Message Protocols for Distributed Systems

with examples from Socket based Client/Server Systems

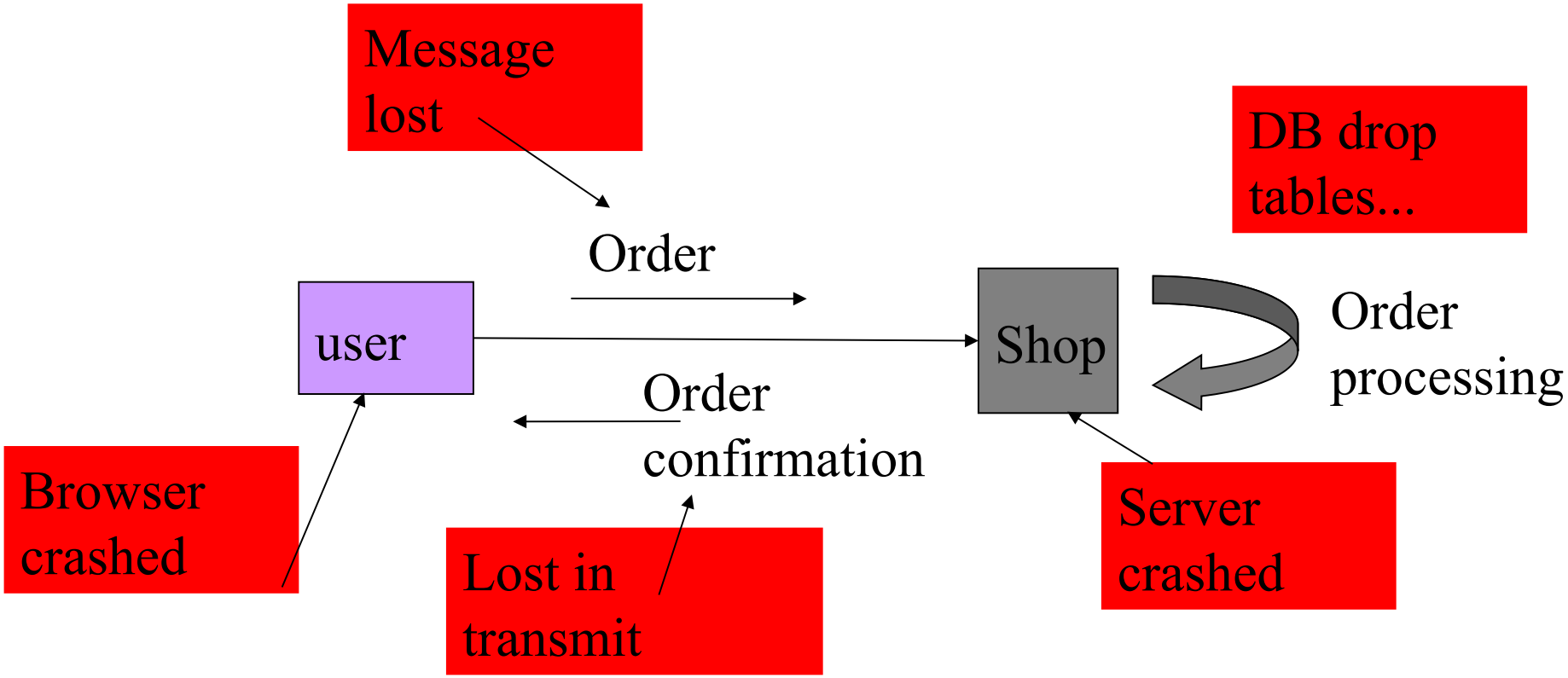
Overview

1. Message Protocols
    - Delivery Guarantees in Point2Point
    - Reliable Broadcast
    - Request Ordering and Causality
2. Programming C/S Systems with Sockets

The Role of Delivery Guarantees

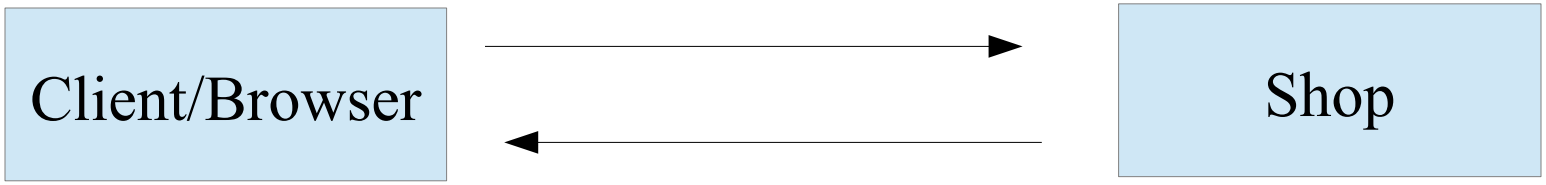
- Problem Scenario: Shop order
- TCP Communication properties
- at-least-once
- at-most-once
- Exactly once?
- Message complexity: the number of messages sent

Shop Order Failure Scenarios



What happens to the order when certain failure types apply? What kind of guarantees do you have? Does TCP Help? What outcomes do you expect?

Concept Exercise: Protocol Design



- You want to receive something eventually
- You don’t want to receive duplicate orders
- Use a browser to place your order
- Use a special fat client to order

Shop Order Failures

Case: Client sends request and receives nothing

a) network problem, server did not receive request, client did not receive response

b) OS problem: OS did receive request but server crashed during work

c) Server problem: Server finished request but response got lost or OS crashed during send.

d) Shop problem: shop out of order, bankrupt, closed.

What are the options for clients and how do they match the possible failures?

Client Options

a) do nothing

network problem, server did not receive request

b) Server problem: OS did receive request but server crashed during work

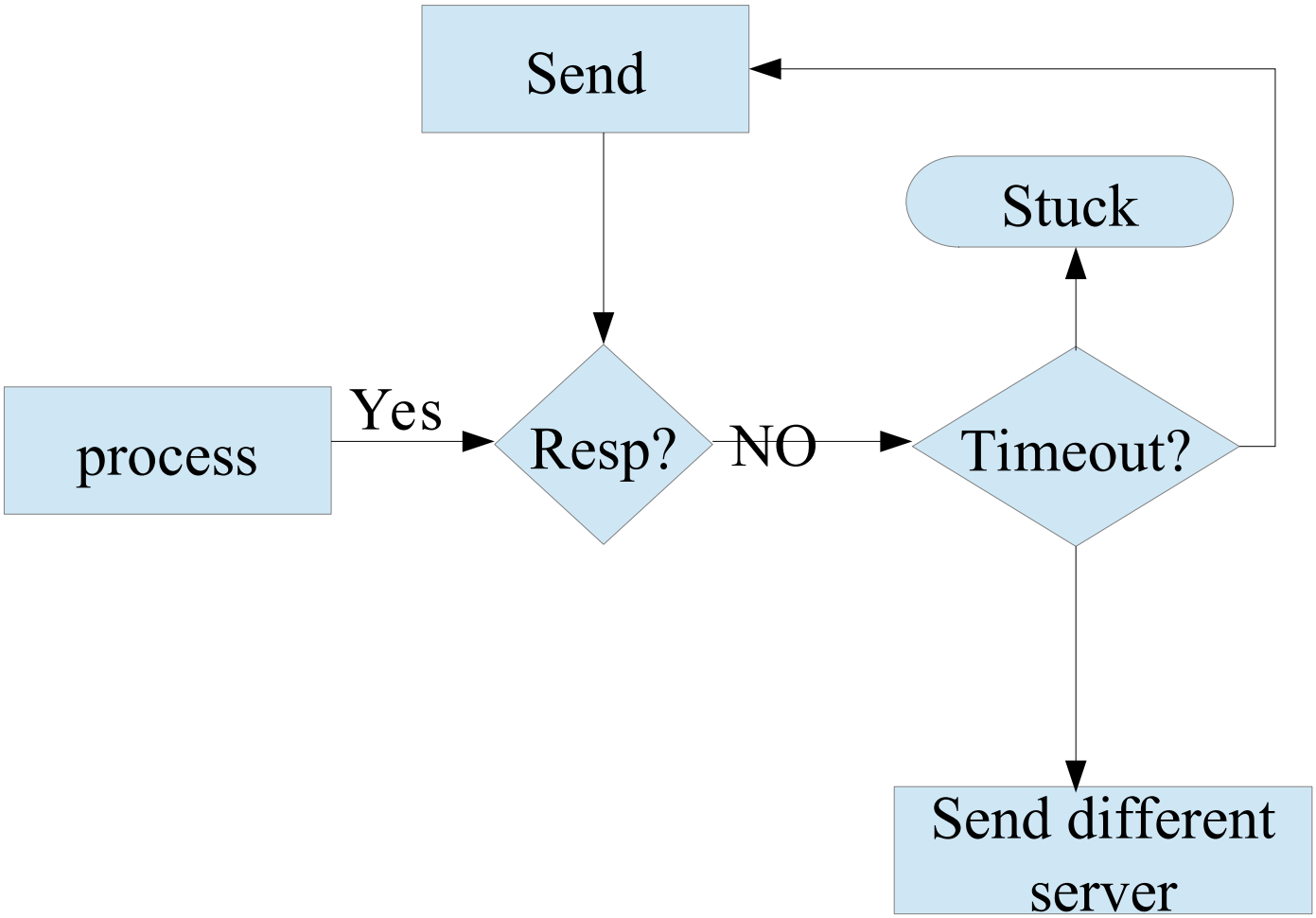
c) OS/Network problem: Server finished request but response got lost or OS crashed during send.

d) Go to different server

e) Chose different shop

Client options: drop request (ok in c), resend request (ok in a and b), send request to different server (ok in a and b). Other client actions lead either to lost or duplicated requests.

Client Options

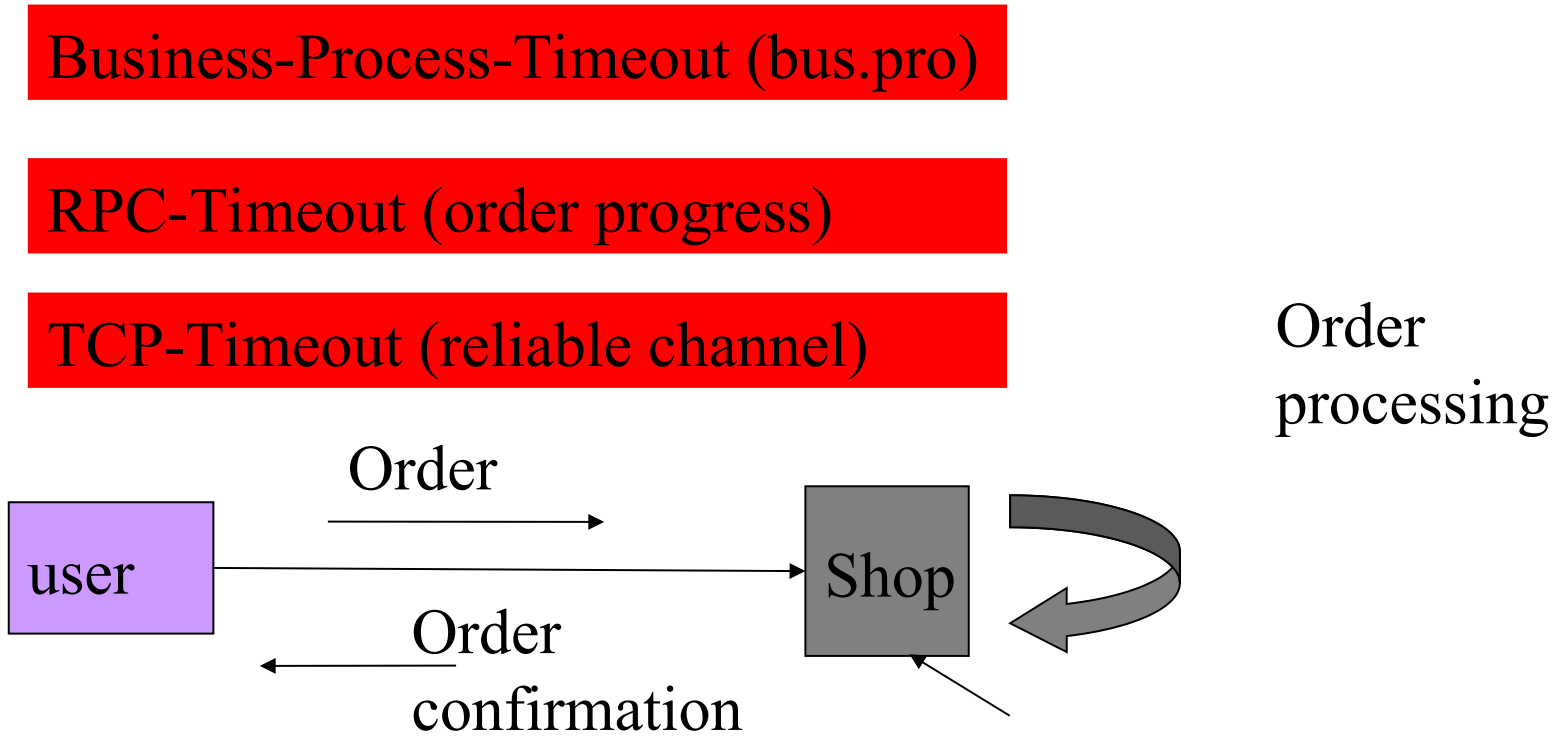


Why is TCP not enough?
TCP communication properties!

- lost messages retransmitted
- Re-sequencing of out of order messages
- Sender choke back (flow control)
- No message boundary protection

These features form a “reliable communication channel”. This does not include proper behavior in case of connection failures! (timeout problem). (Ken Birman, building secure and reliable network applications, chapter 1)

Timeout Levels



In an asynchronous system, we need timeouts on different levels to avoid getting stuck in a process. TCP only covers the lowest layer!

RPC Delivery Guarantees

- Best effort (doesn't guarantee anything)
- At least once (same request several times received)
- At most once (not more than once but possibly not at all)
- Once and only once/ exactly once (does it exist?)

In case of channel break-down TCP does NOT make ANY delivery guarantees. This becomes your job then (or better: a job for your middleware). For a discussion of "exactly once" in the context of real-time streaming software see: http://bravenewgeek.com/you-cannot-have-exactly-once-delivery/

An Important Concept: Idempotent Requests

- Get bank account balance?
- Transer $10 to some account?
- Push elevator button?
- Get /index.html ….?
- Book flight?
- Cancel flight?
- ….

Idempotent means, that a repeated execution of a request will not change state on the server! What kind of delivery guarantee do you need for idempotent service requests?

Idempotency

- Is not a medical condition (Pat Helland)
- The first request needs to be idempotent
- The last request can only be best effort
- Messages may be reordered.
- Your partner may experience amnesia as a result of failures, poorly managed durable state, or load-balancing switch-over to its evil twin.

Pat Helland, http://queue.acm.org/detail.cfm?id=2187821

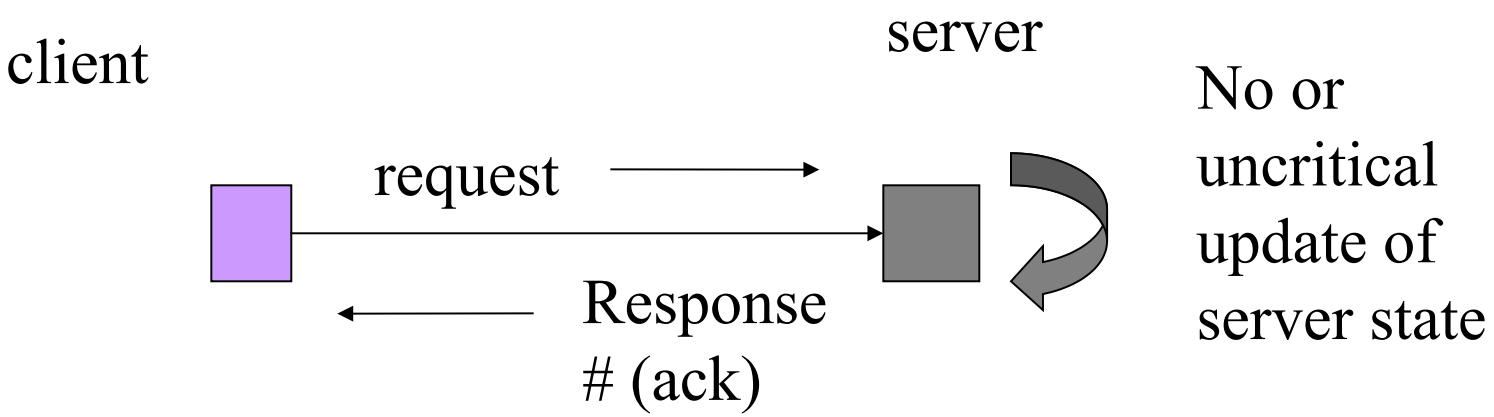
- ….

Idempotency and Server State

- No need to remember a request and its result
- Server can lose its storage
- Concurrent updates might be consistent without concurreny control!

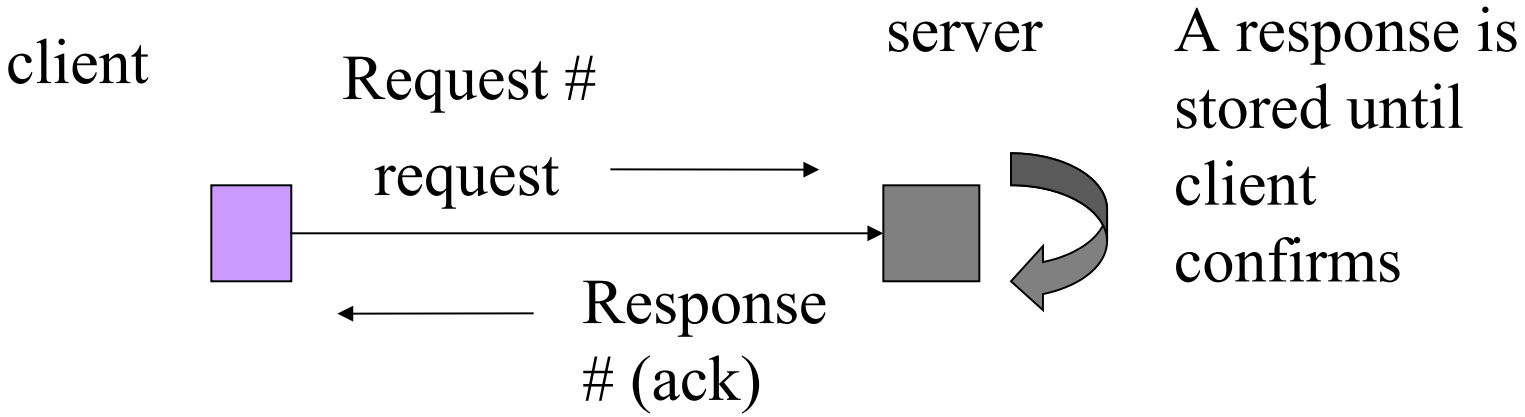
The last point is a critical feature of the new Consistent Replicated Data Types (CRDTs)

„At least once“ implementation for idempotent requests



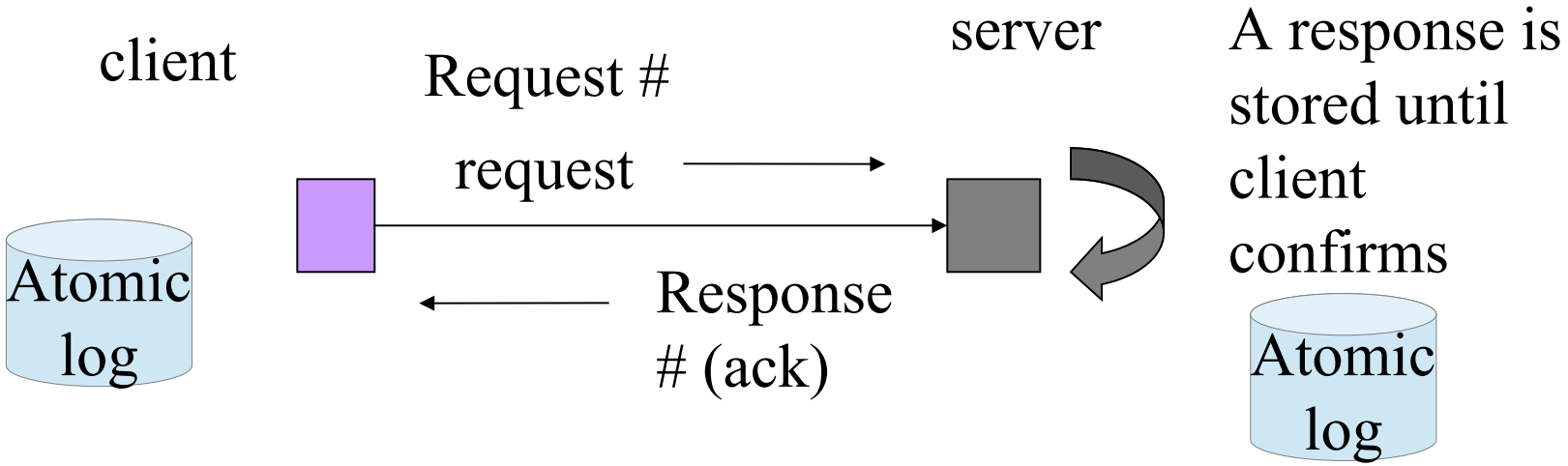
All that is needed is an ack!

„At most once“ implementation for non-idempotent requests



By adding a request number to each request the server can detect duplicate requests and throw them away. The server itself needs to store a response until the client acknowledges that it was received. This creates state on the server!

„Exactly once“ ???



Not possible in asynchronous systems with network failures! But we can do a little bit better with two-phase commit. We need to ensure, that client and server do not forget their decisions! Things like epoch numbers allow garbage collection.

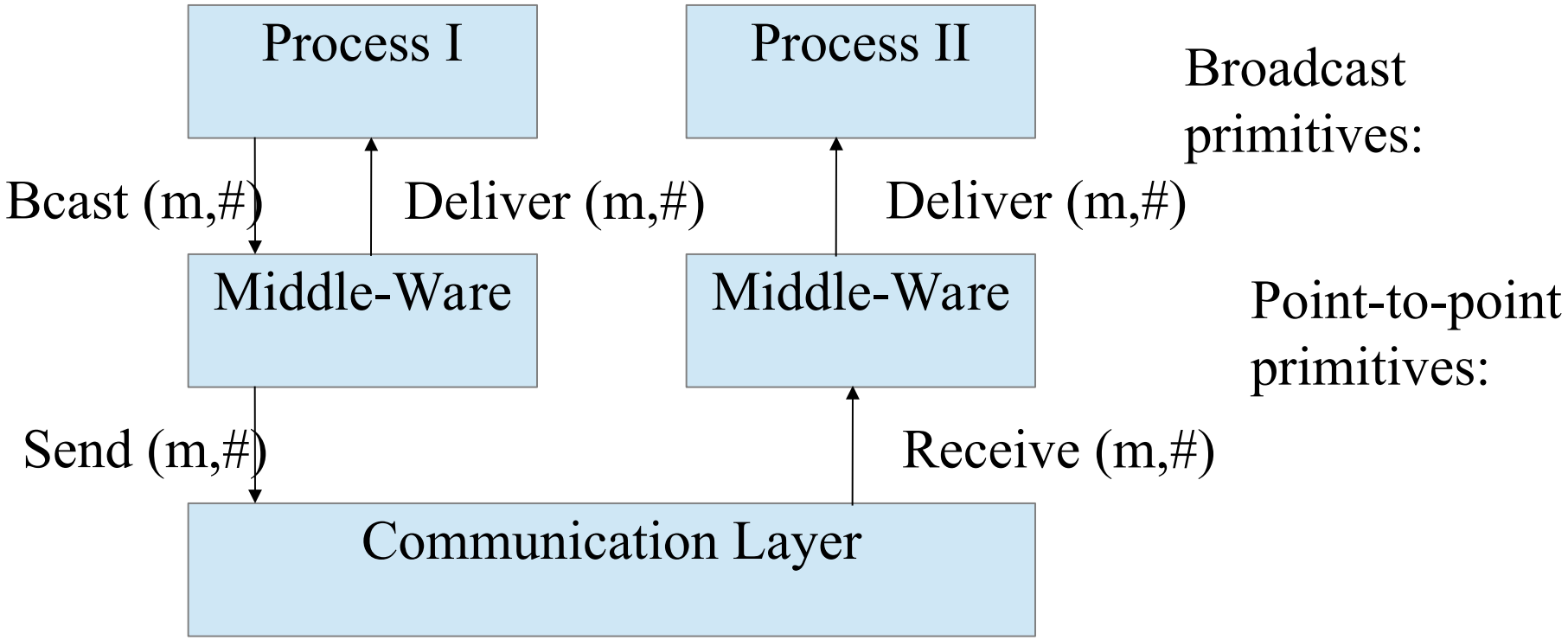
Multi-Point Protocols

Request Order in Multi-Point Protocols

There is no request order:
- from one sender
- between different senders and
- between independent requests of different senders

If your business logic needs some order, it has to be created.
E.g. by using a reliable, fault-tolerant broadcast model.

Fault-tolerant Broadcast Model



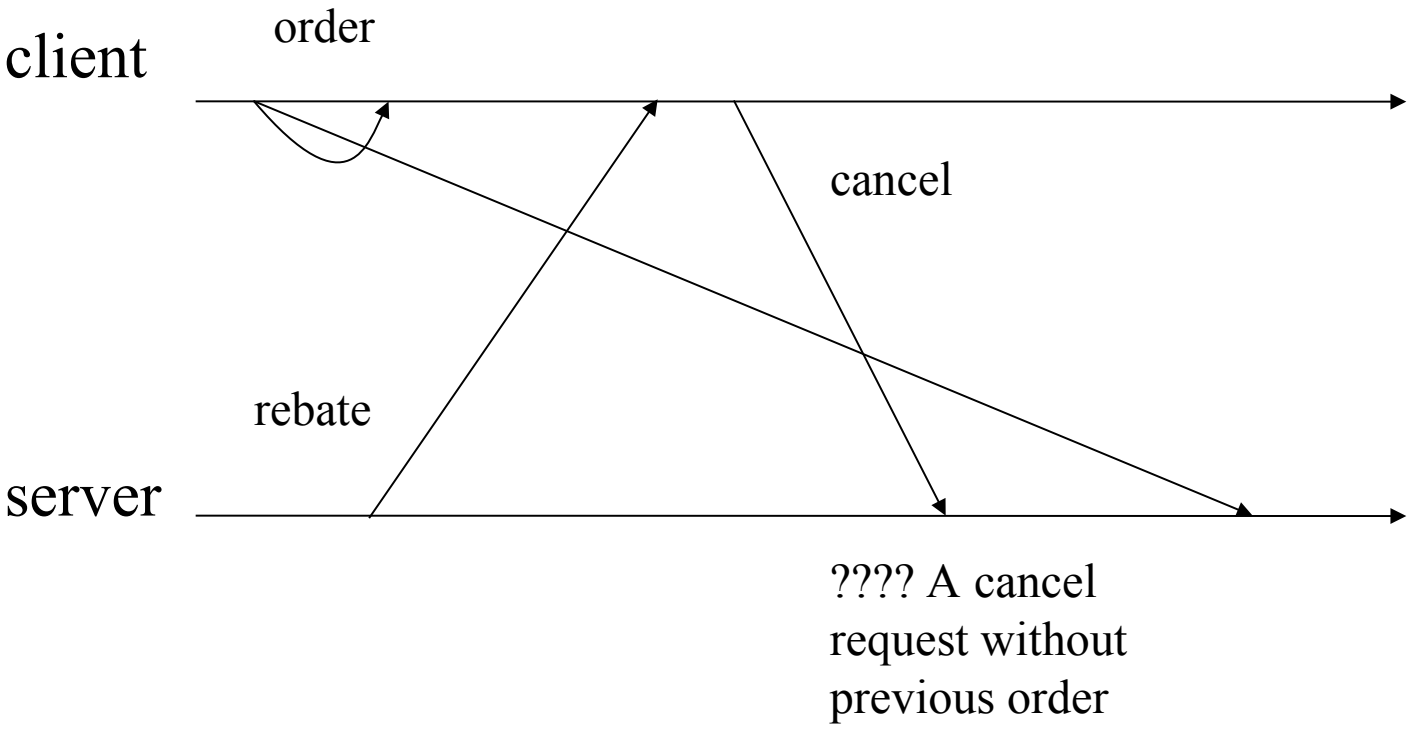
Watch out: messages can be delivered without respect to some order. Or they can be sorted, kept back at the middleware layer and only delivered when a certain order can be guaranteed. Notice the self-delivery of messages by the sending process.

Reliable Broadcast - Request Ordering with Multiple Nodes

- Reliable Broadcast
- Fifo Cast
- Causal Cast
- Absolutely Ordered Casts

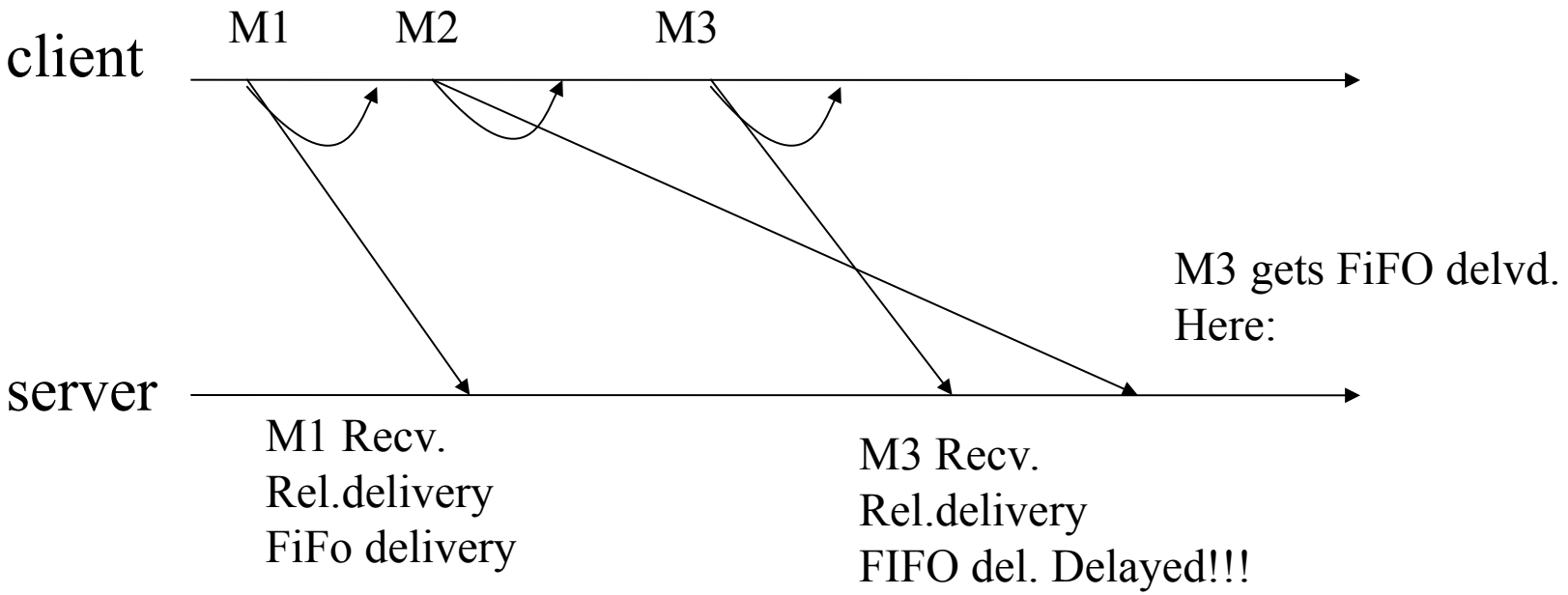
Taken from: C. Karamanoulis and K.Birman

Reliable Broadcast with no Order



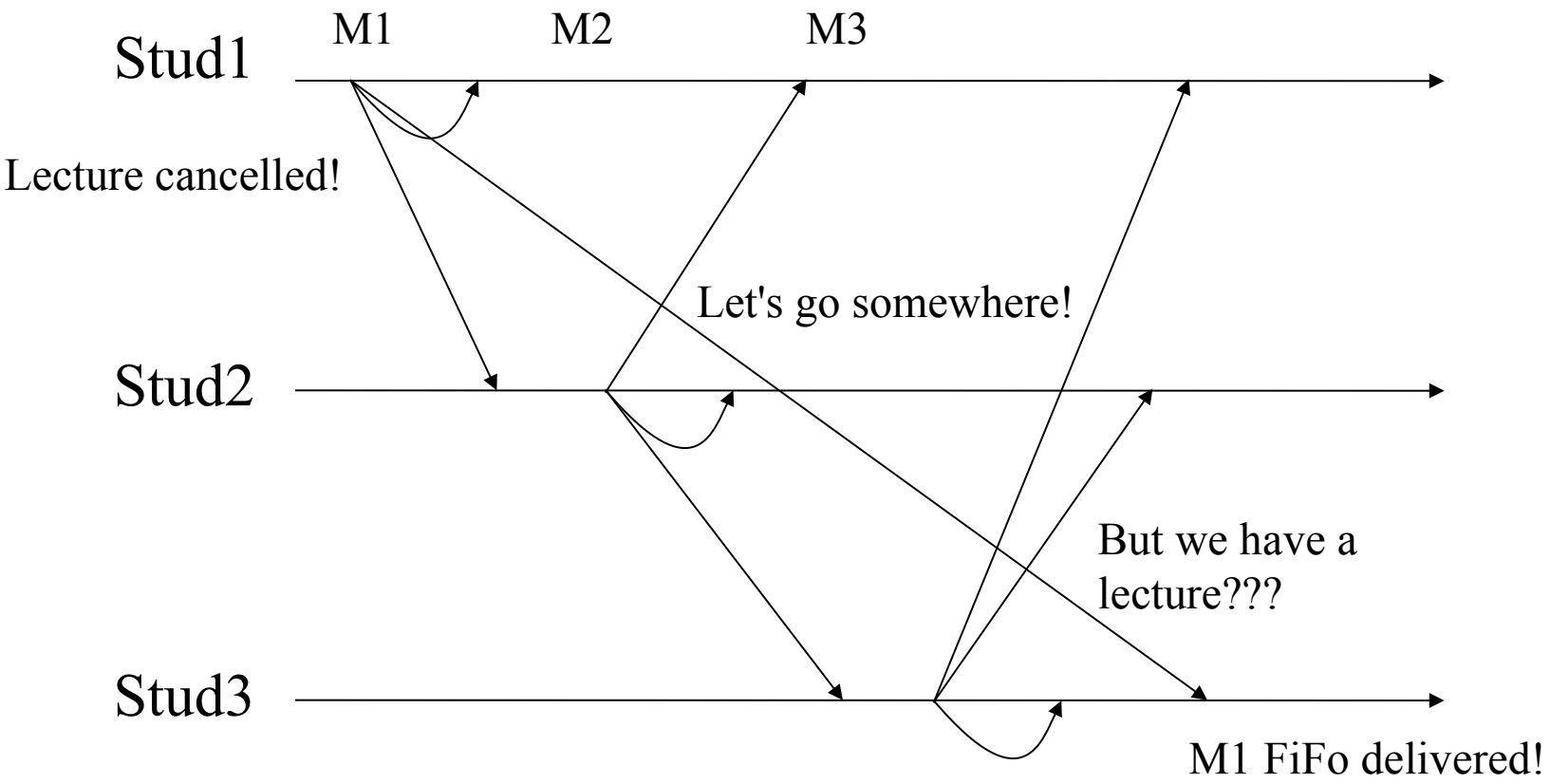
Taken from: C. Karamanoulis, Reliable Broadcasts

Reliable Broadcast with FiFo-Order



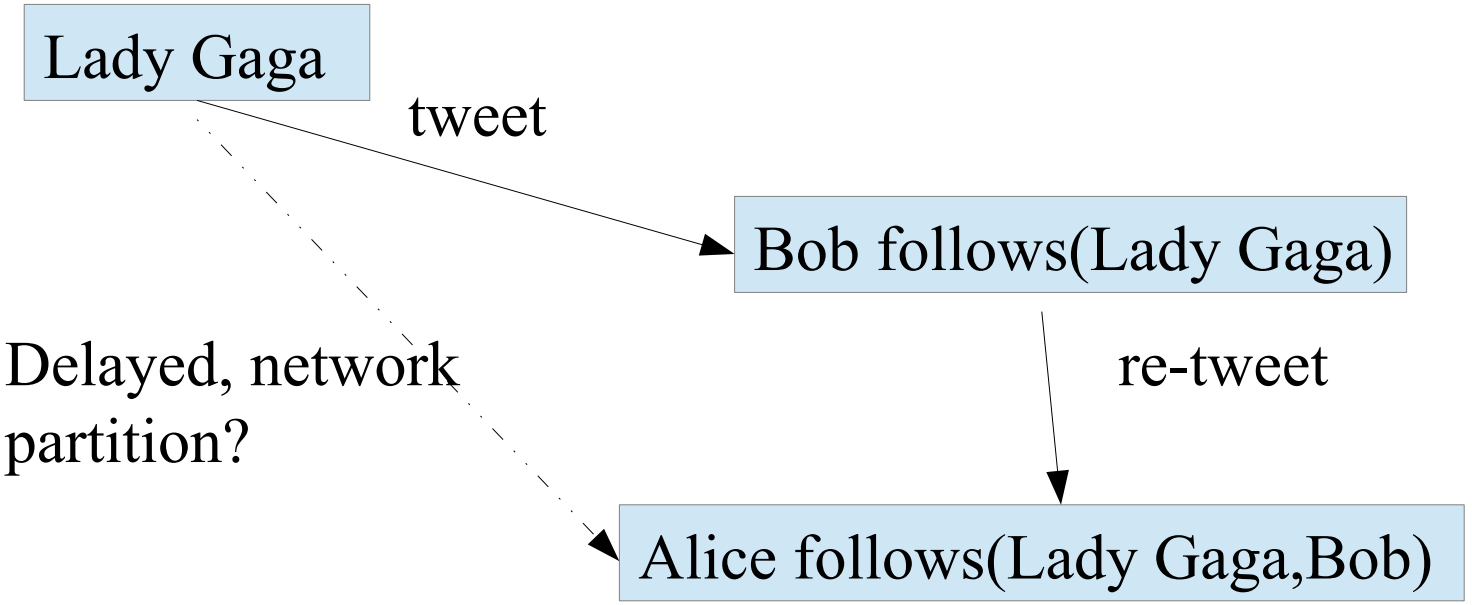
Taken from: C. Karamanoulis,

Causal Violation with FiFO Order



Taken from: C. Karamanoulis, Local Order: If a process delivers a message m before broadcasting a message m', then no correct process delivers m' unless it has previously delivered m.

Twitter Example



Alice gets the re-tweet of Lady Gaga's message, before she gets the original tweet. Is this OK? How could you Prevent it? With partition tolerance?

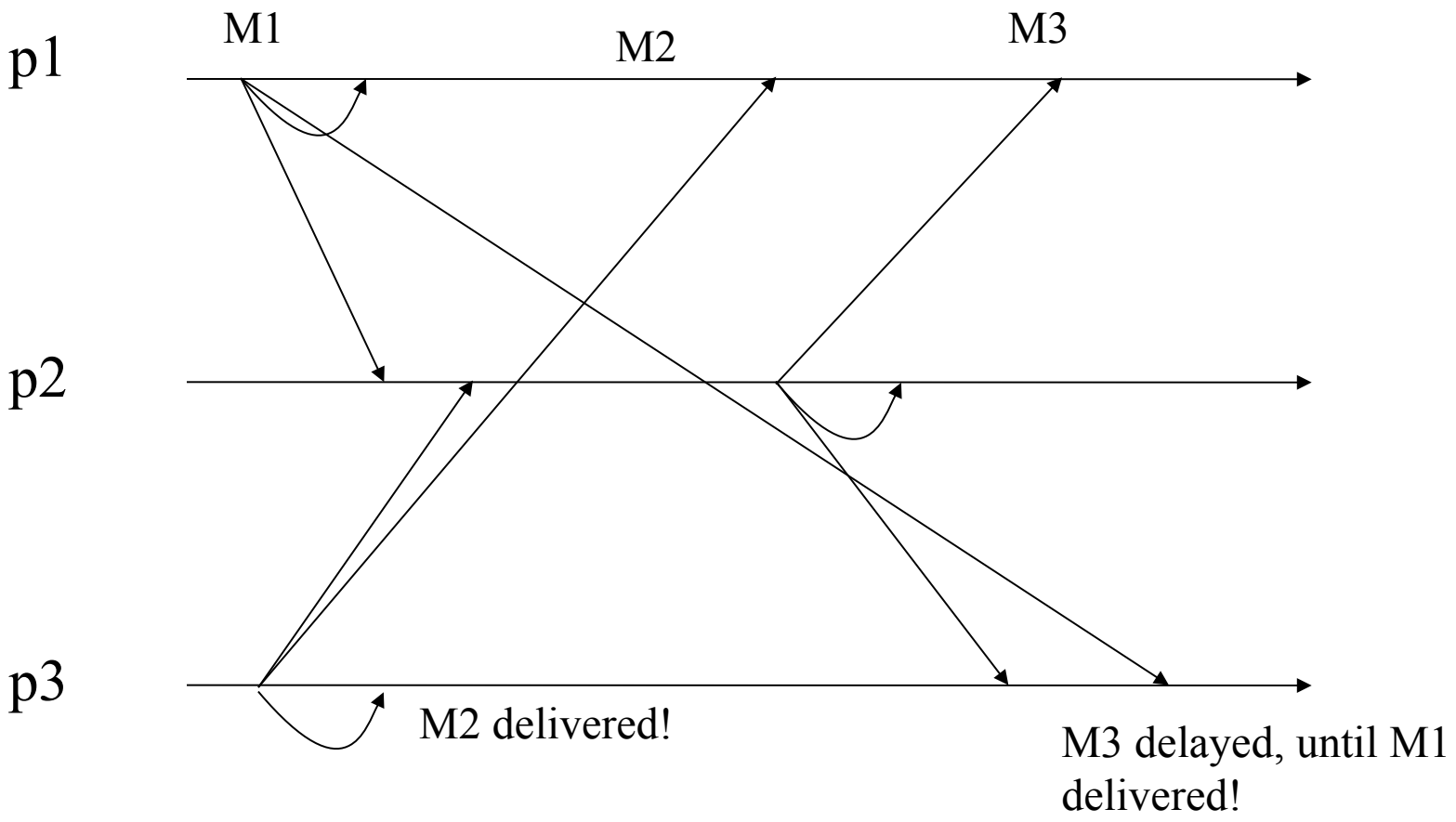
Solutions for Causal Ordered Broadcasts

- Piggyback every message sent with privious messages: Processes which missed a message can learn about it with The next incoming message and then deliver correctly

- Send event history with every message (e.g. using vector Clocks. Delay delivery until order is correct.

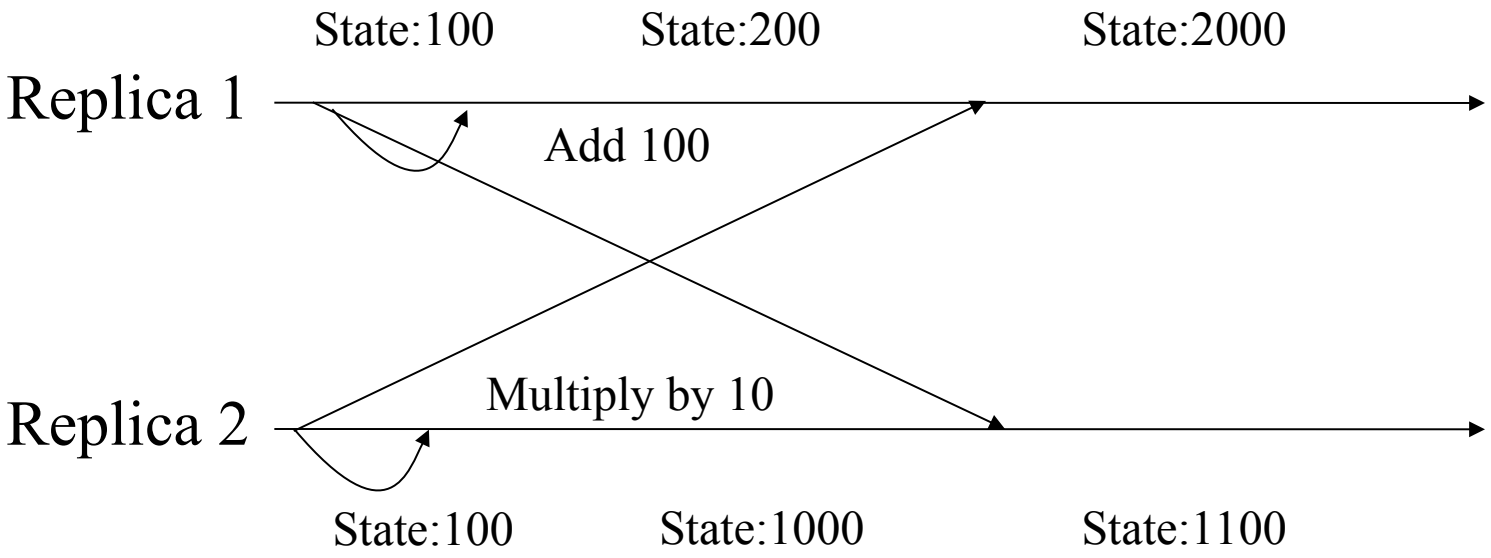
Taken from: C. Karamanoulis and K.Birman. What are the advantages/disadvantages of both solutions?

Causal Re-Ordering



Taken from: C. Karamanoulis,
P3 has delivered M2 to itself, before delivering M1. Is this a problem?
Think about causal dependency and what causes it!

Replication Anomalies with Causal Order



Taken from: C. Karamanoulis, Total Order: If correct processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

Solutions for Atomic Broadcasts (Total Order)

- All nodes send messages to every other node.
- All nodes receive messages, but wait with delivery
- One node has been selected to organize total order.
- This node orders all messages into a total order
- This node sends the total order to all nodes
- All nodes receive the total order and deliver their messages
  According to this order.

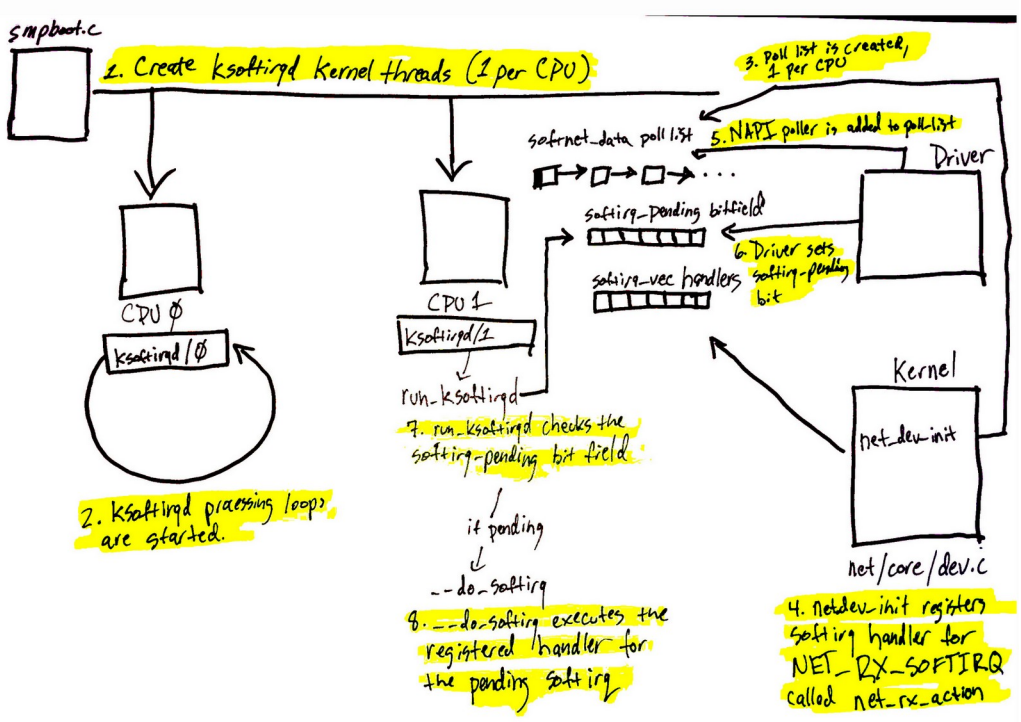
Taken from: K.Birman. What are the advantages/disadvantages of this solution?

Programming Client/Server Systems with Sockets and different I/O Models

Overview

- Below sockets: Linux Networking Stack
- Socket primitives
- Process Model with sockets
- Example of server side socket use
- Transparency and socket programming?
- Security, Performance, Availability, Flexibility etc. of socket based C/S.
- Typical C/S infrastructure (Proxies, Firewalls, LDAP)

Linux Network Stack:Admin Structures



softIRQ kernel threads are created (one per CPU) in spawn_ksoftirqd in kernel/softirq.c with a call to smpboot_register_percpu_thread from kernel/smpboot.c. As seen in the code, the function run_ksoftirqd is listed as thread_fn, which is the function that will be executed in a loop. The ksoftirqd threads begin executing their processing loops in the run_ksoftirqd function.

Next, the softnet_datastructures are created, one per CPU. These structures hold references to important data structures for processing network data. One we'll see again is the poll_list. The poll_list is where NAPI poll worker structures will be added by calls to napi_schedule or other NAPI APIs from device drivers.

net_dev_init then registers the NET_RX_SOFTIRQ softirq with the softirq system by calling open_softirq, as shown here. The handler function that is registered is called net_rx_action. This is the function the softirq kernel threads will execute to process packets.

The call to napi_schedule in the driver adds the driver's NAPI poll structure to the poll_list for the current CPU.

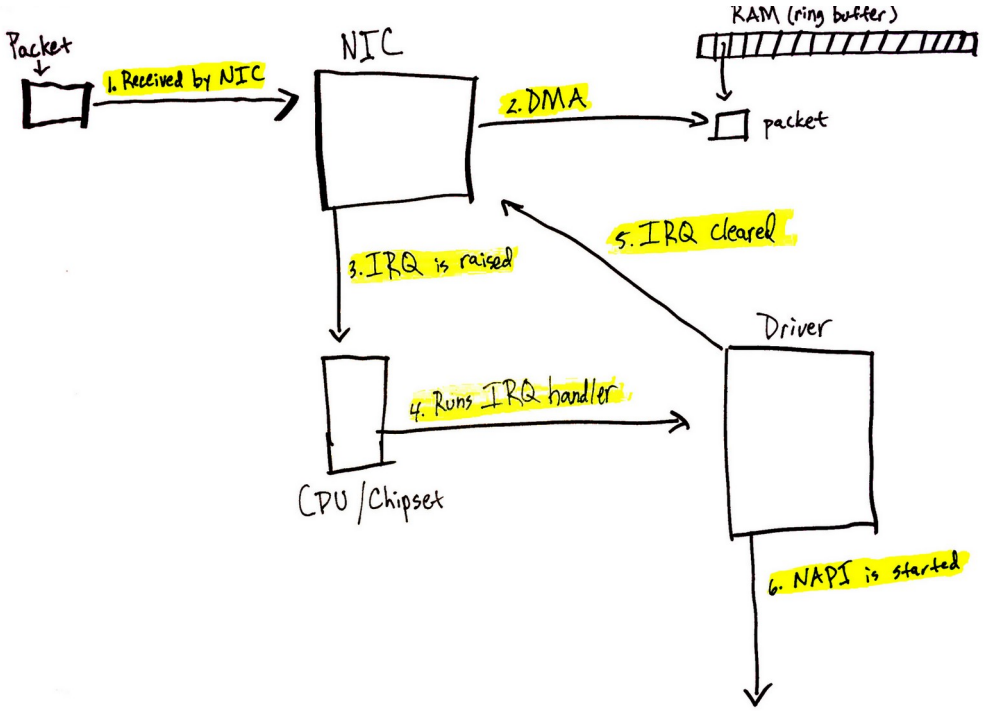
The softirq pending bit is set so that the ksoftirqd process on this CPU knows that there are packets to process.

run_ksoftirqd function (which is being run in a loop by the ksoftirq kernel thread) executes.

__do_softirq is called which checks the pending bitfield, sees that a softIRQ is pending, and calls the handler registered for the pending softIRQ: net_rx_action which does all the heavy lifting for incoming network data processing.

From: http://blog.packagecloud.io/eng/2016/10/11/monitoring-tuning-linux-networking-stack-receiving-data-illustrated/ . See also: http://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data/ for more information.

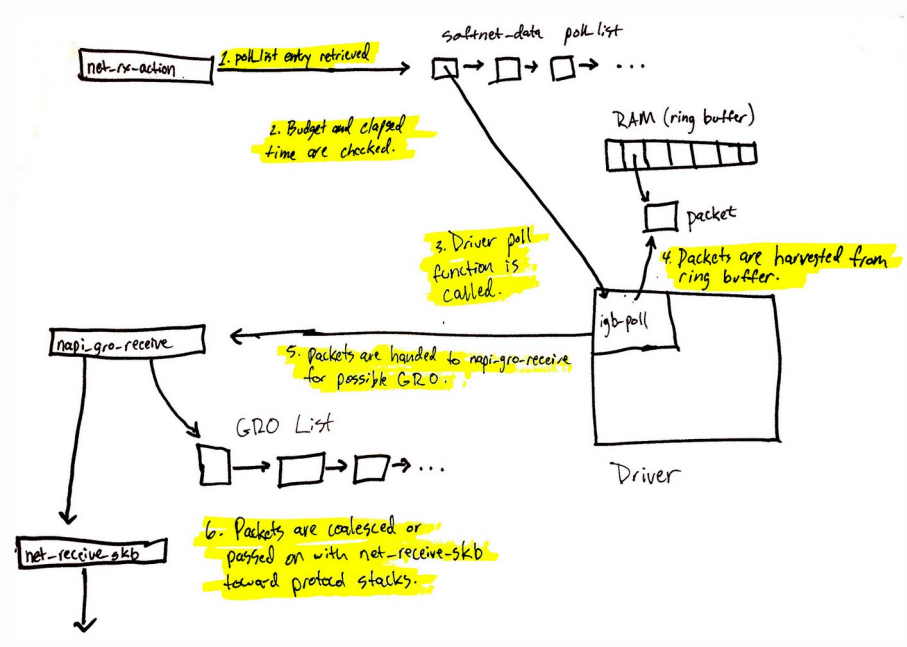
Linux Network Stack:Low Level IRQ



The call to NAPI starts the high-level interrupt processing (SoftIRQ)

From: http://blog.packagecloud.io/eng/2016/10/11/monitoring-tuning-linux-networking-stack-receiving-data-illustrated/ . See also http://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data/ for more information.

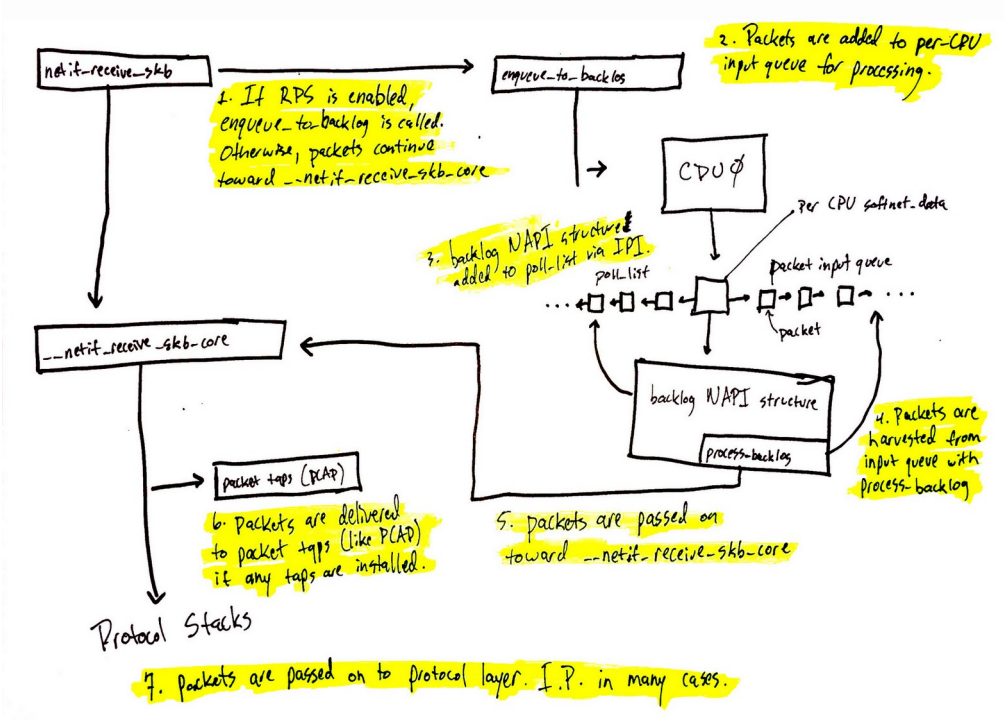
Linux Network Stack: SoftIRQ Handler



The ring buffer with data is processed via SoftIRQ handler. Budget and elapsed time checking ensure fair processing in the kernel.

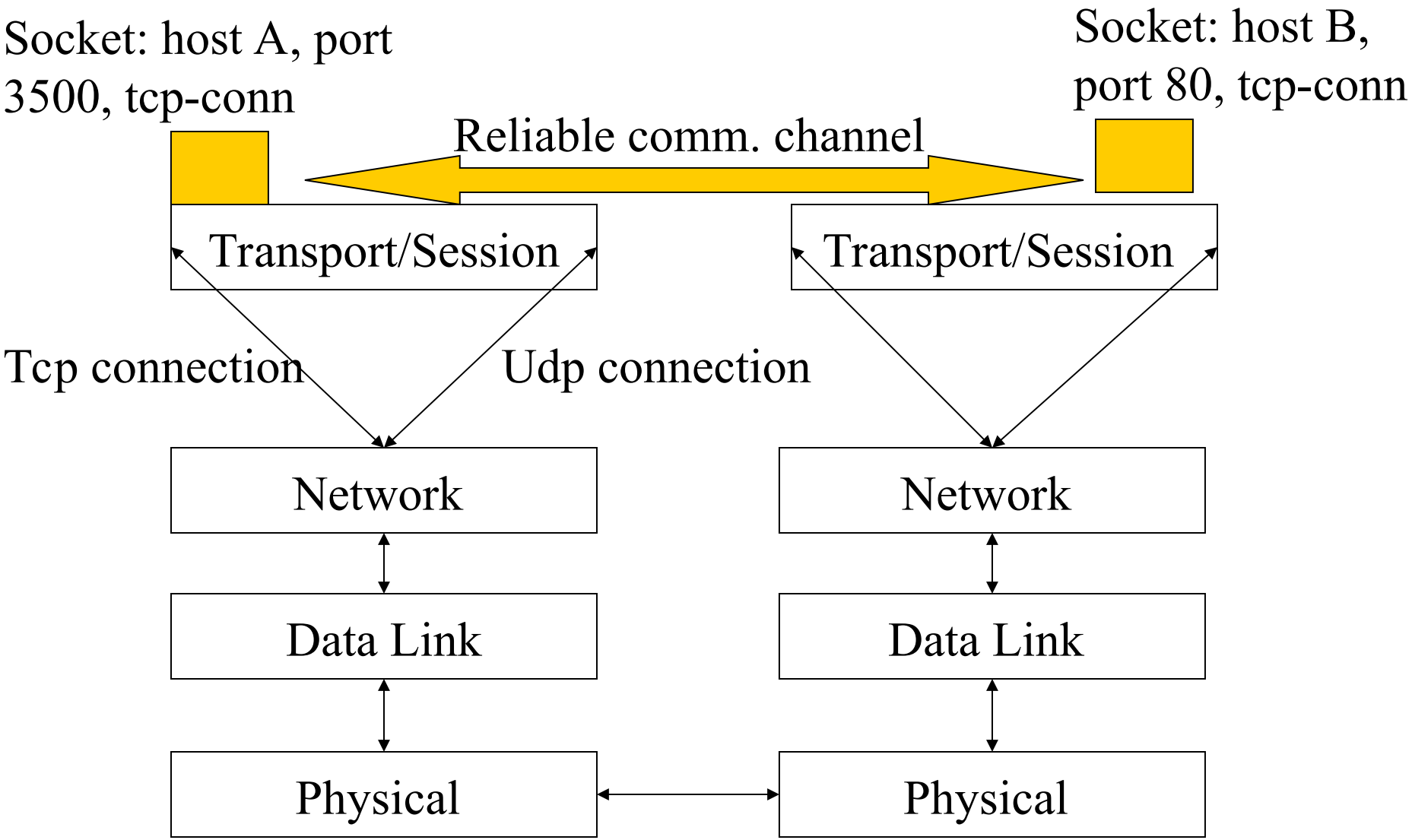
From: http://blog.packagecloud.io/eng/2016/10/11/monitoring-tuning-linux-networking-stack-receiving-data-illustrated/ . See also http://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data/ for more information.

Linux Network Stack: Data Processing



From: http://blog.packagecloud.io/eng/2016/10/11/monitoring-tuning-linux-networking-stack-receiving-data-illustrated/ . See also http://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data/ for more information.

Protocol Stack for Sockets



Socket Properties

- Using either tcp or udp connections
- Serving as a programming interface
- A specification of “Host”, “Port”, “Connection type”
- A unique address of a channel endpoint.

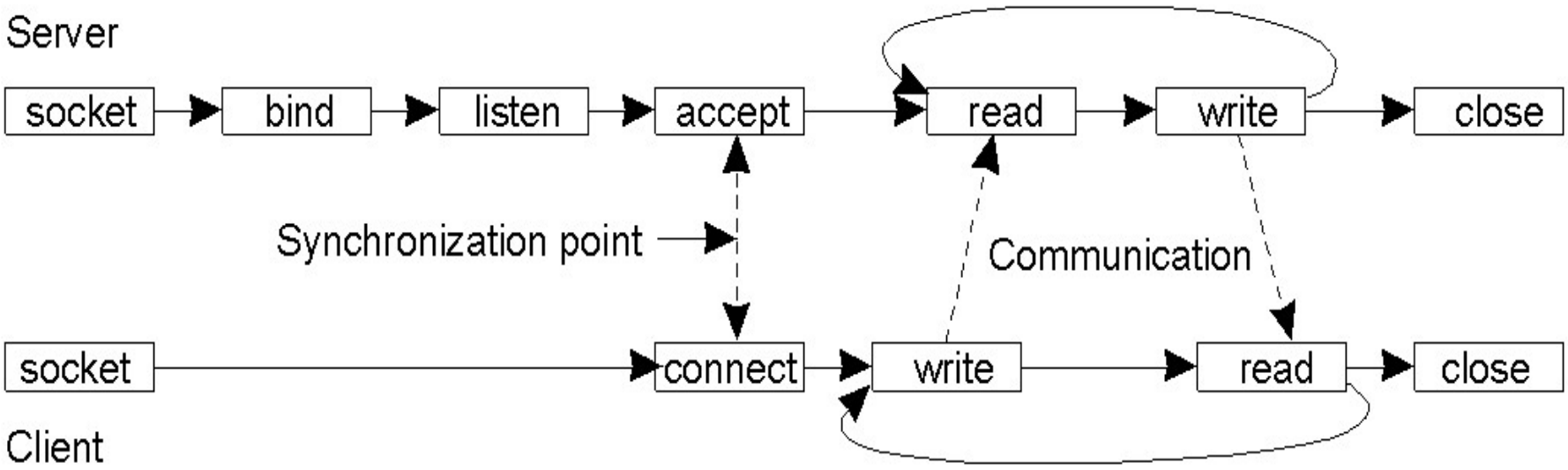
Berkeley Sockets (1)

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Socket primitives for TCP/IP.

From: van Steen, Tanenbaum, Distributed Systems

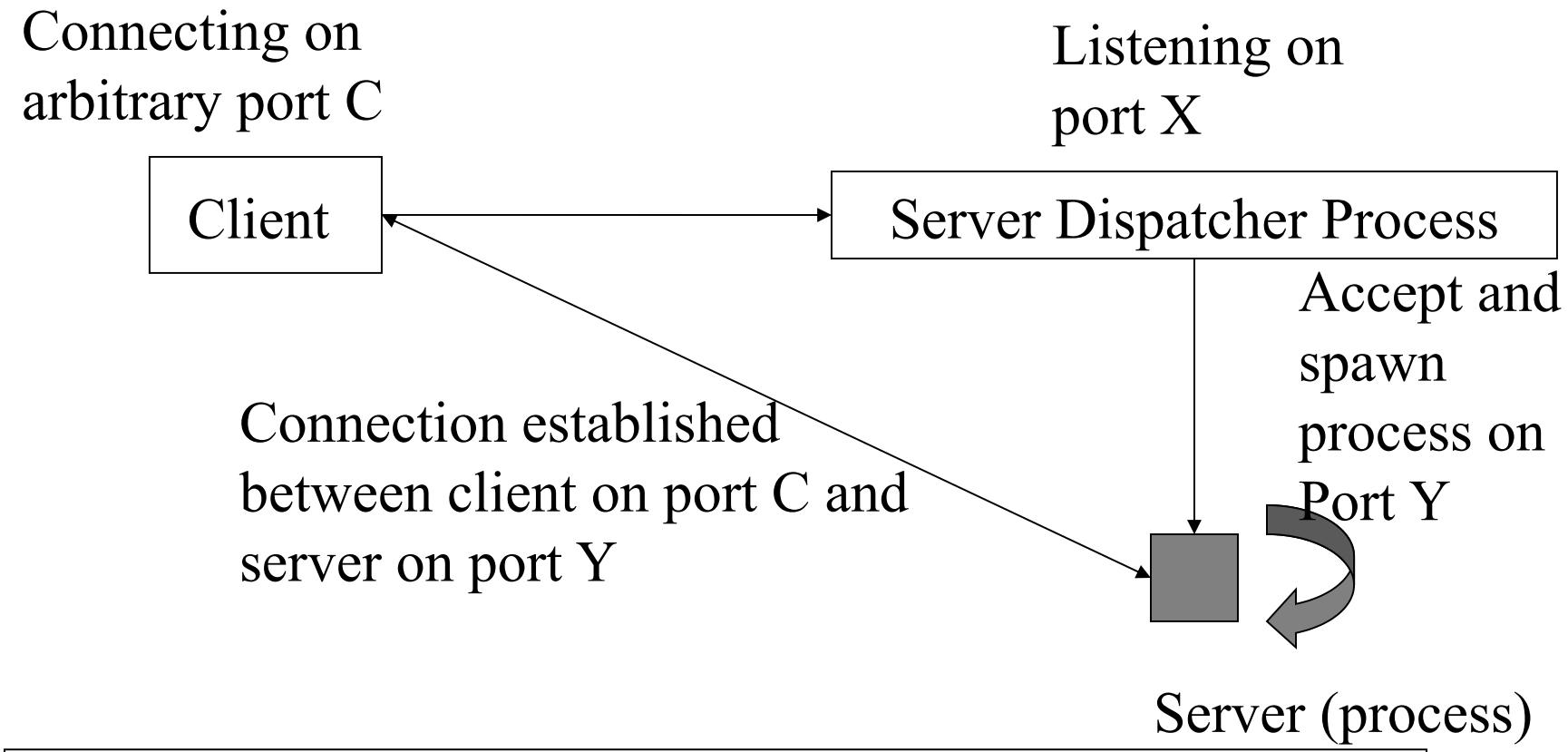
Berkeley Sockets (2)



Connection-oriented communication pattern using sockets.

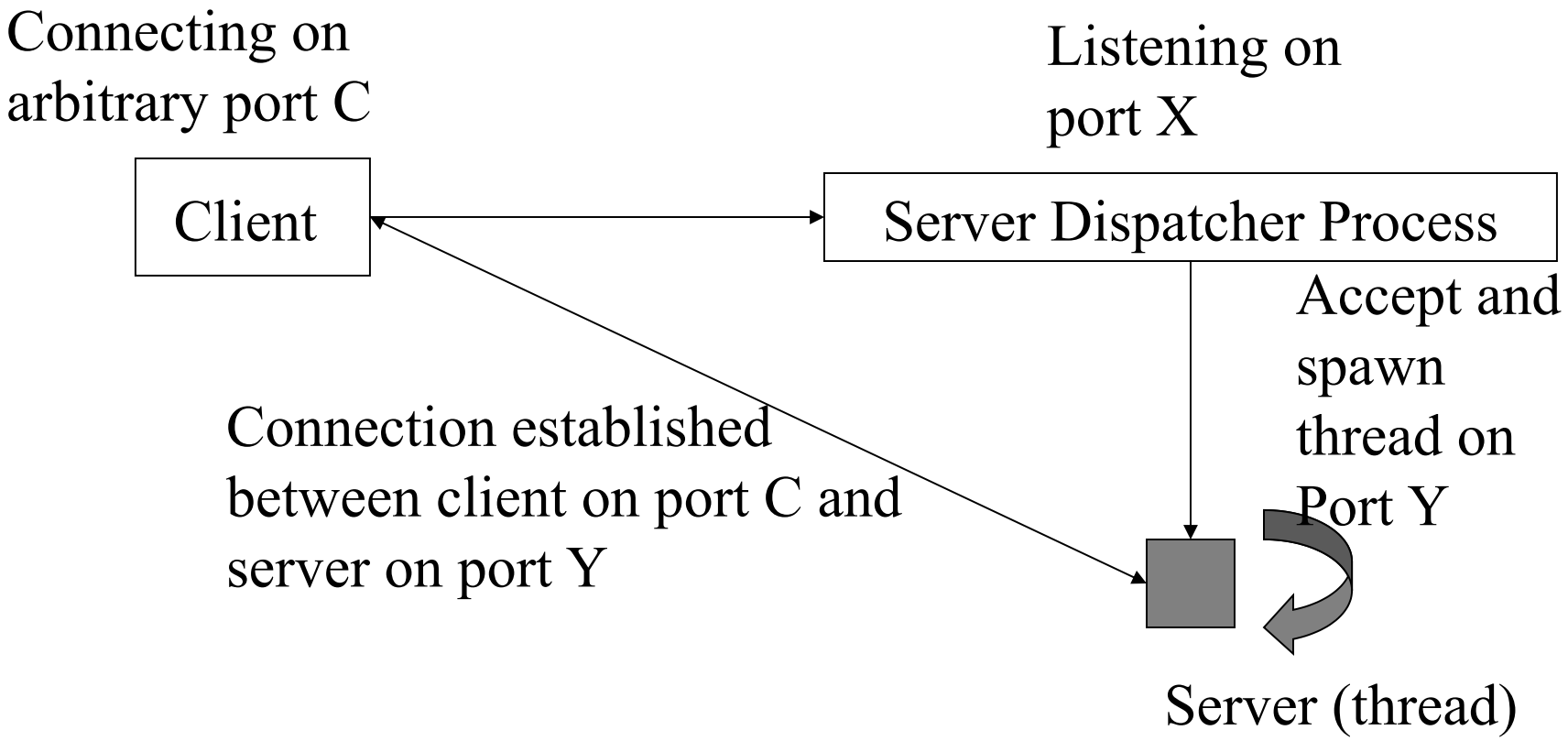
From: van Steen, Tanenbaum, Distributed Systems

Server Side Processing using Processes



After spawning a new process the dispatcher goes back to listening for new connection requests. This model scales to some degree (process creation is expensive and only few processes are possible). Example: traditional CGI processing in web-server

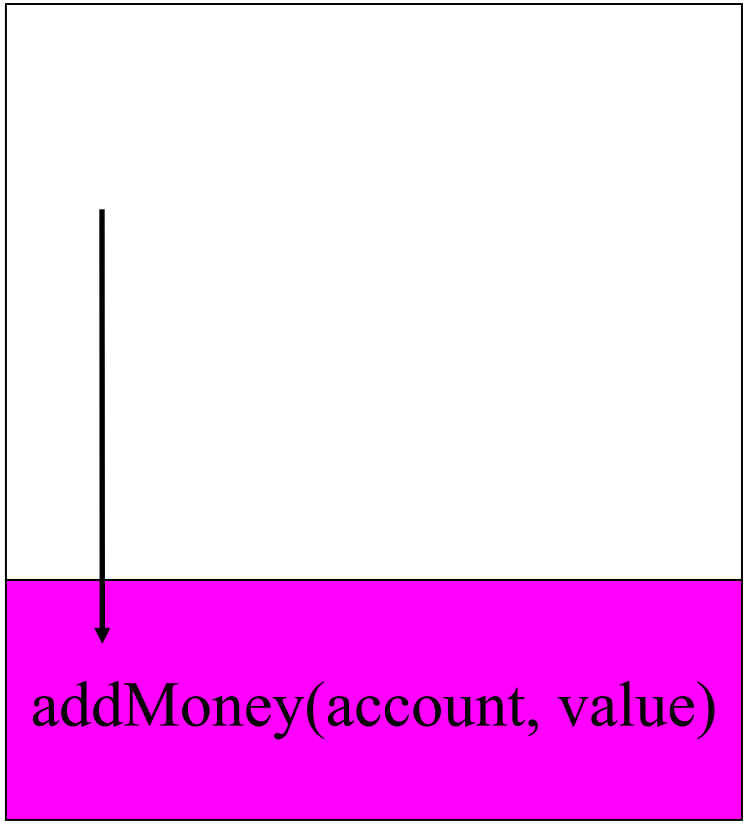
Server Side Processing using Threads



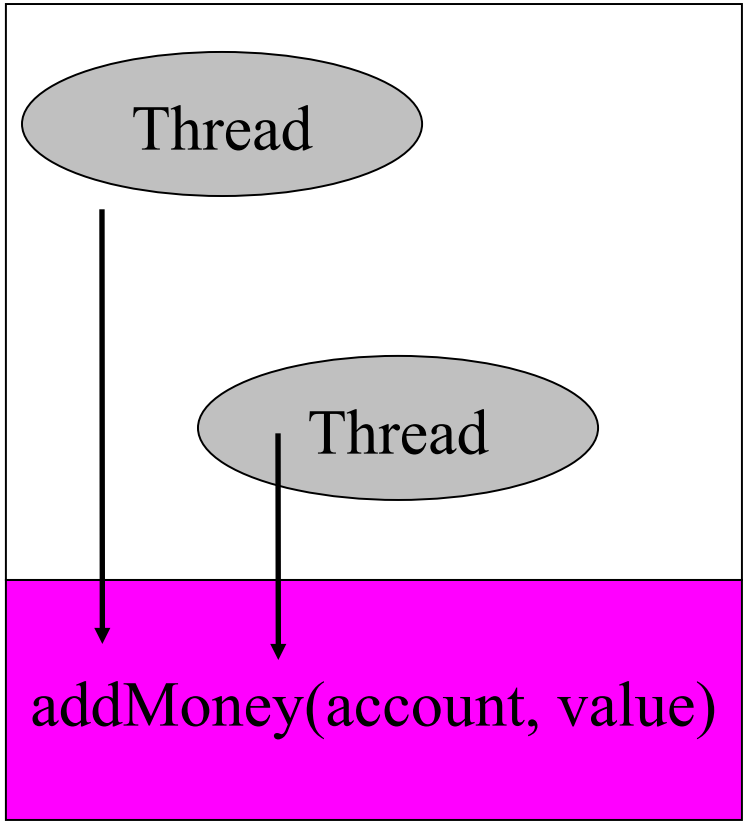
After spawning a new thread the dispatcher goes back to listening for new connection requests. This model scales well (thread creation is expensive but they can be pooled) and a larger number of threads are possible). Example: servlet request processing in servlet engine (aka “web-container”)

Server Side Concurrency

Process per request



Threaded server



In the case of the threaded server the function needs to be re-entrant. No unprotected global variables. Keep state per thread on stack.

Designing a socket based service

a) Design the message formats to be exchanged (e.g. “http1.0 200 OK …). Try to avoid data representation problems on different hardware.

b) Design the protocol between clients and server:

   - Will client wait for answer? (asynchronous vs. synchr. Comm.)

   - Can server call back? (== client has server functionality)

   - Will connection be permanent or closed after request?

   - Will server hold client related state (aka session)?

   - Will server allow concurrent requests?

Stateless or Stateful Service?

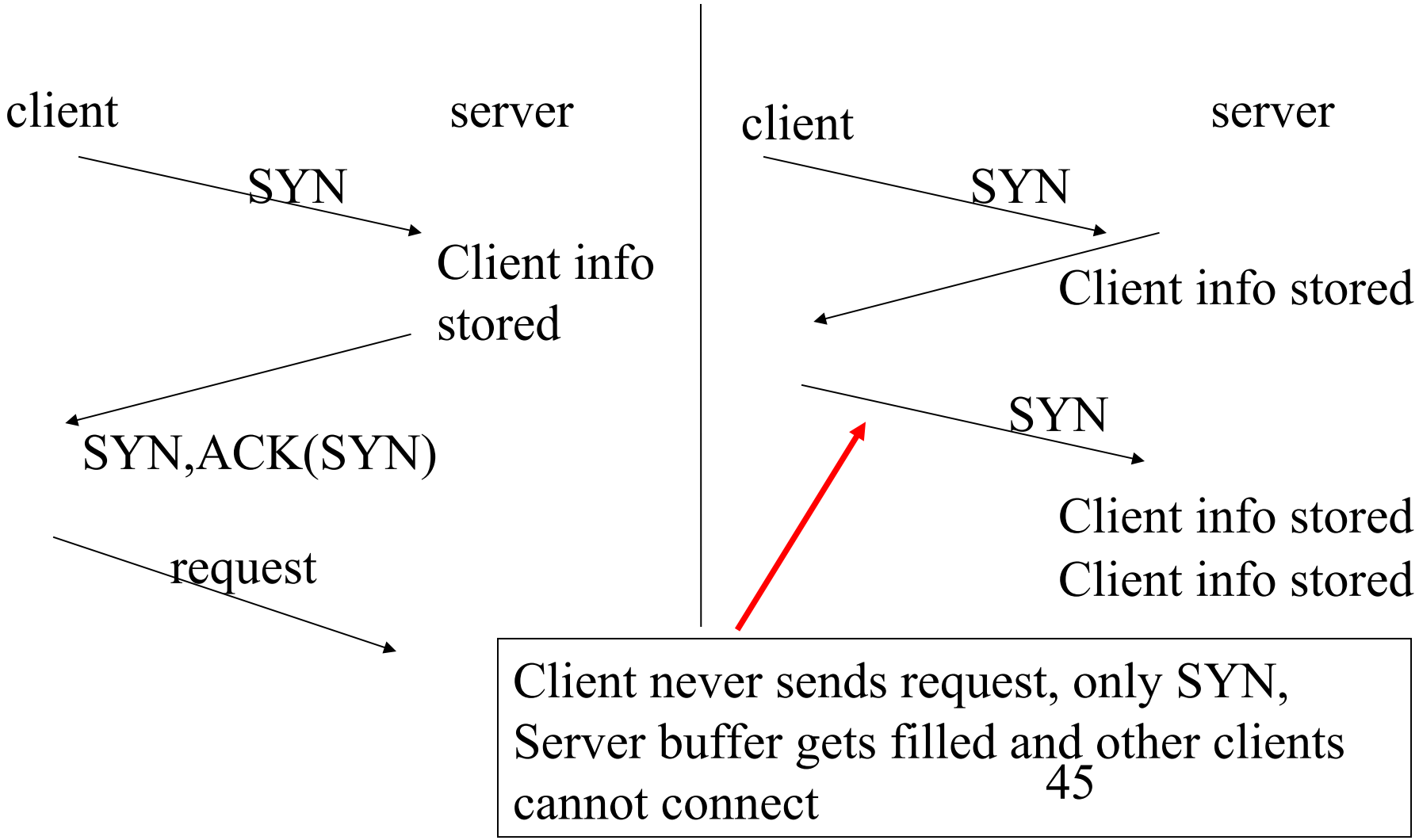
Stateless:

- Scales extremely well
- Makes denial of service attacks harder
- Forces new authentication and authorization per request

Stateful

- Allows transactions and delivery guarantees
- Can lead to resource exhaustion (e.g. out of sockets) on a server
- Needs somehow reliable hardware and networks to succeed.

Server Dangers: Keeping State and expecting clients to behave -TCP SYN flooding



A Client using sockets

1. Define hostname and port number of server host
2. Allocate a socket with host and port parameters
3. Get the input channel from the socket (messages from server)
4. Get output channel from socket (this is where the messages to the server will go)
5. Create a message for the server, e.g. “GET /somefile.html HTTP/1.0”
6. Write message into output channel (message is sent to server)
7. Read response from input channel and display it.

A multithreaded client would use one thread to read e.g. from the console and write to the output channel while the other thread reads from the input channel and displays the server messages on the console (or writes to a file)

A server using sockets

1. Define port number of service (e.g. 80 for http server)
2. Allocate a server socket with port parameter. Server socket does “bind” and “listen” for new connections.
3. “Accept” an incoming connection, get a new socket for the client connection
4. Get the input channel from the socket and parse client message
5. Get output channel from socket (this is where the messages to the client will go)
6. Do request processing (or create a new thread to do it)
7. Create a response message e.g. “HTTP/1.0 2000 \n…”
8. Write message into output channel (message is sent to client)
9. Read new message from client channel or close the connection

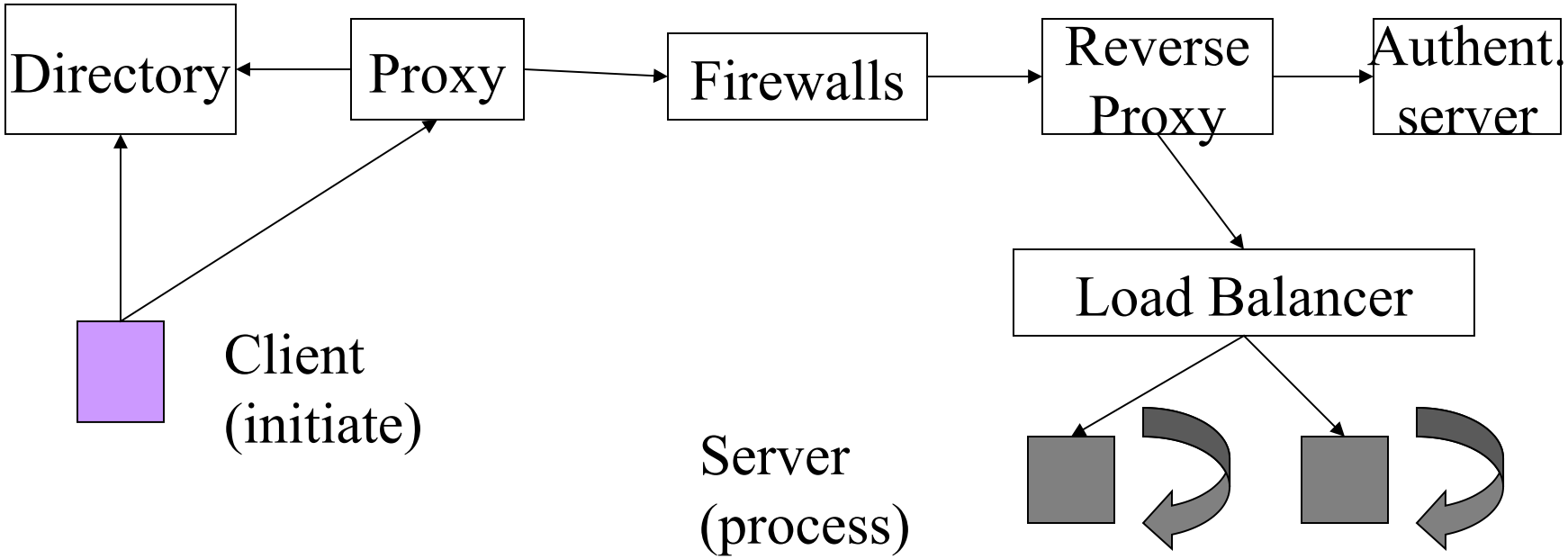
A bare bone server. Could be extended through e.g. a command pattern to match requests with processing dynamically. New commands could get loaded dynamically as well. (“Application Server”)

Distribution Transparency with Sockets?

- Invocation: The server side function cannot be called on the client side. Instead, socket operations must be used and messages defined.
- Location/Relocation/Migration: If service moves, client breaks.
- Replication/Concurrency: No support yet
- Failure: No support yet
- Persistence: No support yet

To be fair: socket based services need to deal with all that but they are still fairly simple to write!

Infrastructure of C/S Systems



Directory: help locate server
Proxy: check client authorization, route via firewall
firewall: allow outgoing calls only

Reverse Proxy: cache results, end SSL session, authenticate client
Authentication server: store client data, authorize client
Load Balancer: distribute requests across servers

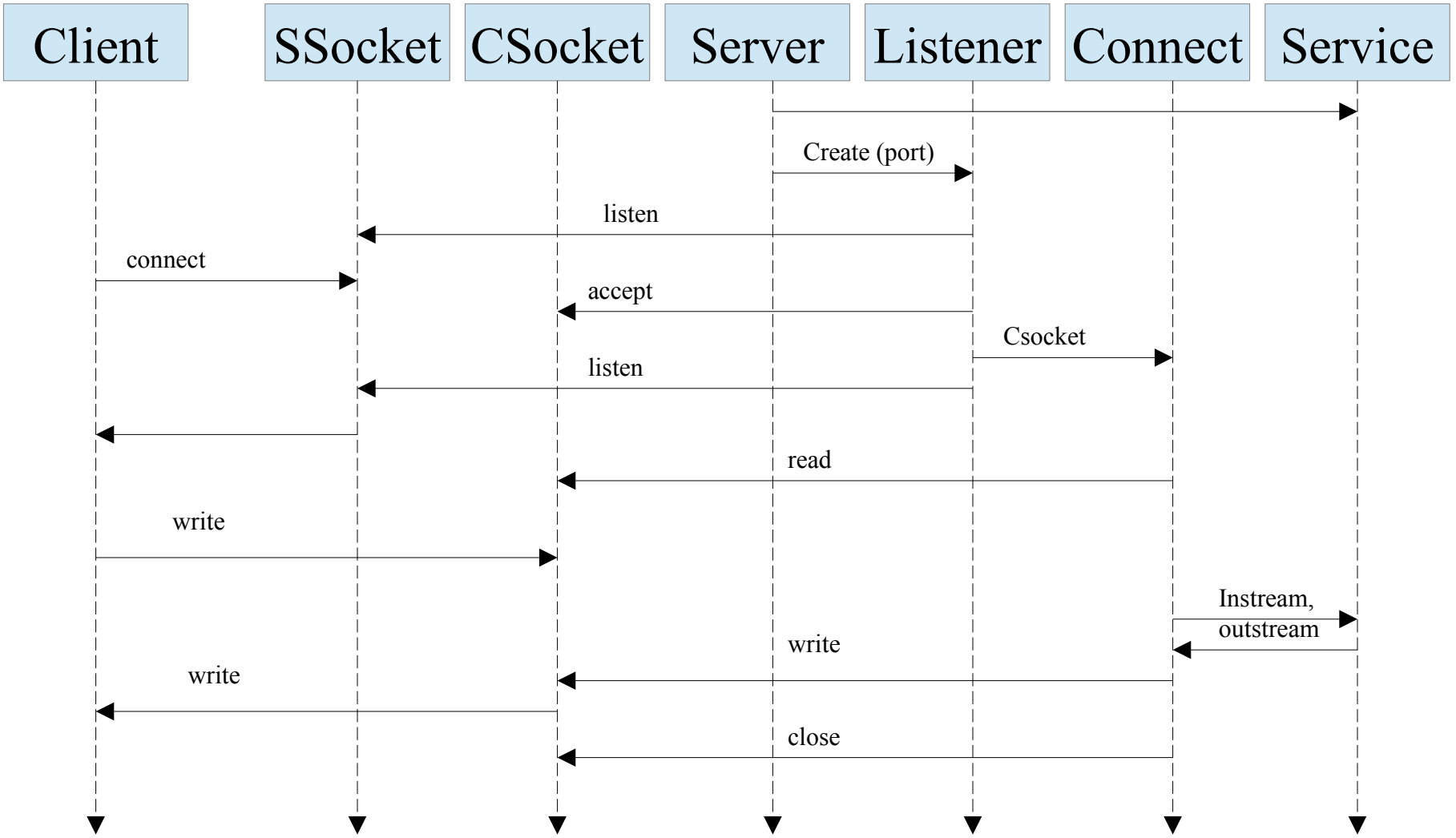
Exercises

Using code pieces from the Java examples book we will:

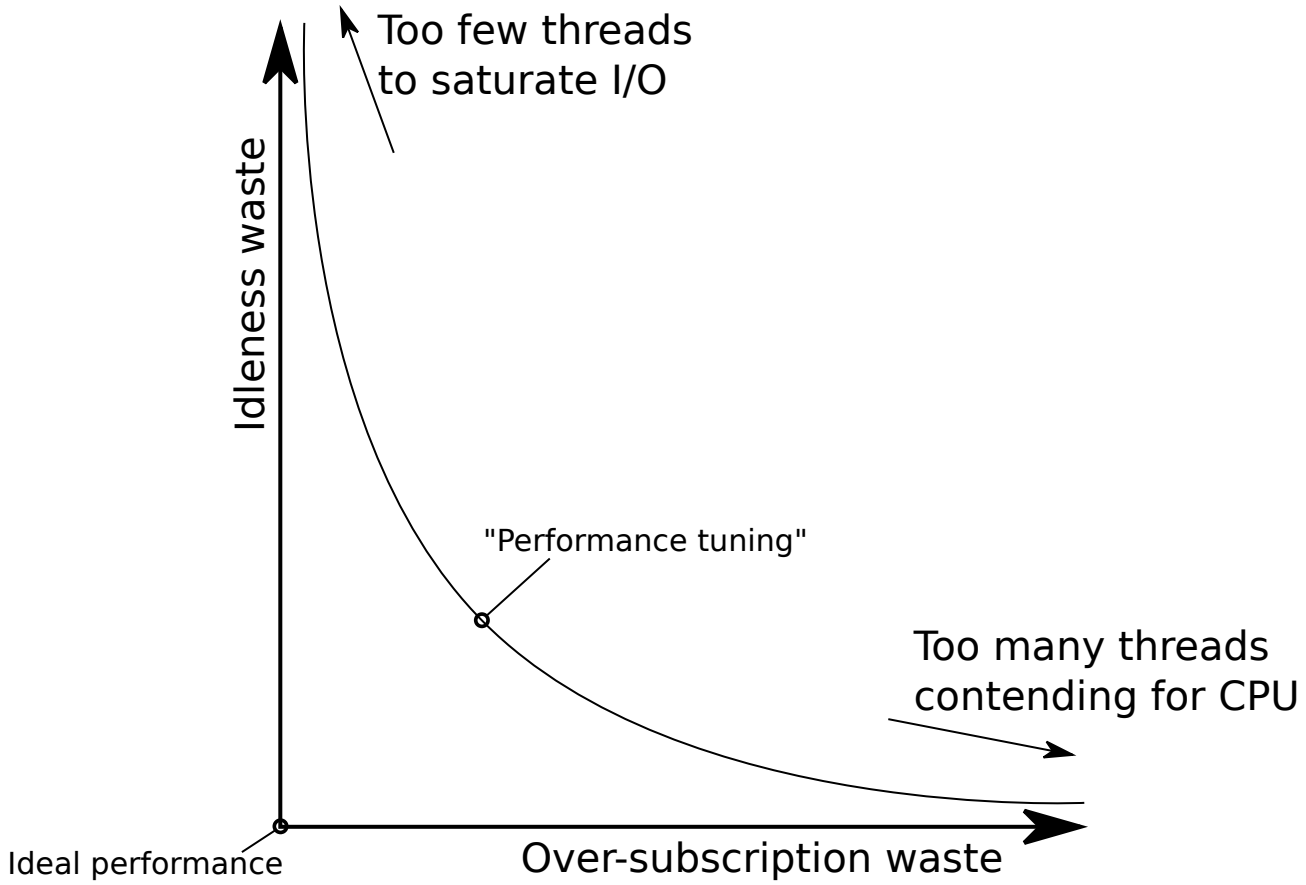
- Extend the application server from server.java
  - security
  - persistence

We will discuss general design issues as well! (patterns etc)

Sequence Diagram Server.java



Synchronous Threading Limitations



https://www.tedinski.com/2018/11/06/concurrency-models.html

Homework for next session on RPC!

Read: Remote Procedure Call

https://christophermeiklejohn.com/pl/2016/04/12/rpc.html

Resources

- Scaling Ruby Apps to 1000 Requests per Minute - A Beginner's Guide
- by Nate Berkopec, http://www.nateberkopec.com/2015/07/29/scaling-ruby-apps-to-1000-rpm.html
- David Flanagan, Java Examples in a Nutshell, O'Reilly, chapter 5. Code: www.davidflanagan.com/javaexamples3
- Ted Neward, Server Based Java Programming chapter 10, Code:www.manning.com/neward3
- Doug Lea, Concurrent Programming in Java
- Pitt, Fundamental Java Networking (Springer). Good theory and sources (secure sockets, server queuing theory etc.)
- Queuing Theory Portal: http://www2.uwindsor.ca/%7Ehlynka/queue.html
- Performance Analysis of networks: http://www2.sis.pitt.edu/~jkabara/syllabus2120.htm (with simulation tools etc.)
- Meet the experts: Stacy Joines and Gary Hunt on WebSphere performance (performance tools, queue theory etc.) http://www-128.ibm.com/developerworks/websphere/library/techarticles/0507_joines/0507_joines.html
- Doug Lea, Java NIO http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf Learn how to handle thousands of requests per second in Java with a smaller number of threads. Event driven programming, Design patterns like reactor, proactor etc.
- Abhijit Belapurkar, CSP for Java programmers part 1-3. Explains the concept of communicating sequential processes used in JCSP library. Learn how to avoid shared state multithreading and its associated dangers.
- Core tips to Java NIO: http://www.javaperformancetuning.com/tips/nio.shtml
- Schmidt et.al. POSA2 book on design patterns for concurrent systems.
- Nuno Santos, High Peformance servers with Java NIO: http://www.onjava.com/pub/a/onjava/2004/09/01/nio.html?page=3 . Explains design alternatives for NIO. Gives numbers of requests per second possible.
- James Aspnes, Notes on Theory of Distributed Systems, Spring 2014, www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf