Distributed Systems

Spring Semester 2020

Lecture 3: RPC and Threads

John Liagouris
liagos@bu.edu

Why is MapReduce good?

- Lots of parallelism
  - maps have each worker doing a independent access to a file and attendant processing
  - reduce tasks are also independent and parallel
- Transparent straggler and failure handling
  - Although master failures are not really discussed

When doesn't work well?

- Small data
  - Overhead of scaling out to many machines
- Small updates in large datasets
  - Well-suited for bulk processing
- Iterative computations
  - Multiple jobs
- Computations where mappers and reducers need to choose input

RPC — Remote Procedure Calls

- One of the Bread and Butter building blocks for distributed system construction.

- Hopefully a particular RPC infrastructure is boring once you get the basic idea and have read the docs.

- Our goal today is to both get a handle on the idea and use.

    - Look at how RPC's work to get into fundamental DS issues.

The idea : libs as services

Client

App Code: hello.go

```
package main

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

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

Service 1

$GOROOT/src/fmt/print.go

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

Service 2

./src/jalib/add.go

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

The idea : libraries services

Client

App Code: hello.go

```
package main

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

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

Service 1

$GOROOT/src/fmt/print.go

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

Service 2

./src/jalib/add.go

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

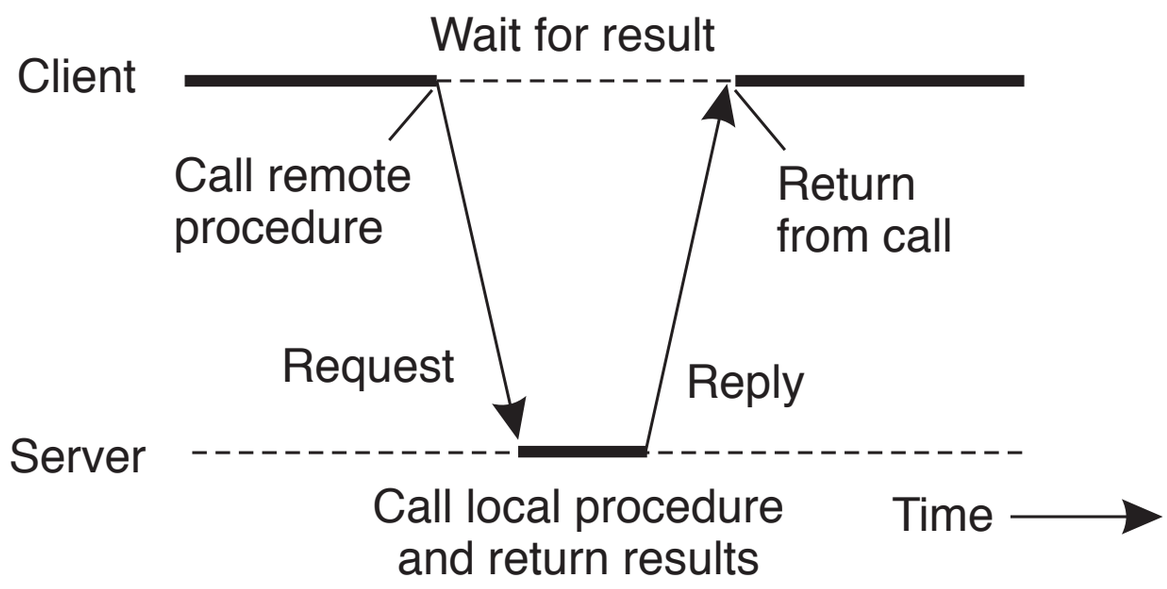
Remote Procedure Call (RPC)

Observations

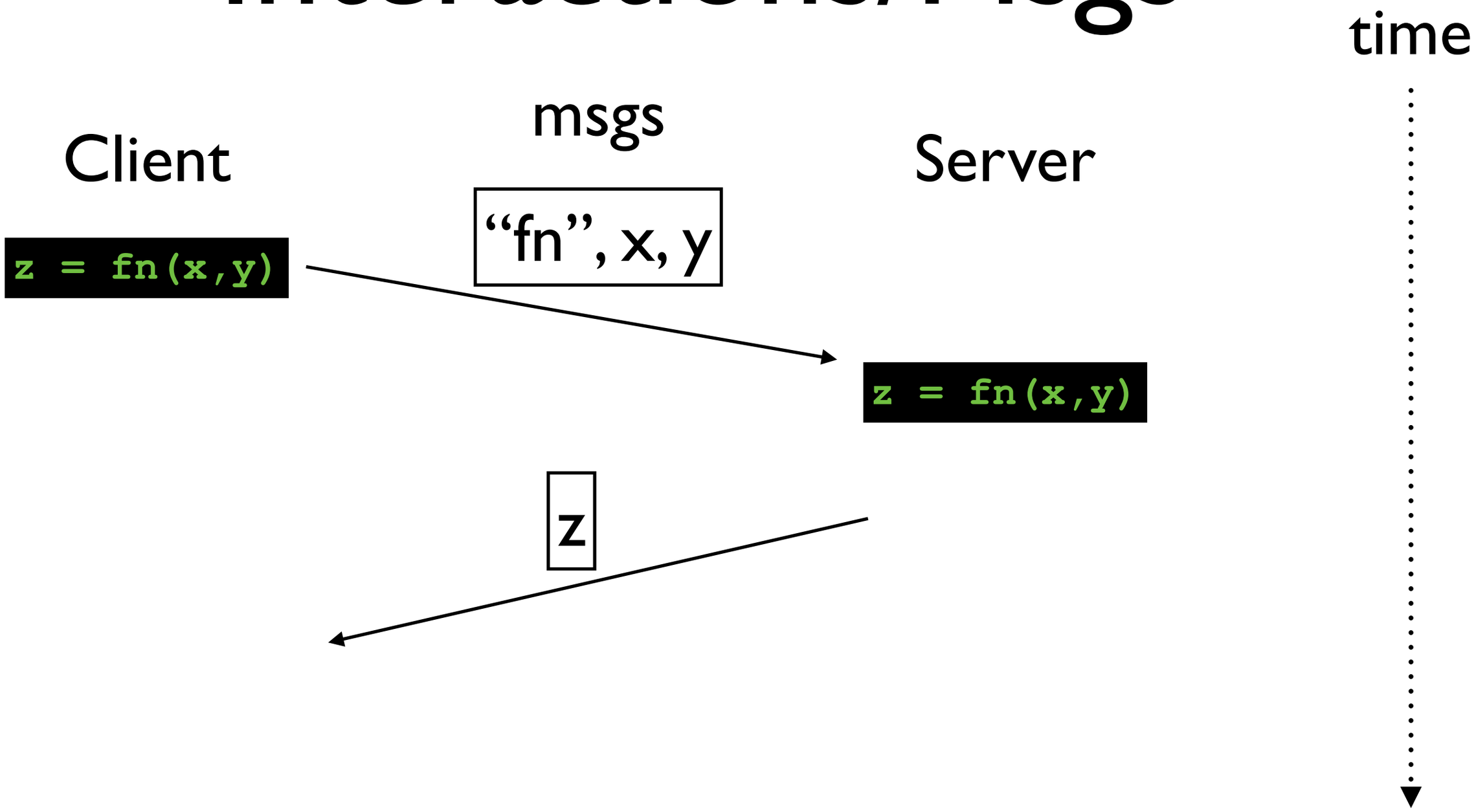
- Application developers are familiar with simple procedure model
- Well-engineered procedures operate in isolation (black box)
- There is no fundamental reason not to execute procedures on separate machine

Conclusion

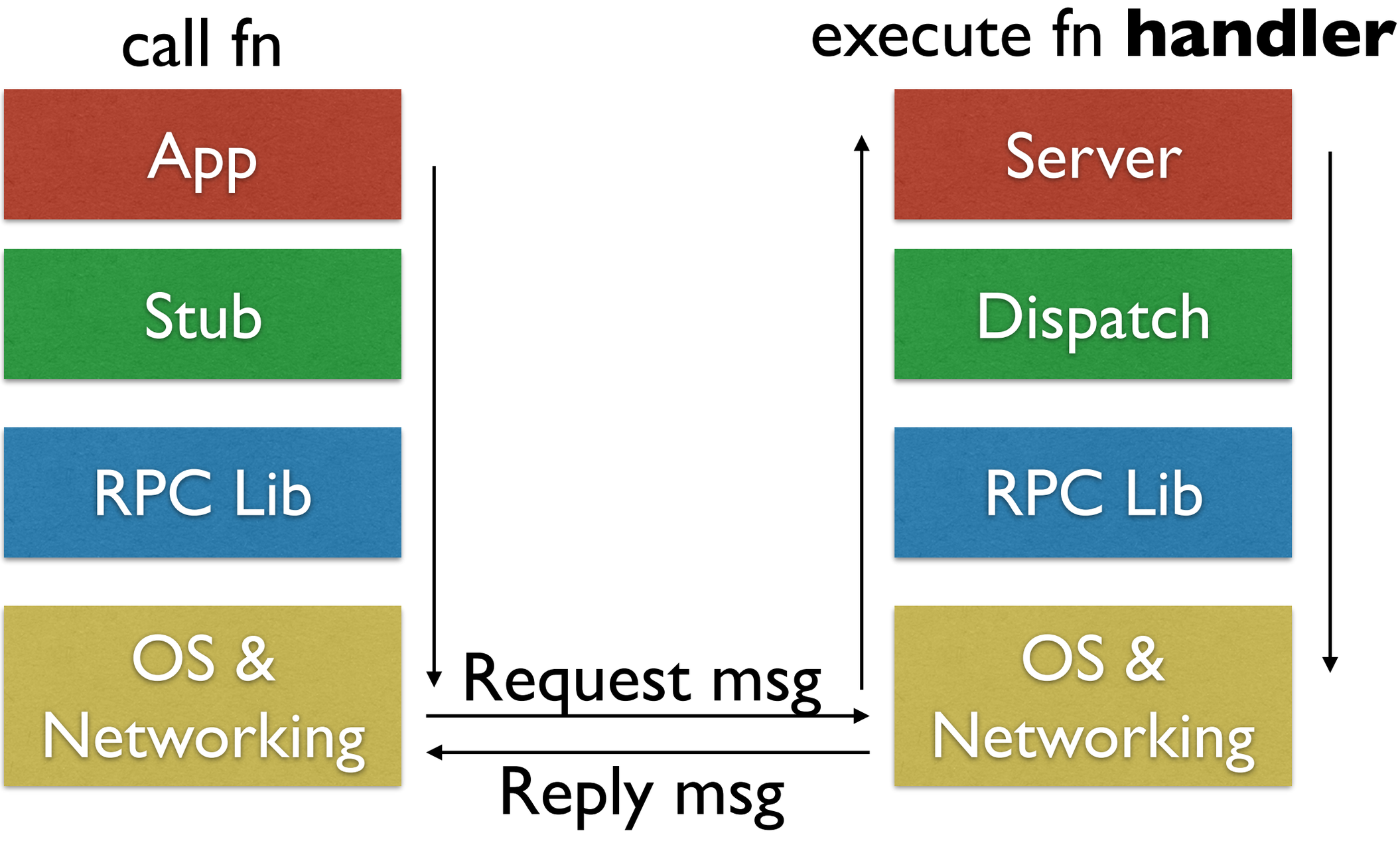
Communication between caller & callee can be hidden by using procedure-call mechanism.



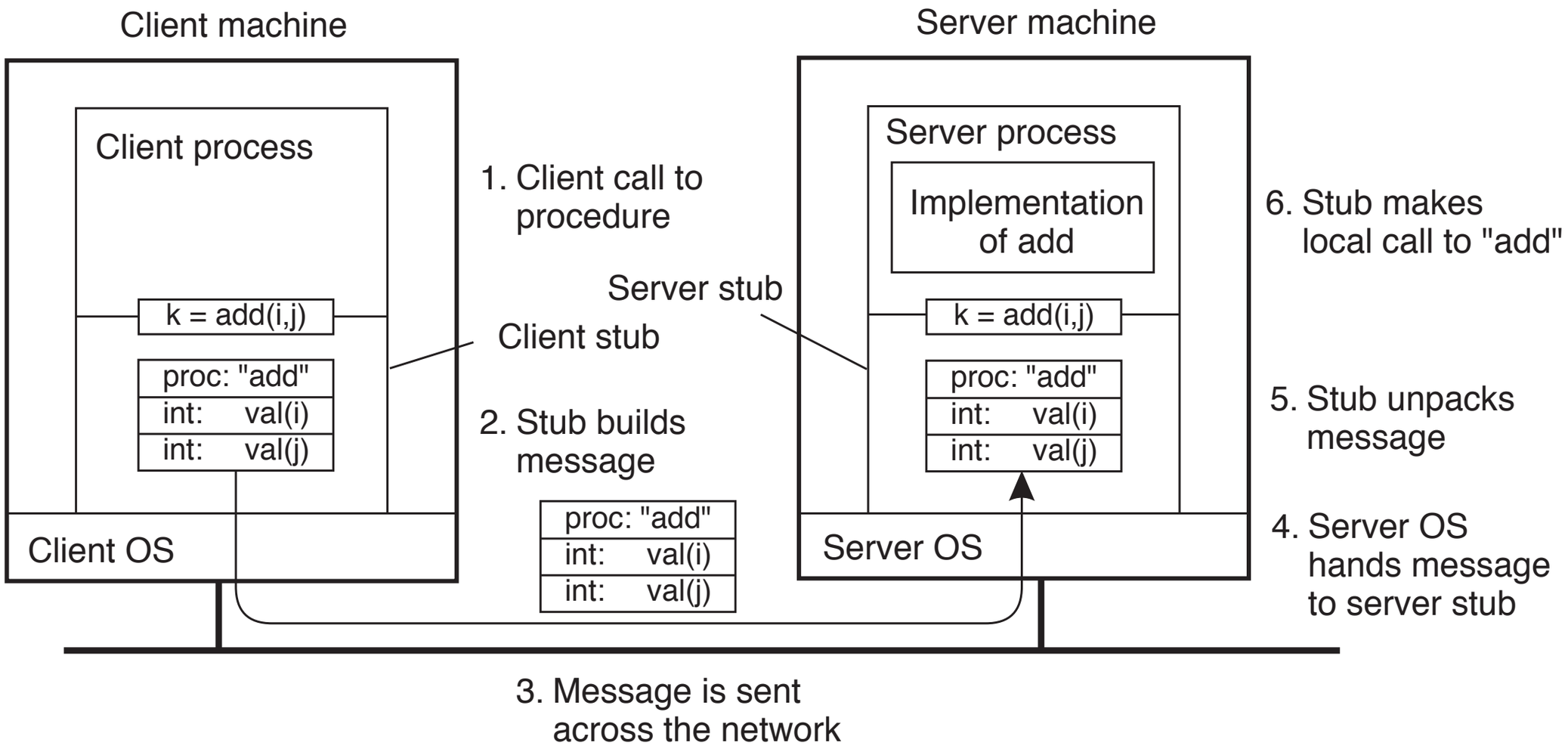
Interactions/Msgs



The SW layers

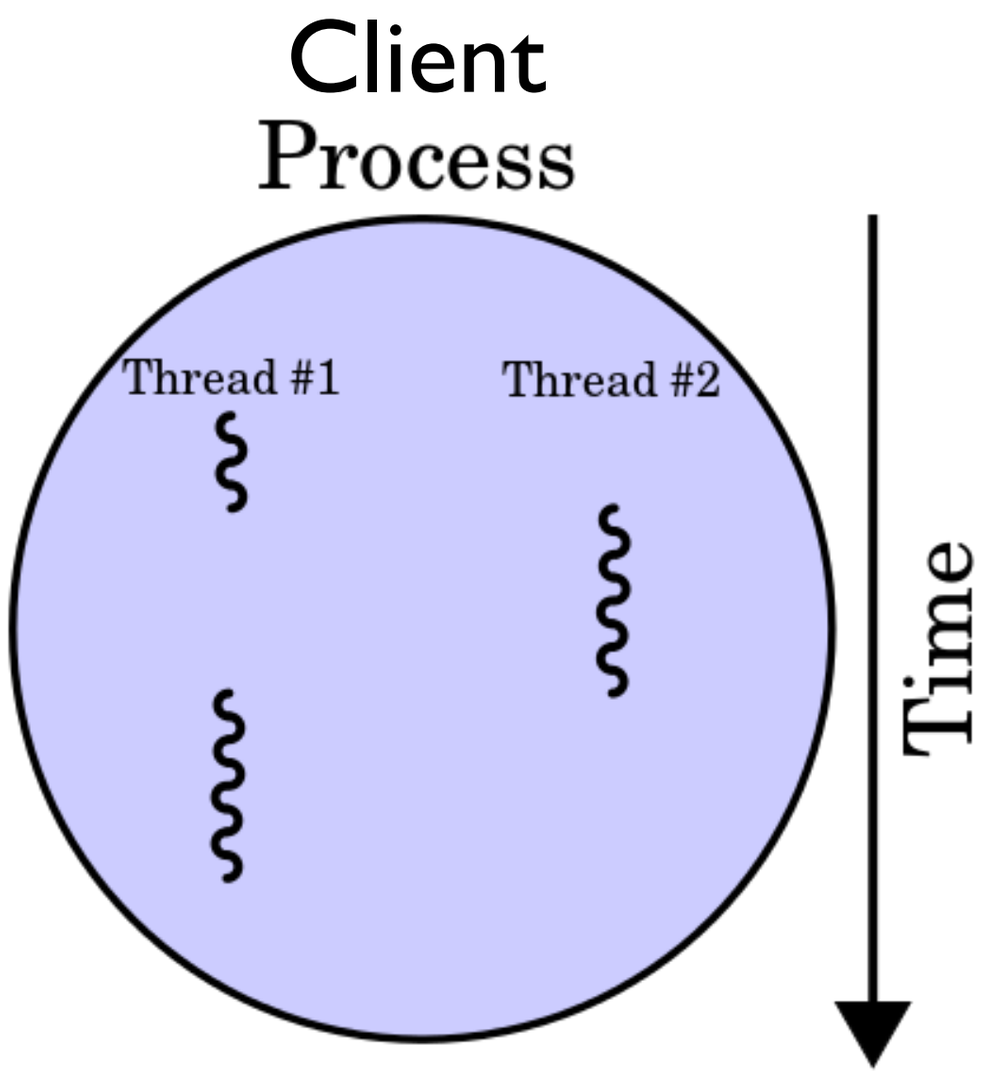


Under the covers

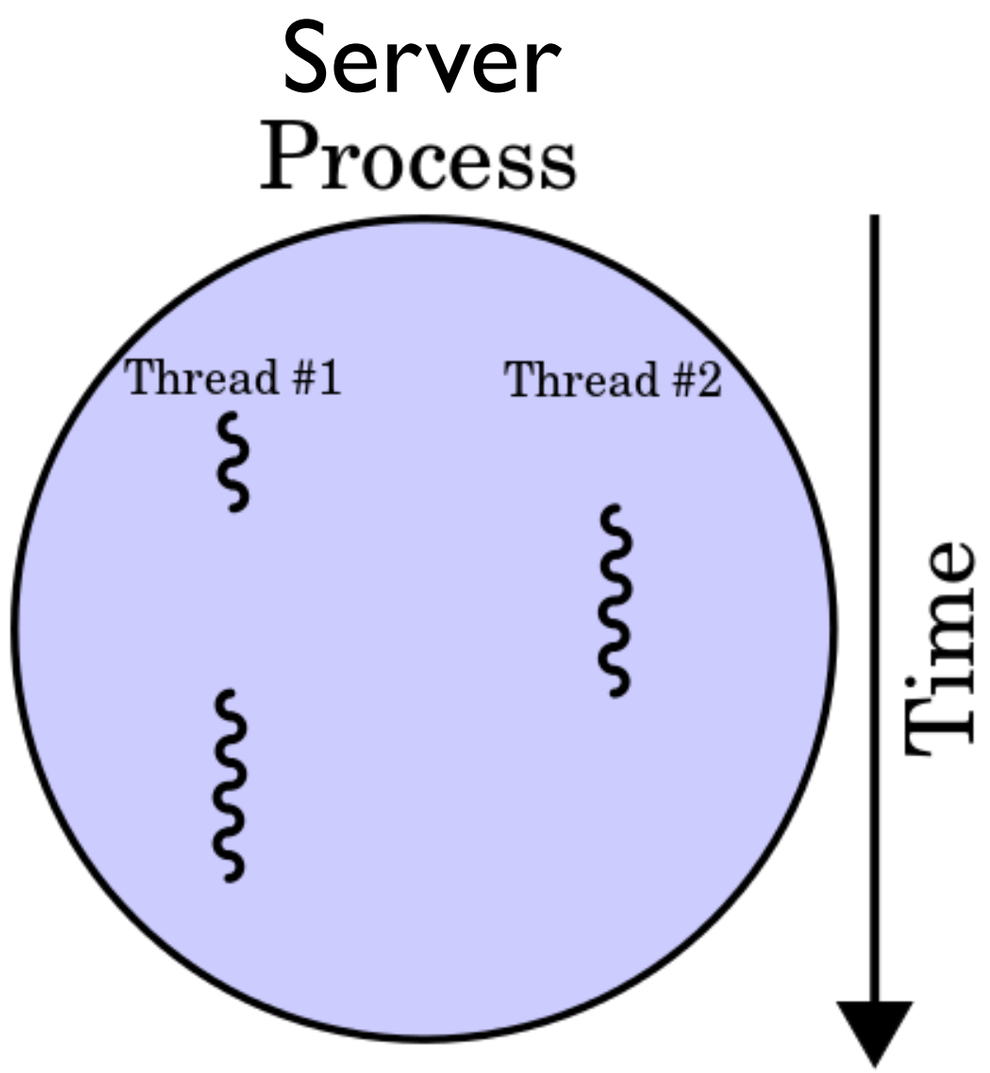


1. Client procedure calls client stub.
2. Stub builds message; calls local OS.
3. OS sends message to remote OS.
4. Remote OS gives message to stub.
5. Stub unpacks parameters and calls server.
6. Server returns result to stub.
7. Stub builds message; calls OS.
8. OS sends message to client's OS.
9.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

Threads



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



rpc handlers might take a long
time — often use threads to
execute many rpcs concurrently
(thread per handler execution)

Failures?

- lost packet,
- broken network,
- slow server,
- crashed server

Failures?

- lost packet,
- broken network,
- crashed server
- slow server

From the Client's perspective failures typically mean that client is waiting for a reply that will never come

if (no reply in X seconds) then ?

At least Once Failure model

- Regardless of failures execute the rpc at least once

At least Once Failure model

- Regardless of failures execute the rpc at least once

```
while true {
```

At least Once Failure model

- Regardless of failures execute the rpc at least once

```
while true {
send request
```

At least Once Failure model

- Regardless of failures execute the rpc at least once

```
while true {
send request
wait X seconds for reply
```

At least Once Failure model

- Regardless of failures execute the rpc at least once

```
while true {
send request
wait X seconds for reply
if reply return
```

At least Once Failure model

- Regardless of failures execute the rpc at least once

```
while true {
  send request
  wait X seconds for reply
  if reply return
}
```

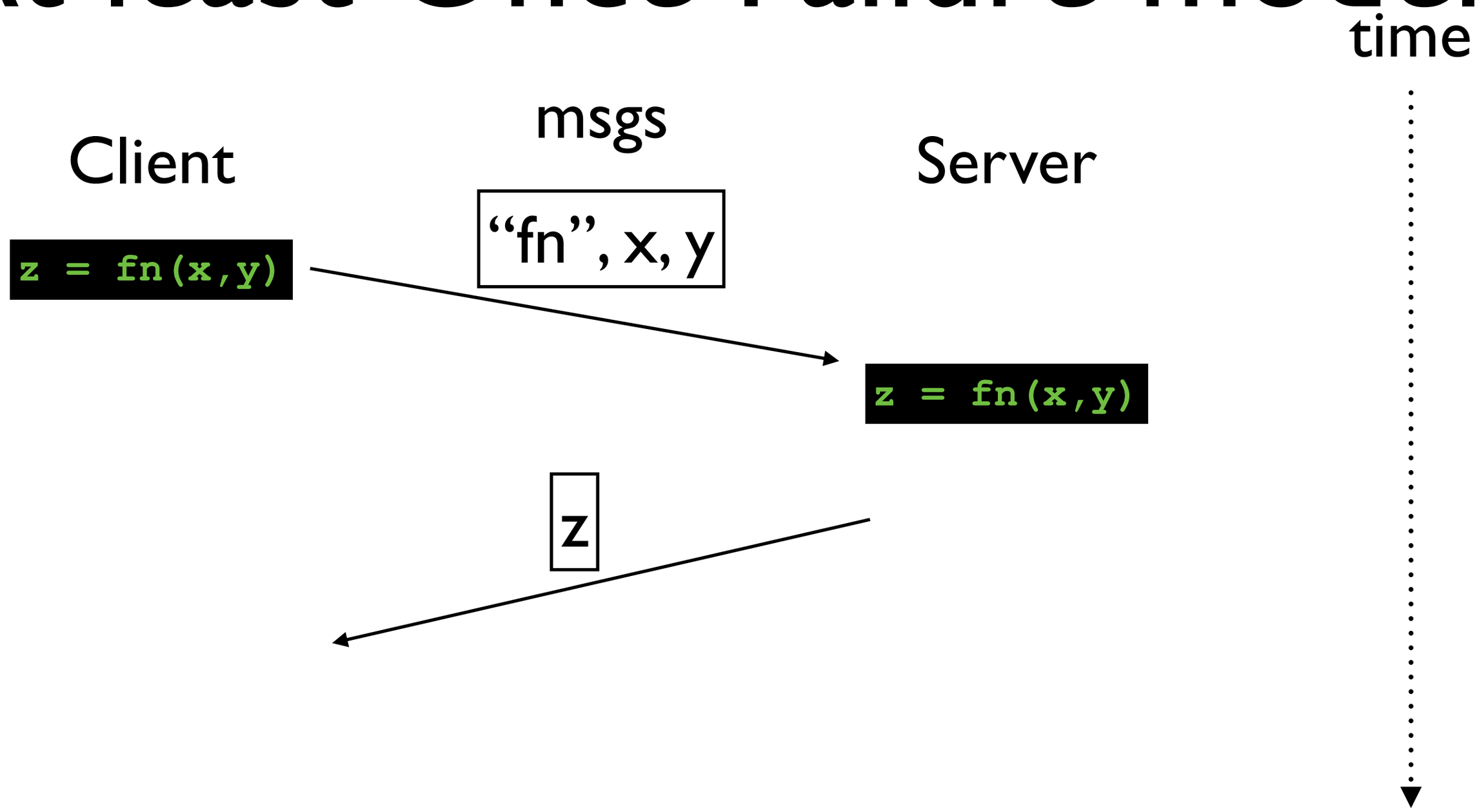
At least Once Failure model

- Regardless of failures execute the rpc at least once

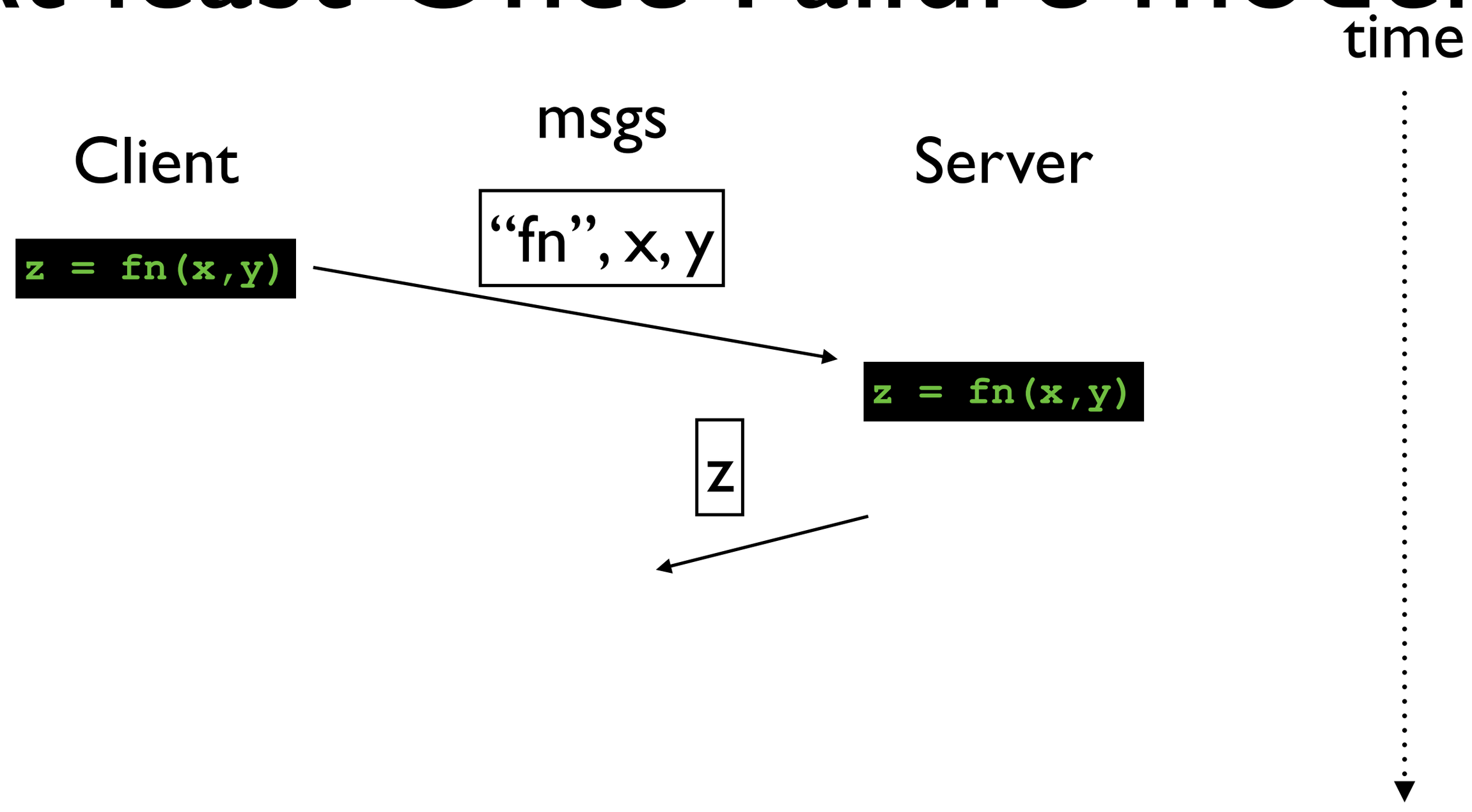
```
while true {
  send request
  wait X seconds for reply
  if reply return
}
```

As long as eventually some or something fixes the problem (eg. reboot server, fix network) then this will always work

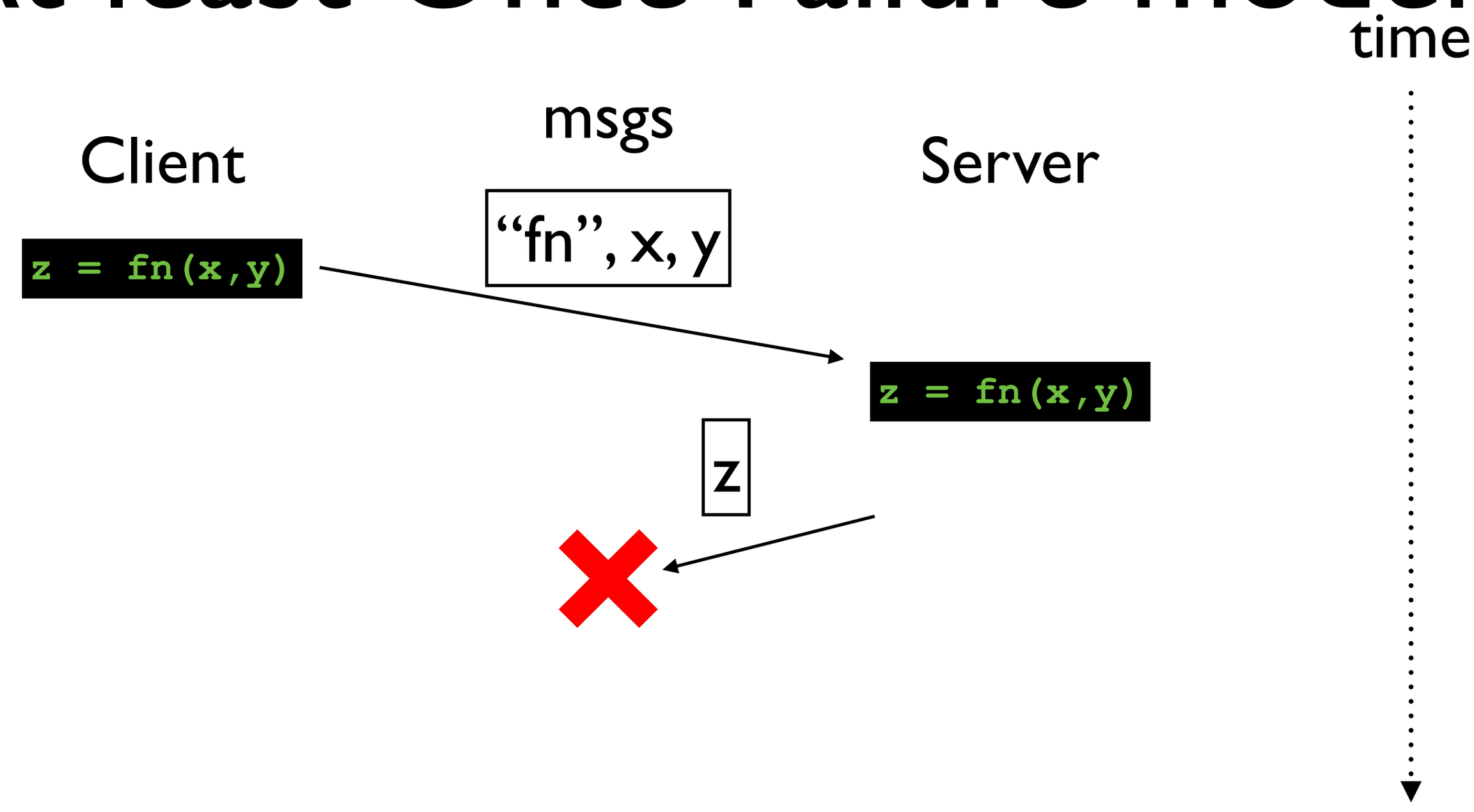
At least Once Failure model



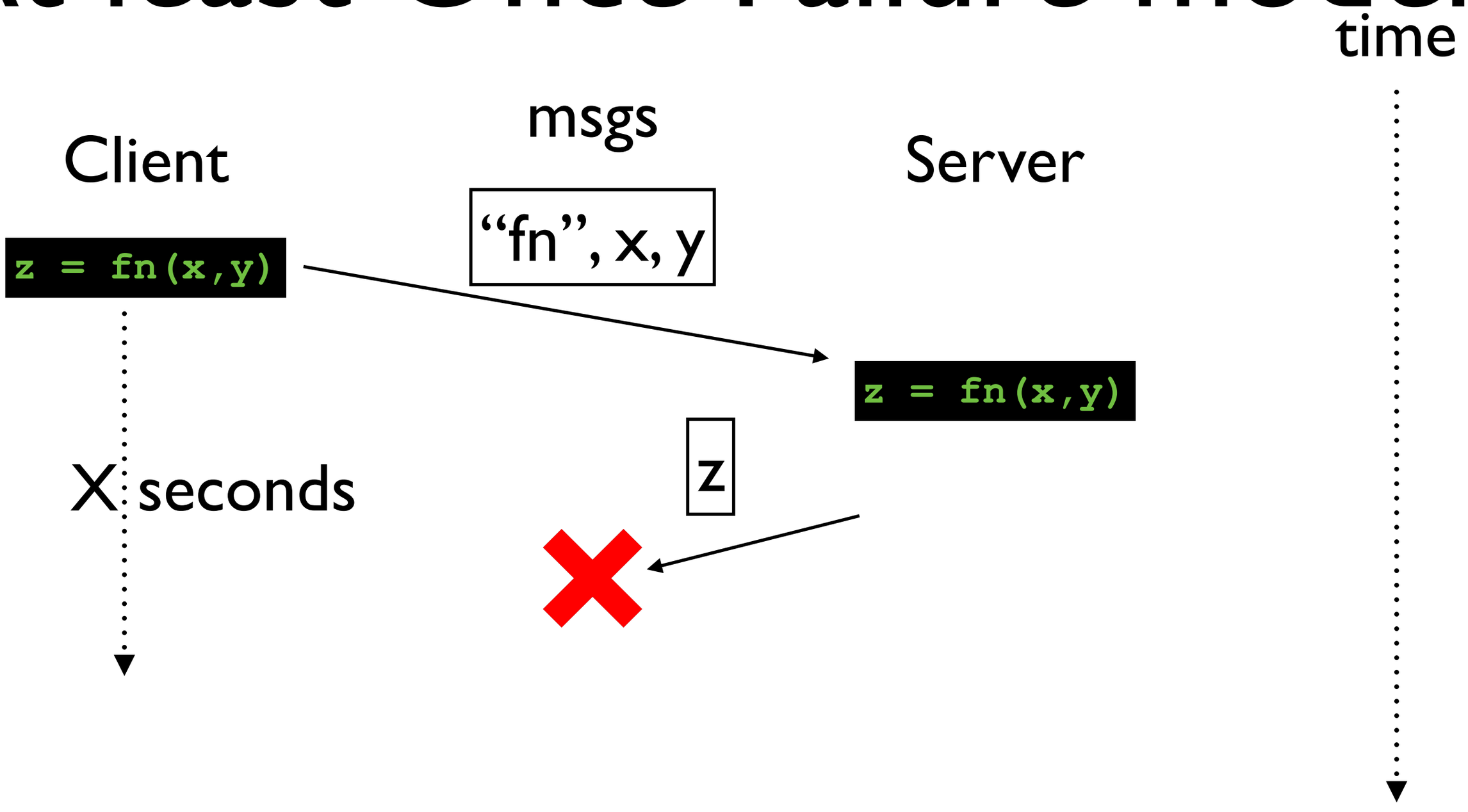
At least Once Failure model



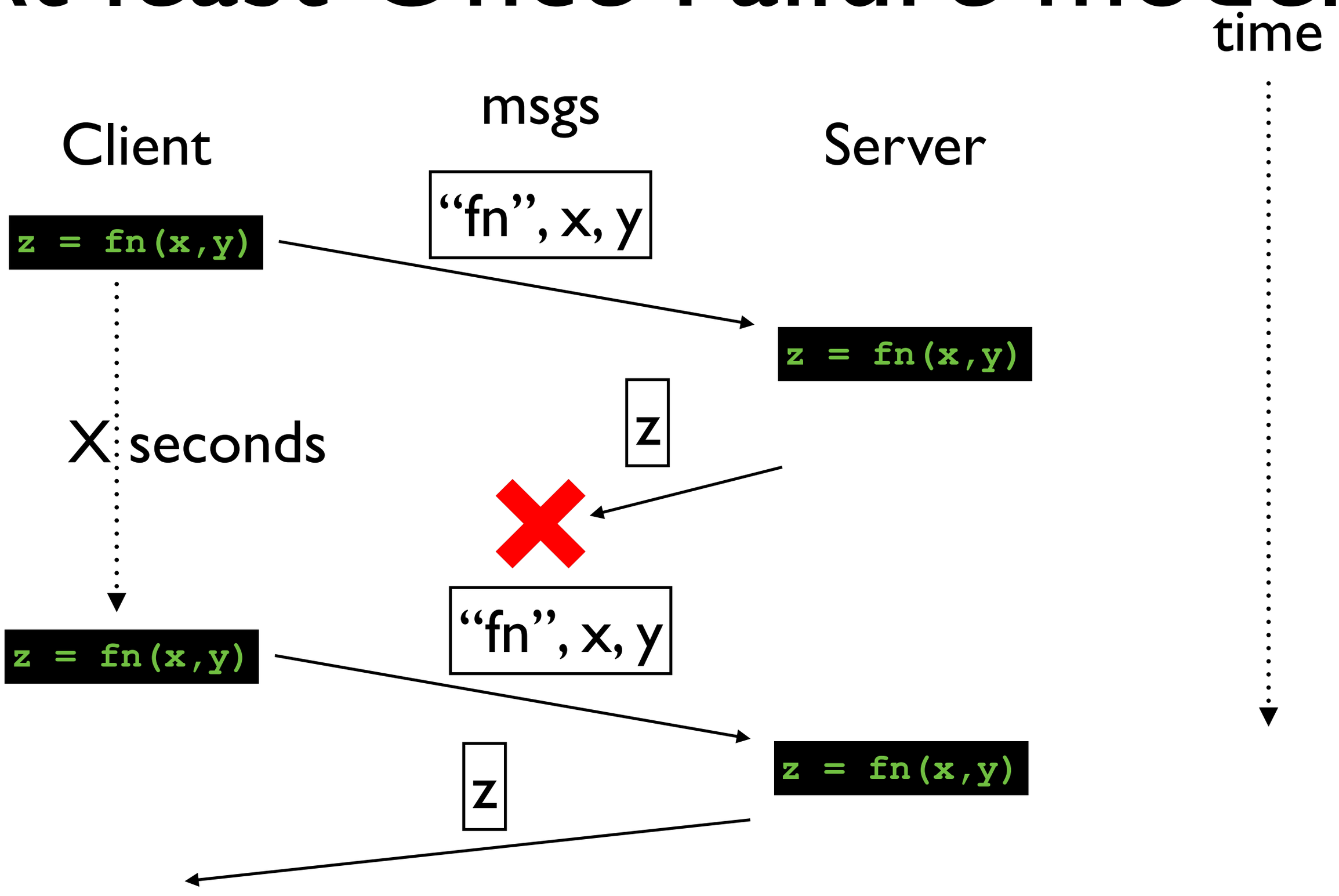
At least Once Failure model



At least Once Failure model



At least Once Failure model



One response but two executions

Assume DB like key,value store

"Idempotent"

put(key,20);
get(key);

Is this ok?

One response but two executions

Assume DB like key,value store

"Idempotent"

put(key,20);
get(key);

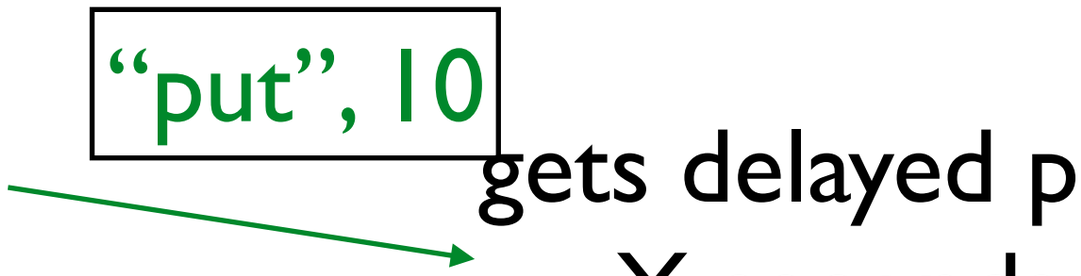
Is this ok?
In general even with
Idempotent functions no.

One response but two executions

"put", 10

```
put(key,10);
put(key,20);
get(key);
```

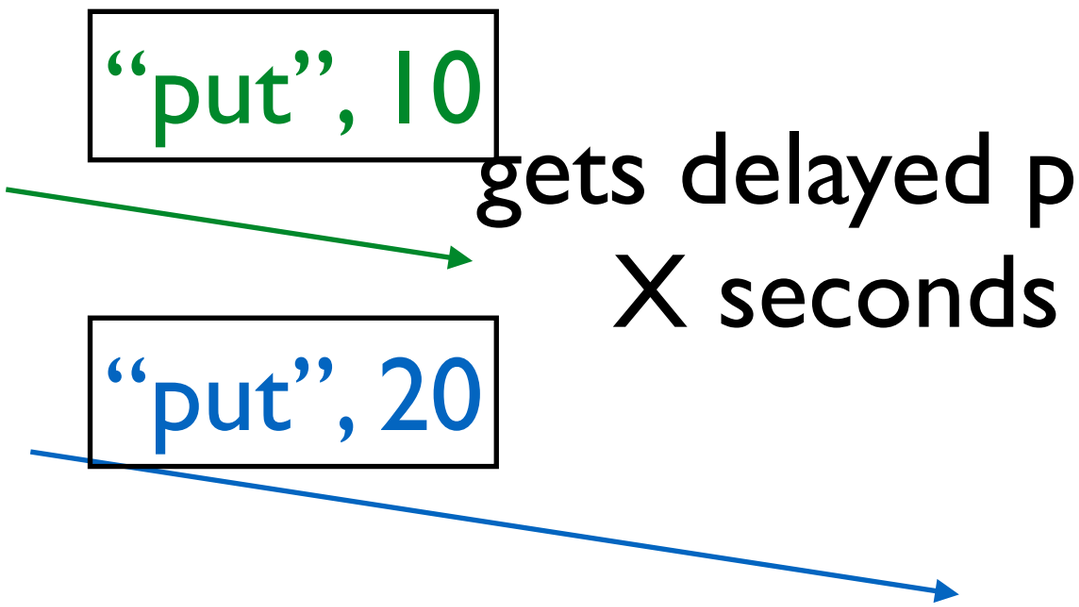
One response but two executions




```
put(key,10);
put(key,20);
get(key);
```

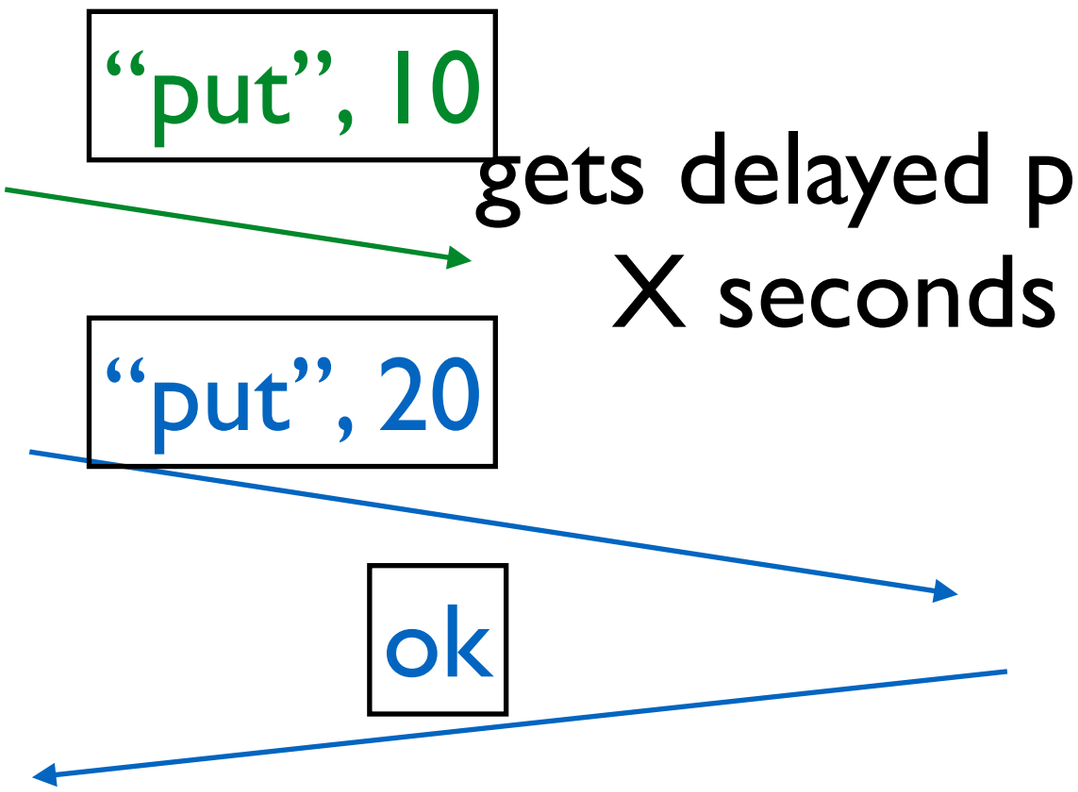
One response but two executions

put(key,10);
put(key,20);
get(key);



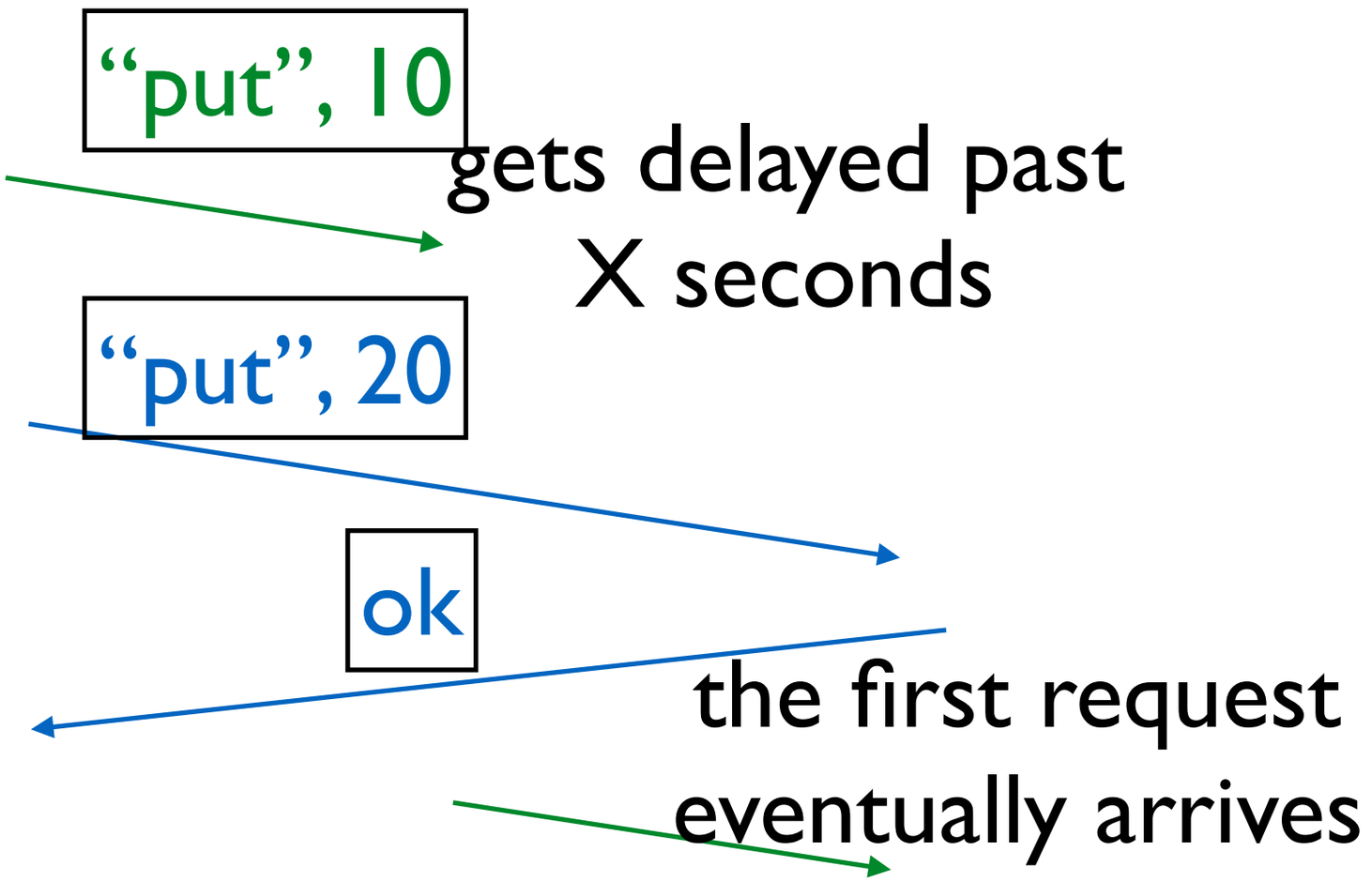
One response but two executions

```
put(key,10);
put(key,20);
get(key);
```



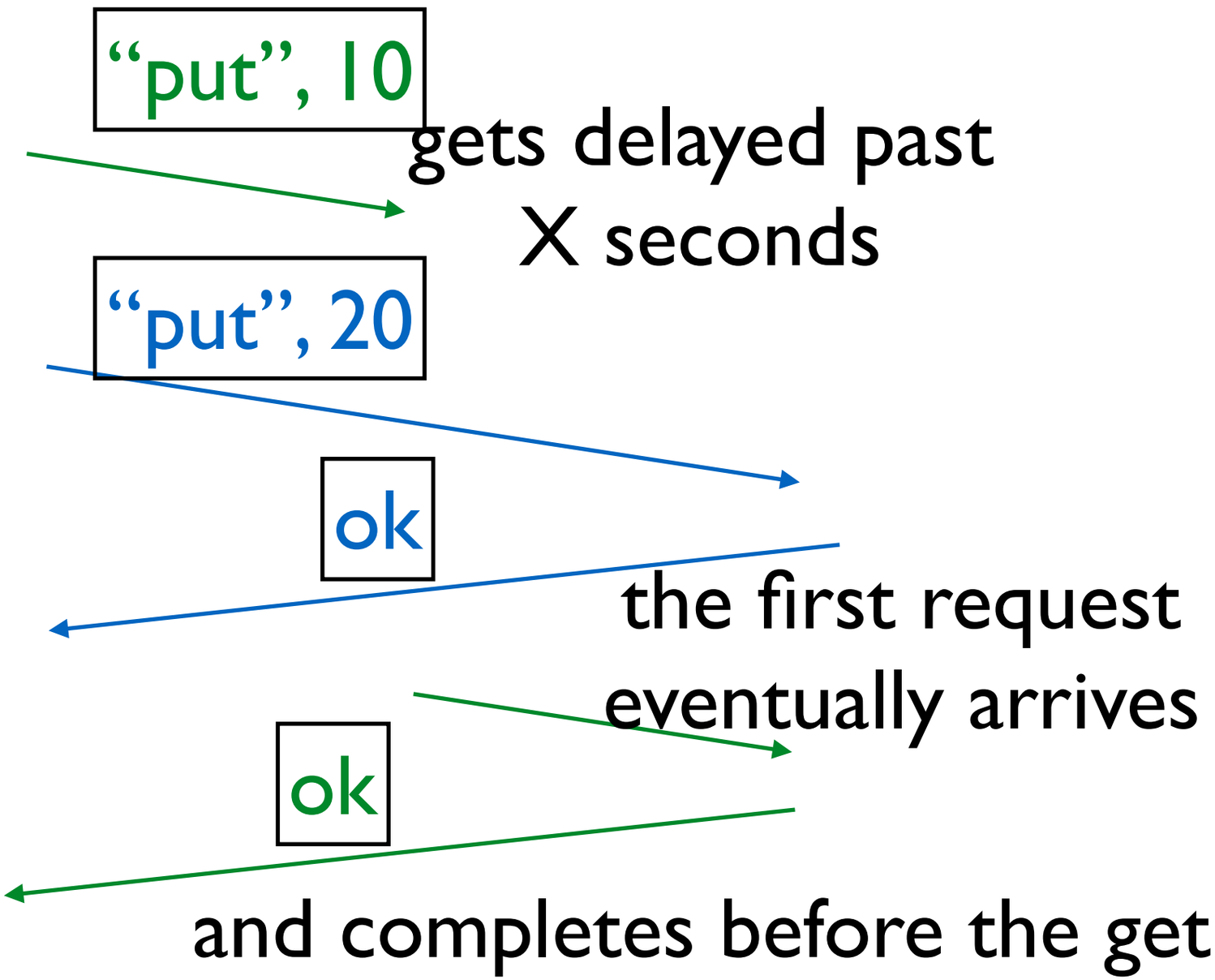
One response but two executions

```
put(key,10);
put(key,20);
get(key);
```



One response but two executions

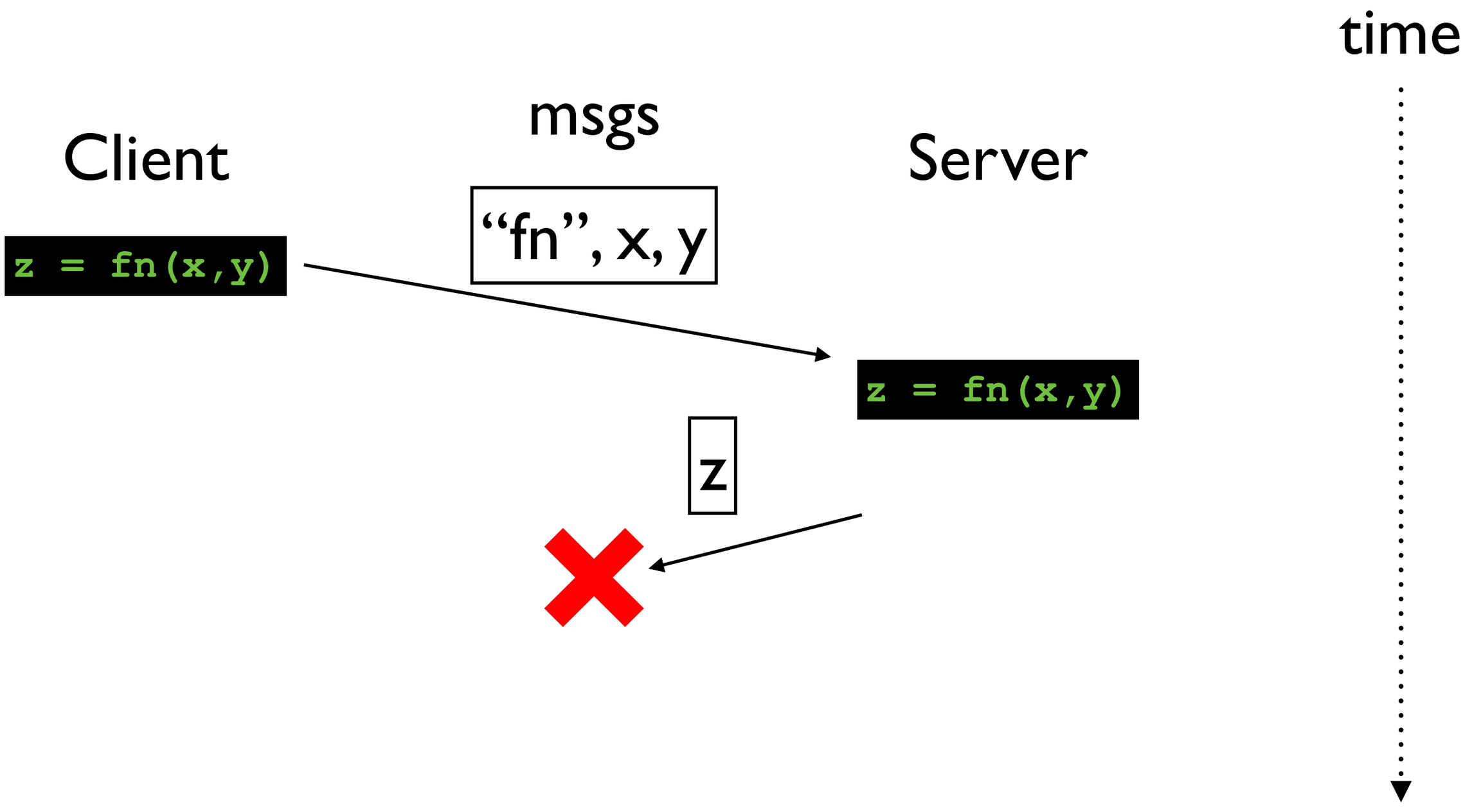
```
put(key,10);
put(key,20);
get(key);
```



At least Once Failure model

- It does work for read-only operations
- or you have a strategy for duplicates (which later labs will require)

BTW this does really happen

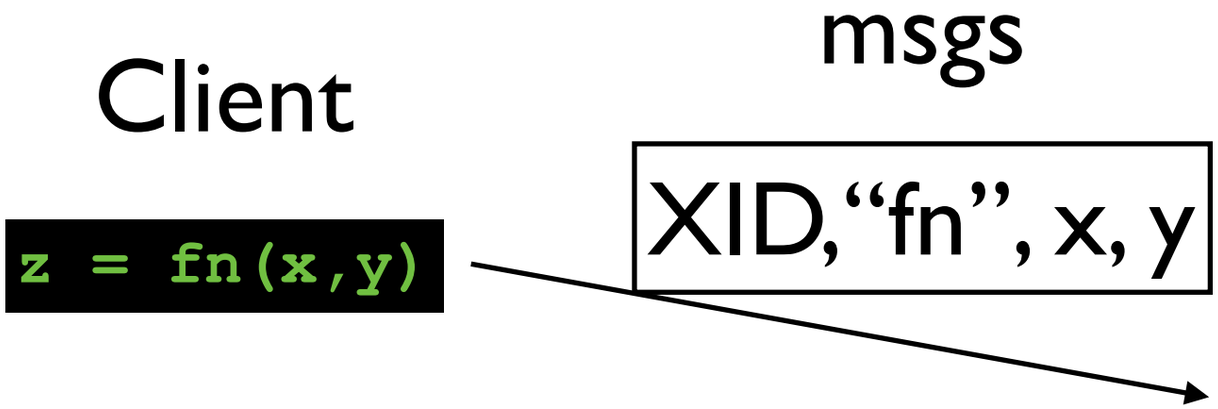


At Most Once

- server detects duplicates and not execute handler
- how to detect?

At Most Once : RPC id

Introduce a unique id per-RPC invocation and some storage




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

At Most Once : RPC id

Introduce a unique id per-RPC invocation and some storage

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

This works but there are some issues

How do we delete things from old and seen?

How do we delete things from old and seen?

Get an ack from the client for XID for which it has received responses

Some related ideas

- send recent retired XID's with next request
- use sequence numbers as XID
  - XID = <client id, seq #>

    <42,0>

    <42,1>

    …

 seq number goes up with every successful response so server knows all prior XID's are retired

There is lots of subtly here

- This is something that you will have to think about and play with
- what happens with sequence numbers if client is allowed to make concurrent requests?
- what happens if duplicate request comes in while the original is still executing?
- What happens if server crashes and is restarted in the face of duplicate requests?

How about Once

- Need at-most once with at least once unbounded retries
- and fault tolerant server implementation

GO RPC

- At most once with respect to a single client server
- Built on top of single TCP connection
    - Thus TCP handles retries and duplicates under the covers
- returns error if reply is not received
    - eg. connection broken (TCP timeout)

Not Good Enough in General

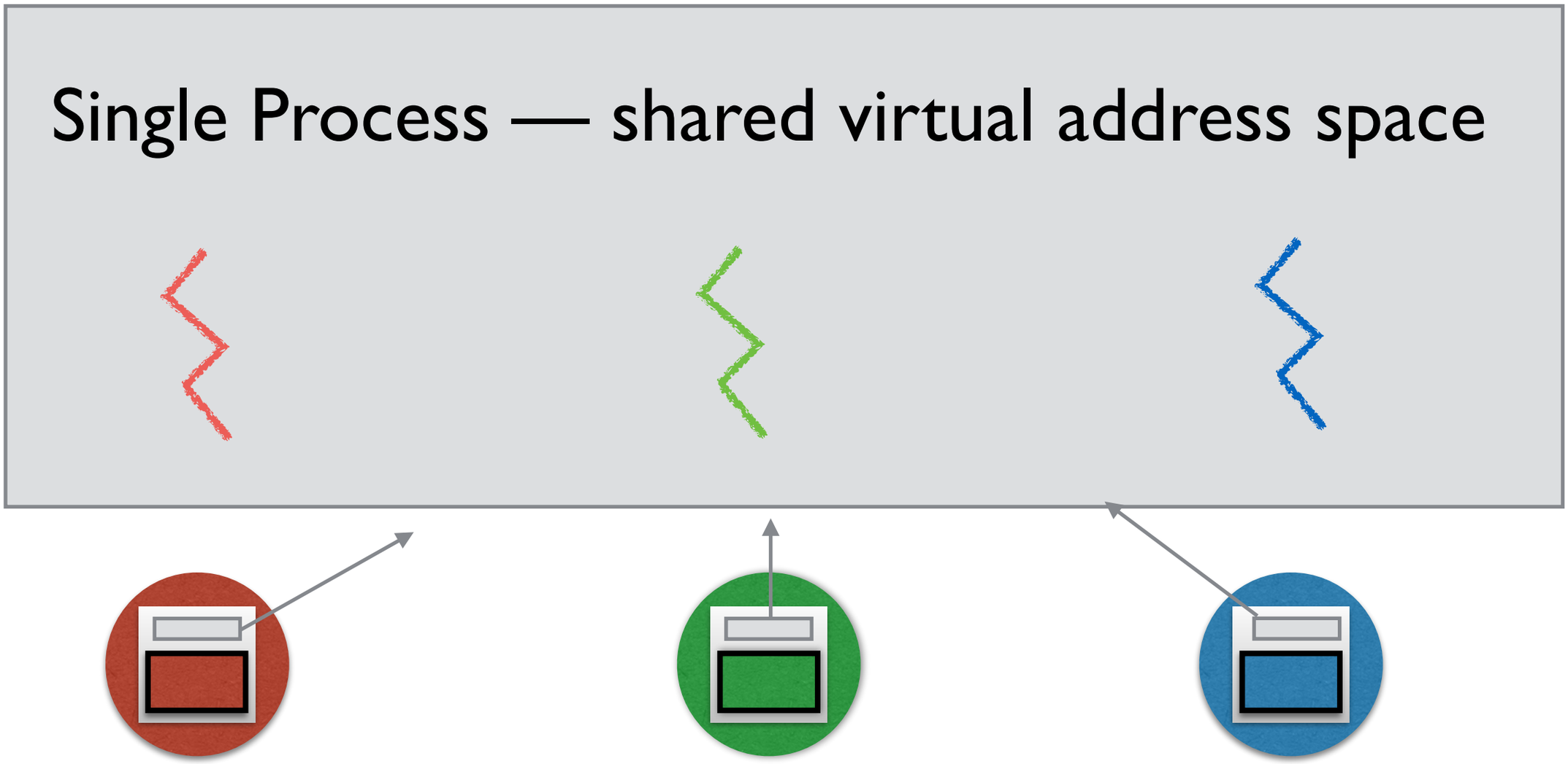
- Lab1: GO RPC will avoid single worker from ever seeing a duplicate request from the master
- But if you don't get a response from a worker and start a request to another worker you now have a duplicate
- GO RPC can't detect this
  - ok in Lab1 — handled at App Level but
  - Lab 2 will explicitly have to deal with duplicates

THREADS

Threads and RPC's go hand in hand

thread = go routines

Basic idea



Basic problem : r/w shared data structures concurrently

MOV 0xdeadbeef, %eax
ADD $1, %eax
MOV %eax, 0xdeadbeef

MOV 0xdeadbeef, %eax
ADD $1, %eax
MOV %eax, 0xdeadbeef

OK

Basic problem : updating shared data structures concurrently

MOV 0xdeadbeef, %eax
ADD $1, %eax
MOV %eax, 0xdeadbeef

MOV 0xdeadbeef, %eax
ADD $1, %eax
MOV %eax, 0xdeadbeef

OK

Basic problem : updating shared data structures concurrently

```
MOV 0xdeadbeef, %eax
ADD $1, %eax
MOV %eax, 0xdeadbeef
```

```
MOV 0xdeadbeef, %eax
ADD $1, %eax
MOV %eax, 0xdeadbeef
```

BAD : RACE

Need to control concurrency when touching shared data structures

MUTUAL EXCLUSION

GO MUTEX aka LOCK

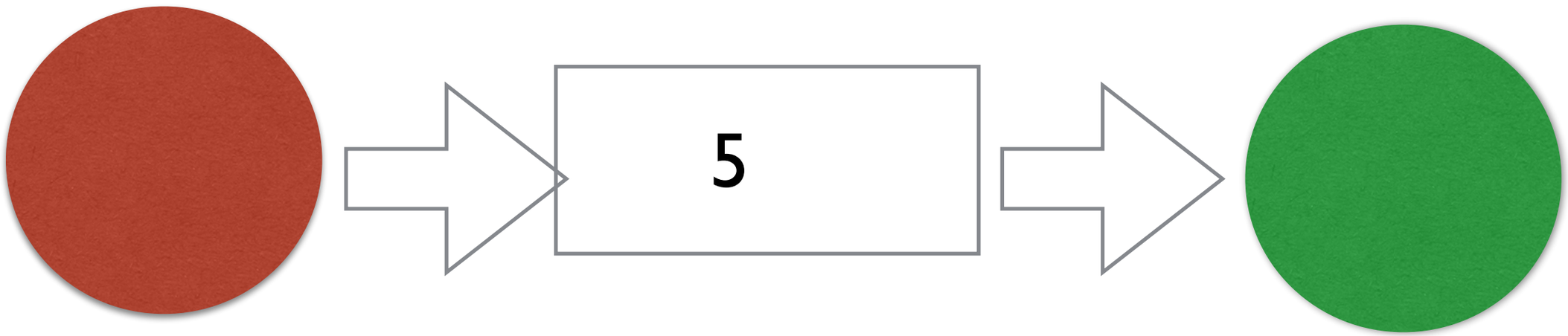
```
import “sync”

var counter int
var lock sync.Mutex

func incCounter() {
    lock.Lock()
    count=count + 1
    lock.Unlock()
}
```

Critical Section
As long as one thread “has” the lock all others will block until the lock is “released”

Channels : Go Channels

var done chan int32
done = make(chan int32)

```
done <- 5                    reply := <- done
```



Both will block as needed. Therefore when either return you can know that both returned and got the value written and read

DEAD LOCK

```
var lockA sync.Mutex;
var lockB sync.Mutex;
```

```
lockA.Lock()
lockB.Lock()
```

```
lockB.Lock()
lockA.Lock()
```