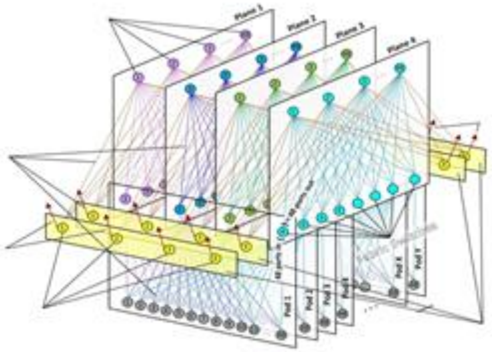
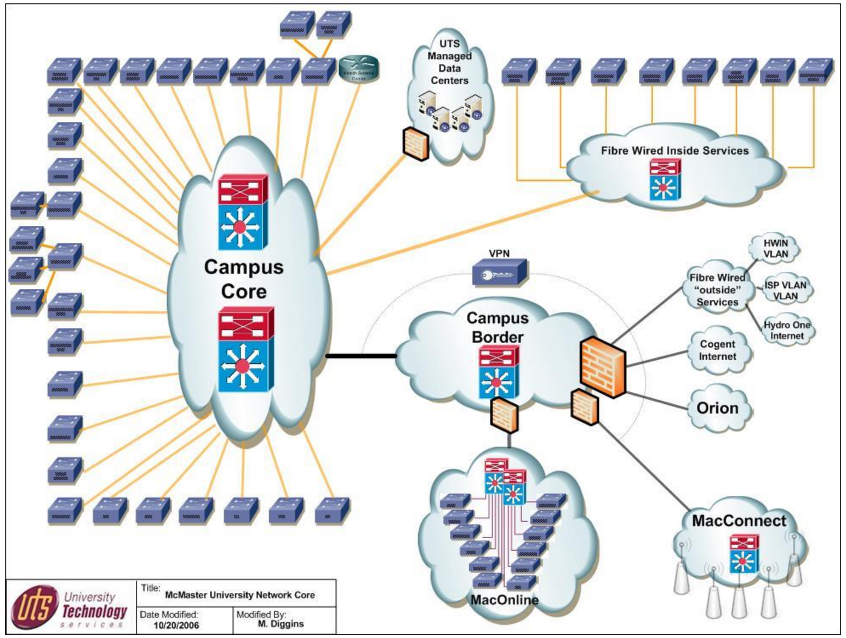


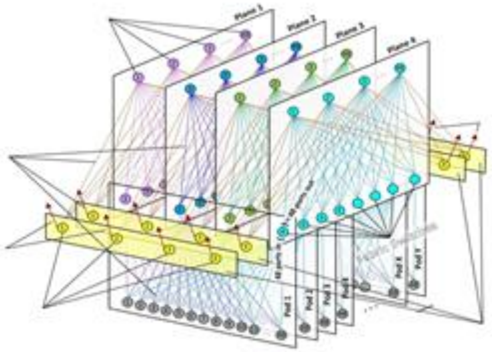
Large Systems:

Design + Implementation + Administration

2024-2025



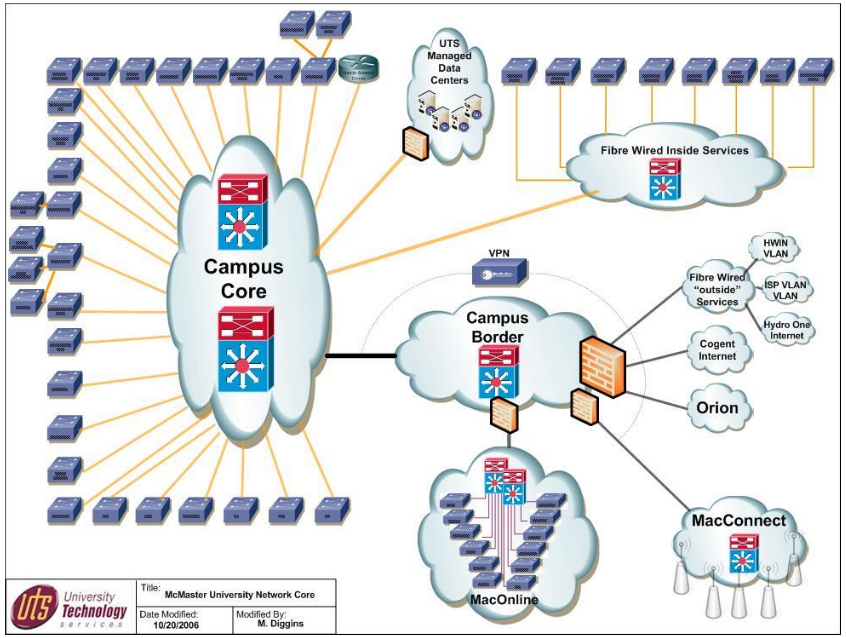
UNIVERSITEIT VAN AMSTERDAM



Large Systems:

Design + Implementation:

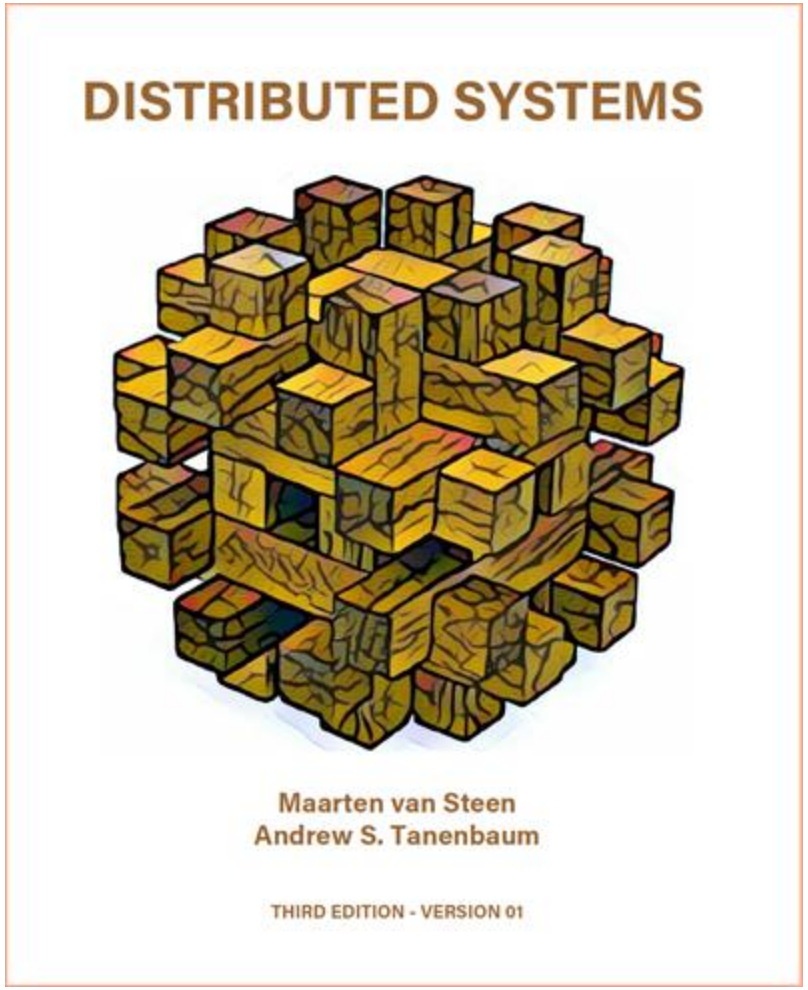
➤ Week3-L5: Communication & Coordination



Shashikant Ilager
shashikantilager.com

11 november 2024

Credits



Slides largely based on "Distributed Systems", 3rd Edition, Maarten van Steen, Andrew S. Tanenbaum

Reading

Chapter 4: Communication

4.1 Introduction

4.2 Remote Procedure Call

4.3 Message-Oriented Communication

4.4 Multicast communication

Chapter 6: Coordination

6.1 Clock synchronization [Optional, background only]

6.2 till p.316, Logical clocks

6.3 Mutual Exclusion

6.4 Election Algorithms [algorithms covered in class]

6.7 Section on Aggregation

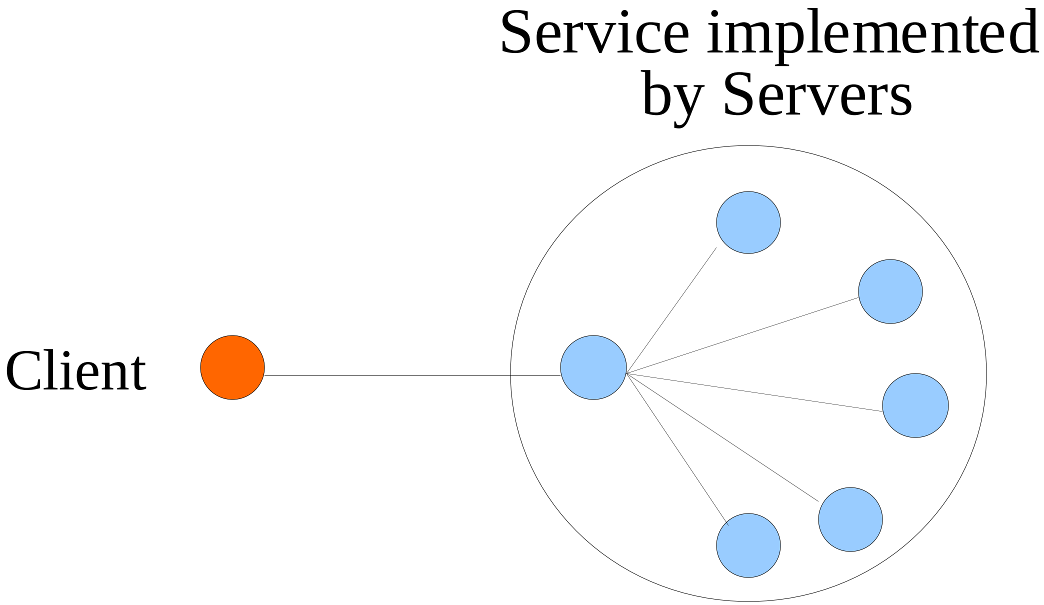
Ref: Distributed Systems, 3rd Ed.

Scaling Techniques

- Bigger machines
- Virtualization
- Asynchronous communication
- Replication & Caching
- Partitioning

Communication (Ch. 4)

- Client-to-Server
  - Hide distribution
- Server-to-Server
  - If multiple Servers provide Service, you need
    1. Group communication
    2. Coordination



Client-Server Communication

- Sending/receiving messages to IP addresses does not hide distribution
- Higher level: Remote Procedure Call
- Client calls procedure (e.g. read()) and gets result
- The fact that read() is executed remotely is hidden
  - Distribution transparency!
  - Natural: implementation of procedure is normally also hidden
- No need to modify the application code
  - Imagine having to rewrite all UNIX tools because the files are now remote...

Case Study: NFS

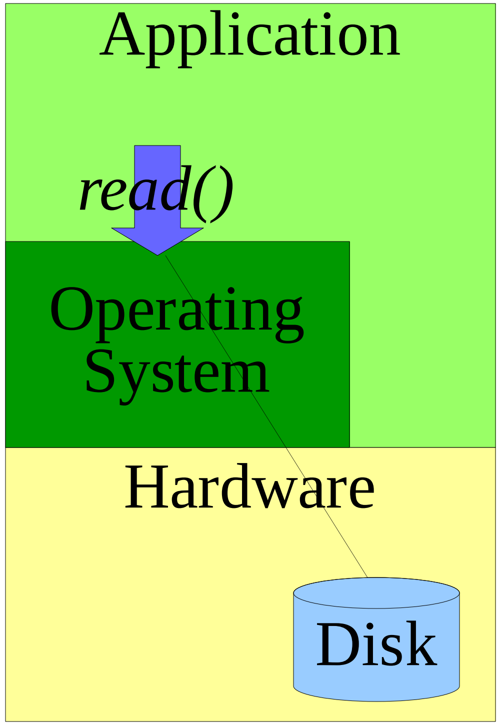
- Network File System
- Read/write files from server as if on local disk
- Local file API in C programming language:
  - `int open(char *filename, int oflag)`
  - `int read(int fd, char *buffer, size_t nbytes)`
  - `int write(int fd, char *buffer, size_t nbytes)`
  - `int close(int fd)`
- Goal: implement this API remotely

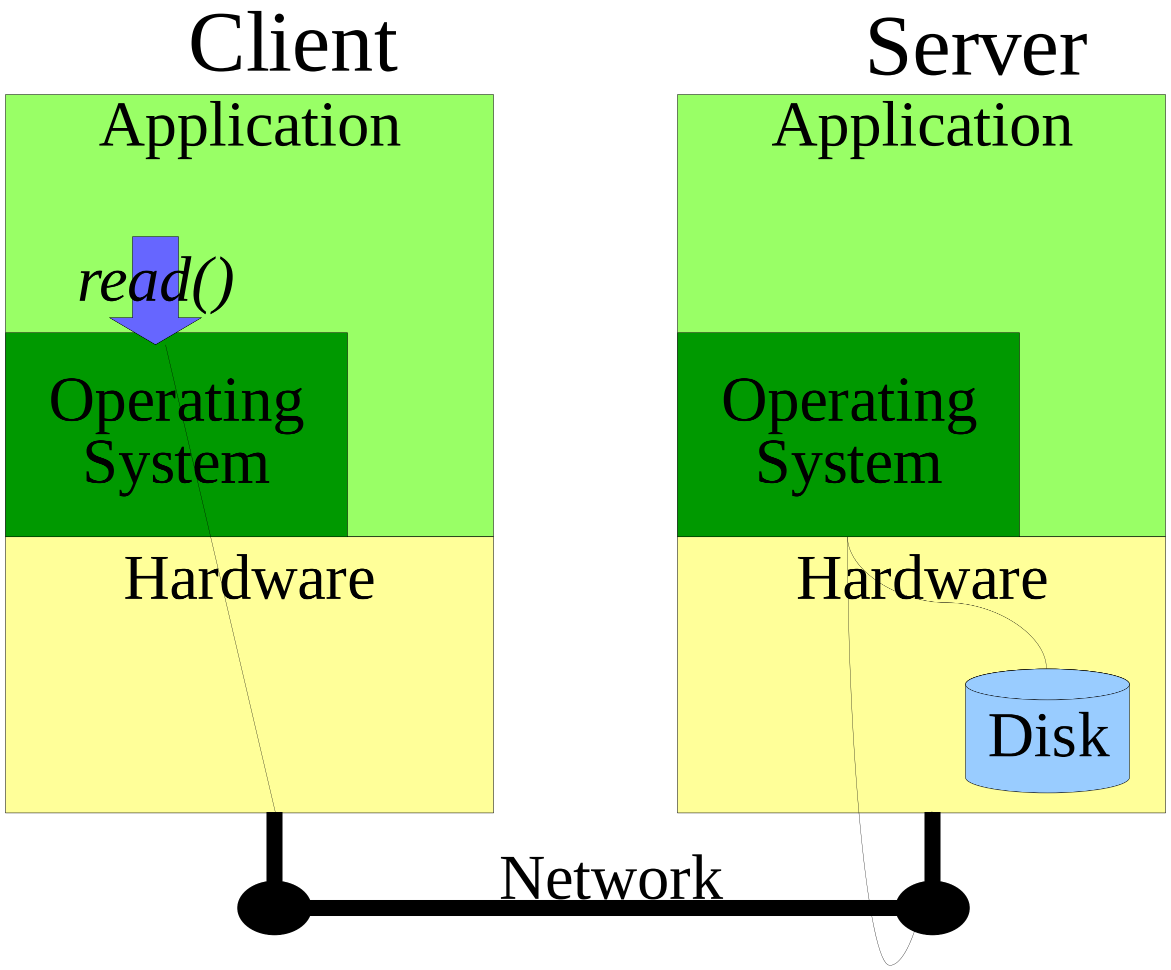
Accessing Files in UNIX

```
char buffer[1024];
int fd = open(“/home/ls24/file.ods”, O_RDWR);
int ret = read(fd, buffer, 1024);
for (int i=0; i<1024; i++)
     buffer[i]=’a’;
ret = write(fd, buffer, 1024);
ret = close(fd);
```

Which part of the file is overwritten?

Local → Remote





Shipping Procedure Calls

Procedure call

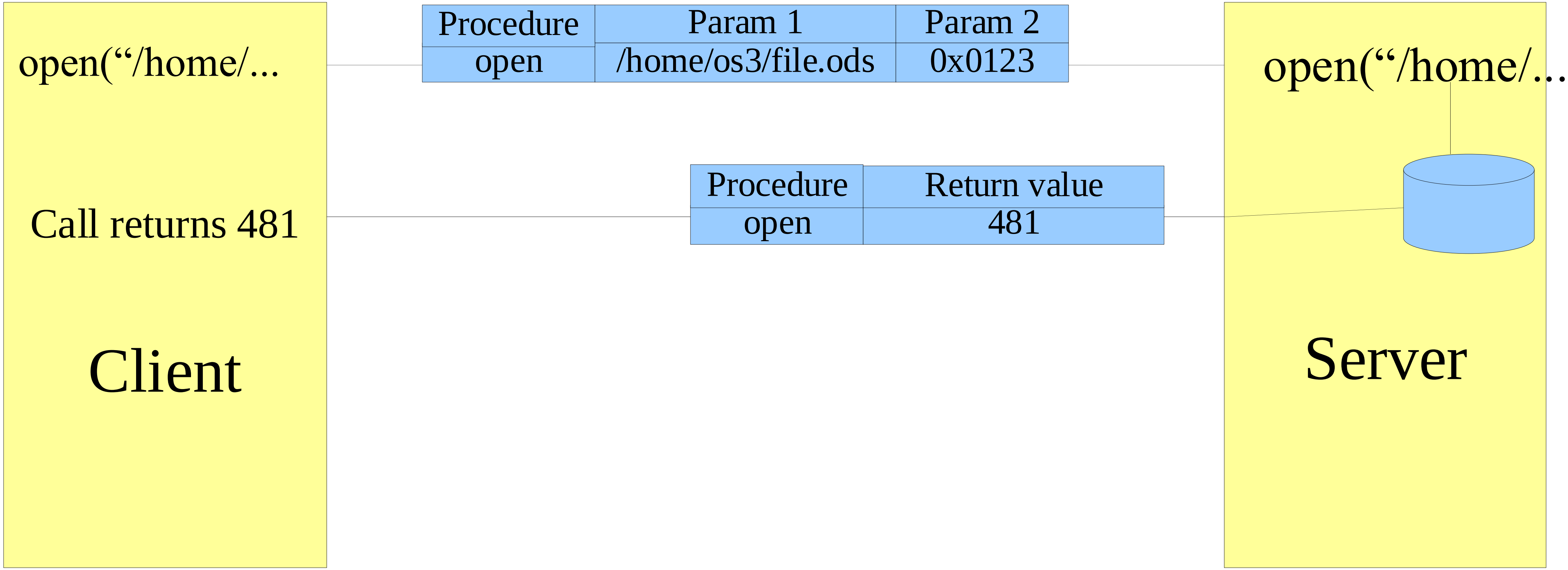
```
int fd = open("/home/ls24/file.ods", 0x0123)
```

Network packet

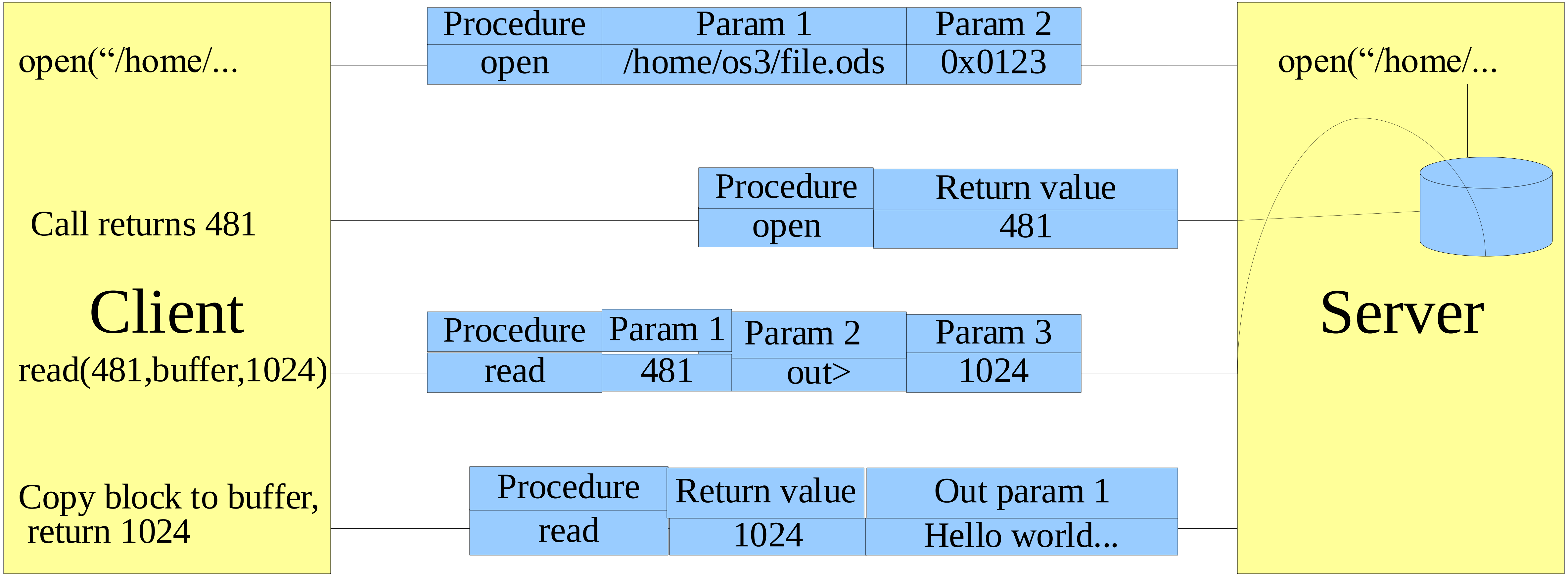
Procedure	Param 1	Param 2
open	/home/ls24/file.ods	0x0123

Parameter marshalling

Remote Procedure Call



Remote Procedure Call (cont'd)



NFS/RPC Issues (1)

- What if NFS server goes down?
  - Soft mount: return error on client call (after a timeout)
  - Hard mount: client call hangs till server back
- Server is stateful:
  - Must remember:
    - filename → file descriptor mapping!
    - offset in file!
  - Must be saved during graceful reboot
  - What about ungraceful reboot, aka crash?

Want Stateless Servers

- Why? No special measures, server can be cattle
- How? Client must send required state in request
- E.g. Compare
  - read(481,buffer,1024)
  - read("/home/os3/file.ods",offset,buffer,1024)

Use cases

CERN IT Department

cloudscaling

Pets are given names like pussinboots.cern.ch

They are unique, lovingly hand raised and cared for

When they get ill, you nurse them back to health

Cattle are given numbers like vm0042.cern.ch

They are almost identical to other cattle

When they get ill, you get another one

Future application architectures tend towards Cattle but Pet support is needed for some specific zones of the cloud

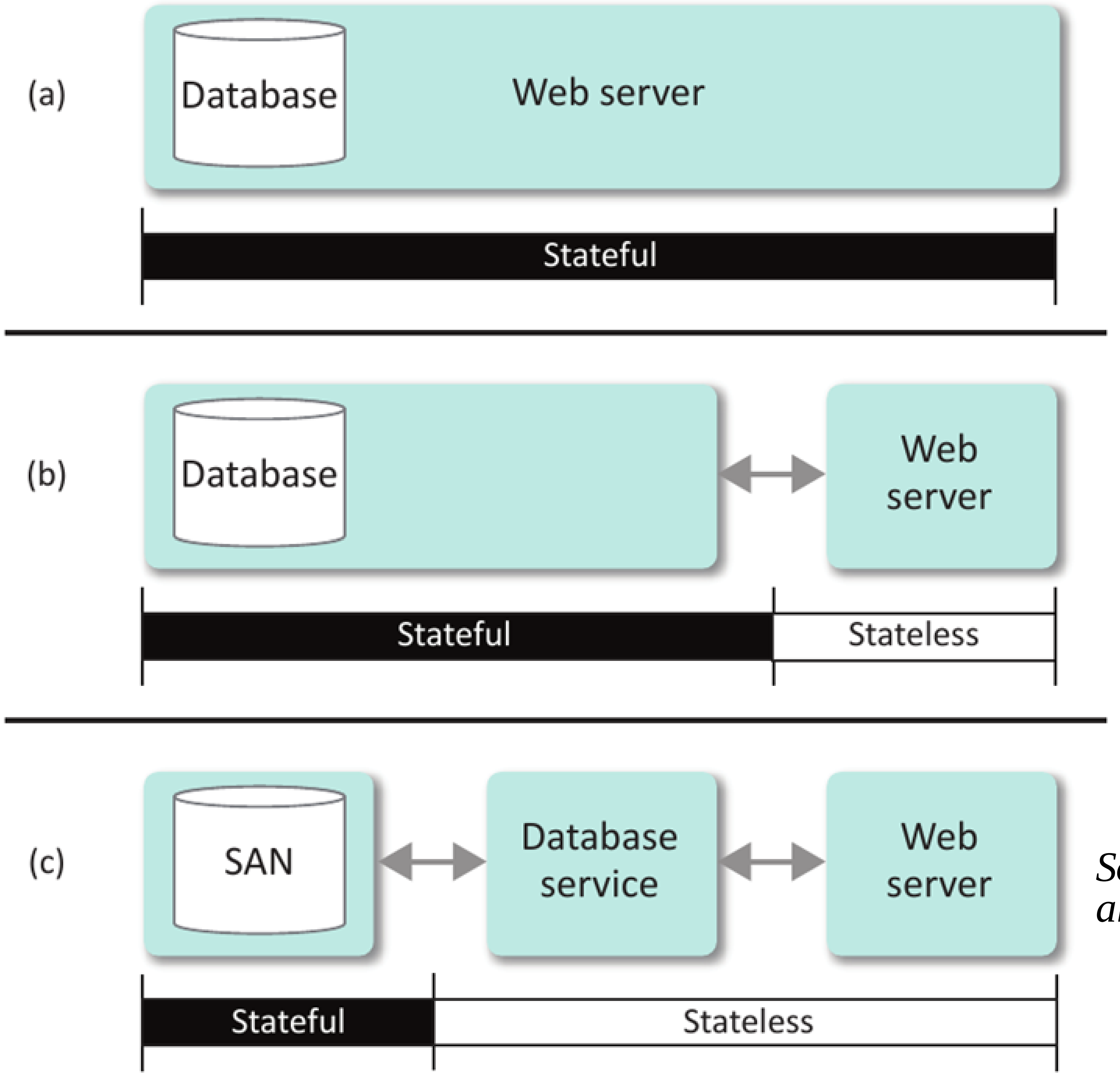
Want Stateless Servers (cont'd)

- The design principle for Web Servers:
  - Representational State Transfer (REST)
  - Operations stateless
  - Transient state sent in client cookies,
  - If unavoidable: Persistent state in DB, distributed cache or both

- See Limoncelli, Chapter 3

See: https://en.wikipedia.org/wiki/Representational_state_transfer

"Pets into Cattle on the Web"



Source: Limoncelli et al. 3rd Ed.

RPC Issues: Parameter marshalling

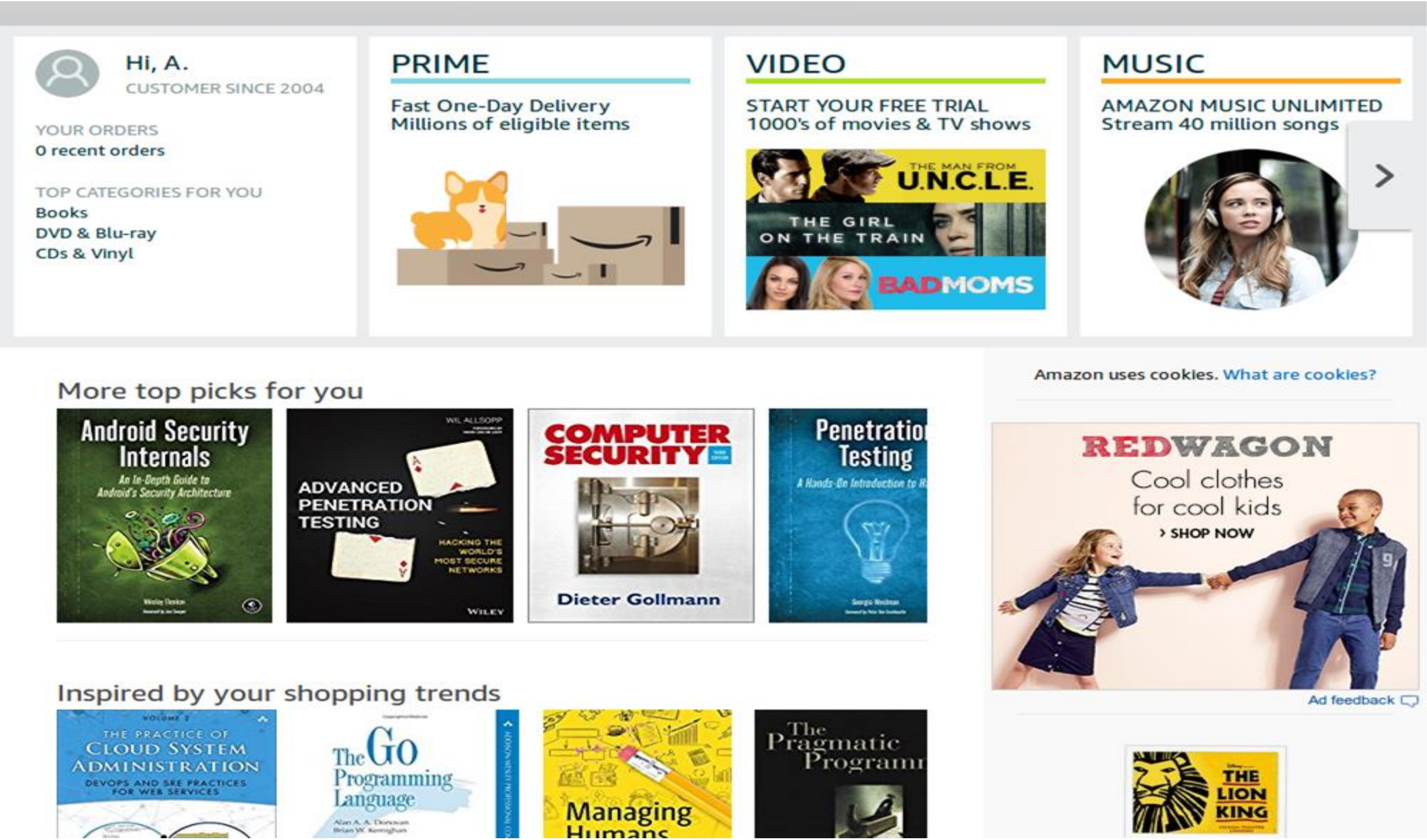
- 3 kinds of parameters
  - in: only read by server
  - out: only written by server
  - inout: read and written by server
- Which is which?
  - int read(int fd, char *buffer, size_t nbytes)
  - int write(int fd, char *buffer, size_t nbytes)
- Data format in the packet needs to be clear, e.g.
  - Integer representation (=big endian, or “network byte order”)
  - String representation

NFS/RPC Issues (3)

- What if parameters are complex data structures?
- What if RPC messages are lost?

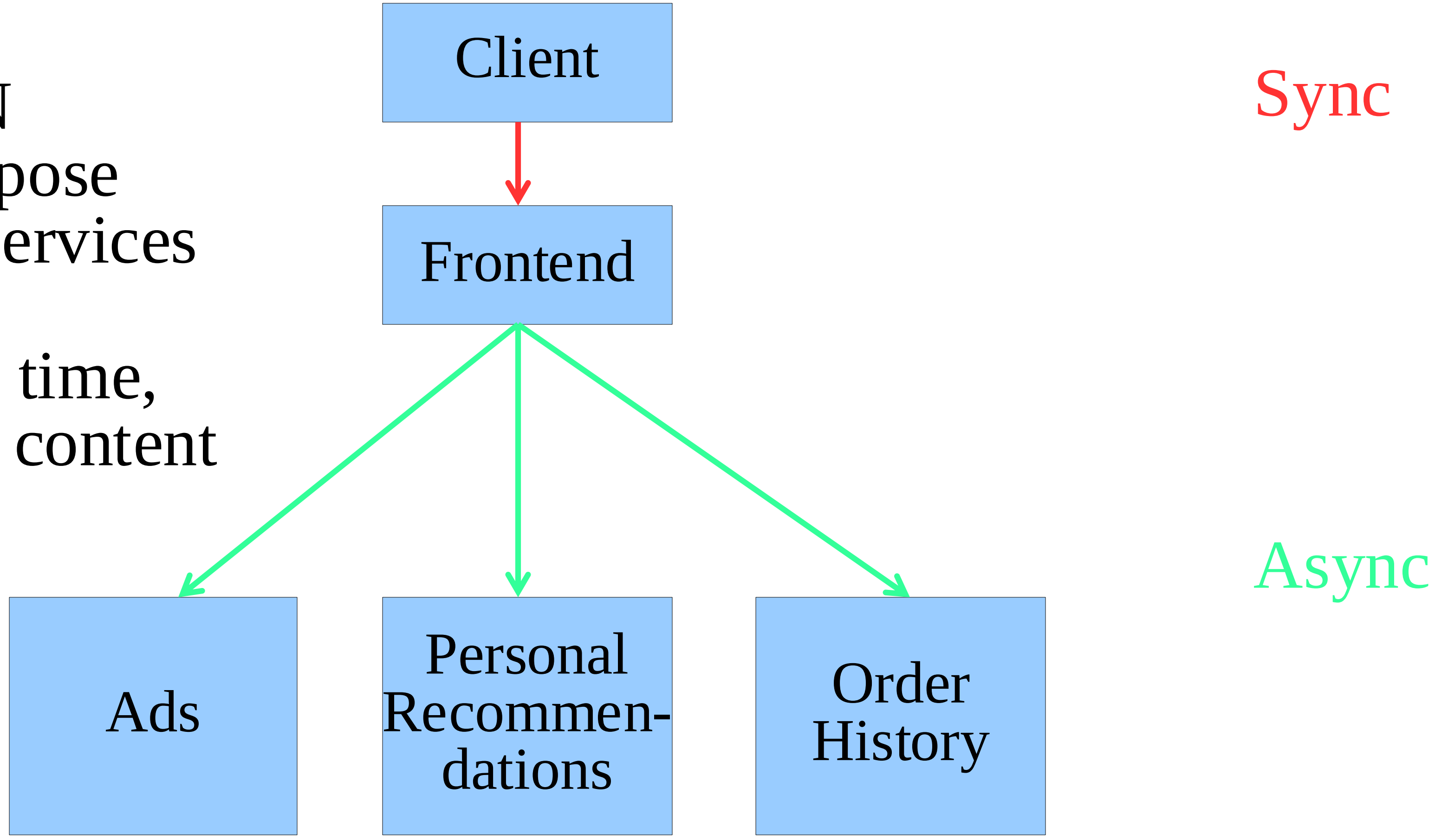
NFS/RPC Issues (3)

- RPC is synchronous
  - Client must wait for reply
- Alternatives:
  - Asynchronous RPC
  - Deferred synchronous RPC
    - Client continues after call
    - Client interrupted via callback when results in
  - Basis for Scalable communication


Hi, A.
CUSTOMER SINCE 2004
YOUR ORDERS
0 recent orders
TOP CATEGORIES FOR YOU
Books
DVD & Blu-ray
CDs & Vinyl
PRIME
Fast One-Day Delivery
Millions of eligible items
VIDEO
START YOUR FREE TRIAL
1000's of movies & TV shows
THE MAN FROM U.N.C.L.E.
THE GIRL ON THE TRAIN
BAD MOMS
MUSIC
AMAZON MUSIC UNLIMITED
Stream 40 million songs
More top picks for you
Android Security Internals
An In-Depth Guide to Android's Security Architecture
WIL ALLSOPP
ADVANCED PENETRATION TESTING
HACKING THE WORLD'S MOST SECURE NETWORKS
WILEY
COMPUTER SECURITY
Dieter Gollmann
Penetration Testing
A Hands-On Introduction to H
Amazon uses cookies. What are cookies?
REDWAGON
Cool clothes for cool kids
› SHOP NOW
Ad feedback
Inspired by your shopping trends
THE PRACTICE OF CLOUD SYSTEM ADMINISTRATION
DEVOPS AND SRE PRACTICES FOR WEB SERVICES
The Go Programming Language
Alan A. A. Donovan
Brian W. Kernighan
Managing Humans
The Pragmatic Programm
THE LION KING

Amazon Web Architecture

Frontend allocates N milliseconds to compose the page. Calls sub services asynchronously.
When reply is not in time, replace with generic content



Client-Server Communication II

- Remote Procedure Call one mechanism
- Object-Oriented variant:
  - Invoke methods on (possibly) remote objects
  - Java RMI
- Other: Message-Oriented Middleware

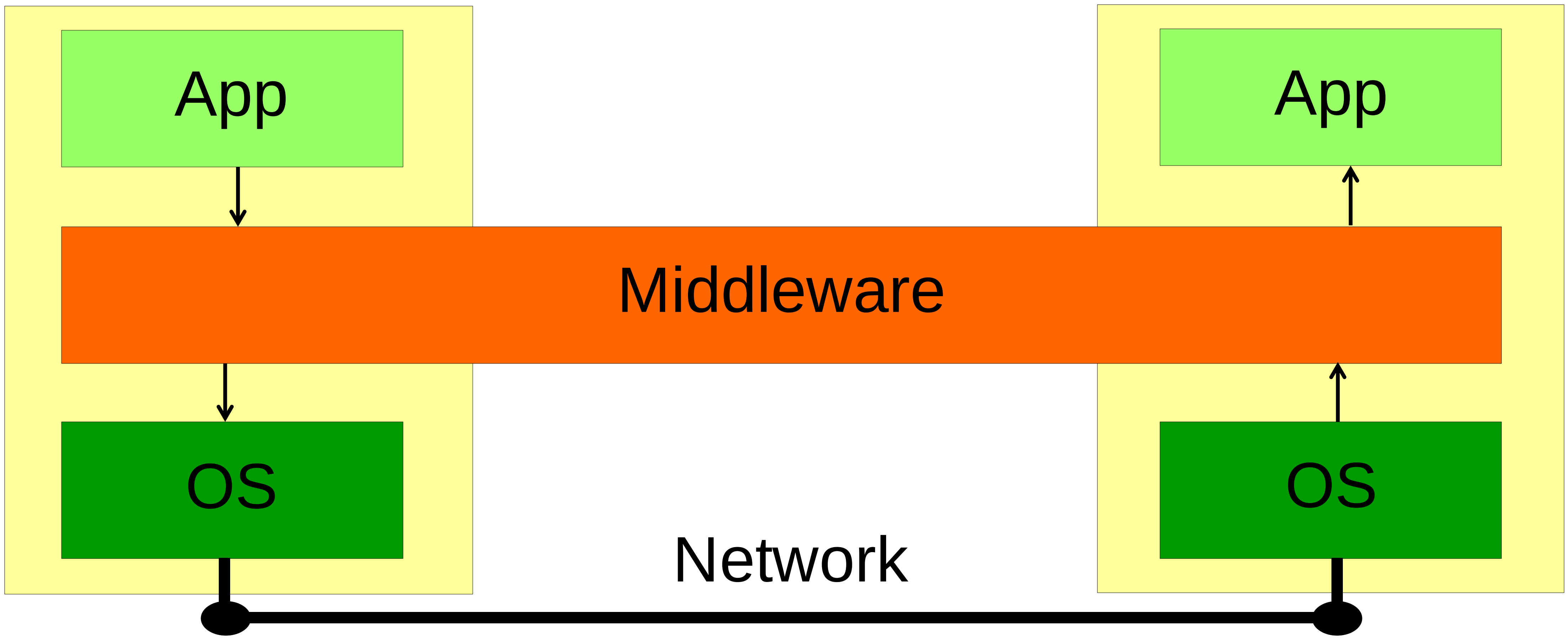
Types of Communication

	Transient	Persistent
Synchronous	RPC	Message-Oriented Middleware*
Asynchronous	Async RPC	Message-Oriented Middleware*

Message-Oriented Middleware (1)

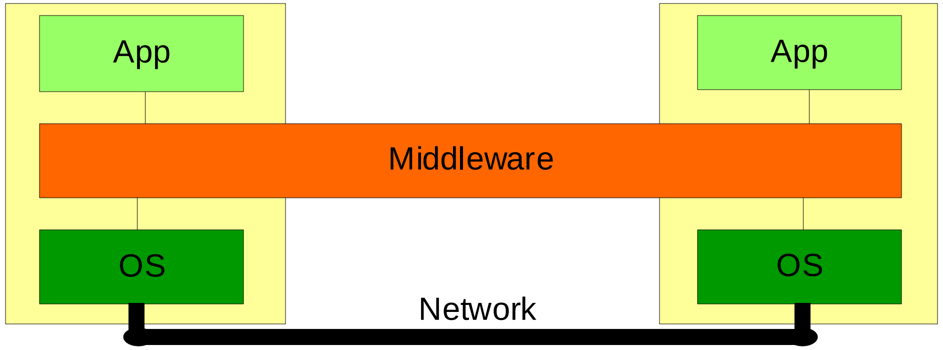
- Clients send messages to abstract destinations
- Middleware layer stores and transports messages
- Messages get pushed to destination, or pulled
- Destination need not be online
  - Asynchronous!
- One-to-One
- One-to-Many

Middleware



Middleware (cont'd)

- Enables parts of application A to communicate with other parts of A
- Enables application A to communicate with application B
- Hides differences in hardware and operating systems
- Responsible for transparencies (distribution, failure, etc)
- Examples:
  - RPC layer (turning calls into network packets)
  - Message-Oriented Middleware



Middleware (cont'd)

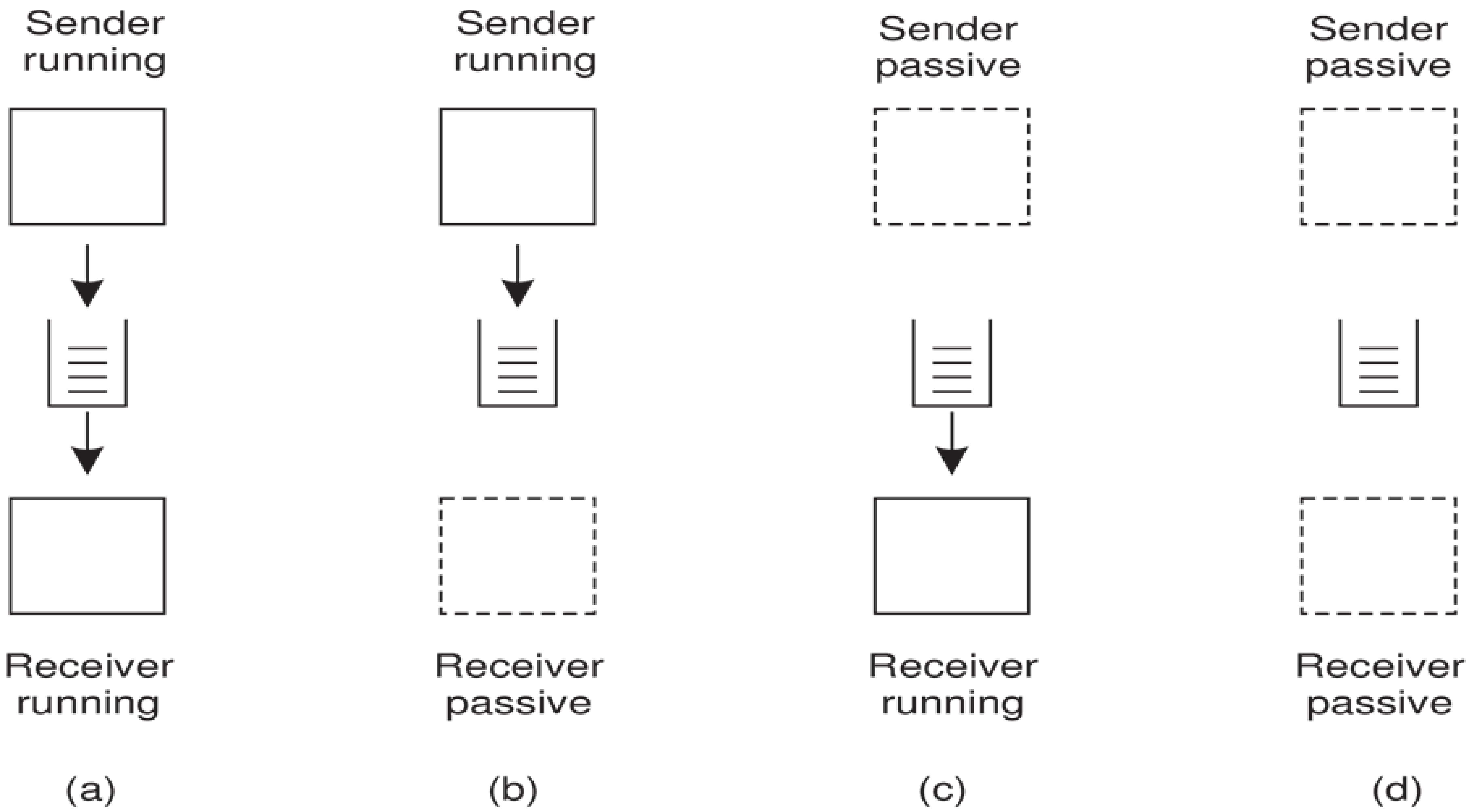
- Hard to write Middleware layers that are generic because
  1. transparency problems hard to solve, or
  2. expensive to solve, so performance becomes poor.

- Need to look at per-application optimizations / solutions.

Message-Oriented Middleware (2)

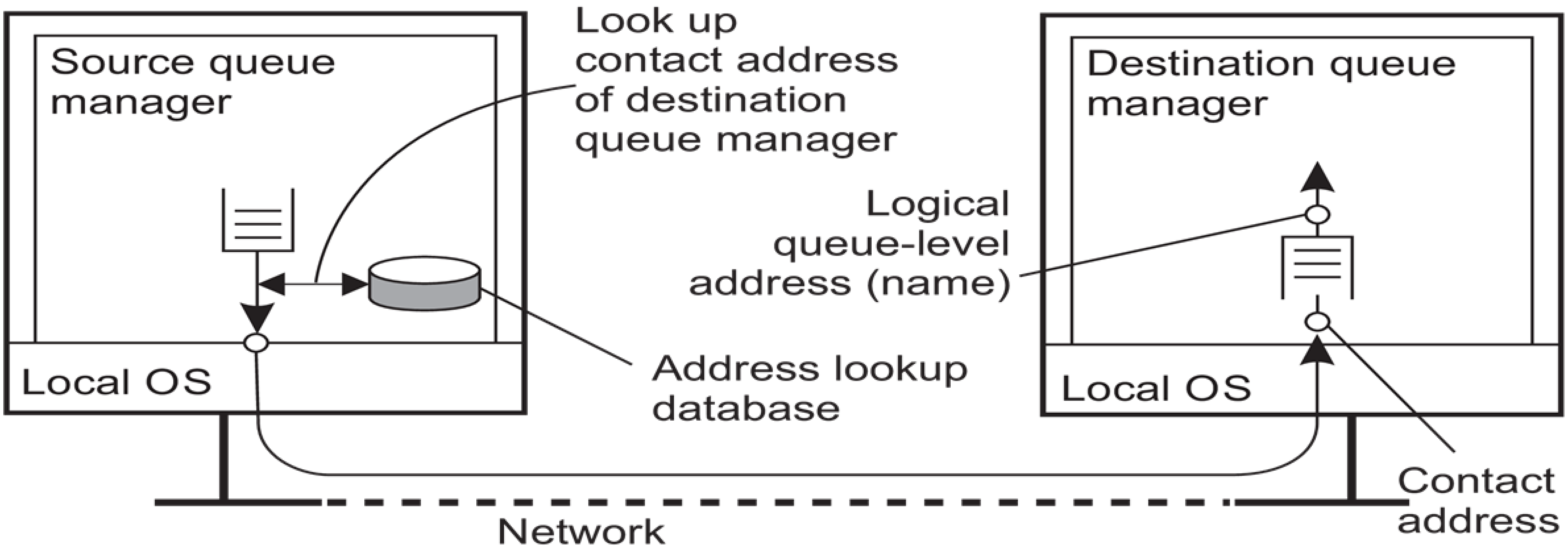
- API: Clients can
  - PUT messages in queue
  - GET messages from the queue (blocking)
  - POLL for messages in queue (non-blocking)
  - NOTIFY: set callback for message arrival
- Middleware stores messages in intermediary servers
- Messages addressed to queue not IP address
- Middleware maps location-independent queue name to IP address
  - Relocation transparency

MOM: Send/Receive Decoupled



Source: Maarten van Steen, "Distributed Systems", 2017

MOM: Decouple Queue and Location

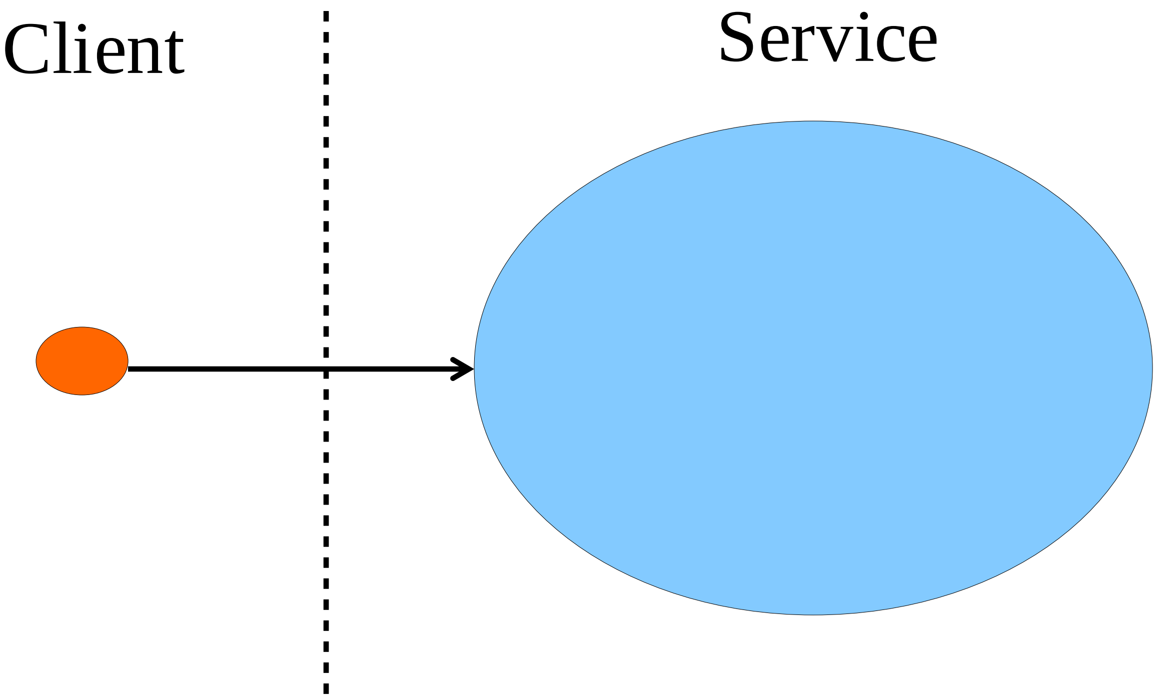


Source: Maarten van Steen, "Distributed Systems", 2017

MOM Summary

- Asynchronous:
  - Send/receive decoupled
- Location transparency
  - Queue and IP address decoupled
- Successful class of middleware
  - Advanced Message Queue Protocol (AMQP)
  - MQ Telemetry Transport (MQTT)
  - eXtensible Messaging and Presence Protocol (XMPP)

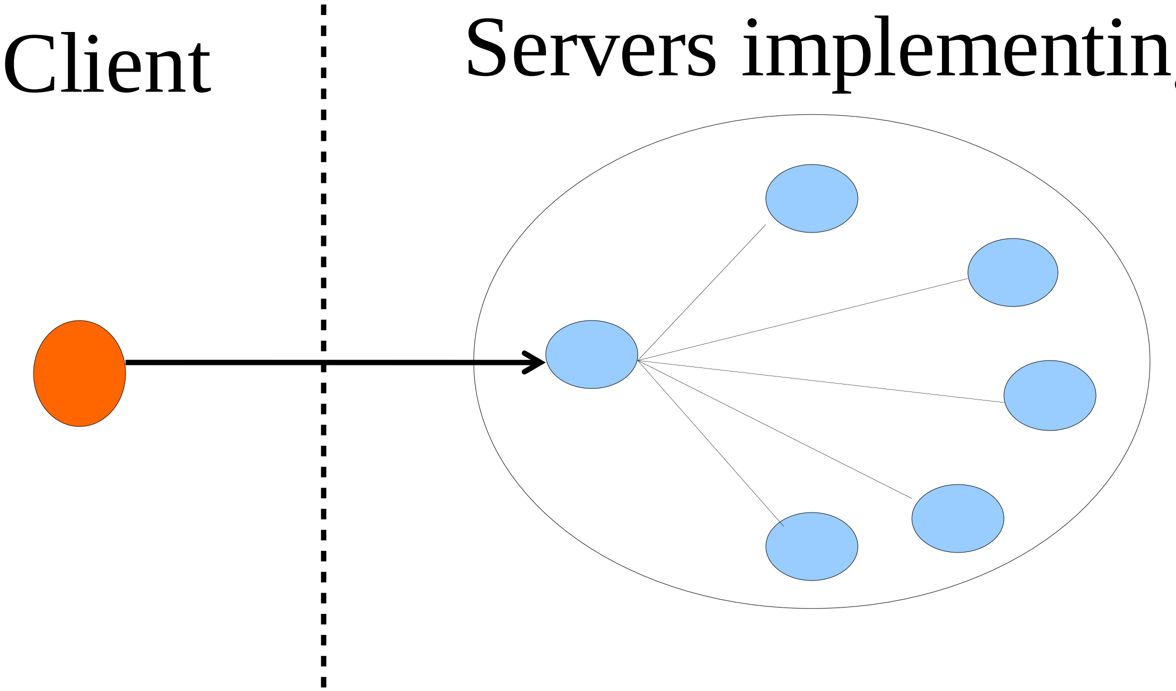
Communication Model



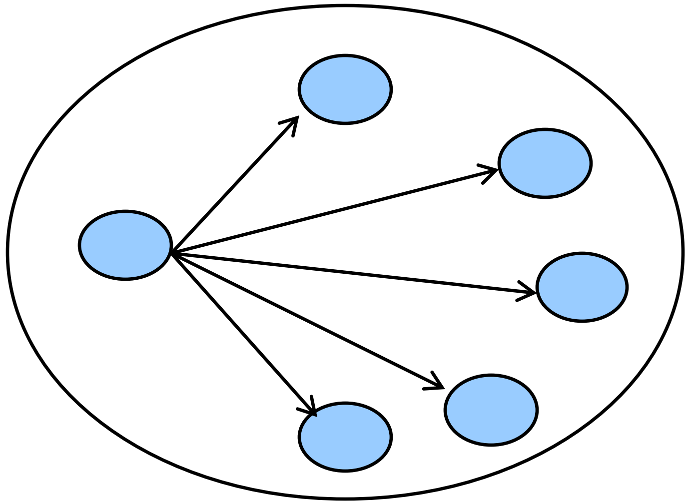
Communication Model

Client

Servers implementing a Service



Server-to-Server Communication



- Need group communication

- Layer 3 multicast (224.0.2.0+)
  - Does not work globally
    - "First, forwarding multicast traffic imposes a great deal of protocol complexity on network service providers.
    - Second, core network infrastructure exposes a far greater attack surface, with particular vulnerability to denial-of-service attacks"
  - Offers only best-effort (UDP) service

[https://en.wikipedia.org/wiki/IP_multicast]

Multicast Requirements

- Layer 3 offers only best-effort (UDP) service
- Want reliable
  - Message must reach all members in group
- Want atomic
  - Message must reach all members or none
- Want dynamic groups
  - Makes atomicity hard!
- Want ordering on messages
  - Causal or Total order

Message Order

- Total
  - All processes see all messages in the same global order
- Causal
  - If message m2 is based on info from message m1, all processes must see m1 before m2

Requirements Hard to Achieve at Scale

- Reliable
  - Acknowledgements (ACKs) swamp sender
  - NACKs mean keeping messages, till when?
- Atomic & dynamic
  - Need to know group members at time of send
  - What if group is huge
  - What if members crash?
- Ordering
  - How to order messages in a scalable way?

Coordination (Ch.6)

- Ordering of messages special case of a group needing to coordinate their actions
- Other examples:
  - Agree on time
  - Mutual Exclusion aka Locking = only one member can perform action
  - Election = select one member to be Primary

Agree on Time

- Use physical clocks
  - Connect servers to an Atomic clock
  - Directly
  - Indirectly
    - Network Time Protocol (NTP)
    - GPS (Atomic clocks in space)
  - Accuracy good enough for some applications to coordinate actions
    - Google Spanner: worldwide database synced via GPS

https://cloud.google.com/spanner?hl=en

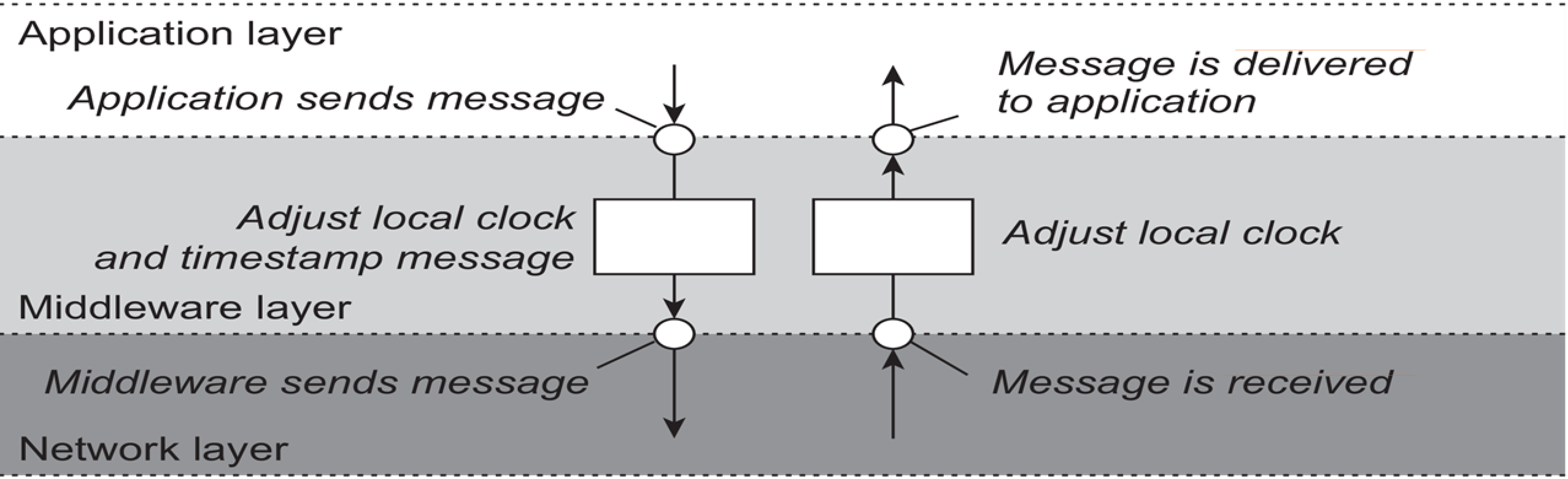
Agree on Time (cont'd)

- Time used to order events
- Observation: Exact time not important, the order is
- Use a Logical Clock to order events
- $C(a)$ = logical timestamp of event a
- Order:
  - If event a happens before b in a process then $C(a) < C(b)$
  - If event a is sending a message and b is receiving that message then $C(a) < C(b)$
- Can be used to impose total order on all messages

Lamport Logical Clocks

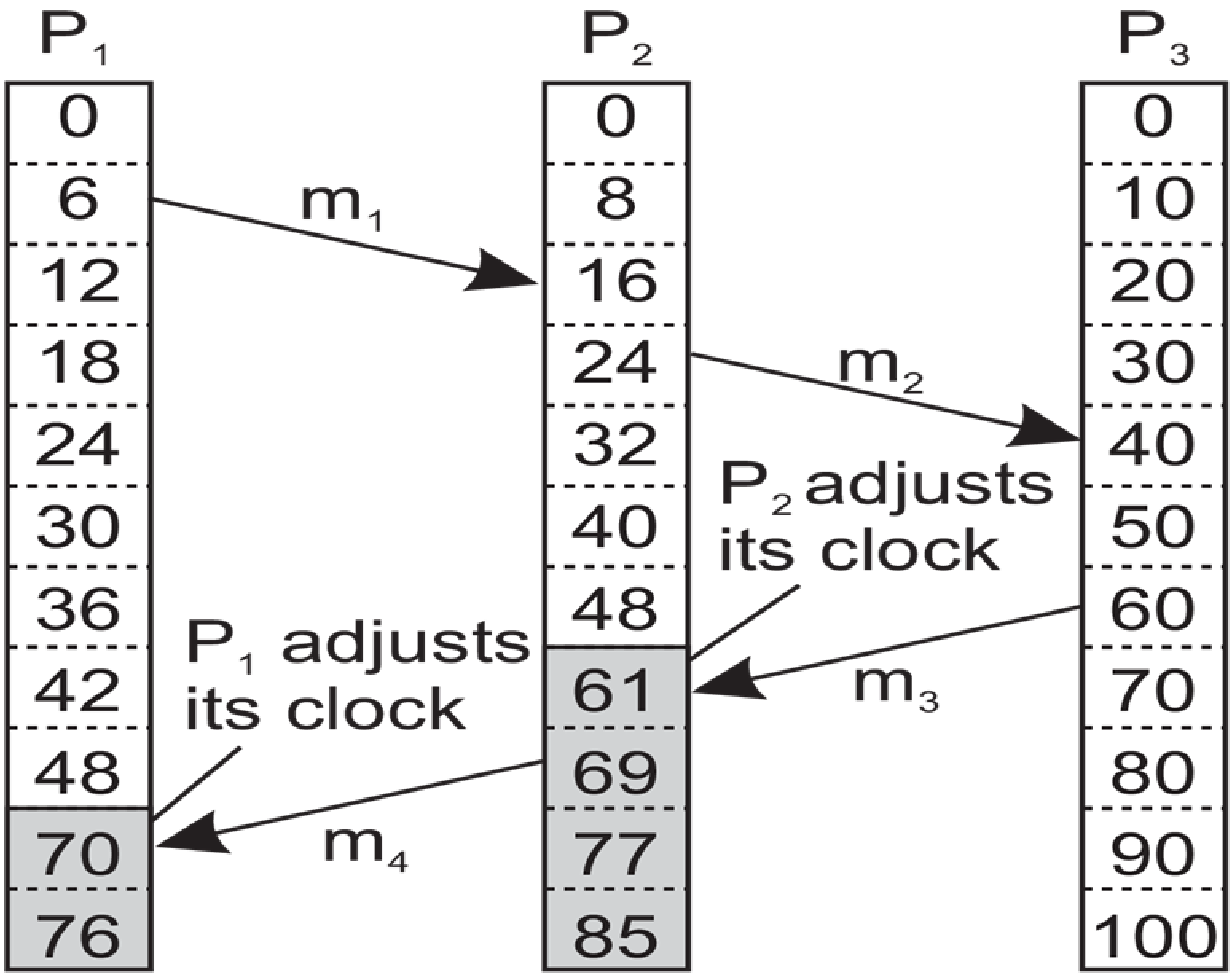
- Each process maintains logical clock C
- On event: $C = C + 1$
  - Sending message
  - Delivering message to application
  - Other
- All messages sent timestamped with C
- On receipt: $C = max(C, ts(m))$

Logical Clock Middleware



Receipt != Delivery

Source: DS3, Fig. 6-9



Source: DS3, Fig. 6-8b

Logical Clocks and Total Order

- Logical Clocks can be used to totally order messages
- Each message carries timestamp
- When message is received, each process sends an ACK to all processes
- Process can deliver message with lowest timestamp to application when ACKs received from all processes

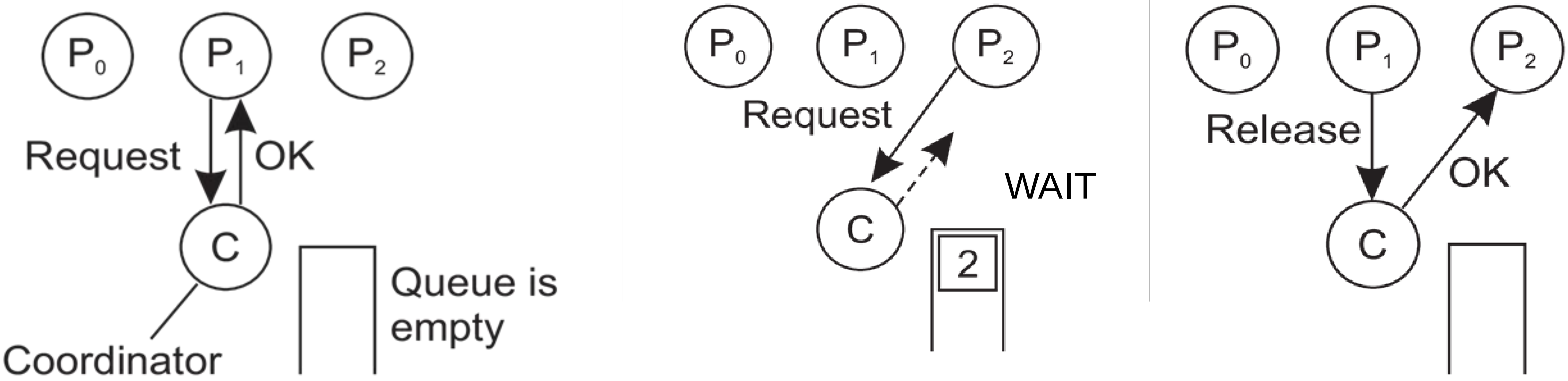
Mutual Exclusion

- = only one member of group can perform action, i.e. hold lock
- E.g. write to replicated database
- Various solutions:
  - Central Lock Coordinator
  - Using Total Order
  - Token Ring

MutEx: Central Coordinator

- Process that wants lock sends REQUEST to coordinator
- If not locked, coordinator sends OK
- If locked, coordinator sends WAIT, remembers
- When process is done, send RELEASE to coordinator
- If other processes waiting for lock, coordinator sends OK to next in queue

MutEx: Central Coordinator (cont'd)



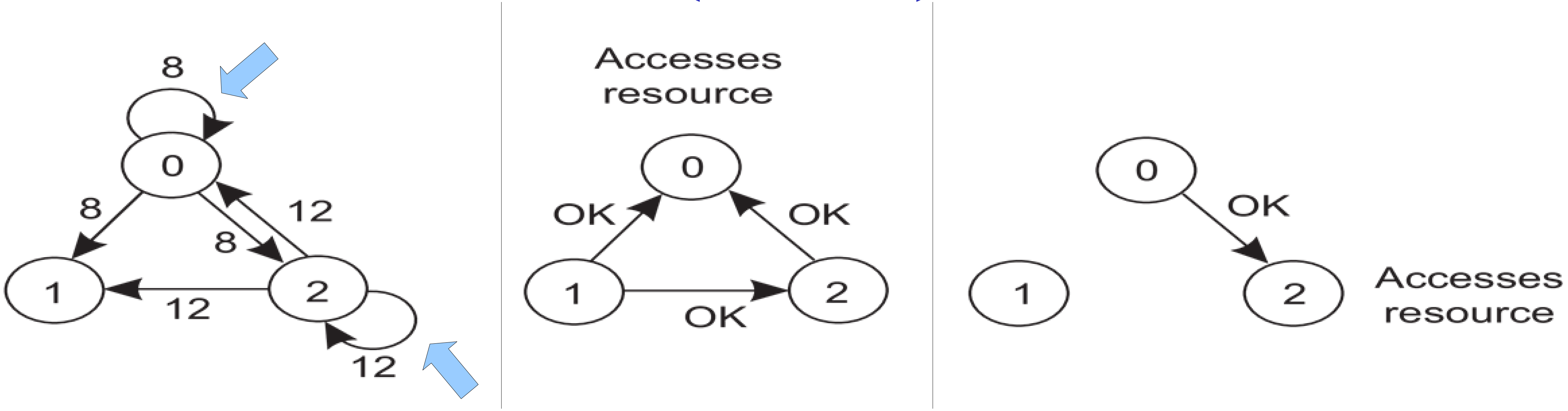
- Pros: Simple, Fair
- Cons: Single Point of Failure, Performance Bottleneck

Source: DS3, Fig. 6-15

MutEx: Total Order

- All messages are timestamped using Logical Clock
- Process A that wants lock sends REQUEST to all
- Process B receives REQUEST and:
  - If not holding lock and does not want lock: send OK
  - If holding lock: no reply, queue
  - If wants lock: compare $ts(\text{REQUEST}_B)$ to $ts(\text{REQUEST}_A)$
    - $ts(\text{REQUEST}_A) < ts(\text{REQUEST}_B) \rightarrow$ send OK to A
    - Else: no reply, queue
- When process A has received OK from all: Perform action
- When done, A sends OK to all in his queue

MutEx: Total Order (cont’d)



- P0 and P2 both interested in lock

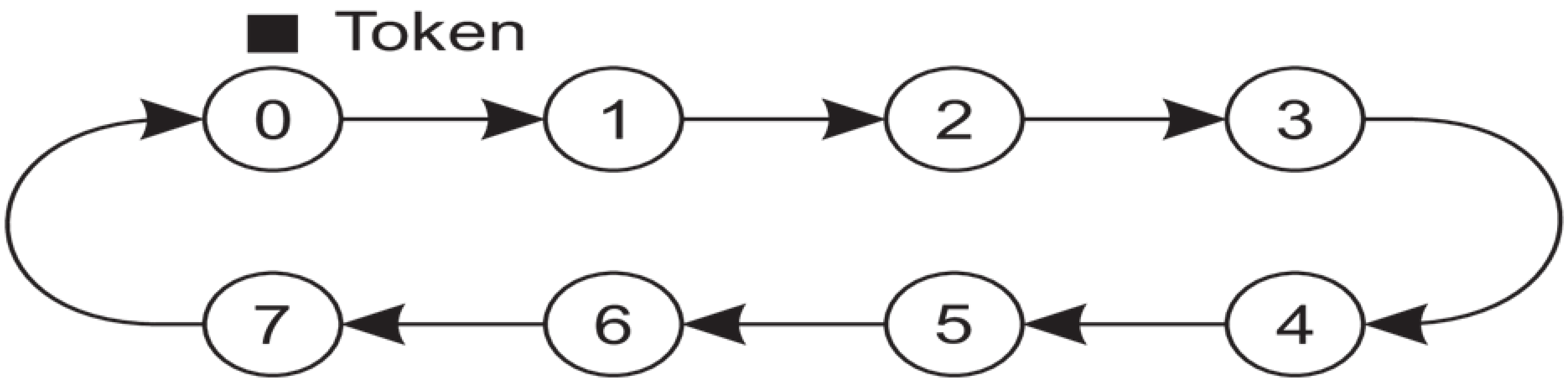
- Pros: No Single Point of Failure
- Cons: N Points of Failure, Need Group Knowledge

Source: DS3, Fig. 6-16

MutEx: Token Ring

- Processes organized in ring
- One process holds token
  - If wants lock: keep token
  - If not: pass to next in ring

MutEx: Token Ring (cont'd)



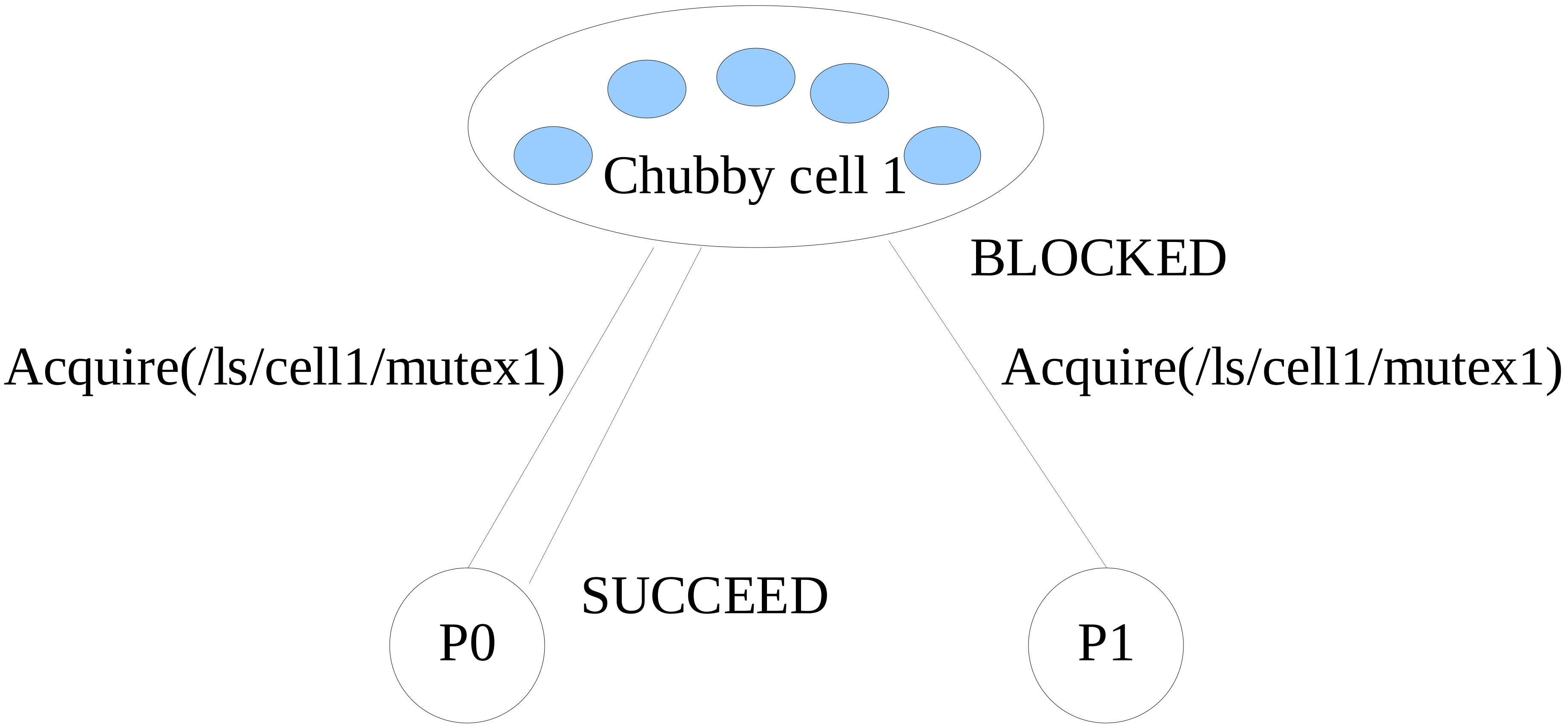
- Pros: Fair
- Cons: Token loss, crashes require group knowledge

Source: DS3, Fig. 6-17

MutEx: Shocking!

- Central coordinator is not looking so bad...

MutEx at Google: Chubby

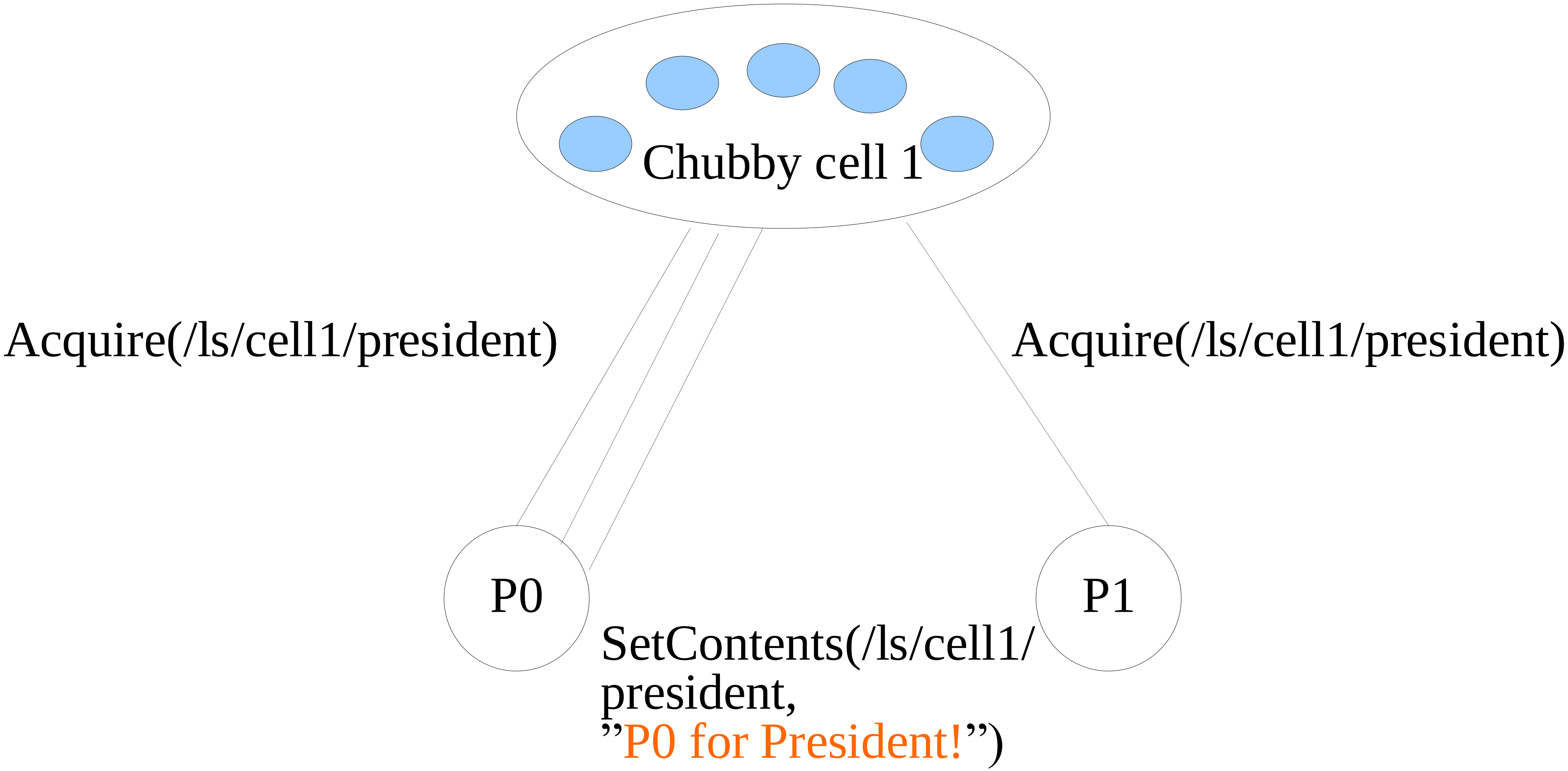


Source: "Distributed Systems" 5th Ed, Coulouris et al, p. 941

Election

- E.g. Select one group member to be Primary

Election using Chubby



Source: "Distributed Systems" 5th Ed, Coulouris et al, p. 942

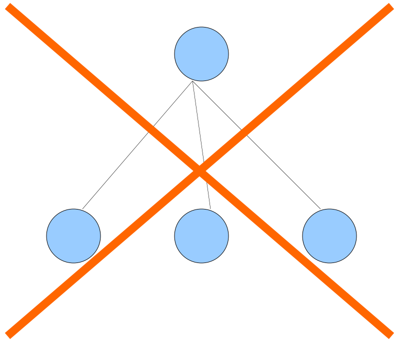
Election using etcd

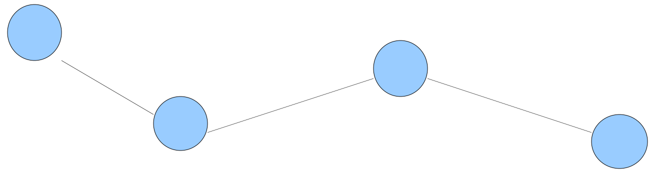
- Processes write (key,value)
  - key = “election key for our distributed application”
  - value = Unique process ID
- etcd writes are atomic
- First write succeeds, process becomes Primary
- Others see key exists, read which process became Primary

Source: https://akazuko.medium.com/leader-election-using-etcd-10301473843c

Gossip-based Coordination

- Gossip
  - There is no global information channel to reach all nodes
  - Nodes only learn information from direct interaction with other nodes
- How?
  - Pick node at random and exchange information
  - Results in rapid information dissemination

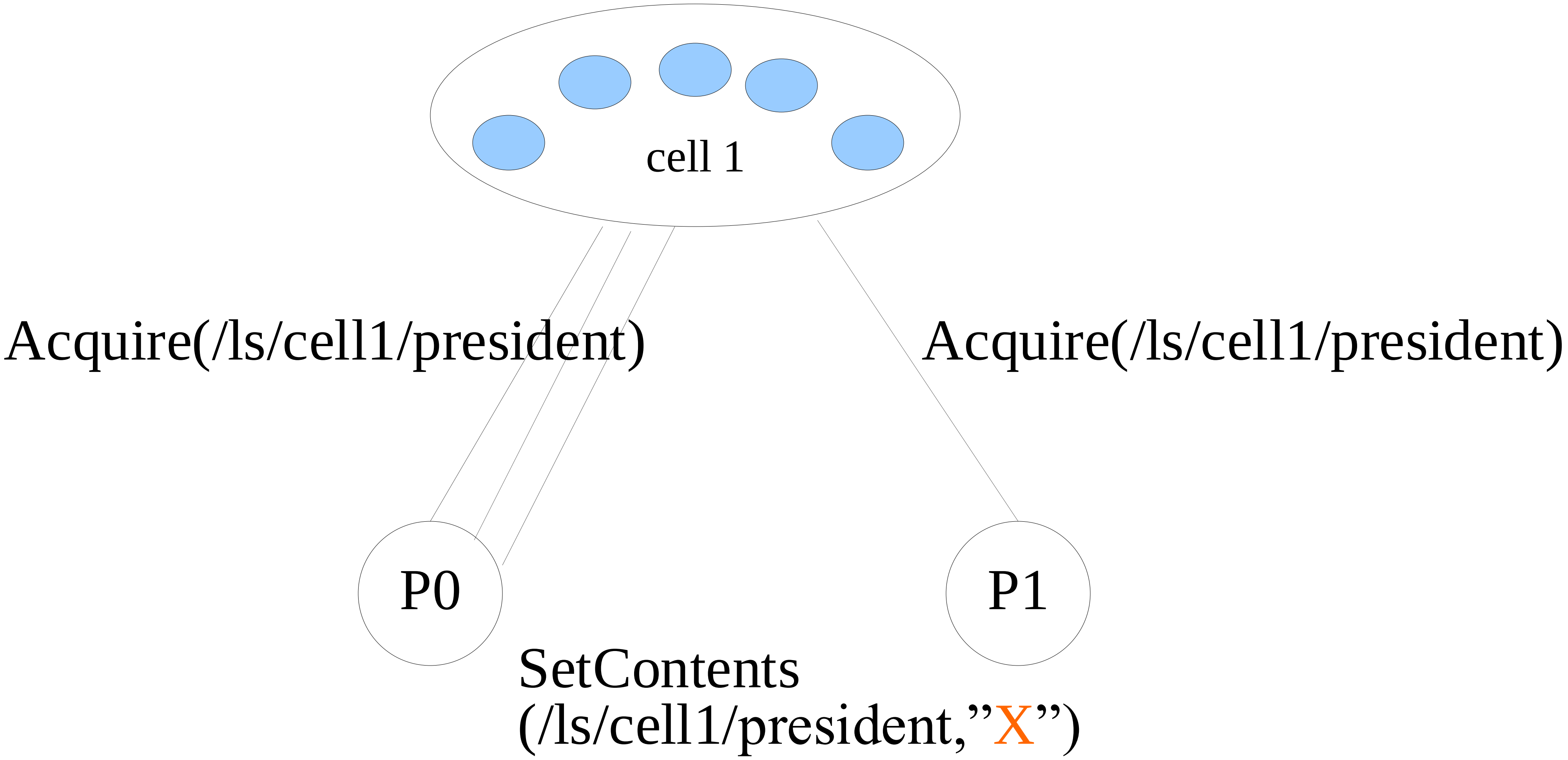




Recap

- What is Middleware?
- What properties are ideal in group communication?
- What nice properties does MOM have?
- What are common downsides of distributed algorithms (e.g. MutEx)?

Recap



Source: “Distributed Systems” 5th Ed, Coulouris et al, p. 942

Scaling Techniques

- Bigger machines
- Virtualization
- Asynchronous communication
- Replication & Caching
- Partitioning