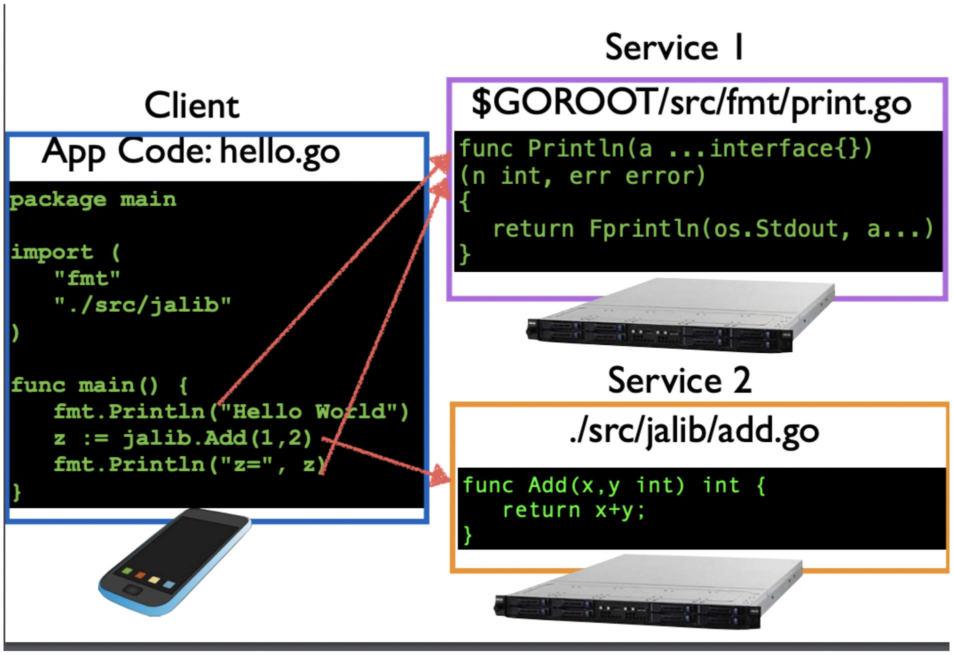
CAS CS 350
Lecture 4

MapReduce

1. Example
    a. File with #pid, age, diag, glucose level, heart rate, …
        135, 36, d1, 120, …, ….
    b. Problem: Average glucose level for all patient between 40 and 60 years old and group by diagnosis code in terms of MapReduce
        i. Map phase
            1. input: partition of file
            2. Output (require intermediate key value pair)
                a. Key: diag
                b. value: glucose level
            3. The filtering of age (age between 40 and 60) within the map phase
        ii. Data Shuffle (between map and reduce phase)
            1. The distribution of key-value pair inside map to appropriate reduce function
        iii. Reduce phase
            1. Reduce (key, iterator) → reduce (diag, [glucose 1, glucose 2, …]
            2. Diag → iterator
            3. Sum the glucose level per diagnosis
            4. Record the count for each key
            5. Divide Sum by the count to calculate the average
            6. Output → (diagnosis, average) → [(d1, avg), (d2, avg), …]
    c. What happens if there exists a skew in data (d1 has huge amounts of data while other diagnoses do not contain many data)?
        i. Ex) (d1, g1), (d1, g2), (d2, g3), (d3, g4), (d1, g5) → many diagnosis for d1
        ii. Combiner function between map phase and reduce phase that adds all the glucose values (g) for each key diagnosis (d)→ reduce intermediate amount of data
            1. d1, (322, 3)
            2. d2, (g3, 1)
            3. d3, (g4, 1)
2. RPC - Remote Procedure Calls
    a. One of the bread and butter building blocks for distributed system construction
    b. Hopefully a particular RPC infrastructure is boring once you get the basic idea and have read the docs
    c. Our goal today is to both get a handle on the idea and use

3. The idea: libs as services

    a. 

```
package main

import (
    "fmt"
    "./src/jalib"
)

func main() {
    fmt.Println("Hello World")
    z := jalib.Add(1,2)
    fmt.Println("z=", z)
}
```

```
func Println(a ...interface{})
(n int, err error)
{
   return Fprintln(os.Stdout, a...)
}
```

```
func Add(x,y int) int {
   return x+y;
}
```

    b. Functions as services:

        i. Functions do not have to be stored in the same machine

        ii. Different applications/users can use the same function

4. Remote Procedure Call (RPC)

    a. Observations:

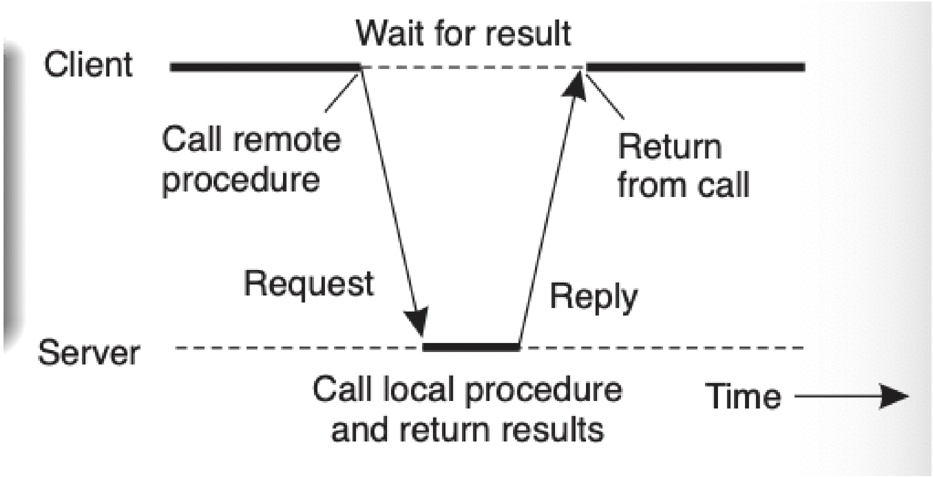
        i. Application developers are familiar with simple procedure model

        ii. Well-engineered procedures operate in isolation (black box)

        iii. There is no fundamental reason not to execute procedures on separate machine

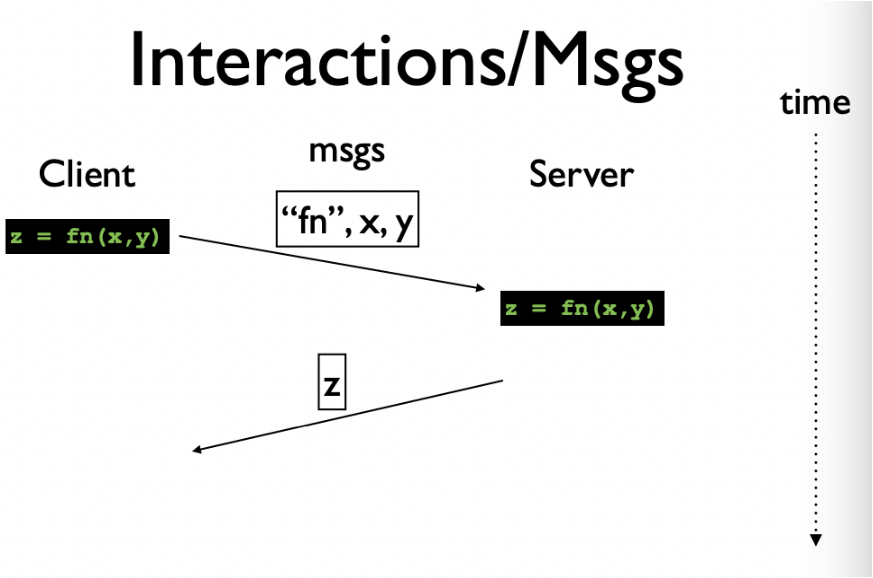
    b. Conclusion

        i. Communication between caller & callee can be hidden by using procedure-call mechanism

        ii.

5. Interactions/Msgs



Interactions/Msgs

Client

msgs

Server

time

`z = fn(x,y)`

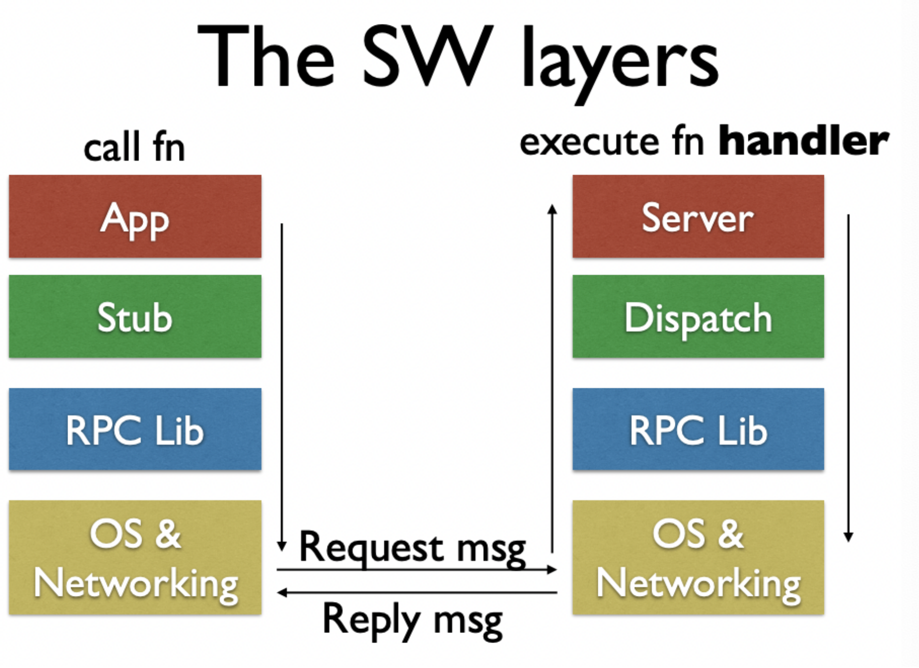
“fn”, x, y

`z = fn(x,y)`

z

a.

6. The SW layers



The SW layers

call fn

execute fn **handler**

App

Stub

RPC Lib

OS & Networking

Server

Dispatch

RPC Lib

OS & Networking

Request msg

Reply msg

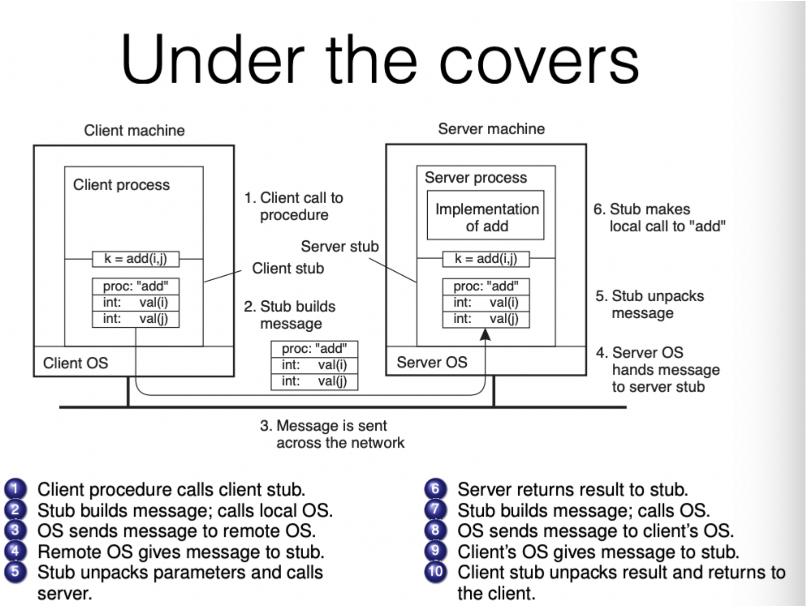
a.

b. Program initiates actual RPC

c. The application using RPC communicates and request the RPC

d. How to do this: take an object → break into bytes → send to RPC library → send to operating system → send it to other server and repeat the process (reconstruct the object → … → send to application on the server side)

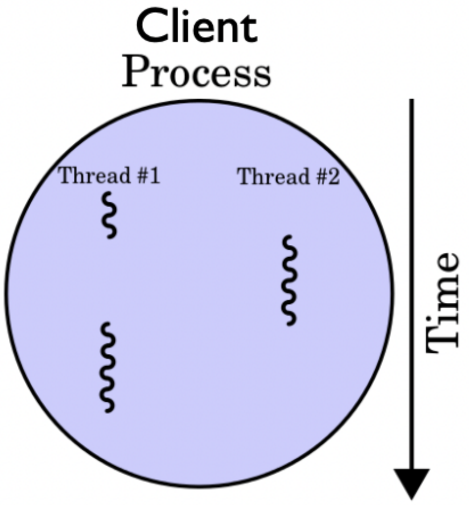
7. Under the covers

a. 

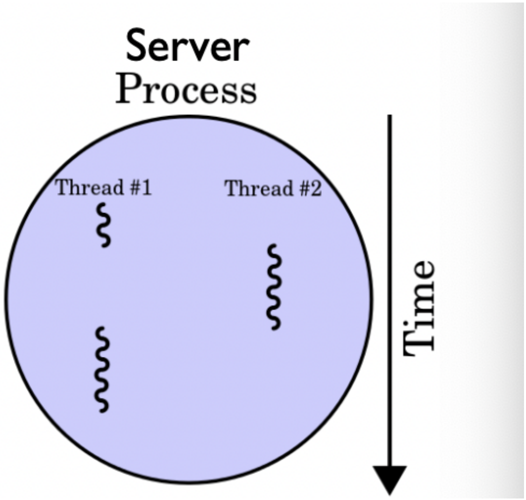
Under the covers

1. Client procedure calls client stub.
2. Stub builds message; calls local OS.
3. OS sends message to remote OS.
4. Remote OS gives message to stub.
5. Stub unpacks parameters and calls server.
6. Server returns result to stub.
7. Stub builds message; calls OS.
8. OS sends message to client's OS.
9. Client's OS gives message to stub.
10. Client stub unpacks result and returns to the client.

8. Threads

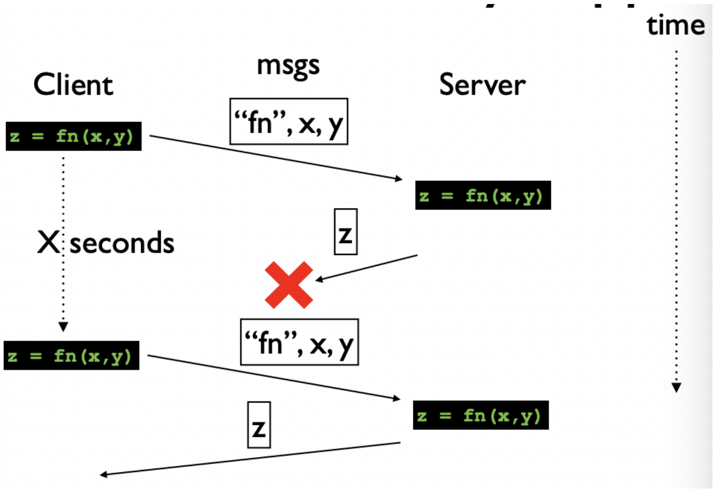
a. 

b. Might have multiple threads all of which could concurrently be making rpc's to one more servers



c.
d. Rpc handlers might take a long time - often use threads to execute many rpcs concurrently (thread per handler execution)

9. Failures?
    a. Lost packet
    b. Broken network
    c. Slow server
    d. Crashed server
    e. From the client's perspective, failures typically mean that client is waiting for a reply that will never come
10. If (no reply in X seconds) then?
    a. At least once failure model

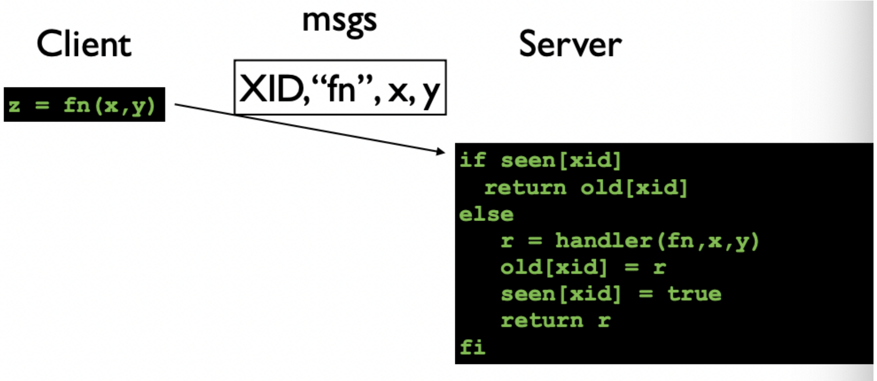


i.
ii. Regardless of failures execute the rpc at least once
iii. While true{

```
While true{
        Send request
        Wait x seconds for reply
        If reply return
}
```

        iv. As long as eventually some or something fixes the problem (eg. robot server, fix network), then this will always work
        v. It does work for read-only operations
        vi. Or you have a strategy for duplicates (which later labs will require)

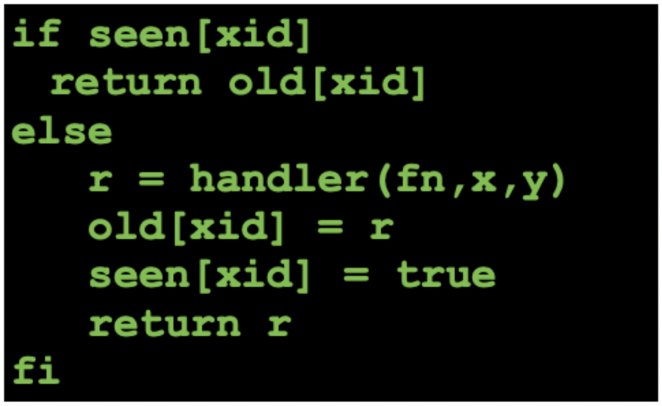
    b. At most once
        i. Server detects duplicates and not execute handler
        ii. Easy way to detect duplicates
            1. RPC id
                a.



```
z = fn(x,y)
```

```
if seen[xid]
  return old[xid]
else
   r = handler(fn,x,y)
   old[xid] = r
   seen[xid] = true
   return r
fi
```

                b.



```
if seen[xid]
  return old[xid]
else
   r = handler(fn,x,y)
   old[xid] = r
   seen[xid] = true
   return r
fi
```

                c. Introduce a unique id per-RPC invocation and some storage → if time runs out, and tries again, it uses the same id as the old id
                d. Server side: Server has to maintain all these ids (which can lead to running out of memory)
                e. This works but there are some issues
                f. How do we delete things from old and seen?
                    i. Get an ack from the client for XID for which it has received responses