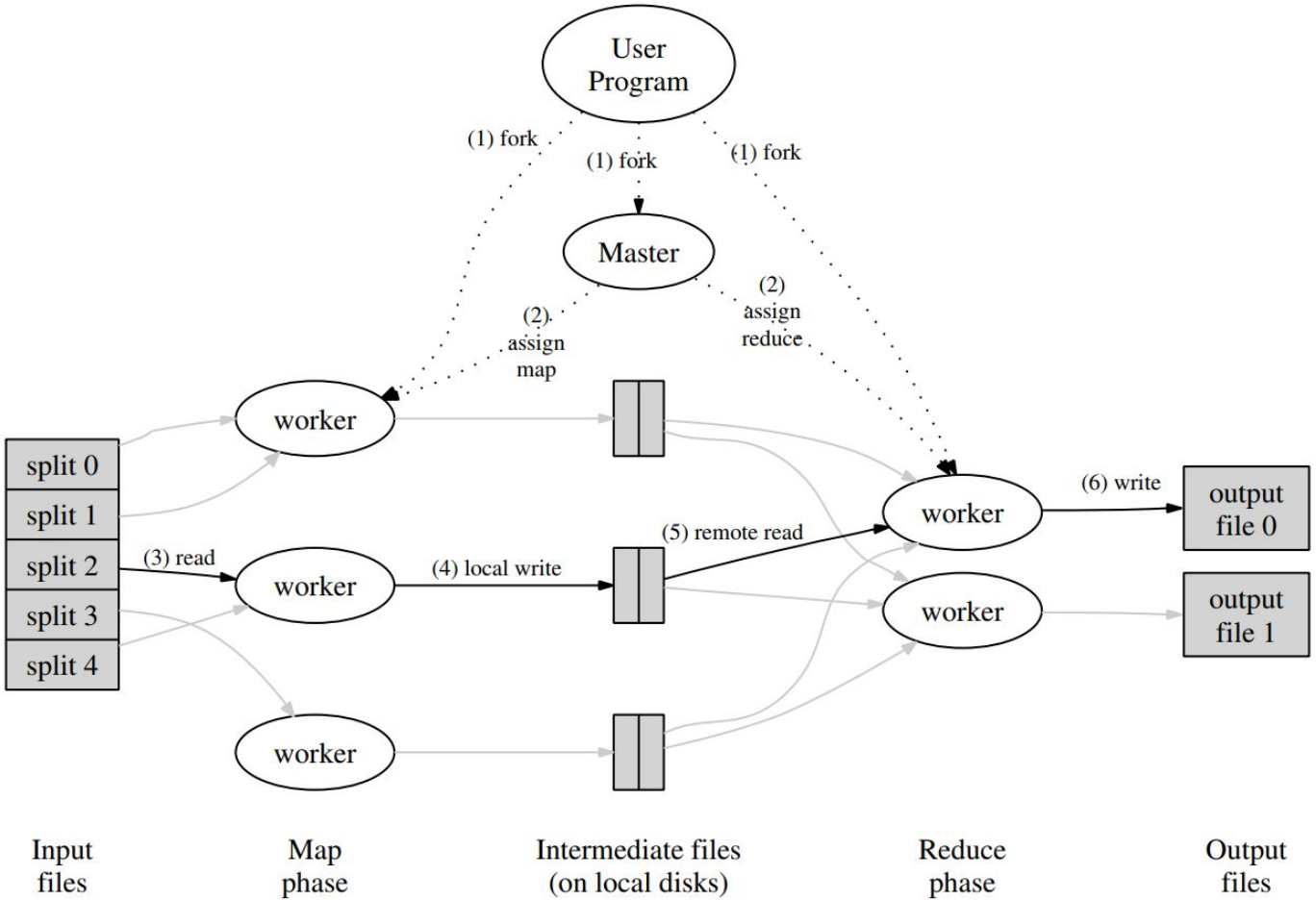
Ryan Stutsman Office Hours Friday 15:30 to 17:00 in MEB 3436

Candy Office Hours Wednesday 15:00 to 16:30 in MEB 3115

cs6963.slack.com

Prep for next time:
Have Go Tour done; especially spend some time looking at concurrency module
Watch at least for first 4.5 minutes of the Raft video.
Make sure you requested cs6963 group access in gitlab.

Get going on Lab 1; lots of moving parts, takes a bit of playing to get rolling.


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

# RPC “Transparency”

Client:

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

Server:

```
fn(x, y) {
  // Compute.
  return x + y
}
```

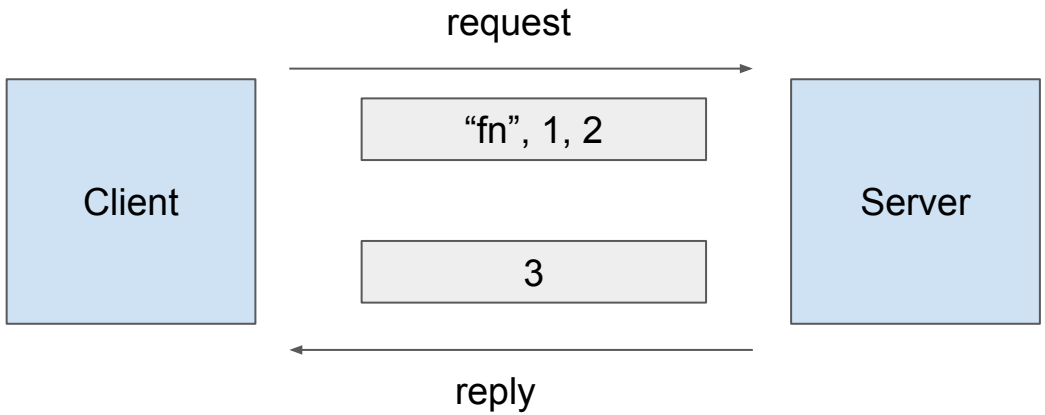
# RPC “Transparency”

Client:

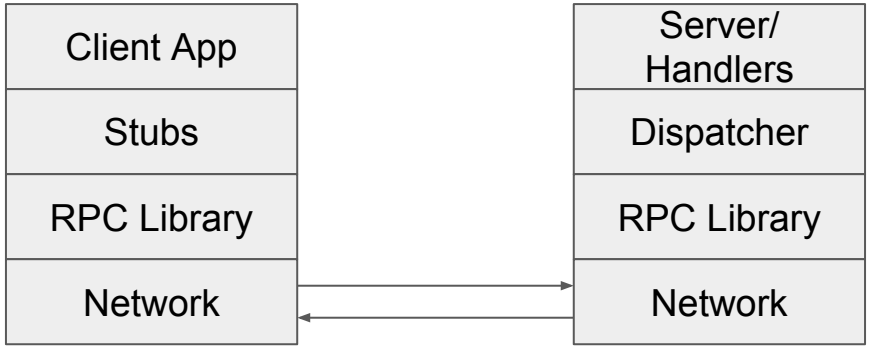
```
z = fn(x, y)
```

Server:

```
fn(x, y) {
  // Compute.
  return x + y
}
```



# RPC Overview



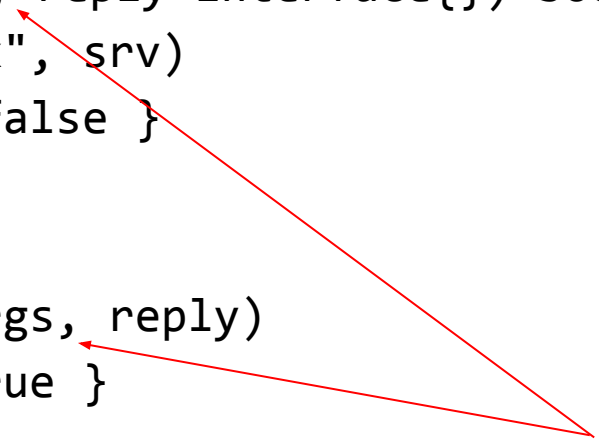
RPC Library

# Using the Go RPC Library (from Lab 1 code)

```
func call(srv string, rpcname string,
          args interface{}, reply interface{}) bool {
  c, errx := rpc.Dial("unix", srv)
  if errx != nil { return false }
  defer c.Close()

  err := c.Call(rpcname, args, reply)
  if err == nil { return true }

  fmt.Println(err)
  return false
}
```



Go RPC
Supplies
the magic here.
(“Marshaling”)

Stubs

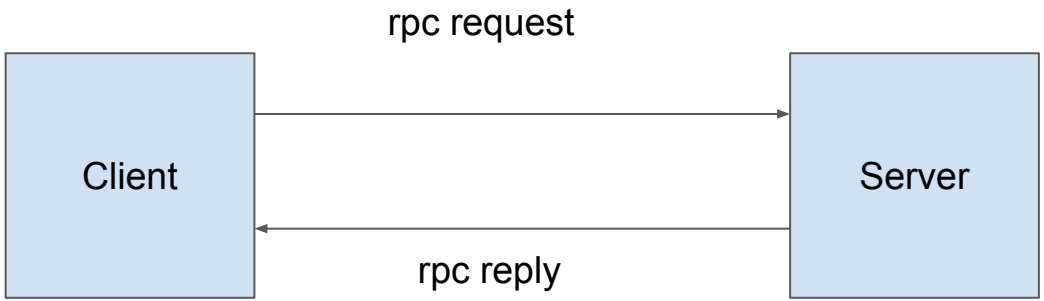
# RPC “Glue” Code (Lab 1 Worker)

```
type RegisterArgs struct { Worker string }
// Tell the master we exist and ready to work
func (wk *Worker) register(master string) {
  args := new(RegisterArgs)
  args.Worker = wk.name
  ok := call(master, "Master.Register", args, new(struct{}))
  if ok == false {
    fmt.Printf("Register: RPC %s register error\n", master)
  }
}
```

```
// Register is called by workers via RPC after they have started
// up to report that they are ready to receive tasks.
func (mr *Master) Register(args *RegisterArgs, _ *struct{}) error
{
  mr.Lock()
  defer mr.Unlock()
  debug("Register: worker %s\n", args.Worker)
  mr.workers = append(mr.workers, args.Worker)
  go func() { mr.registerChannel <- args.Worker }()
  return nil
}
```

# RPC Scenario: bank withdrawal

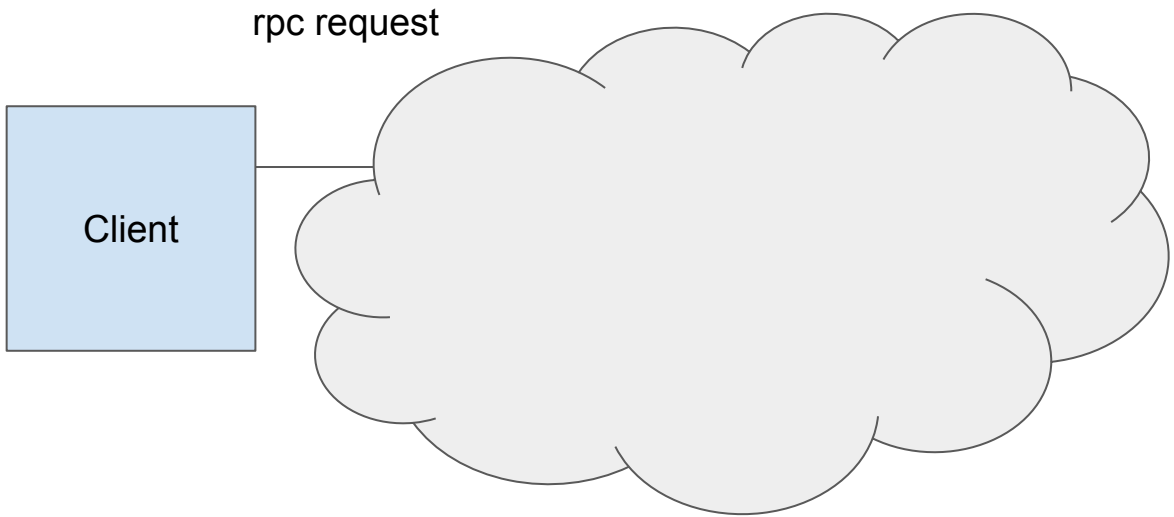
deduct(amount)



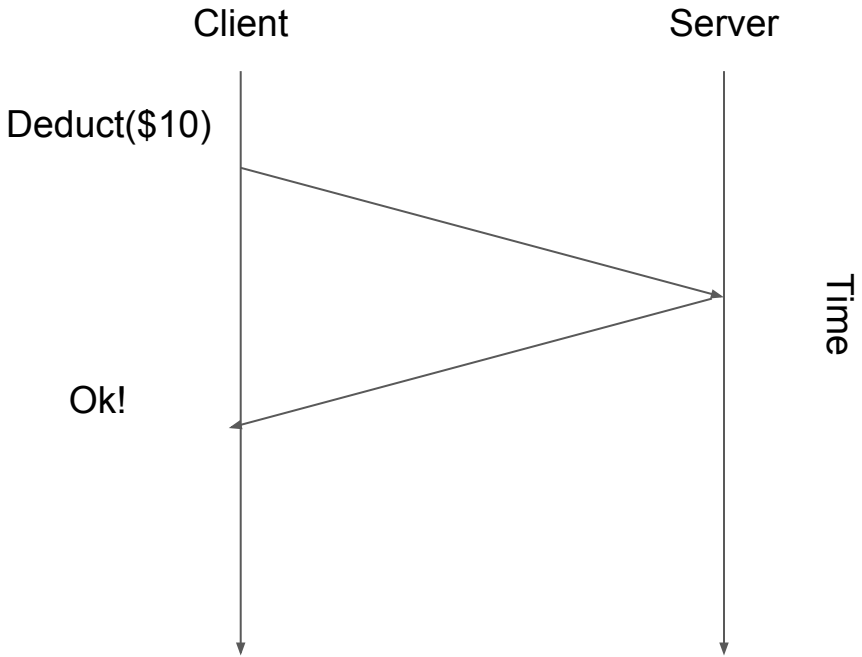
# RPC Failure?

deduct(amount)

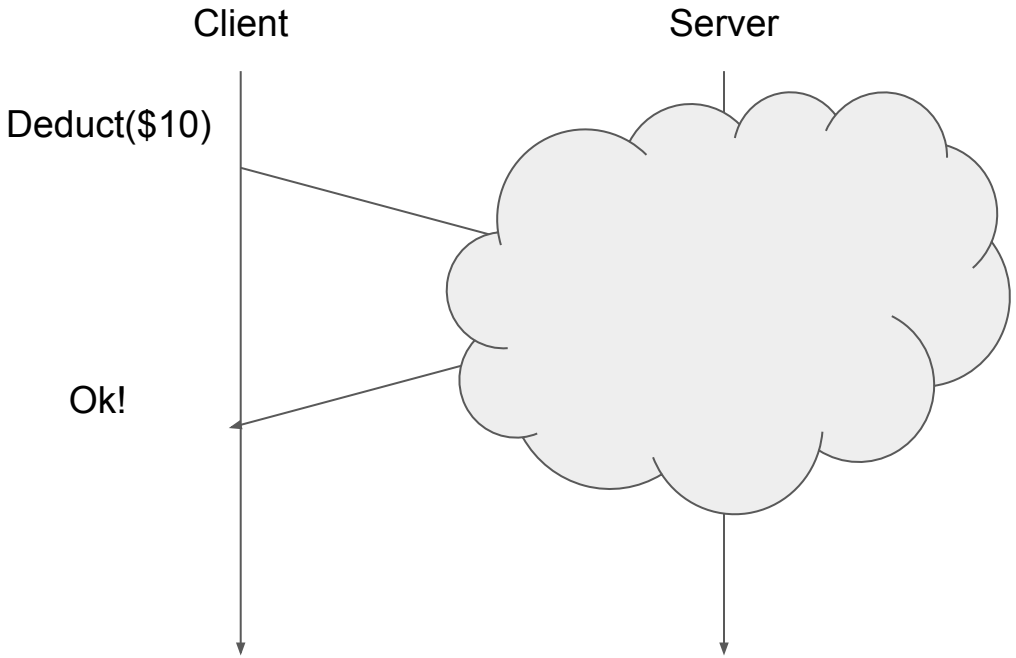
What if we get no reply?



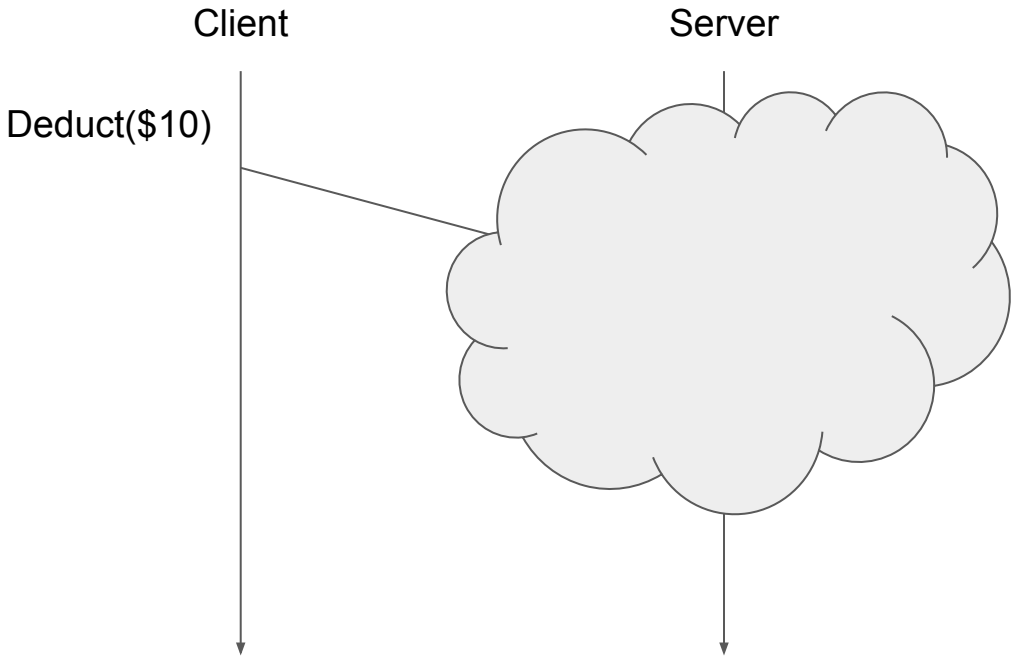
# At-least Once Semantics



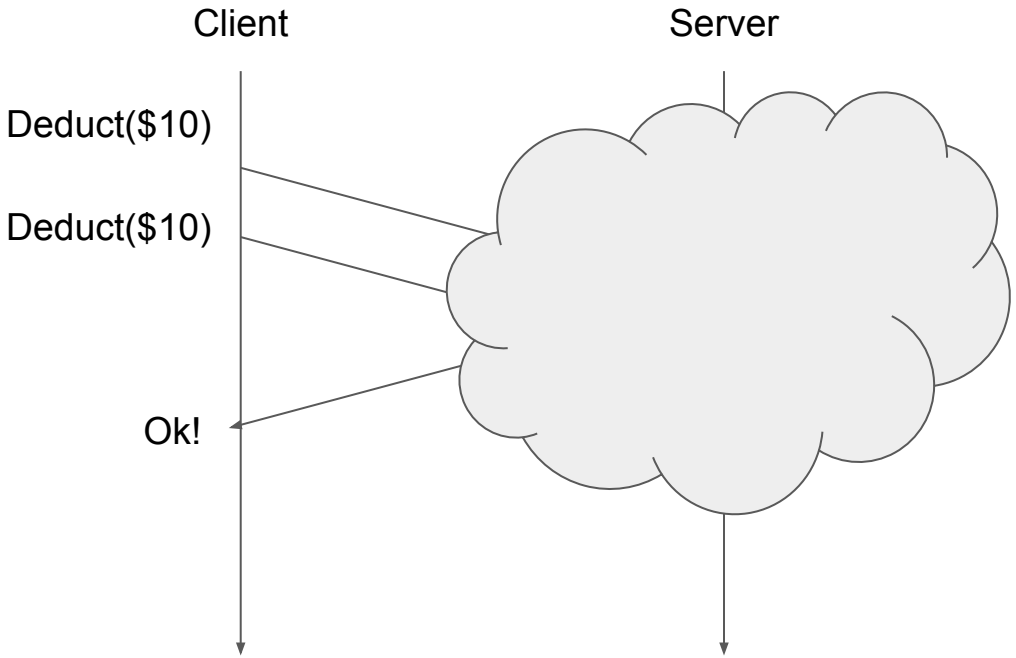
# At-least Once Semantics



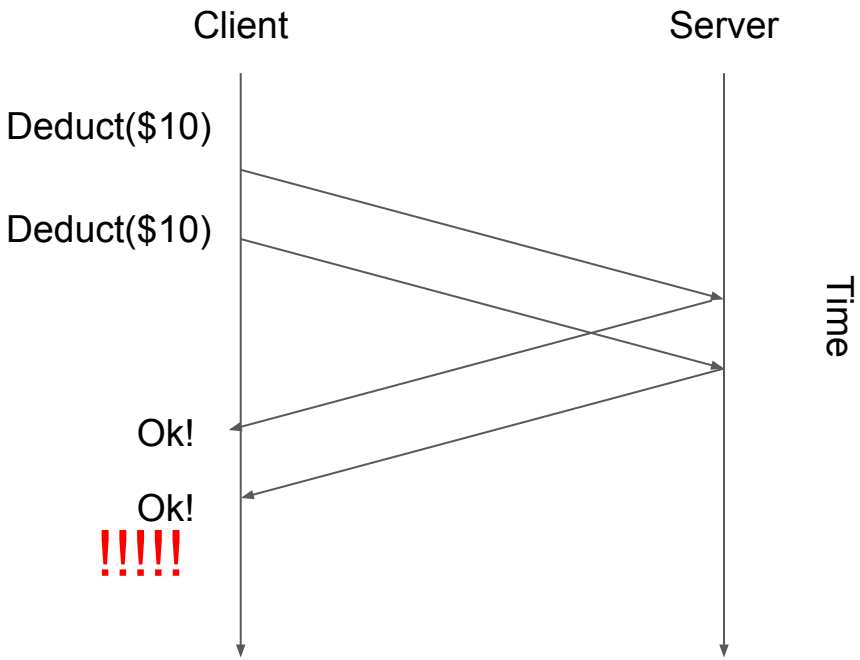
# At-least Once Semantics



# At-least Once Semantics



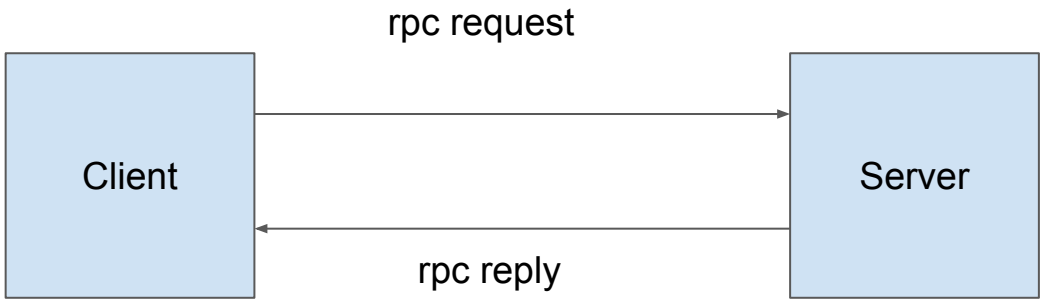
# At-least Once Semantics



# Scenario #2: simple key-value store

get(k) returns last put v or nil

put(k, v) sets k to v

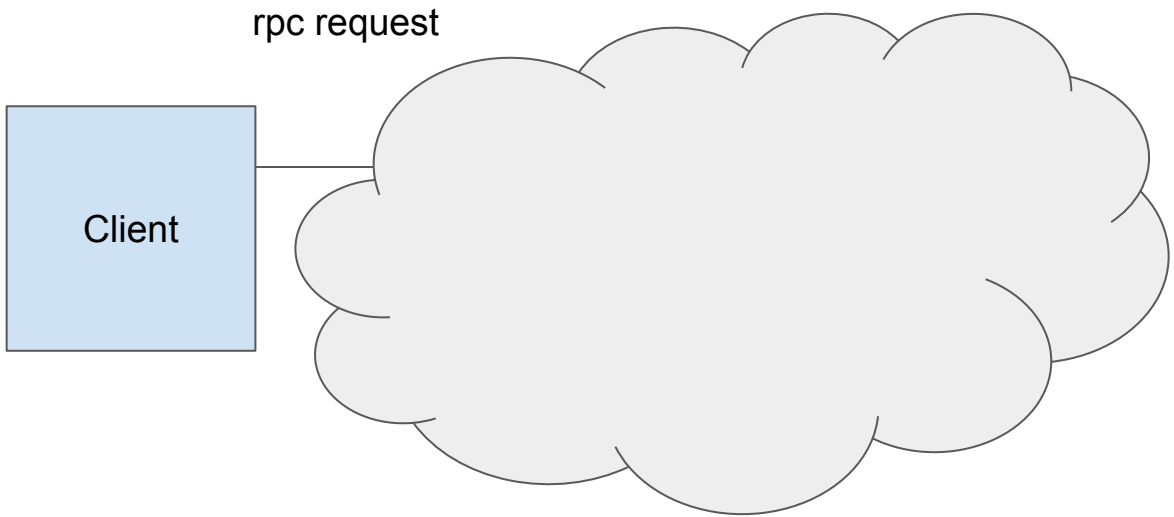


# RPC Failure?

get(k) returns last put v or nil

put(k, v) sets k to v

What if we get no reply?



# What is the final value of k with at-least once?

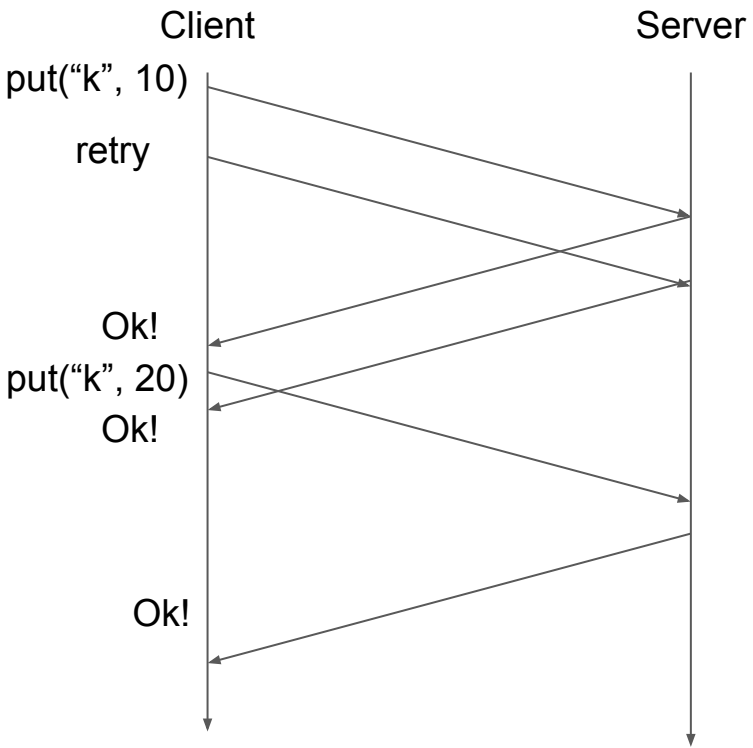
put(“k”, 10)

put(“k”, 20)

# What is the final value of k with at-least once?

put(“k”, 10)

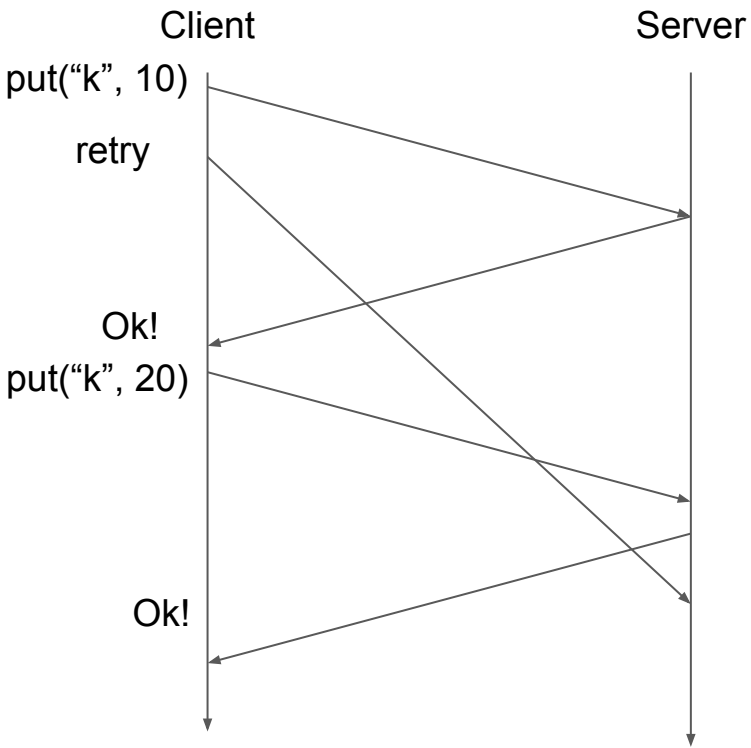
put(“k”, 20)



# What is the final value of k with at-least once?

put(“k”, 10)

put(“k”, 20)



# Is at-least once ever ok?

# Is at-least once ever ok?

get(k)? Ok.

put(k, v)? Not so good.

# Is at-least once ever ok?

get(k)? Ok.

put(k, v)? Not so good.

Yes. If it’s ok to repeat operations. “Read-only” ops are a good example.

Yes. If application has a plan for eliminating duplicates (“idempotent”…).

- e.g. delete_order(8128) where order number 8128 is never reused.

# Better Behavior?

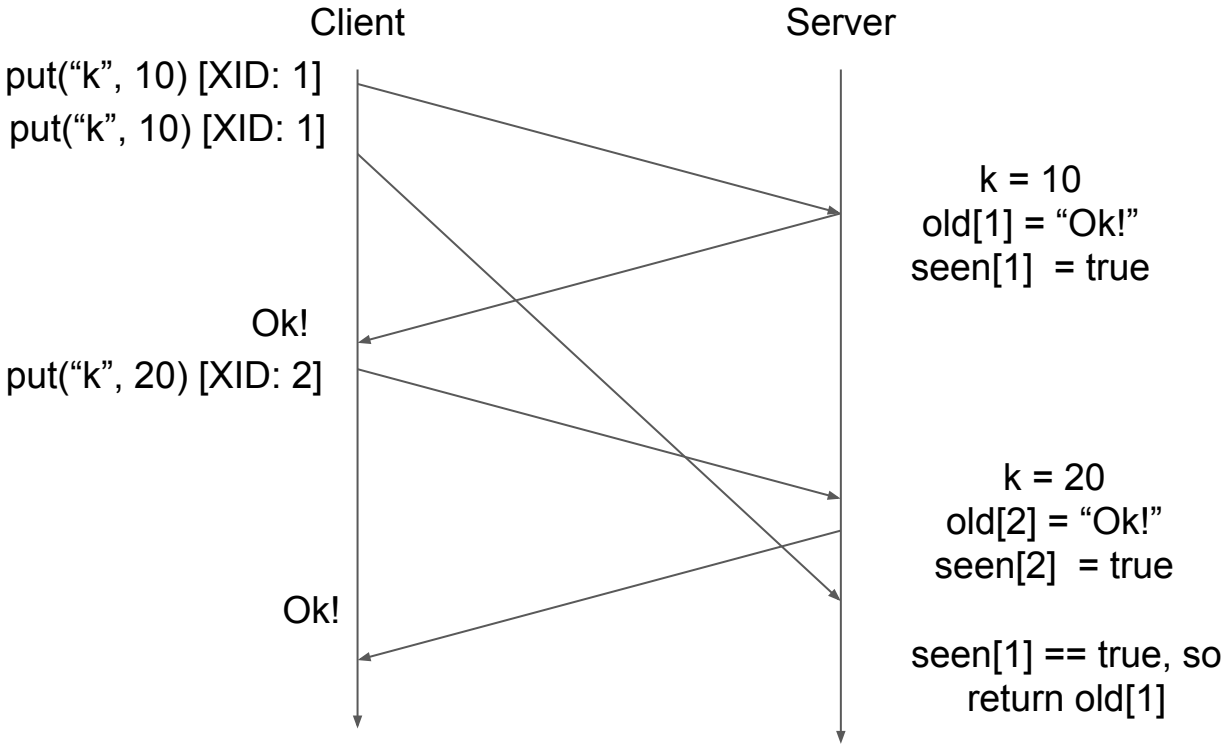
# At-most Once Server Dispatch

```
if seen[xid]:
   r = old[xid]
 else
   r = handler()
   old[xid] = r
   seen[xid] = true
```

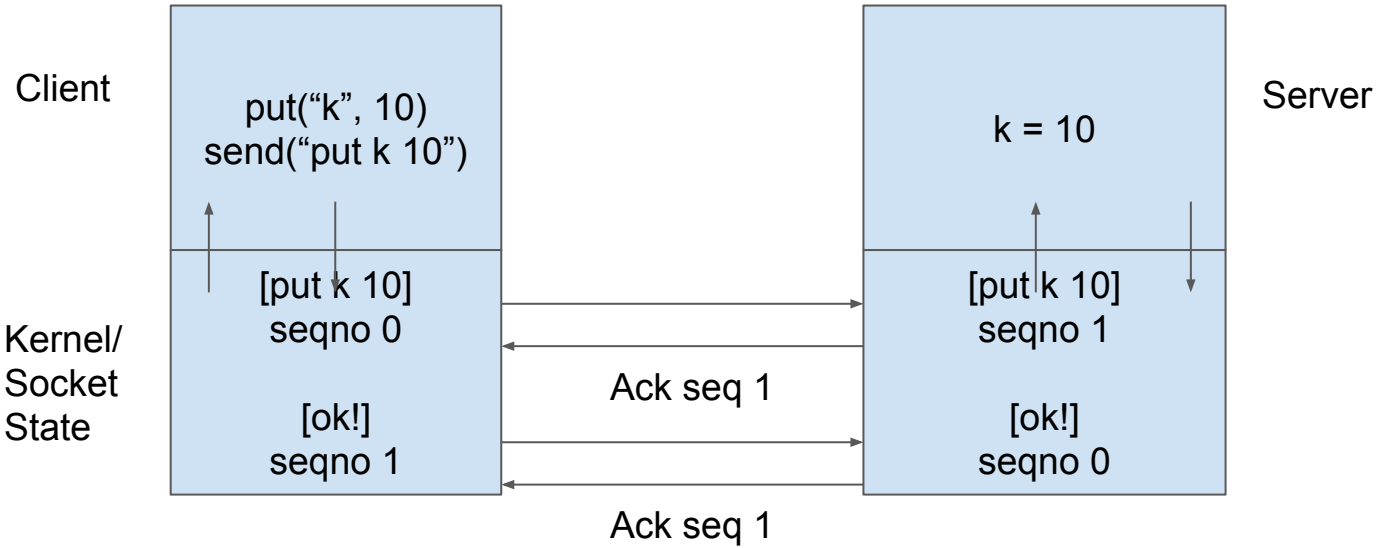
# What is the final value of k with at-most once?

put(“k”, 10)

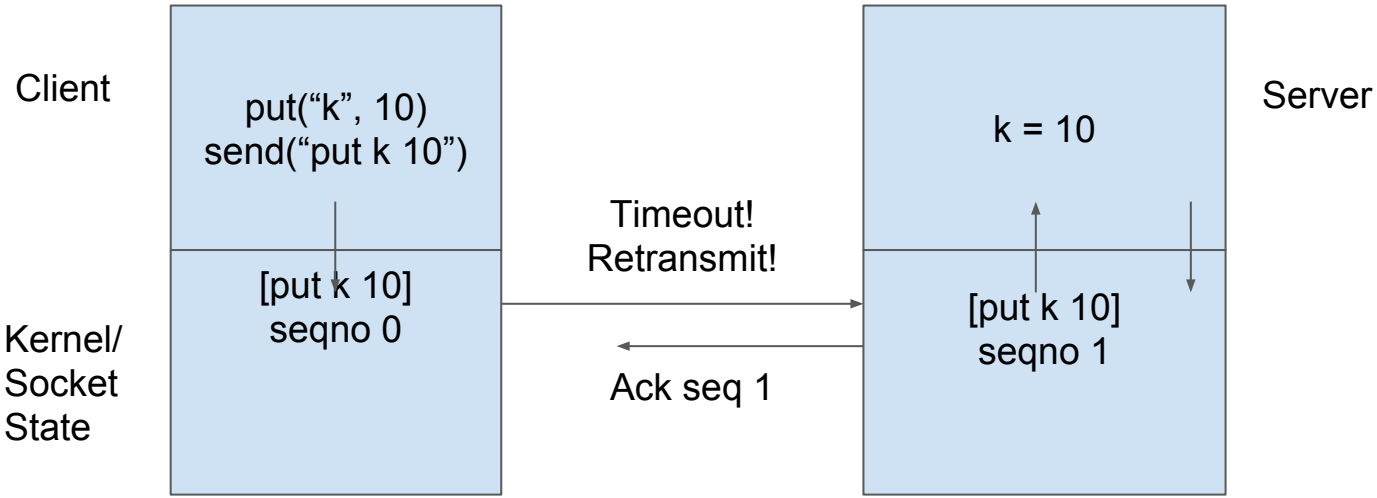
put(“k”, 20)



# Go RPC and TCP



# Go RPC and TCP




“Eliminates” duplicates and request reordering.
Also, “eliminates” the need to cache response at application.

# All done?

Does this fix all of the issues in Lab 1?

Consider DoTask()

doTask(‘worker-1’, map, ‘input-file.txt-1’, 3 reducers)

… Worker-1 starts running…

… TCP connection times out …

doTask(‘extra-worker’, map, ‘input-file.txt-1’, 3 reducers)

… Extra-worker starts running ...

# Threads

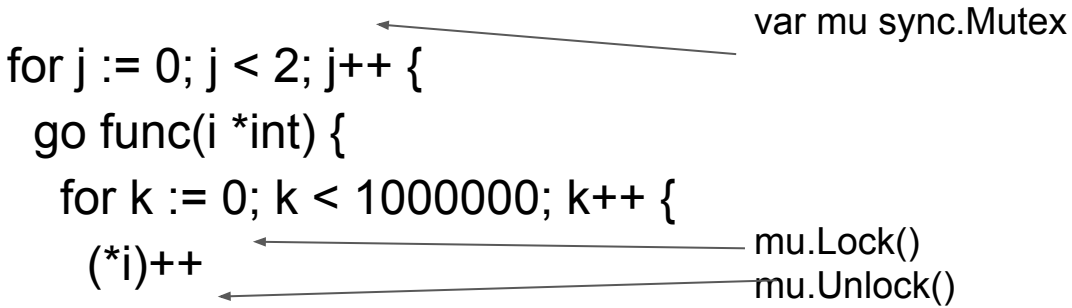
```
done := make(chan int)
 i := 0
 go func(i *int) {
   for k := 0; k < 1000000; k++ {
     (*i)++
   }
   done <- 1
 }(&i)
 <-done
 fmt.Printf("Final result is %v\n", i)
```

# Threads

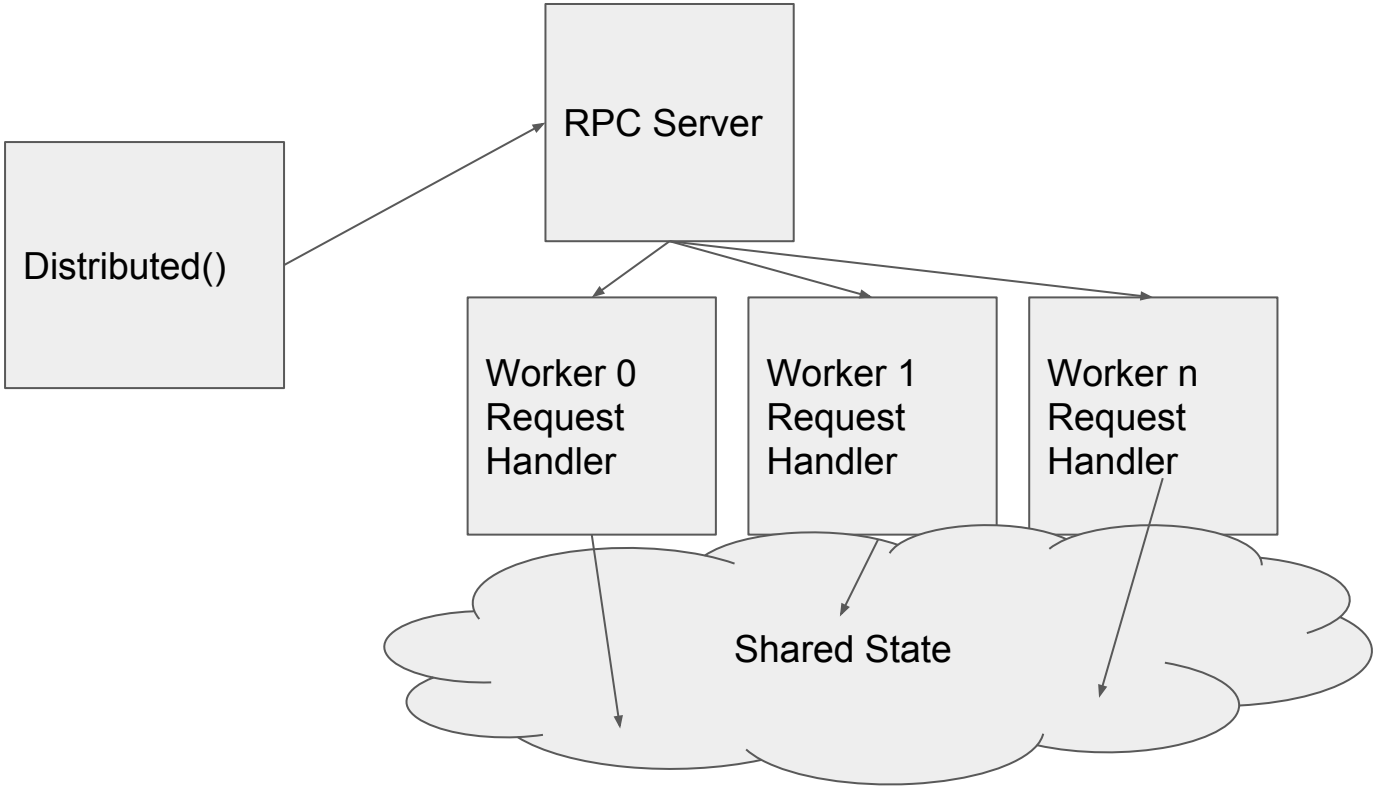
```
for j := 0; j < 2; j++ {
  go func(i *int) {
    for k := 0; k < 1000000; k++ {
      (*i)++
    }
    done <- 1
  }(&i)
}
<-done
<-done
fmt.Printf("Final result is %v\n", i)
```

Output: 1021646

# Threads


var mu sync.Mutex
for j := 0; j < 2; j++ {
go func(i *int) {
for k := 0; k < 1000000; k++ {
mu.Lock()
(*i)++
mu.Unlock()
}
done <- 1
}(&i)
}
<-done
<-done
fmt.Printf("Final result is %v\n", i)

# MapReduce Master



```
func f() int { return 1 }

var x int
done := false
go func() {
    x = f()
    done = true
}

for done == false {
}
```

```
func f() int { return 1 }

var x int
done := make(chan bool)
go func() {
    x = f()
    done <- true
}

<- done
```

# Idempotence?

**Idempotence** is a funky word that often hooks people. **Idempotence** is sometimes a confusing concept, at least from the academic definition. From a RESTful service standpoint, for an operation (or service call) to be **idempotent**, clients can make that same call repeatedly while producing the same result.

What is Idempotency? - REST API Tutorial
www.restapitutorial.com/lessons/idempotency.html

*About this result • Feedback*

# Idempotence: Copied from “the textbook”

An idempotent operation is an operation that can be performed repeatedly with the same effect as if it had been performed exactly once.

For example, an operation to add an element to a set is an idempotent operation because it will always have the same effect on the set each time it is performed, whereas an operation to append an item to a sequence is not an idempotent operation because it extends the sequence each time it is performed. A server whose operations are all idempotent need not take special measures to avoid executing its operations more than once.

# Computer Science Meaning (Wikipedia)

The term **idempotent** is used to describe an operation that will produce the same results if executed once or multiple times.[7] This may have a different meaning depending on the context in which it is applied. In the case of methods or subroutine calls with side effects, for instance, it means that the modified state remains the same after the first call.

This is a very useful property in many situations, as it means that an operation can be repeated or retried as often as necessary without causing unintended effects. With non-idempotent operations, the algorithm may have to keep track of whether the operation was already performed or not.

# Examples (from Wikipedia)

A function looking up a customer's name and address in a database is typically idempotent, since this will not cause the database to change.

Changing a customer's address is idempotent, because the final address will be the same no matter how many times it is submitted.

Placing an order for a car for the customer is typically not idempotent, since running the call several times will lead to several orders.

Canceling an order is idempotent, because the order remains canceled no matter how many requests are made.

Idempotence isn’t closed under composition.

- That is, combining two idempotent operations doesn’t always yield an idempotent operation.

Idempotence is defined on two back-to-back runs of the *same* exact operation.

- Message reordering?
- Concurrency?