Distributed Systems
[Fall 2012]

Topic: Synchronization

Lec 6: Local Synchronization Primitives

Slide acks: Jinyang Li, Dave Andersen, Randy Bryant

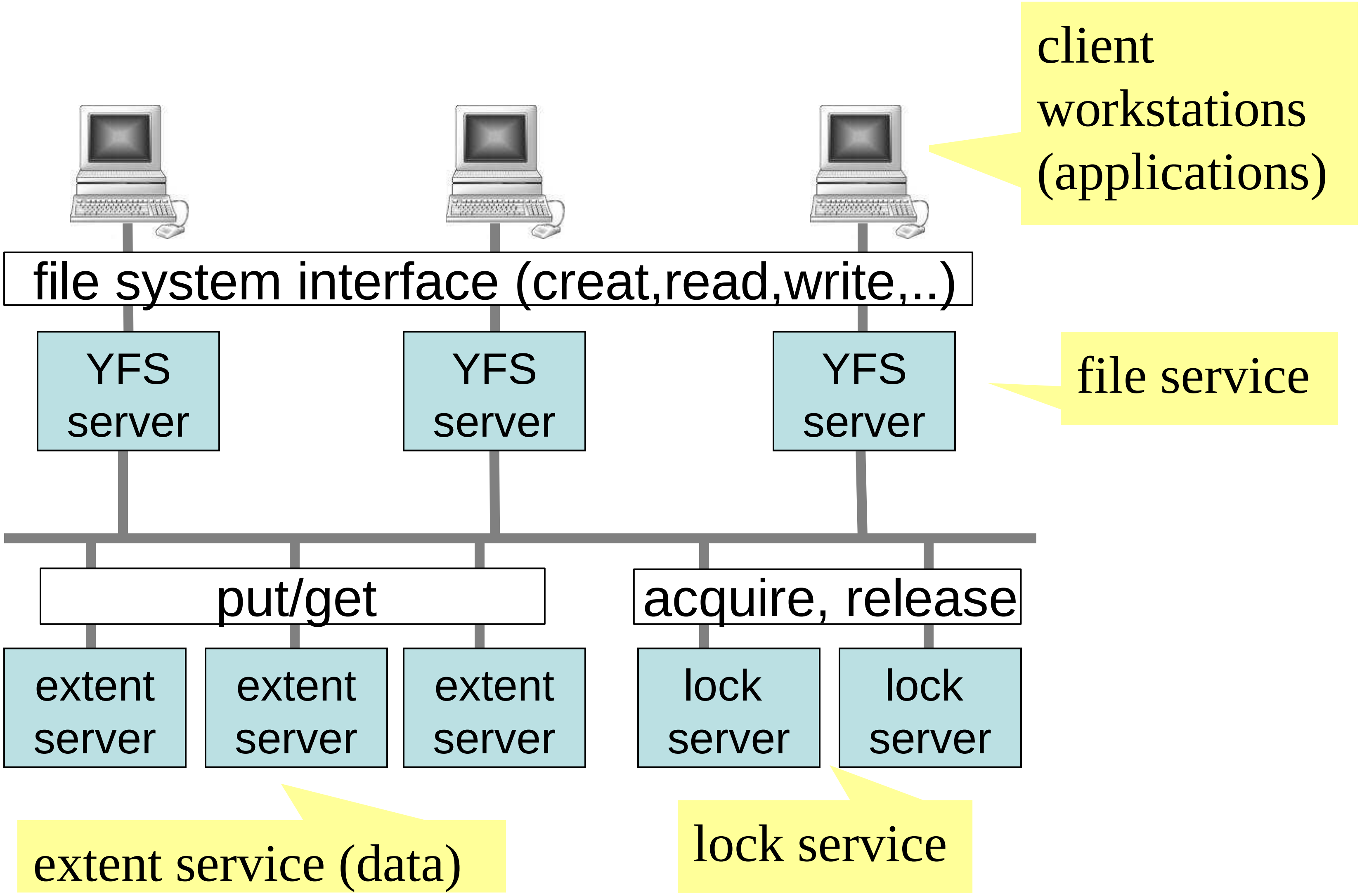
(http://www.news.cs.nyu.edu/~jinyang/fa10/notes/ds-lec2.ppt,

http://www.cs.cmu.edu/~dga/15-440/F12/lectures/05-concurrency.txt)

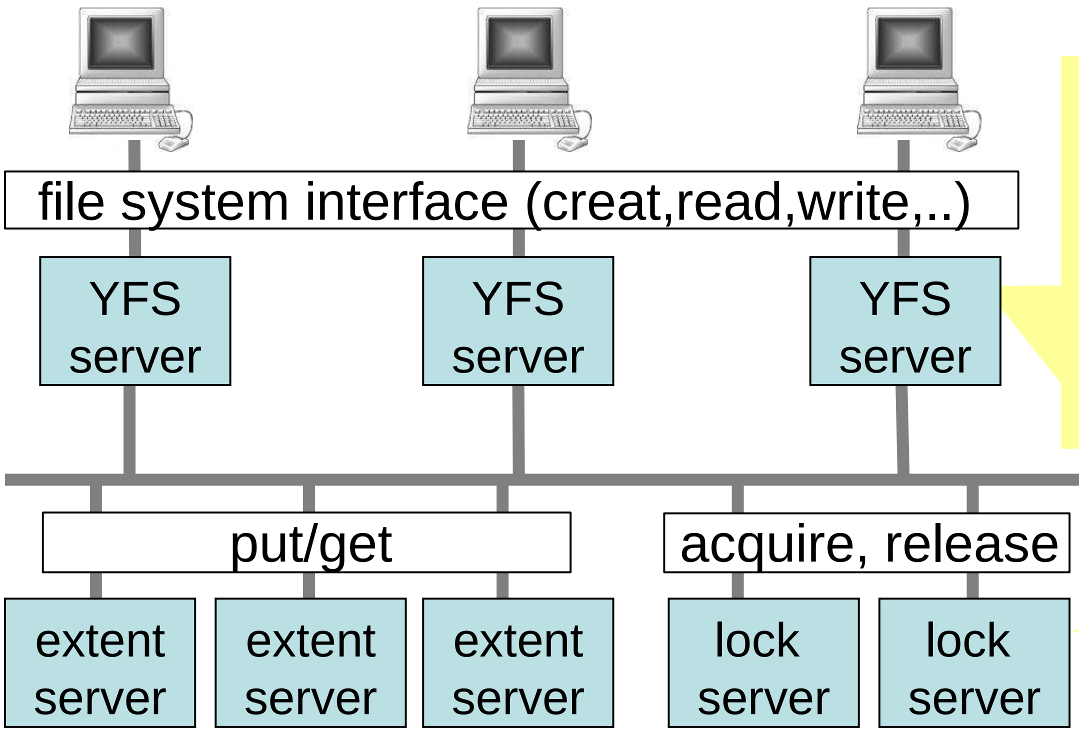
Outline

- Example motivation: YFS
- Local synchronization primitives
  - Semaphores
  - Conditional variables
- Next time: distributed synchronization primitives

YFS Design (Reminder)



YFS Design



YFS Service:

- serves file system requests
- it's stateless (i.e., doesn't store data)
- uses extent service to store data
- incrementally scalable with more servers

Lock Service:

- ensure consistent updates by multiple yfs servers
- replicated for fault tolerance

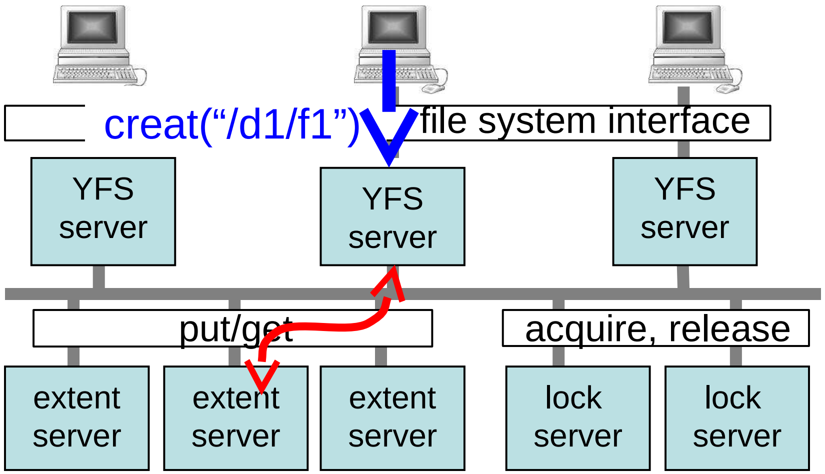
Extent Service:

- stores the data (dir and file contents)
- replicated for fault tolerance
- incrementally scalable with more servers

YFS Server Implements FS Logic

- Application program:

  creat("/d1/f1", 0777)



- YFS server:
  1. GET root directory's *data* from Extent Service
  2. Find Extent Server address for dir "/d1" in root dir's *data*
  3. GET "/d1"s *data* from Extent Server
  4. Find Extent Server address of "f1" in "/d1"'s *data*
  5. If not exists

     alloc a new data block for "f1" from Extent Service

     add "f1" to "/d1"'s data, PUT modified "/d1" to Extent Serv.

Concurrent Accesses Cause Inconsistency

time

App 1: creat("/d1/f1", 0777)

Server S1:

…

GET "/d1"

Find file "f1" in "/d1"

If not exists

…

PUT modified "/d1"

App 2: creat("/d1/f2", 0777)

Server S2:

…

GET "/d1"

Find file "f2" in "/d1"

If not exists

…

PUT modified "/d1"

What is the final result of "/d1"? What should it be?

Solution: Use a Lock Service to Synchronize Access

time

App 1: creat(“/d1/f1”, 0777)

Server S1:

…

ACQUIRE(“/d1”)

GET “/d1”

Find file “f1” in “/d1”

If not exists

…

PUT modified “/d1”

RELEASE(“/d1”)

App 2: creat(“/d1/f2”, 0777)

Server S2:

…

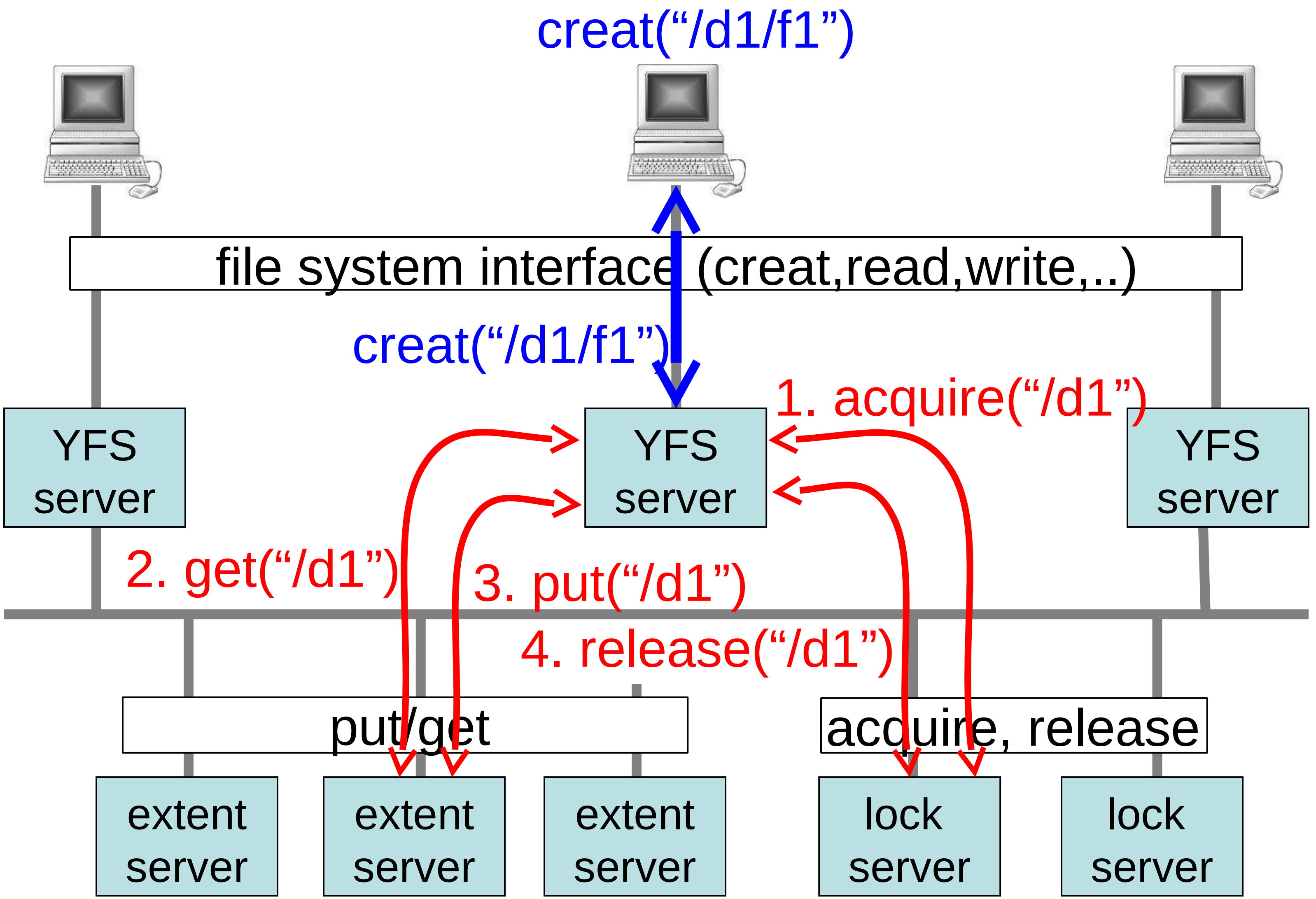
ACQUIRE(“/d1”) // blocks

GET “/d1”

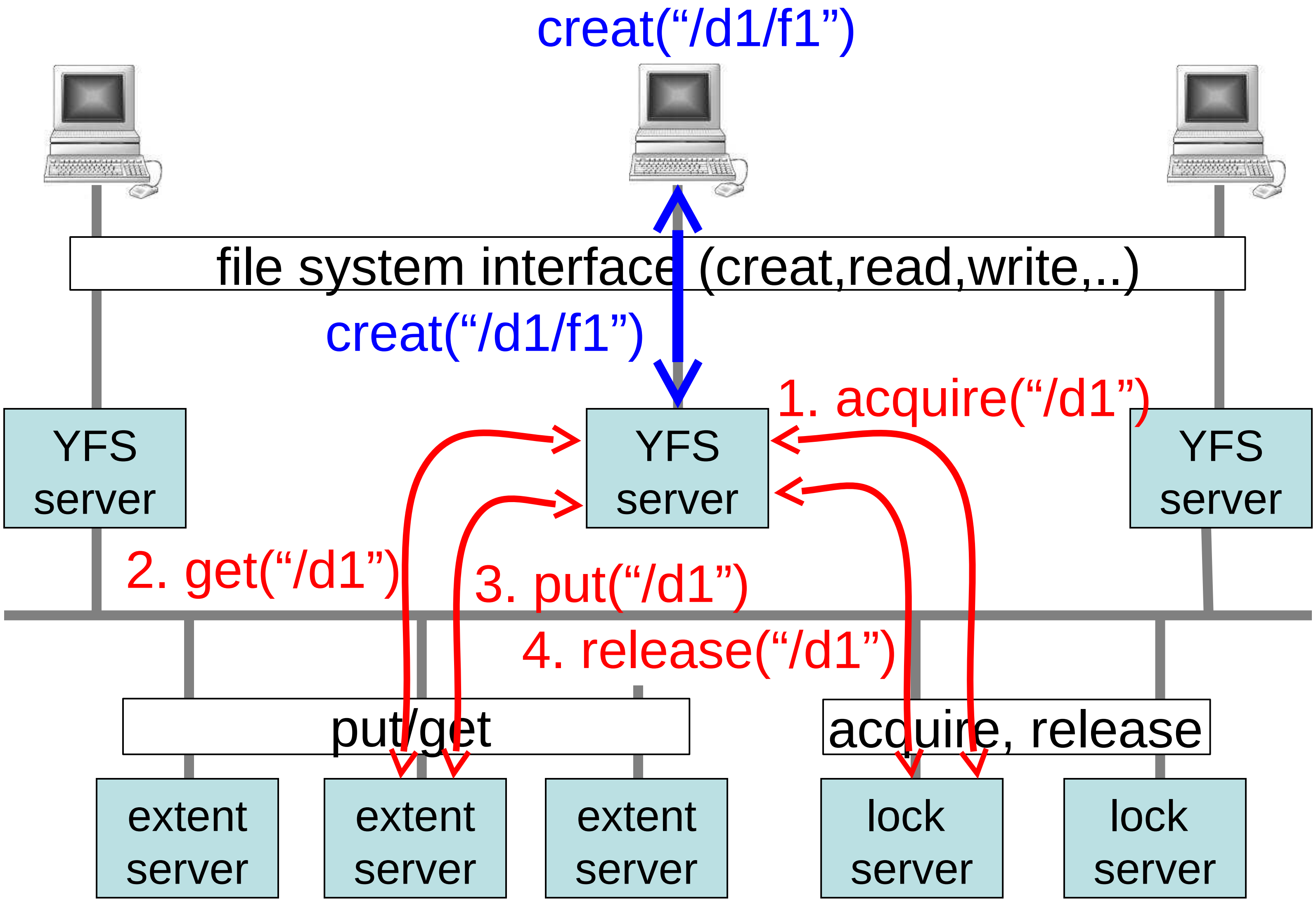
…

RELEASE(“/d1”)

Putting It Together



Another Problem: Naming



Any issues with this version?

Another Problem: Naming

time

App 1: creat("/d0/d1/f1", 0777)

Server S1:

…

acquire("/d0/d1")

GET "/d0/d1"

Find file "f1" in d1

If not exists

…

PUT modified "/d0/d1"

Release("/d0/d1")

App 2:

Server S2:

…

rename("/d0", "/d2")

rename("/d3", "/d0")

…

Same problem occurs for reading/writing files, if we use their names when identifying them on the server.

Solution: Using GUIDs

time

App 1: creat("/d0/d1/f1", 0777)

Server S1:

…

acquire(d1_id)

d1 = GET d1_id

Find file "f1" in d1

If not exists

…

PUT(d1_id, modified d1)

release(d1_id)

App 2:

Server S2:

…

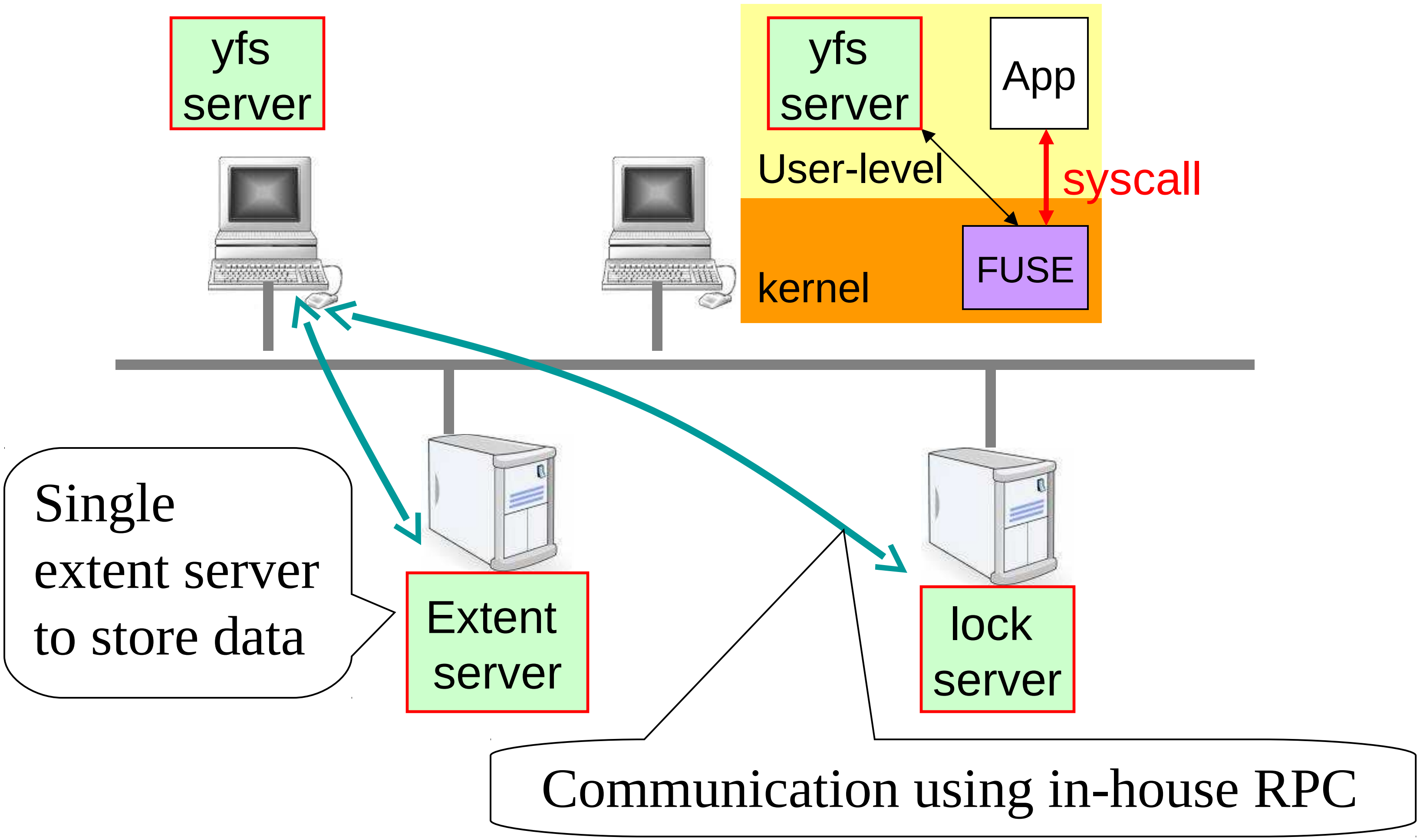
rename("/d0", "/d2")

rename("/d3", "/d0")

…

- GUIDs are globally-unique, location-independent names
- GUID principle is pervasive in FSes and DSes!
  - Think of inode numbers

(Paranthesis) HW 2-7
Build a Simplified YFS Version



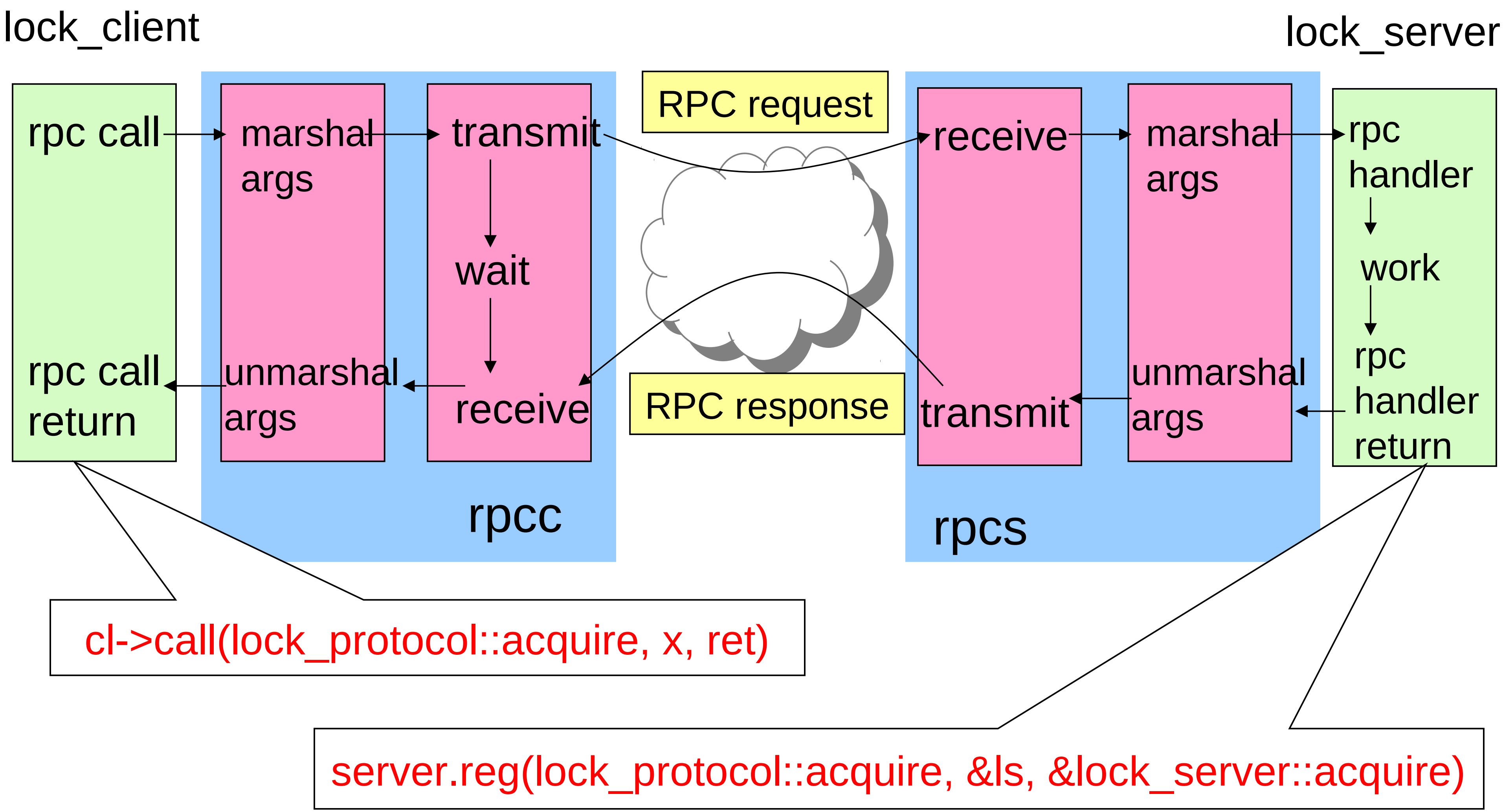
HW 2: Lock Server

- Lock service consists of:
  - Lock server: grant a lock to clients, one at a time
  - Lock client: talk to server to acquire/release locks
- Correctness:
  - At most one lock is granted to any client at any time
- Additional requirement:
  - acquire() at client does not return until lock is granted

HW 2 Steps

- Step 1: Implement server lock and client lock code
  - Use in-house RPC mechanism:
    - It's a bit more primitive than Thrift and the like (see next slide)
- Step 2: Implement at-most-once semantics
  - Why?
  - How do we do that?
- Due next Sept 25 before midnight
  - Absolutely no extensions!
  - Lab is significantly more difficult than HW 1, so start now working on it **NOW**!
  - Work in layers, submit imperfect but on time!

YFS's RPC library (End Paranthesis)



Outline

- Detailed YFS lab introduction
  - Plus Lab 1 description
  - From last lecture's slides, which we didn't cover
- Local synchronization primitives
  - Locks   // already talked about these
  - Semaphores
  - Conditional variables
- Next time: distributed synchronization
  - Many of the primitives here distribute or build upon local primitives

Locks are great. Why others?

- Locks are very low level, and often times you need more to accomplish a synchronization goal
  - Hence, people have developed a variety of synchronization primitives that raise level of abstraction

Semaphores

- Integer variable x that allows interaction via 2 operations:
  - **x.P()**:

```
while (x == 0) wait;
--x
```

  - **x.V()**:

```
++x
```

- Both operations are done atomically
  - All steps take place without any intervening operations
- When do we use semaphores?

Example: Thread-Safe FIFO Queue

```
q.Initialize():
   initialize state

q.Insert(x):
   add item into queue

q.Remove():
   block until queue not empty;
   return item at head of queue

q.Flush():
   clear queue
```

- Assume that we already have a sequential queue implementation: sq
  - But sq is not thread-safe!

- So, how do we make q thread-safe?

FIFO Queue with Mutexes

```
q.Initialize():
    q.sq = NewSQueue()
    q.mutex = 1

q.Insert(x):
    q.mutex.lock()
    q.sq.Insert(x)
    q.mutex.unlock()

q.Remove():
    q.mutex.lock()
    x = q.sq.Remove()
    q.mutex.unlock()
    return x

q.Flush():
    q.mutex.lock()
    q.sq.Flush()
    q.mutex.unlock()
```

Are we done?

FIFO Queue with Mutexes

```
q.Initialize():
    q.sq = NewSQueue()
    q.mutex = 1

q.Insert(x):
    q.mutex.lock()
    q.sq.Insert(x)
    q.mutex.unlock()
```

```
q.Remove():
    q.mutex.lock()
    x = q.sq.Remove()
    q.mutex.unlock()
    return x

q.Flush():
    q.mutex.lock()
    q.sq.Flush()
    q.mutex.unlock()
```

Are we done?

Nope: Remove doesn't block when buffer's empty

FIFO Queue with Semaphores

- Use semaphore to count number of elements in queue

```
q.Initialize():
    q.sq = NewSQueue()
    q.mutex = 1
    q.items = 0

q.Insert(x):
    q.mutex.lock()
    q.sq.Insert(x)
    q.mutex.unlock()
    q.items.V()
```

```
q.Remove():
    q.items.P()
    q.mutex.lock()
    x = q.sq.Remove()
    q.mutex.unlock()
    return x

q.Flush():
    q.mutex.lock()
    q.sq.Flush()
    q.items = 0
    q.mutex.unlock()
```

Are we done?

FIFO Queue with Semaphores

- Use semaphore to count number of elements in queue

```
q.Initialize():
    q.sq = NewSQueue()
    q.mutex = 1
    q.items = 0

q.Insert(x):
    q.mutex.lock()
    q.sq.Insert(x)
    q.mutex.unlock()
    q.items.V()
```

q.Flush() →

```
q.Remove():
    q.items.P()
    q.mutex.lock()
    x = q.sq.Remove()
    q.mutex.unlock()
    return x

q.Flush():
    q.mutex.lock()
    q.sq.Flush()
    q.items = 0
    q.mutex.unlock()
```

Are we done?

Nope: Just Insert & Remove work fine,

but Flush messes things up

Fixing Race with Mutex?

```
q.Initialize():
    q.sq = NewSQueue()
    q.mutex = 1
    q.items = 0


q.Insert(x):
    q.mutex.lock()
    q.sq.Insert(x)
    q.mutex.unlock()
    q.items.V()
```

```
q.Remove():
    q.mutex.lock()
    q.items.P()
    x = q.sq.Remove()
    q.mutex.unlock()
    return x

q.Flush():
    q.mutex.lock()
    q.sq.Flush()
    q.items = 0
    q.mutex.unlock()
```

Are we done?

Yes, from a correctness perspective

Nope, from a liveness perspective -- deadlock

Condition Variables

- Condition variables provide synchronization point, where one thread suspends until activated by another
- Condition variable always associated with a mutex
- cvar.Wait():
  - Must be called after locking mutex
  - Atomically: { release mutex & suspend operation }
  - When resume, lock the mutex (may have to wait for it)
- cvar.Signal():
  - If no thread suspended, then no-op
  - Wake up one suspended thread

FIFO Queue with Condition Variable

```
q.Initialize():
    q.sq = NewSQueue()
    q.mutex = 1
    q.cvar = NewCond(q.mutex)


q.Insert(x):
    q.mutex.lock()
    q.sq.Insert(x)
    q.cvar.Signal()
    q.mutex.unlock()
```

```
q.Remove():
    q.mutex.lock()
    if q.sq.IsEmpty():
        q.cvar.Wait()
    x = q.sq.Remove()
    q.mutex.unlock()
    return x

q.Flush():
    q.mutex.lock()
    q.sq.Flush()
    q.mutex.unlock()
```

Are we done?

Still Nope: Wait() has 3 steps:

- Unlock
- Wait for signal
- Lock

(Unlock and Wait for signal: atomic)

q.Flush() → (between Wait for signal and Lock)

Thread-Safe FIFO Queue

```
q.Initialize():
    q.sq = NewSQueue()
    q.mutex = 1
    q.cvar = NewCond(q.mutex)


q.Insert(x):
    q.mutex.lock()
    q.sq.Insert(x)
    q.cvar.Signal()
    q.mutex.unlock()
```

```
q.Remove():
    q.mutex.lock()
    while q.sq.IsEmpty():
        q.cvar.Wait()
    x = q.sq.Remove()
    q.mutex.unlock()
    return x

q.Flush():
    q.mutex.lock()
    q.sq.Flush()
    q.mutex.unlock()
```

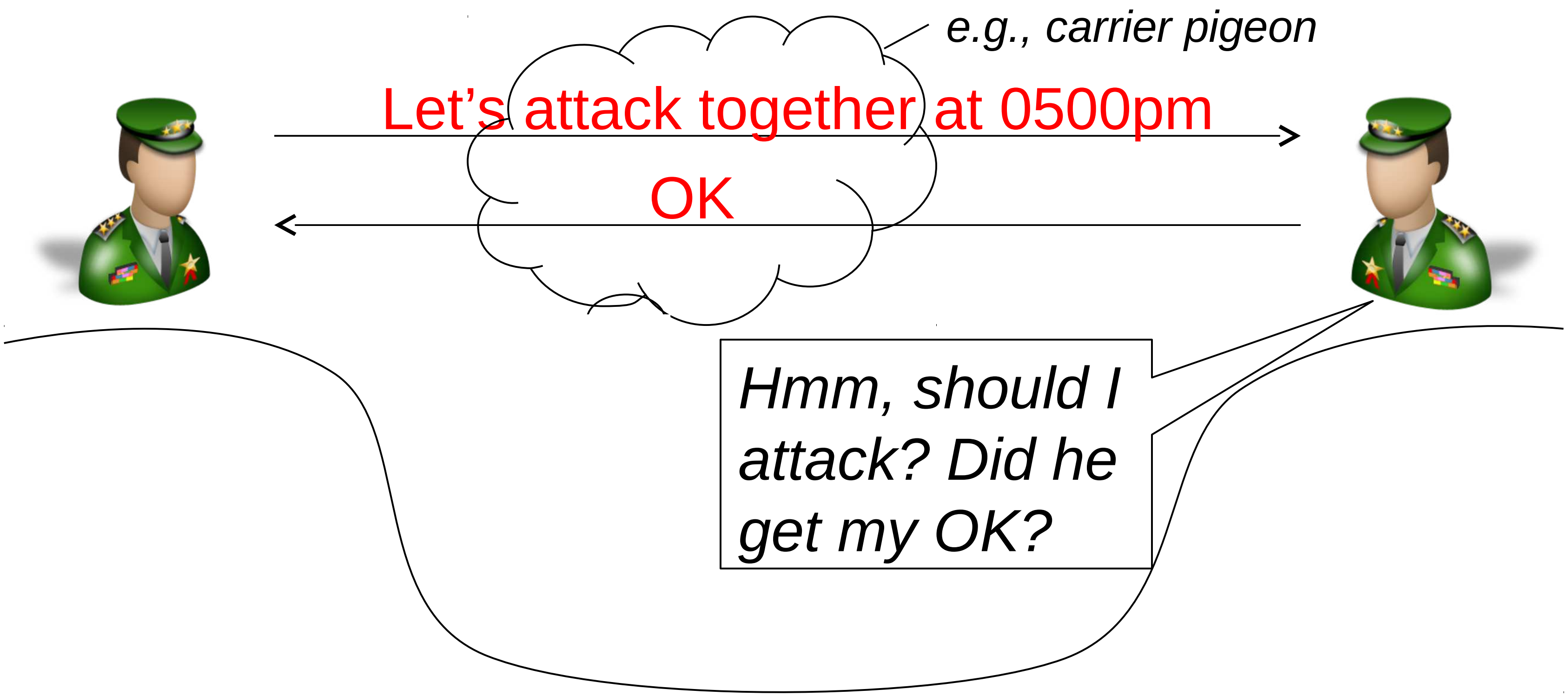
- Actually, one could build this using mutexes
  - Build semaphore using mutex
  - Build cond variable using semaphore + mutex
  - But, boy, it’s a mind-bender to do that – that’s why you want higher level of abstraction

Synchronization in Distributed Systems

- As we've already seen in YFS Lab, distributed systems have similar issues:
  - Multiple processes on different machines share the same resource: the data (or a printer, or the user's screen, …)

- Synchronization is even hairier in distributed systems, as you have to worry about failures, not just overlappings
  - E.g.: when you get a lock, you may not even know you've gotten it ☺

Analogy: The Generals' Dilemma



- Same with distributed systems: machines need to agree on how to progress via an unreliable medium
- Things can get even messier if generals/machines can also fail (or even worse, go rogue)…

Distributed Synchronization Mechanisms

- Logical clocks: clock synchronization is a real issue in distributed systems, hence they often maintain logical clocks, which count operations on the shared resource
- Consensus: multiple machines reach majority agreement over the operations they should perform *and* their ordering
- Data consistency protocols: replicas evolve their states in pre-defined ways so as to reach a common state despite different views of the input
- Distributed locking services: machines grab locks from a centralized, but still distributed, locking service, so as to coordinate their accesses to shared resources (e.g., files)
- Distributed transactions: an operation that involves multiple services either succeeds or fails at all of them

- We're going to look at all of these in the following lectures

Next Time

- Clocks in distributed systems
  - Clock synchronization problem
  - Logical (protocol) clocks