Mutex
    keyvalue map[string]string
}

func (kv *KV) Put(args *PutArgs, reply *PutReply) error {
    kv.mu.Lock()
    defer kv.mu.Unlock()
    kv.keyvalue[args.Key] = args.Value
    reply.Err = OK
    return nil
}
```

- RPC handlers have a certain signature (two arguments, second being a pointer, return type error)
- RPC handlers must be exported (First letter capitalized)

```go
func (kv *KV) get(key string) string {
    ...
}
```
Example: starting RPC server

```go
func startServer() {
    rpcs := rpc.NewServer()
    kv := KV{keyvalue: make(map[string]string)}
    rpcs.Register(&kv)
    l, e := net.Listen("tcp", "::8888")
    go func() {
        for {
            conn, err := l.Accept()
            if err == nil {
                go rpcs.ServeConn(conn)
            } else {
                break
            }
        }
        l.Close()
    }()
}
```

Server handles each connection and request in a separate thread
Example: client-side

type KVClient struct {
    clt *rpc.Client
}

func NewKVClient() *KVClient {
    clt, err := rpc.Dial("tcp", ":8888")
    return &KVClient{clt: clt}
}

func (c *KVClient) Put(key string, value string) {
    args := &PutArgs{Key: key, Value: val}
    reply := PutReply{}
    err := c.clt.call("KV.Put", args, &reply)
}
Example: putting it together

```go
func main() {
    startServer()
    client := NewKVClient()
    client.Put("nyu", "New York University")
    client.Put("cmu", "Carnegie Mellon University")
    fmt.Printf("Get value=%s\n", client.Get("nyu"))
}
```
**RPC software structure**

### Client application
- Stubs
- RPC library
- Network

- construct RPC request
  - marshaled args
  - some RPC handler id
- send request, wait for reply

### Server RPC handlers
- dispatcher
- RPC library
- Network

- Identify handlers to call
- Unmarshal args
- Invoke RPC handler w/ args
- Marshall reply
- Send RPC response back
- send request, wait for reply

---

**Diagram notes:**
- RPC request flows from Client to Server.
- RPC response flows from Server to Client.
- Network components involved in request and response transport.
Details of RPC library

• Which server function to call?
  – In Go RPC, it’s specified in Call(“KV.Put”, ...)

• Marshalling: format data structure into byte stream
  – Go RPC forbids channels, functions in RPC args/reply

• Binding: how does client know which server to talk to?
  – Go RPC: client supplies server host name
  – (rpcbind) A name service maps service names to some server host
RPC challenge

• What to do about failures?
  – lost packets, broken network, slow server, crashed server

• What does a failure look like to the client RPC library?
  – Client does not see a response from the server
  – Client cannot distinguish between the 2 scenarios:
    • Server has never see/processed request
    • Server has processed request, but reply is lost
RPC behavior under failure: at-least-once?

- A simple scheme to handle failure
  - RPC library waits for response for a while
  - If none arrives, re-send the request (re-establish network connection if necessary)
  - Repeat several times
  - If still no response, return an error to application client.
Perils of at-least-once semantics

```go
er1 := clt.Call("KV.Put", &PutArgs{"k", 10}, ...)
er2 := clt.Call("KV.Put", &PutArgs{"k", 20}, ...)
```

• What’s the expected value stored under “k” if err1==nil and err2== nil?
• Could it be otherwise?
Perils of at-least-once semantics

Client

Put $k=10$

Put $k=10$

Put $k=10$, reply=OK

Put $k=20$

Put $k=20$, reply=OK

Server

K=10 overwrites k=20
Perils of at-least-once

• Is at-least-once semantics ever okay?
  – If it’s ok to repeat operations, e.g. read-only operations
  – If application has its own plan for copying with duplicates
At-most-once RPC semantics

• Idea:
  – RPC library detects duplicate requests, returns previous reply instead of re-running handler

• Client uses unique ID (xid) with each request, (same xid for re-send)

• Server:

```go
if oldreply, ok := seen[xid]; ok {
    reply = oldreply
} else {
    reply = handler()
    seen[xid] = reply
}
```
Complexities in realizing at-most-once

• How to ensure XID is unique?
  – random numbers (must be big)
  – unique client-id + sequence #

• How to eventually discard old RPC replies?
  – Possible solution-1:
    • xid = unique client-id + sequent #
    • clients include x in request, indicating “seen all replies <= x”
    • server discards replies <= x
  – Possible solution-2:
    • client agrees to retry for < 5 minutes
    • server discards after 5+ minutes
Complexities in realizing at-most-once

• How to handle dup. request while original is in the middle of execution?
  – Server does not know the reply yet
  – Must not run twice
  – Solution: “pending” flag per executing RPC; wait or ignore

• What if an at-most-once server crashes and restarts?
  – If server state is in-memory, server will forget.
  – Possible solution:
    • server uses a unique number “nonce” upon each startup,
    • client obtains server-nonce upon startup, and includes it in every RPC request.
    • server rejects all requests with an old nonce
At-most-once semantics

err = clt.Call(...)  

What are the possible scenarios?

1. if err == nil  
2. if err != nil  

1. Handler is executed exactly once  
2. Handler is executed >=1 times  
3. Handler is not executed
Go RPC semantics

• Go RPC is “at-most-once”
  – Client open TCP connection
  – write request to TCP connection
  – TCP may retransmit, but server’s TCP filters out duplicates
  – No retry in Go RPC library (e.g. will NOT create another TCP connection if original one fails)
  – Go RPC returns error if it does not get a reply
    • after a TCP connection error
What about “exactly-once”? 

• If a RPC call returns error, how should the application client respond?
  – Re-transmit to the same server?
  – Re-transmit to a different server replica?
  – Applications need solutions to avoid duplicates

• Lab 3 will handle this in the context of a fault-tolerant key-value service
  – capable of handling unbounded client retransmits