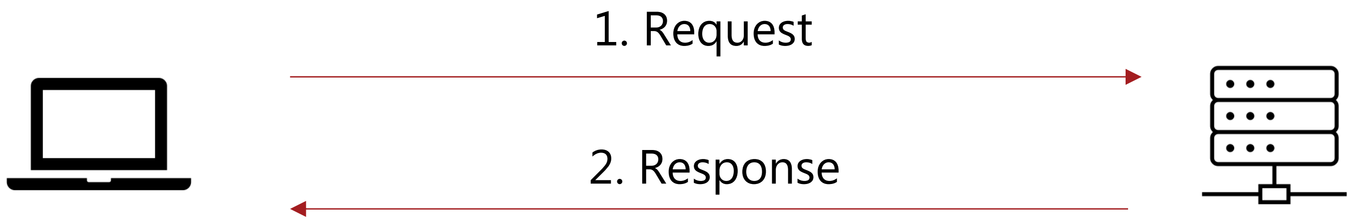
High performance parallel systems

Lecture 8 – Network protocols and applications

Kenneth Skovhede, NBI, 2020-12-10

UNIVERSITY OF COPENHAGEN

Client/Server - or Request/Response



Defined in RFC 2616

HTTP Request

Verb Path Protocol version

```
GET /index.html HTTP/1.1\r\n
Host: www.example.com\r\n
User-Agent: my-client-v0.1\r\n
Cookie: abc=123;def=567\r\n
X-test: yes!\r\n
\r\n
```

No body for a GET request

Headers

End of header

HTTP Response

Protocol version Status code and text

```
HTTP/1.1 200 OK\r\n
Server: MyWeb/2.2.14\r\n
Content-Length: 32\r\n
Content-Type: text/html\r\n
Connection: Closed\r\n
\r\n
<html><body>hello!</body></html>
```

Body

Minimal HTTP Request

$$\underbrace{\texttt{GET}}_{\text{Verb}}\ \underbrace{\texttt{/index.html}}_{\text{Path}}\ \underbrace{\texttt{HTTP/1.1}}_{\text{Protocol version}}\texttt{\textbackslash r\textbackslash n}$$

```
GET /index.html HTTP/1.1\r\n
Host: www.example.com\r\n
\r\n
```

Minimal HTTP Response

Protocol version / Status code and text

```
HTTP/1.1 200 OK\r\n
\r\n
OK
```

More than one request pr. connection:

- Client sends
  - `Connection: Keep-Alive`
- Server responds
  - `Connection: Keep-Alive`
  - `Keep-Alive: timeout=5, max=50`
- All requests and responses must have explicit `Content-Length`, or boundary marker

HTTP methods

- `GET` Idempotent operation, caching possible
- `PUT`
- `POST`
- `PATCH`
- `DELETE`

Update resource, caching not allowed

- (`PROPFIND`) Non-standard operation, used for WebDAV

The querystring is part of the URL

https://www.google.com/search?q=meaning+of+life

```
GET /search?q=meaning+of+life HTTP/1.1\r\n
Host: www.google.com\r\n
\r\n
```

Caching allowed, by client, server and proxies

Url format is:

```
<protocol> :// <server> </path> <?querystring>
```

Path is refering to the path on the local system but is often restricted to a particular folder.

```
/                          => /var/www/
/search                    => /var/www/search
/profile/data/set1.txt     => /var/www/profile/data/set1.txt
/..                        => /var/
```

Usually forbidden

Updating a shopping cart - POST

```
POST /cart/add HTTP/1.1
Host: api.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 34

Pizza+with%20cheese=1&salad+bowl=2
```

OR

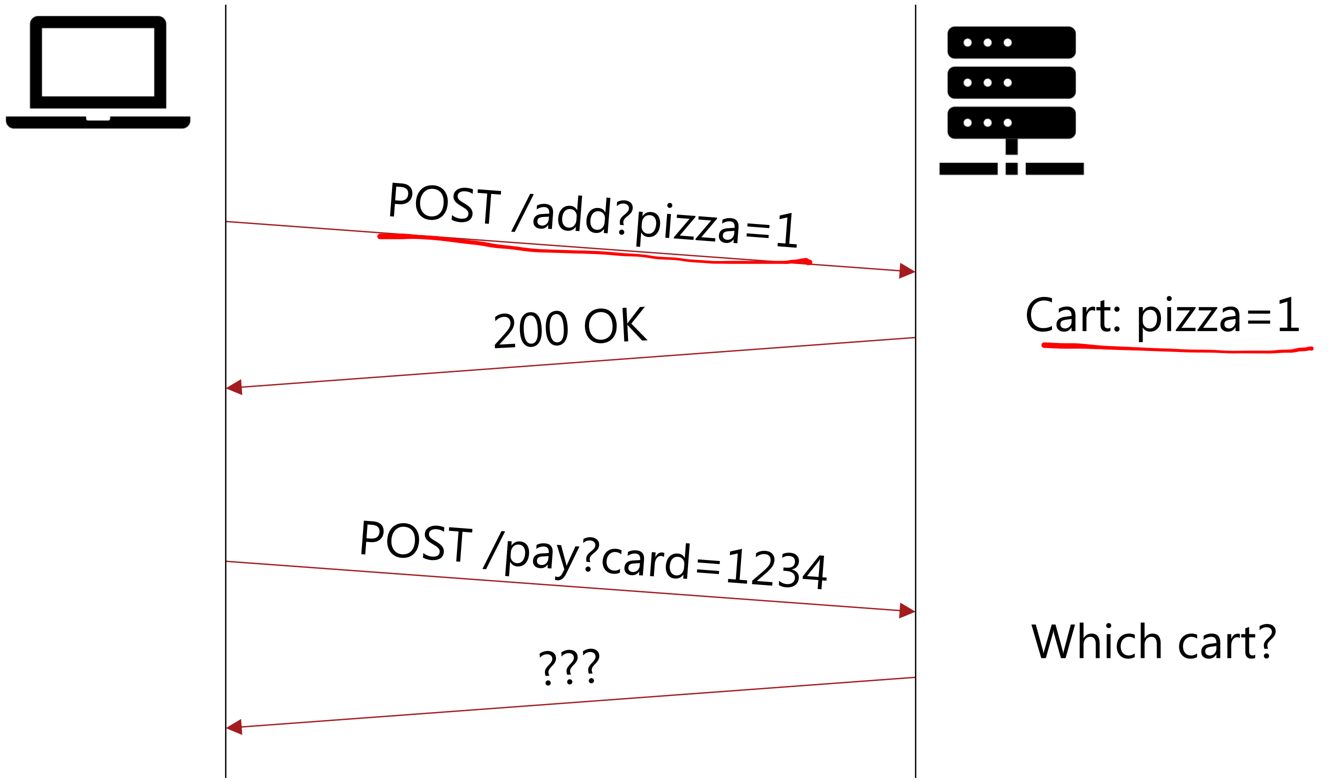
```
POST /cart/add HTTP/1.1
Host: api.example.com
Content-Type: multipart/form-data;boundary="boundary"

--boundary
Content-Disposition: form-data; name="pizza with cheese"

1
--boundary
Content-Disposition: form-data; name="salad bowl"

2
--boundary--
```

HTTP request/response is stateless



Cookies for state management

First visit from client

```
GET /cart HTTP/1.1\r\n
Host: api.example.com\r\n
\r\n
```

1.

Server creates session

```
HTTP/1.1 200 OK\r\n
Set-Cookie: sessionid=1234;Max-Age=500
\r\n
OK
```

Subsequent requests carry the cookie

```
POST /cart/add?pizza=1 HTTP/1.1\r\n
Host: api.example.com\r\n
Cookie: sessionid=1234
\r\n
```

2.

Server matches with session

```
HTTP/1.1 200 OK\r\n
\r\n
OK
```

```
PUT /cart/add?pizza=2 HTTP/1.1\r\n
Host: api.example.com\r\n
Cookie: sessionid=1234
\r\n
```

3.

Server can refresh (extend) session

```
HTTP/1.1 200 OK\r\n
Set-Cookie: sessionid=1234;Max-Age=500
\r\n
OK
```

HTTP status codes

- 1xx – connection messages, not about the request
  - 101 – switch protocols, e.g. HTTP/2.0
- 2xx – success messages
  - 200 OK
  - 201 Created
  - 204 No content
- 3xx – redirect messages, not completed
  - 300 – redirect (legacy, vague semantics)
  - 301 – moved permanently
  - 302 – found (or moved temporarily)
  - 304 – not modified
- 4xx – client request error
  - 400 – bad request
  - 401 – not authorized
  - 404 – not found
  - 414 – uri too long
- 5xx – server handling error
  - 500 – internal server error

Sockets – What is it?

Abstraction for communication, meant to mimic file operations

File

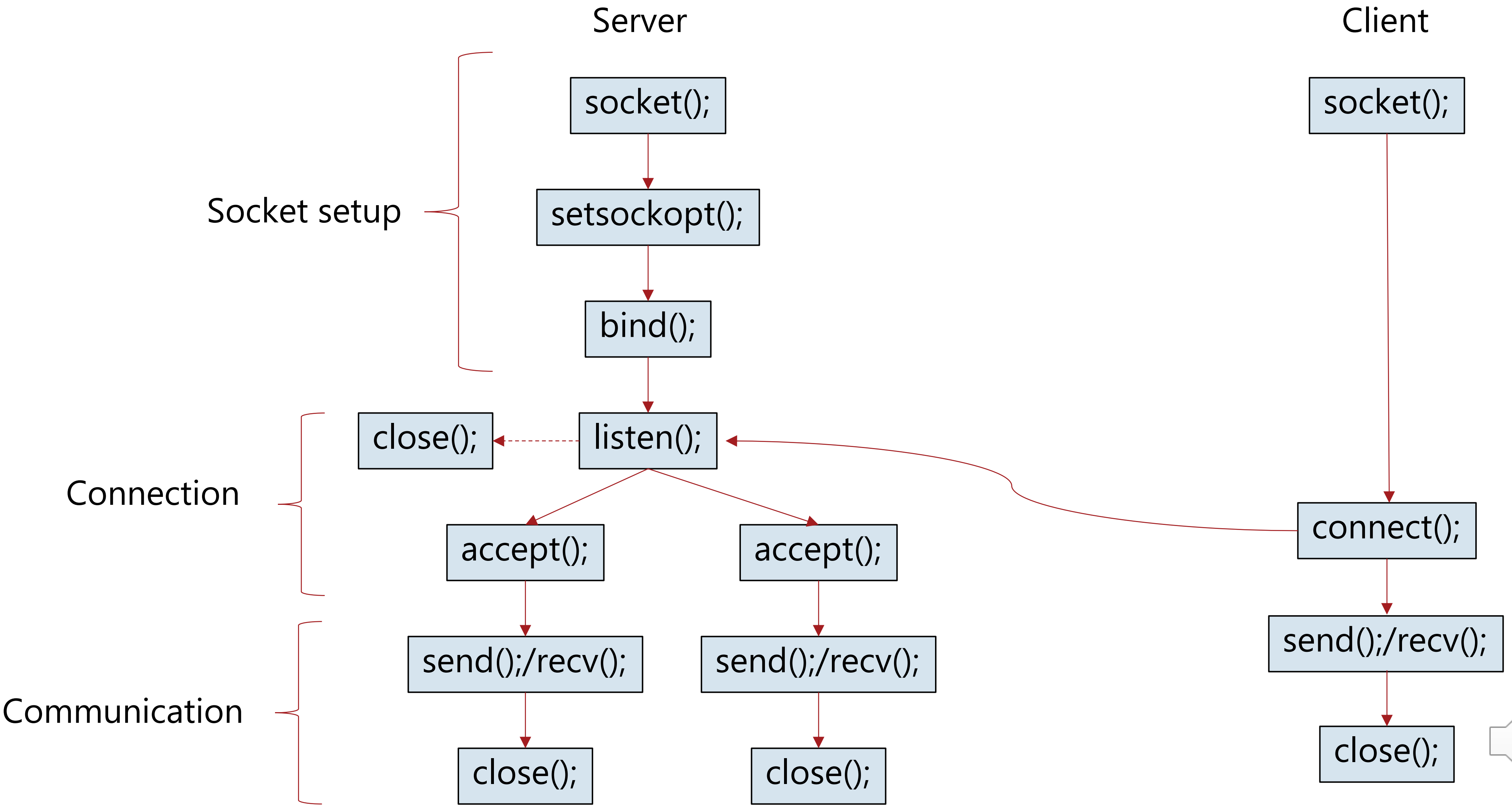
```
handle = open(path);
read();
write();
close();
```

Socket

```
handle = socket(domain, type, protocol);
... extra steps here ...
send();
recv();
close();
```

Note that we cannot change position in socket data (i.e. no `seek()`)

Socket states



The methods

In C language

```c
int server_fd, new_socket, valread;
struct sockaddr_in address;
char buffer[1024] = {0};
int addrlen = sizeof(address);

// Warning: No error checks!!!
int server_fd = socket(AF_INET, SOCK_STREAM, 0);
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons( PORT );
bind(server_fd, (struct sockaddr *)&address, sizeof(address);
listen(server_fd, 3);

new_socket = accept(server_fd,
     (struct sockaddr *)&address, (socklen_t*)&addrlen);
while(1) {
  valread = read( new_socket , buffer, 1024);
  if (valread <= 0)
        break;
  send(new_socket, buffer, valread);
}
```

In Python

```python
import socket
with socket.socket(socket.AF_INET,
  socket.SOCK_STREAM) as s:

  s.bind(('127.0.0.1', 6531))
  s.listen()
  conn, addr = s.accept()
  with conn:
    while True:
      data = conn.recv(1024)
      if not data:
        break
      conn.sendall(data)
```

ZeroMQ

Extends traditional sockets with more communication models:

- Request/response (like HTTP)
- Publish/subscribe, many-to-many
- Push/pull, producers->consumers->collectors

ZeroMQ – Publisher in Python

```
import zmq
import random
import sys
import time

port = "5556"
context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.bind("tcp://*:%s" % port)

while True:
    topic = random.randrange(9999,10005)
    messagedata = random.randrange(1,215) - 80
    print "%d %d" % (topic, messagedata)
    socket.send("%d %d" % (topic, messagedata))
    time.sleep(1)
```

Based on code from: https://learning-0mq-with-pyzmq.readthedocs.io/en/latest/pyzmq/patterns/pubsub.html

ZeroMQ – Subscriber in Python

```
import sys
import zmq

port = "5556"
context = zmq.Context()
socket = context.socket(zmq.SUB)

print "Collecting updates from server..."
socket.connect ("tcp://localhost:%s" % port)

topicfilter = "10001” # Subscribe to zipcode, default is NYC, 10001
socket.setsockopt(zmq.SUBSCRIBE, topicfilter)

total_value = 0
for update_nbr in range (5):
    string = socket.recv()
    topic, messagedata = string.split()
    total_value += int(messagedata)
    print topic, messagedata

print "Average value for topic '%s' was %dF" % (topicfilter, total_value / update_nbr)
```

Based on code from: https://learning-0mq-with-pyzmq.readthedocs.io/en/latest/pyzmq/patterns/pubsub.html

Distributed systems

Has most of the problems from shared memory systems

- But no fast locks

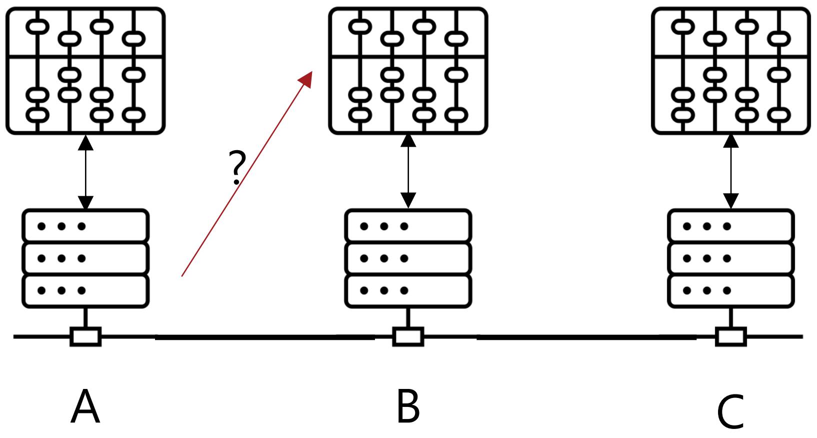
Each exchange must be with messages

- Adds latency to each operation

Machines may crash or go offline at any moment

- Networks can partition forming two or more groups that all consider themselves “global”

MPI – Message Passing Interface



No shared memory
Need to do request/response
Data can change in between requests
No total ordering of events

MPI concepts

- **Communicator** – a “group” for communication
  - Using “**MPI_COMM_WORLD**” for every process
- **Size** - the number of processes in the communicator
- **Rank** – The “id” or index of a given process

Point-to-point:

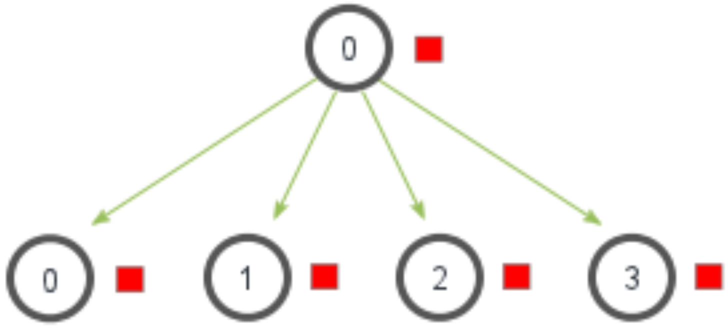
- **mpi_send()**
- **mpi_recv()**

Synchronization:

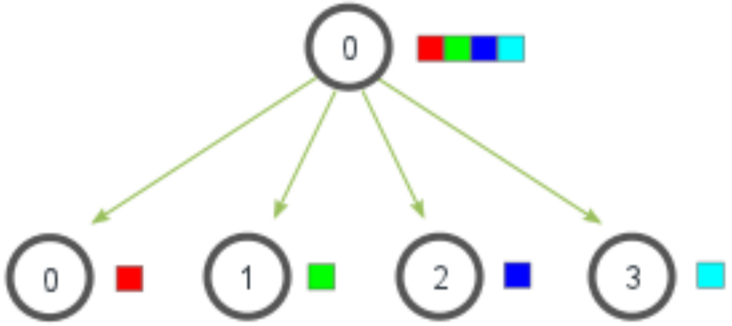
- **mpi_barrier()**

MPI – communication

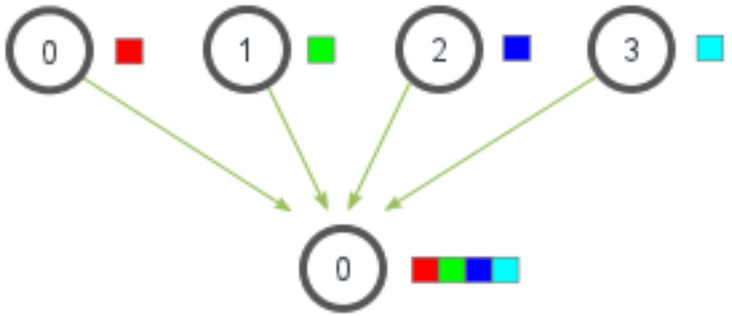
MPI_Bcast()



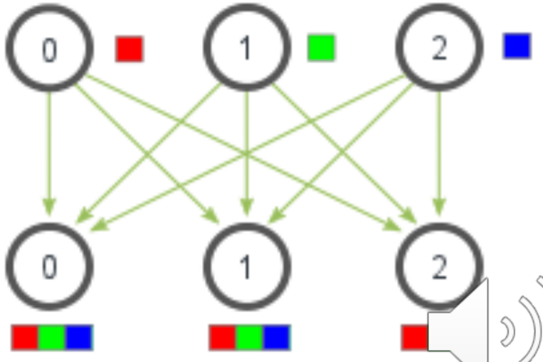
MPI_Scatter()



MPI_Gather()



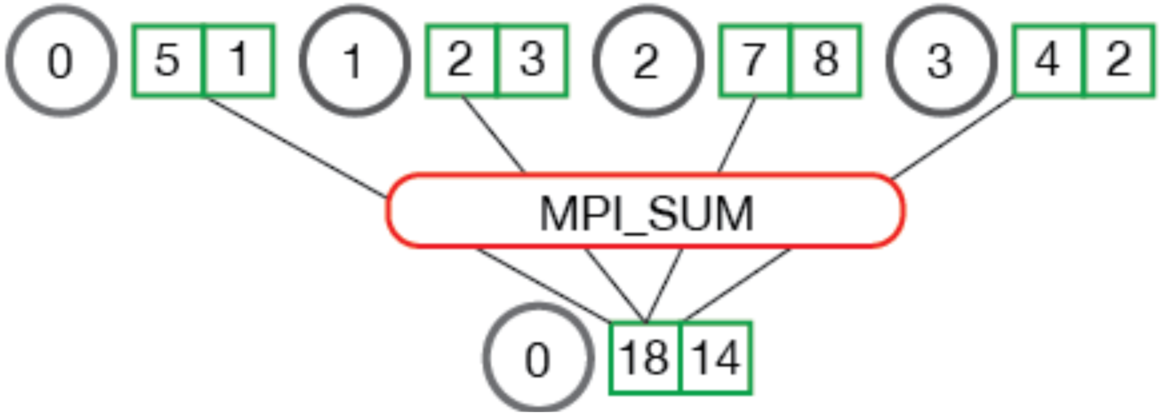
MPI_Allgather()



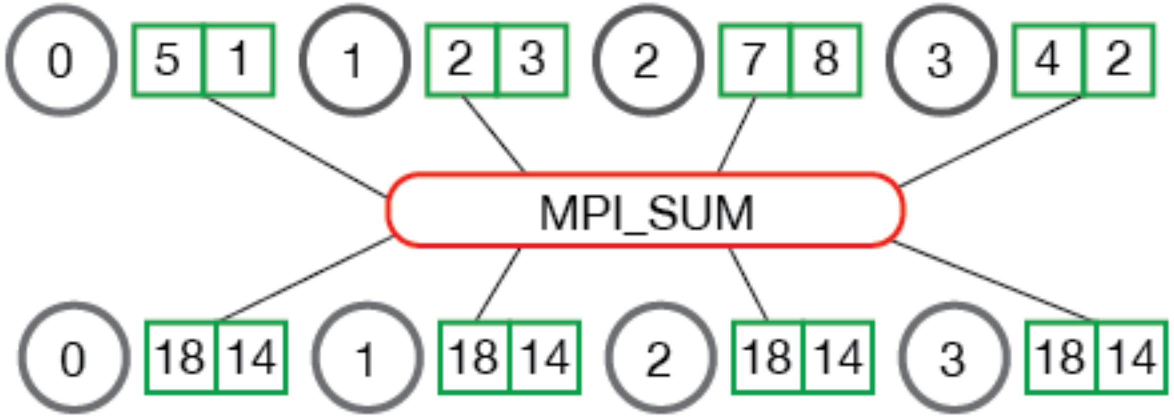
Images from: https://mpitutorial.com

MPI – reduction

MPI_Reduce()



MPI_Allreduce()



Images from: https://mpitutorial.com

MPI with Python

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1)
elif rank == 1:
    data = comm.recv(source=0)
    print('On process 1, data is ',data)
```

Example from: https://rabernat.github.io/research_computing/parallel-programming-with-mpi-for-python.html

Peer-to-peer

- Each node is both a client and server
- No pre-defined coordinators nodes
- Handles joining and leaving
- Resources scale with the amount of participants

Examples
- Skype
- Bitcoin
- Kademlia

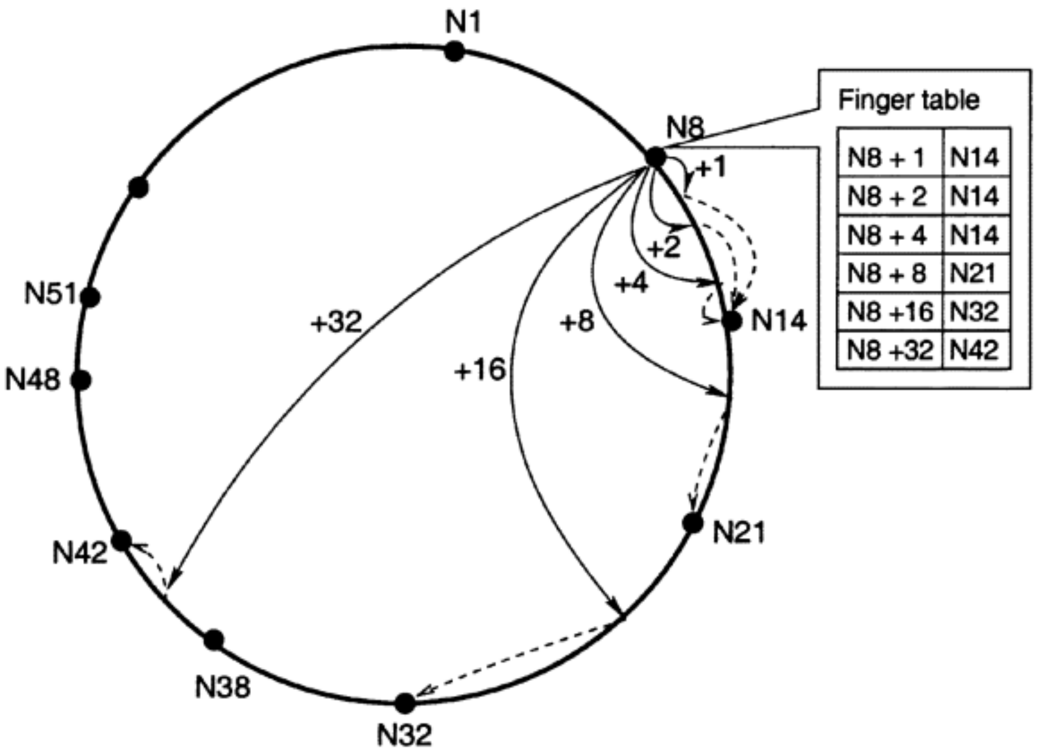
Peer-to-peer example: Kademlia

Basic operation:

- A distributed hash table
- Store key/value pairs
- Retrieve value for given key

Four messages

- PING – liveliness check
- STORE – write a value into the DHT
- FIND_NODE – locate node with data for a given key
- FIND_VALUE – locate data for a given key



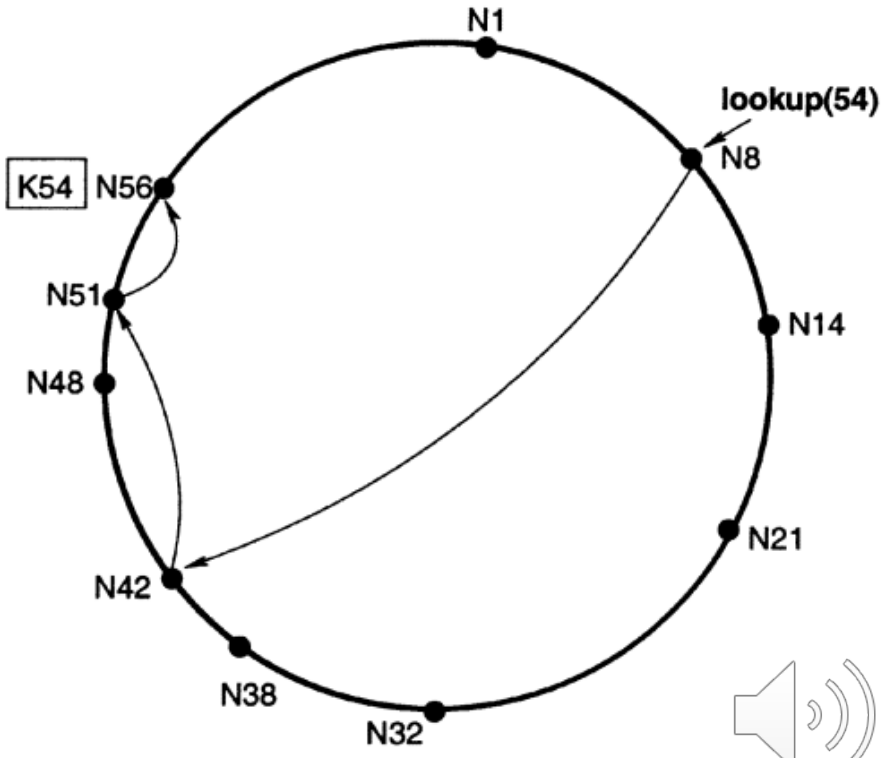


Image from: https://medium.com/unitychain/intro-to-dht-e98425fc05f1

Applications on a network

Need to carefully balance workloads

- Automatic balancing systems are great

Avoid any single-point-of-failure

- Really hard in practice

Resilience to duplicate and lost messages

- Almost the same as evil adversary

Hadoop - HDFS

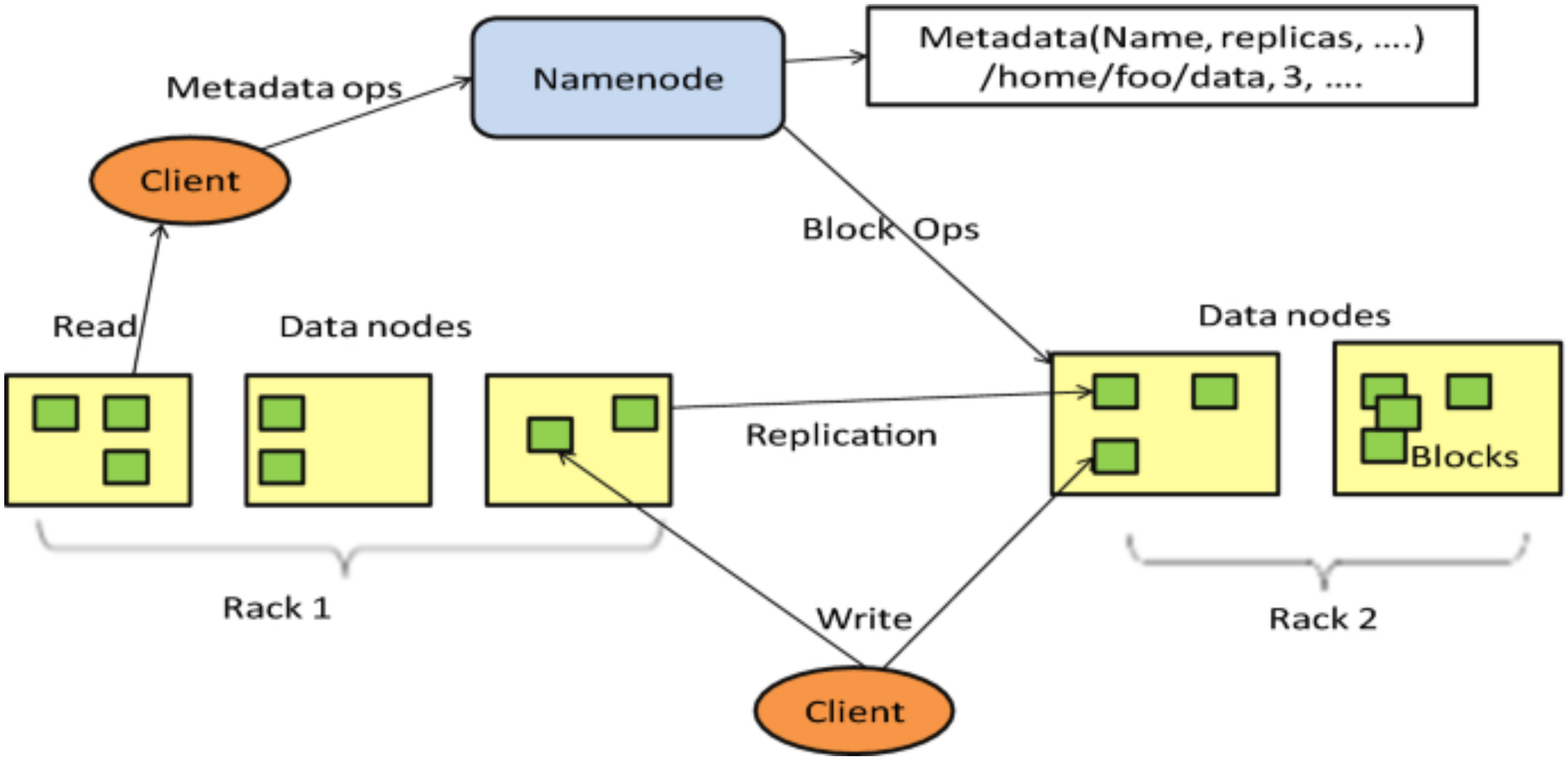
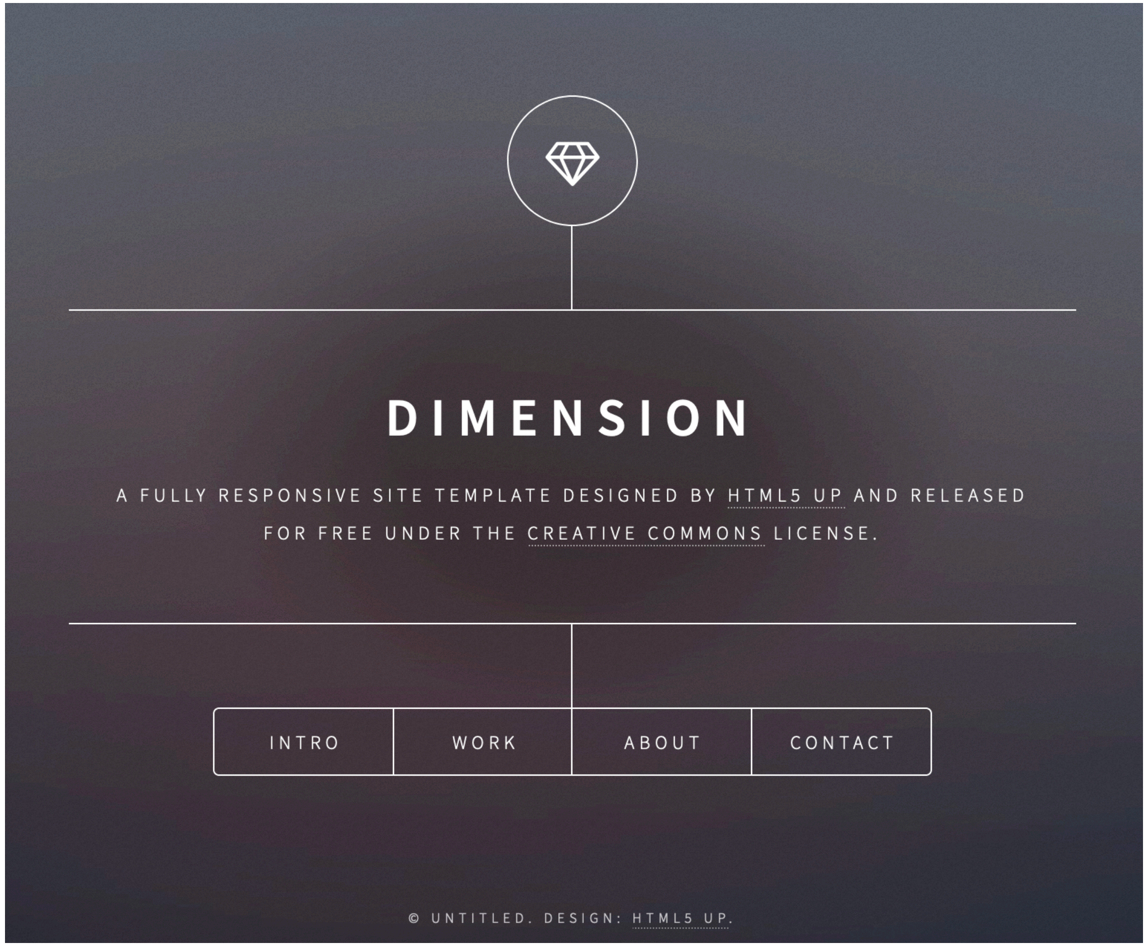


Image from: https://link.springer.com/article/10.1186/s40537-020-00356-z

Assignment #3 – A web server



Example of a web-page with static files: https://github.com/cloudacademy/static-website-example

Assignment #3 – Suggested tasks

1. Set up a server that listens to requests
   - Can use suggestion from https://realpython.com/python-sockets/
2. Write a parser that validates a HTTP/1.1 request
   - Read up on RFC 2616 but *don't read it all!*
   - Be sure to respond correctly to non-`GET` requests
3. Map URL to local path
   - `GET /images/bg.jpg` => /home/user/static-website-example/images/bg.jpg
   - Beware of tricky paths, eg. `GET ../../passwd`
4. Copy local file to socket
   - May want to set `Content-Type` and `Content-Length` headers