Distributed Systems - Lecture 4

Verteilte Systeme (BVS2)

Prof. Dr. Pascal Cerfontaine • Zoom 644 2622 6965 • 263566

Table of contents

Zoom

https://th-koeln.zoom-x.de/j/64426226965

Meeting-ID: 644 2622 6965

Kenncode: 263566

Please post your questions or upvote existing questions during the lecture!

Source: Slides adapted from the BVS2 lecture with permission from Prof. Dr. Carsten Vogt

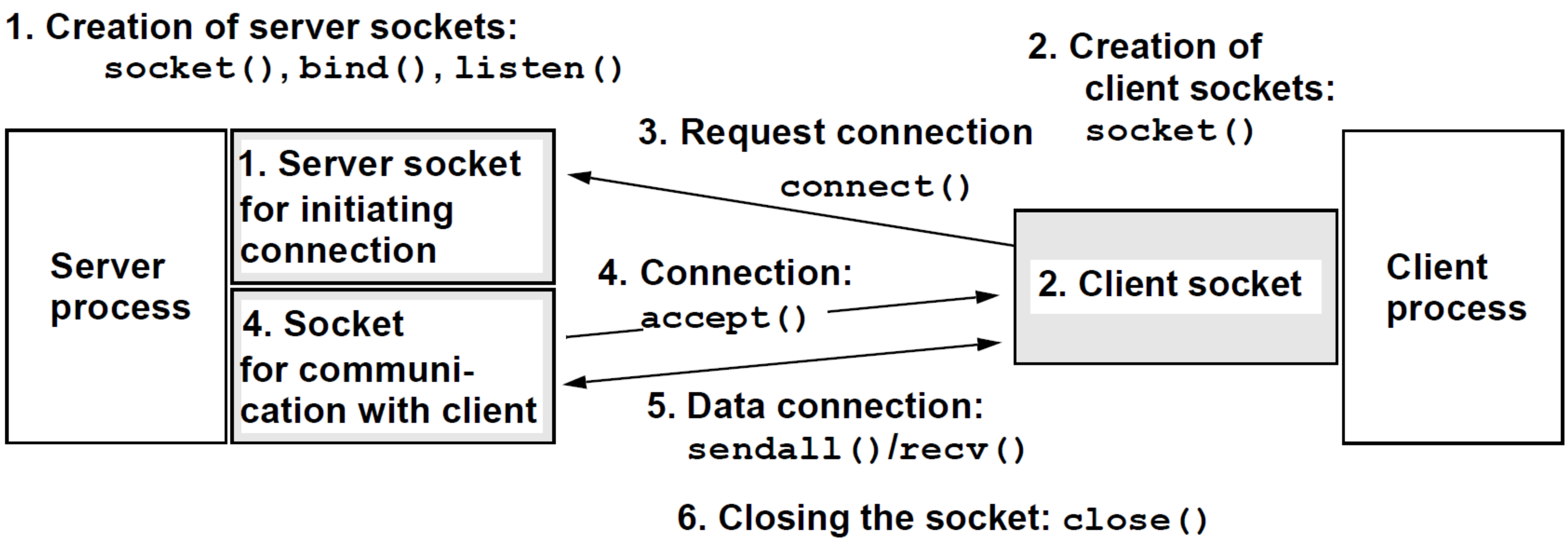
Summary (Last Week)

- **Message Passing**
  - Direct or indirect (via mailbox or port)
  - Synchronous (blocking) vs. asynchronous (non-blocking) communication
  - “Rendezvous” = blocking send + blocking receive
- **Ports and Sockets**
  - Ports: Logical endpoints for client-server coordination
  - Sockets: API to protocol stacks (TCP/IP, UDP/IP)
    - Domains: UNIX (local), Internet (network)
    - Types: Stream (TCP), Datagram (UDP), Raw (manual)
- **Client-Server Communication**
  - Server socket: Waits for connections
  - Client socket: Initiates connection or sends datagrams
- **Programming Model**
  - Setup: Bind → Listen → Accept (server); Connect (client)
  - Supports both local (UNIX) and network (Internet) communications

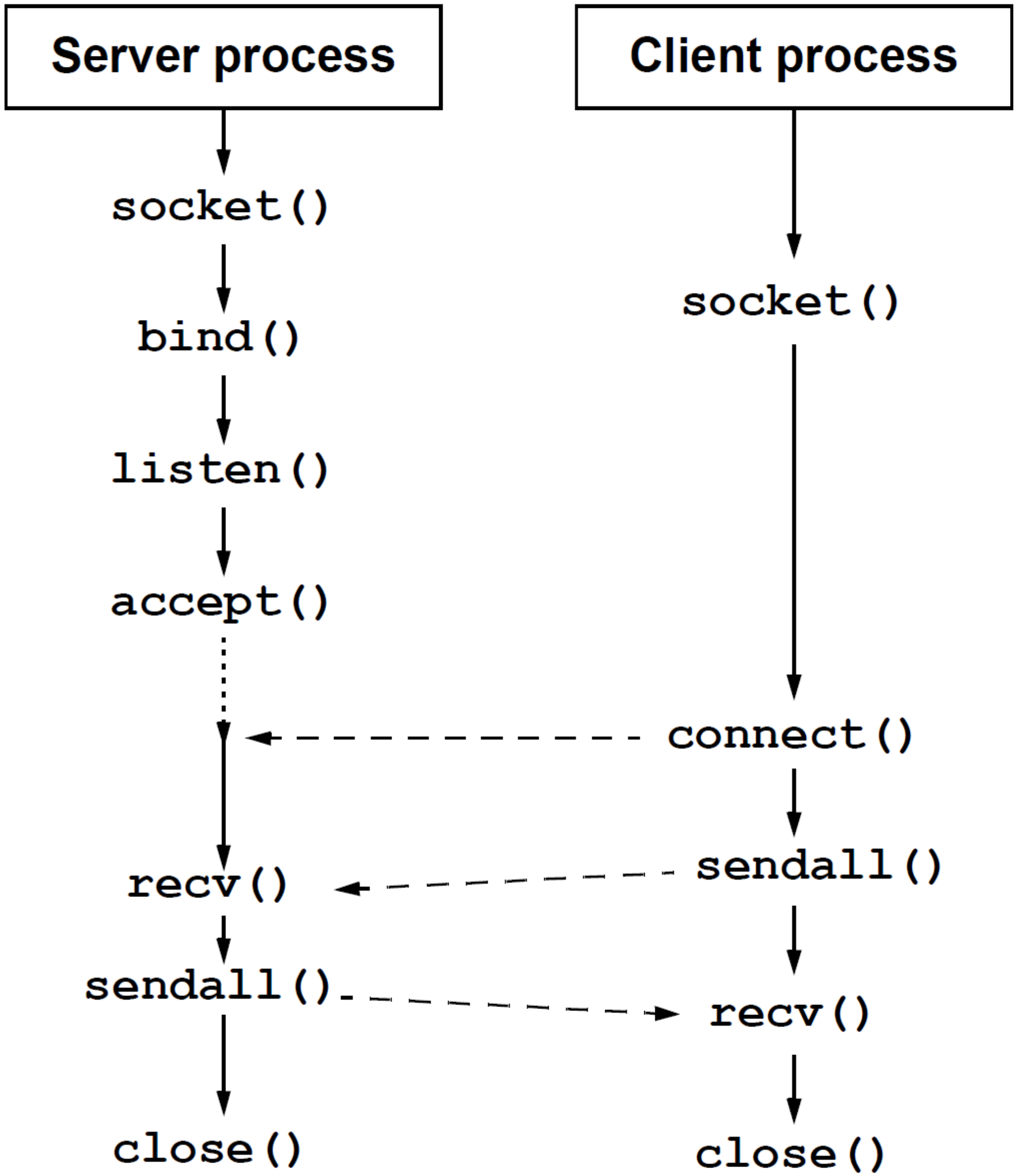
2 Communication in Distributed Systems

2.3 Programming with Sockets

Stream Sockets: Program Steps



Stream Sockets - Timeline



Internet Domain Stream Sockets Example: Client

```
import socket

HOST = 'nebsy.nt.fh-koeln.de'  # Server's IP address (use localhost for local testing)
PORT = 62423         # Server's port

def send_message(message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
        client_socket.connect((HOST, PORT))
        client_socket.sendall(message.encode())
        response = client_socket.recv(1024)
        print("Server response:", response.decode())

if __name__ == "__main__":
    msg = input("Enter a message to send to the server: ")
    send_message(msg)
```

Internet Domain Stream Sockets Example: Server

```python
import socket

HOST = '0.0.0.0'  # Listen on all available interfaces
PORT = 62423      # Port to listen on

def start_server():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
        server_socket.bind((HOST, PORT))
        server_socket.listen()
        print(f"Server listening on port {PORT}...")

        while True:
            conn, addr = server_socket.accept()
            with conn:
                print(f"Connected by {addr}")
                while True:
                    data = conn.recv(1024)
                    if not data:
                        break
                    response = f"Received message: {data.decode()}"
                    conn.sendall(response.encode())

```

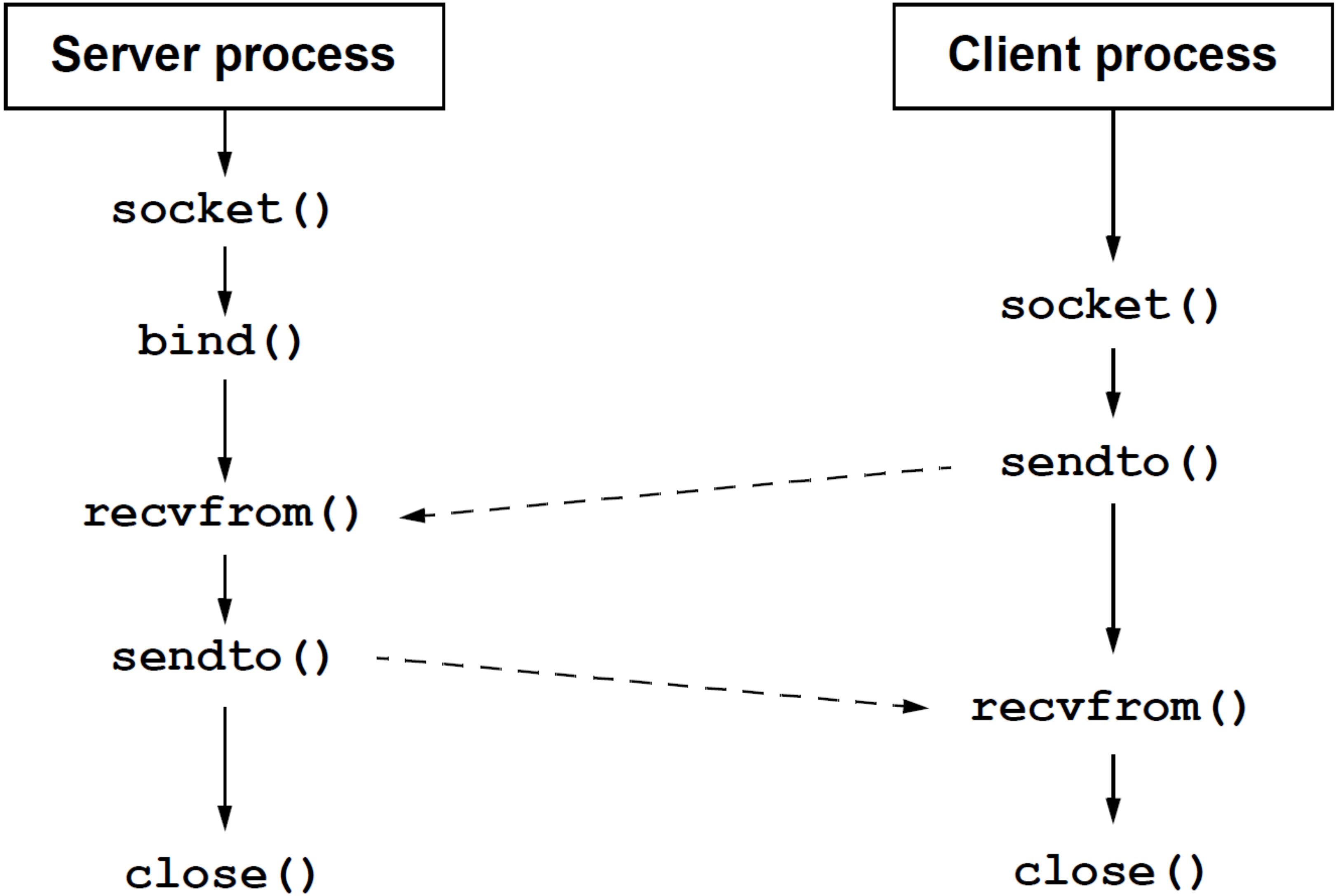
(See uploaded Python files for full code listing)

Stream Sockets - Programming Remarks

- Server can send a result back to the client over the same connection
- Server: `conn.sendall(response.encode())`
- Client: `client_socket.recv(1024)`
- If the number of bytes is unknonw:

```python
# Buffer to hold the received data
received_data = b""

# Keep receiving data until the server closes the connection
while True:
    chunk = sock.recv(4096)  # You can adjust buffer size as needed
    if not chunk:
        # No more data (connection closed)
        break
    received_data += chunk

print("Received data:")
print(received_data.decode('utf-8', errors='replace'))
```

Datagram Sockets - Timeline



Internet Domain Datagram Sockets Example: Client

```python
import socket

def udp_client(server_host='127.0.0.1', server_port=12345):
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as client_socket:
        while True:
            message = input("Enter message (or 'exit' to quit): ")
            if message.lower() == 'exit':
                break

            client_socket.sendto(message.encode(), (server_host, server_port))
            data, _ = client_socket.recvfrom(1024)
            print(f"Received from server: {data.decode()}")

if __name__ == "__main__":
    udp_client()
```

Internet Domain Datagram Sockets Example: Server

```python
import socket

def udp_server(host='127.0.0.1', port=12345):
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as server_socket:
        server_socket.bind((host, port))
        print(f"UDP server listening on {host}:{port}")

        while True:
            data, addr = server_socket.recvfrom(1024)  # Buffer size is 1024 bytes
            print(f"Received from {addr}: {data.decode()}")

            response = f"Echo: {data.decode()}"
            server_socket.sendto(response.encode(), addr)

if __name__ == "__main__":
    udp_server()
```

(See uploaded Python files for full code listing)

Raw Sockets

The application must create the packet header itself and can read the packet header

- This is in contrast to stream sockets and datagram sockets where this is automatically handled by the protocol implementation
- Advantage: The application has access to the headers
- e.g., for experimenting with header entries
- e.g., for sniffing

Select and Poll

- A single functional call that waits for incoming data on several sockets
- Enables server to serve several clients
- Can add timeouts

Select

- `select.select()` monitors multiple sockets (or files) for readability, writabilityand errors
- Useful in non-blocking or asynchronous I/O

```python
import select
readable, writable, errored = select.select(inputs, outputs, inputs, timeout)
```

Poll

- `poll()` is more scalable than `select()` (better for many sockets)
- Works with a poll object and file descriptors

```python
poller = select.poll()
poller.register(socket.fileno(), select.POLLIN)
```

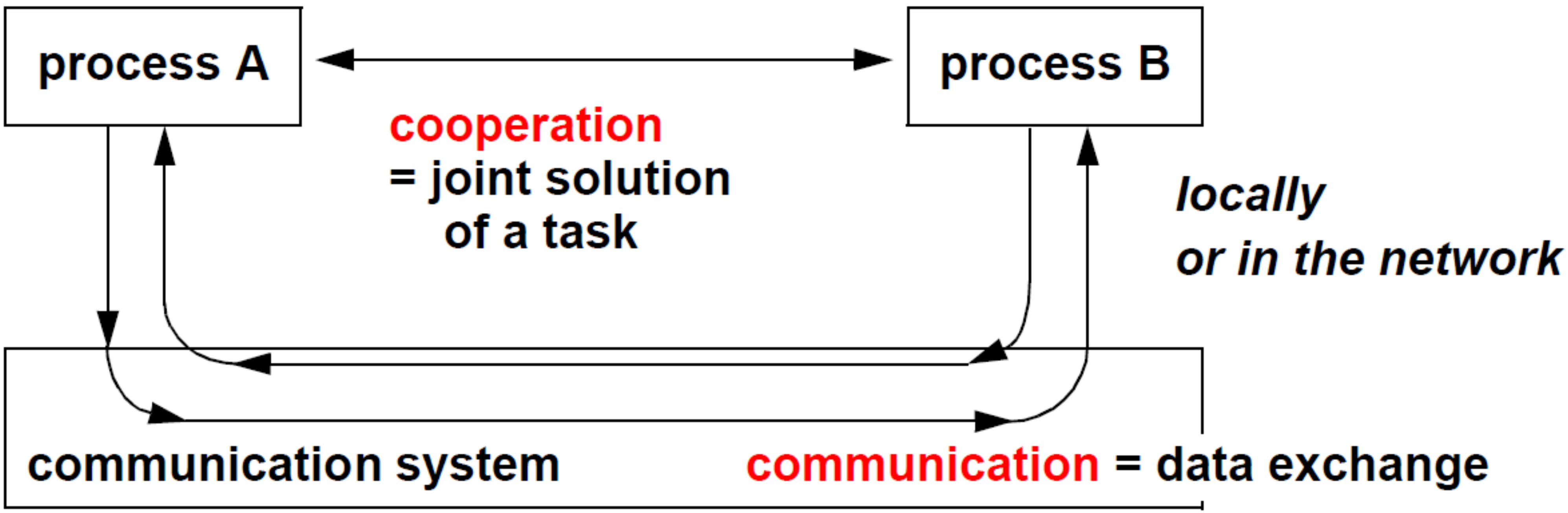
- Events are checked using `poller.poll(timeout)`

Select Example

```python
import socket, select

server = socket.socket()
server.bind(('localhost', 12345))
server.listen()

inputs = [server]

while True:
    readable, _, _ = select.select(inputs, [], [])
    for s in readable:
        if s is server:
            conn, addr = server.accept()
            inputs.append(conn)
        else:
            data = s.recv(1024)
            if data:
                print("Received:", data.decode())
            else:
                inputs.remove(s)
                s.close()
```

3 Cooperation

Communication and Cooperation



- Often: cooperation based on the **client-server** paradigm
  → In this chapter:
  - Fundamental properties of client-server systems
  - Briefly: **peer-to-peer** as an alternative
  - Implementation techniques for client-server systems

3.1 The Client-Server Model: Fundamental Properties

Client-Server System: Roles

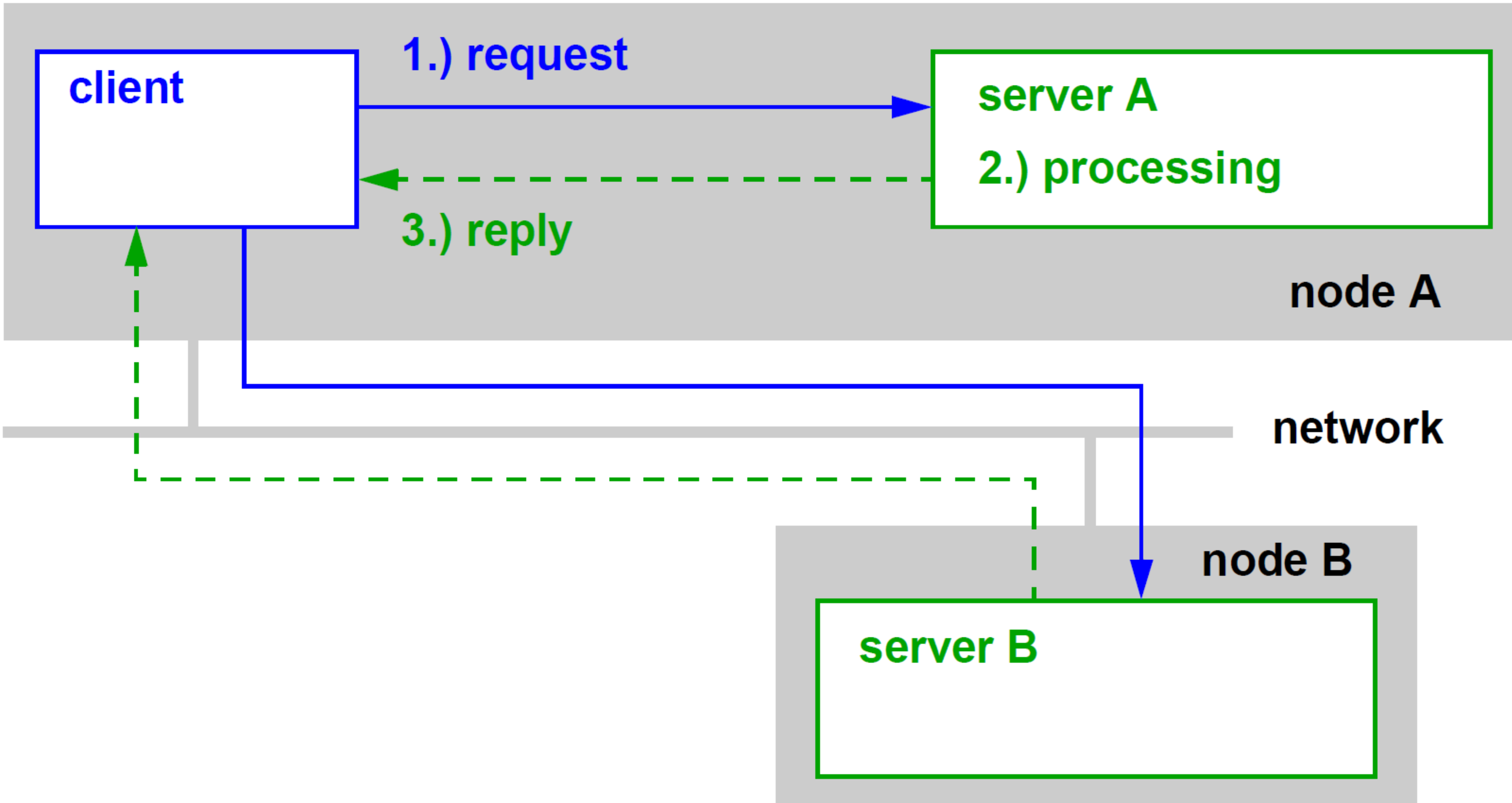
Roles for Processes / Threads / Computer Nodes:

- **Server**: Provider of services
  - e.g., access to databases, file systems, documents
- **Client**: User of services
  - Calls services locally or over the network

Dual Role:

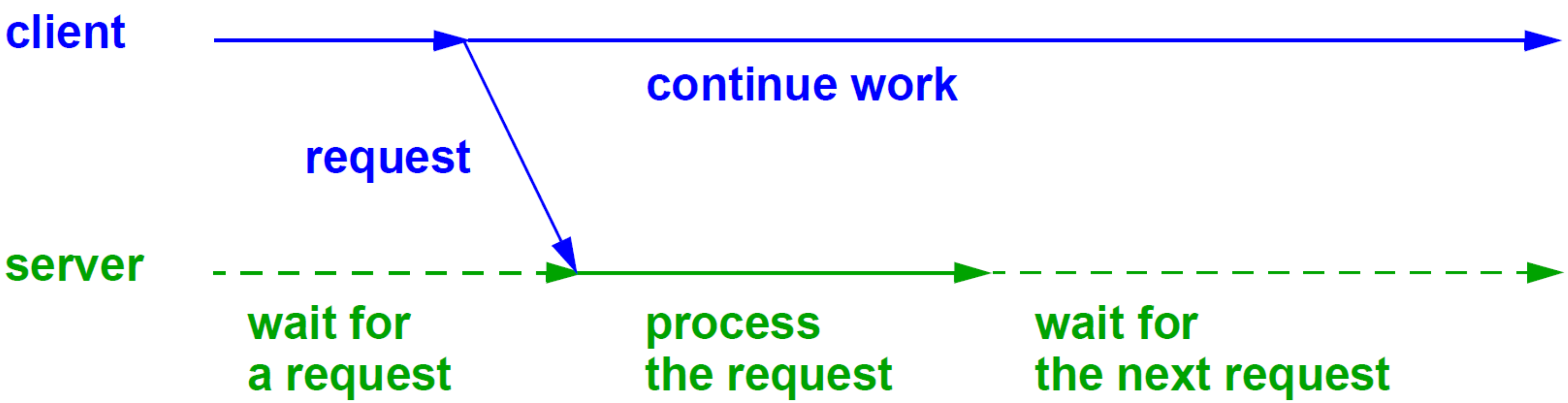
- A process / thread / computer node can be both:
  - A **server**: provides some services
  - And a **client**: uses other services

Client-Server Cooperation: Steps

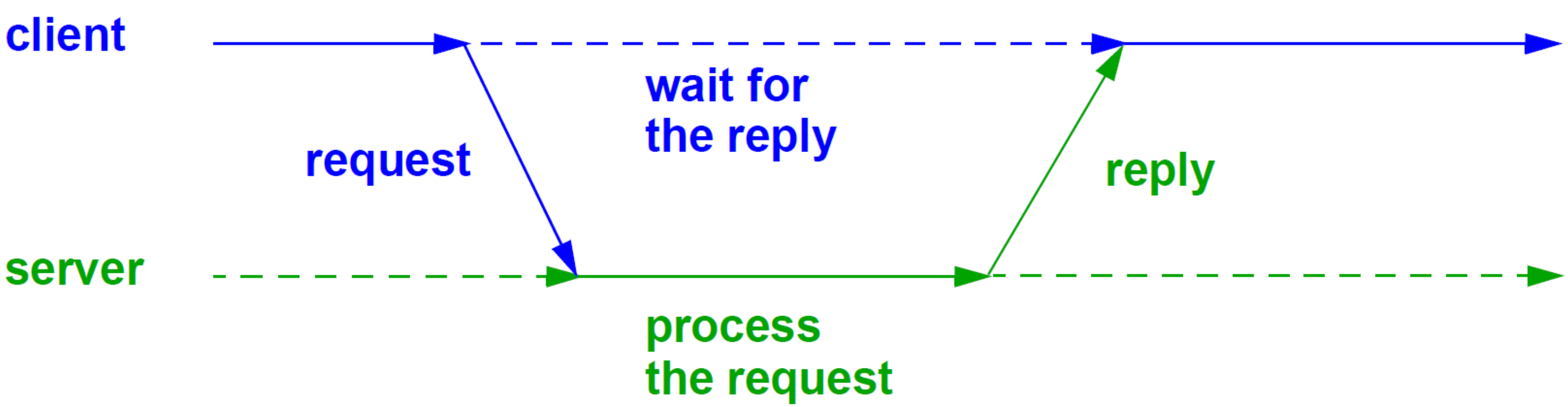


Client-Server Cooperation: Timing Behaviour

a.) Cooperation without reply:

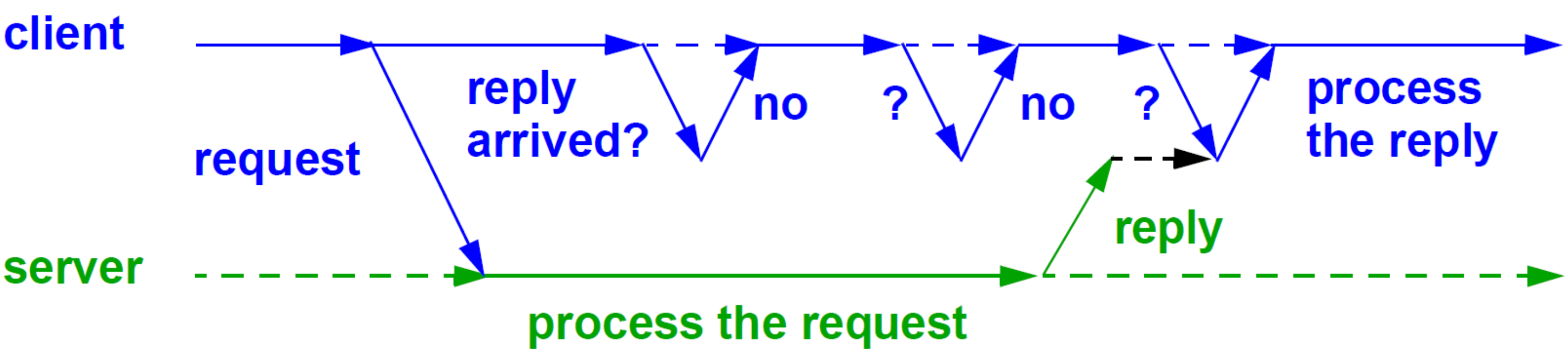


b.1.) Synchronous cooperation with reply

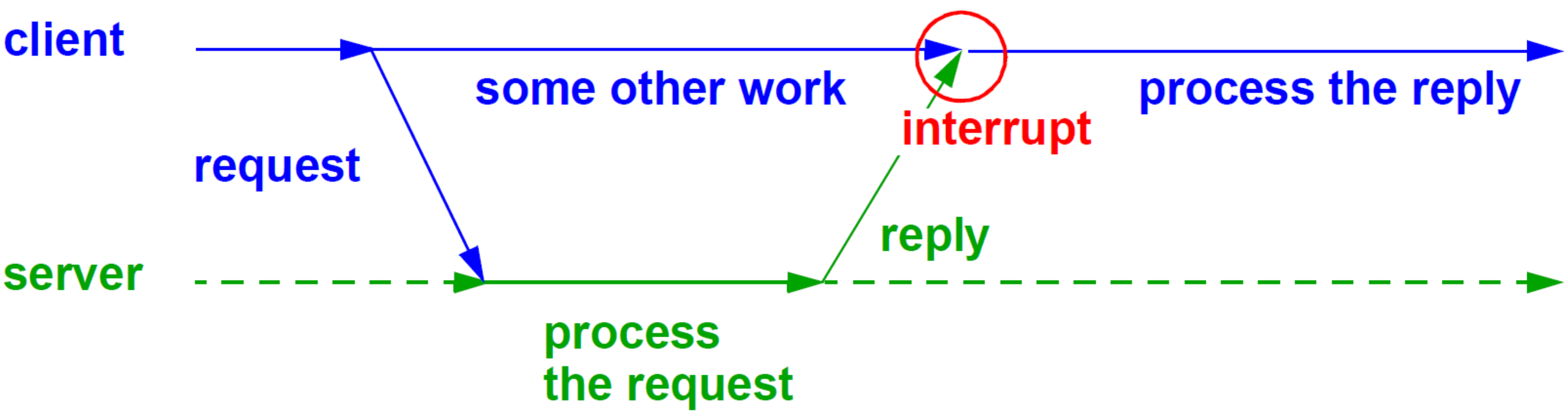


Client-Server Cooperation: Timing Behaviour

**b.1.1.) Asynchronous cooperation with reply:
Client polling for the reply**



**b.1.2.) Asynchronous cooperation with reply:
Reply interrupting the client**



Client-Server Cooperation: Possible Errors

- Messages can be:
  - Lost
  - Distorted
  - Duplicated
- The server might be:
  - Down
  - Unreachable

Consequences:

- It is **not guaranteed** that a request:
  - Is processed at all
  - Is not processed **multiple times**

→ Additional mechanisms are needed to provide different types of **guarantees**

Client-Server Cooperation: Guarantees & Mechanisms

"At Most Once" Guarantee

- A request is processed **at most once**
- **Mechanism**: Sequence numbers
  - Each request has a unique number
  - Server discards requests with numbers it has already seen

"At Least Once" Guarantee

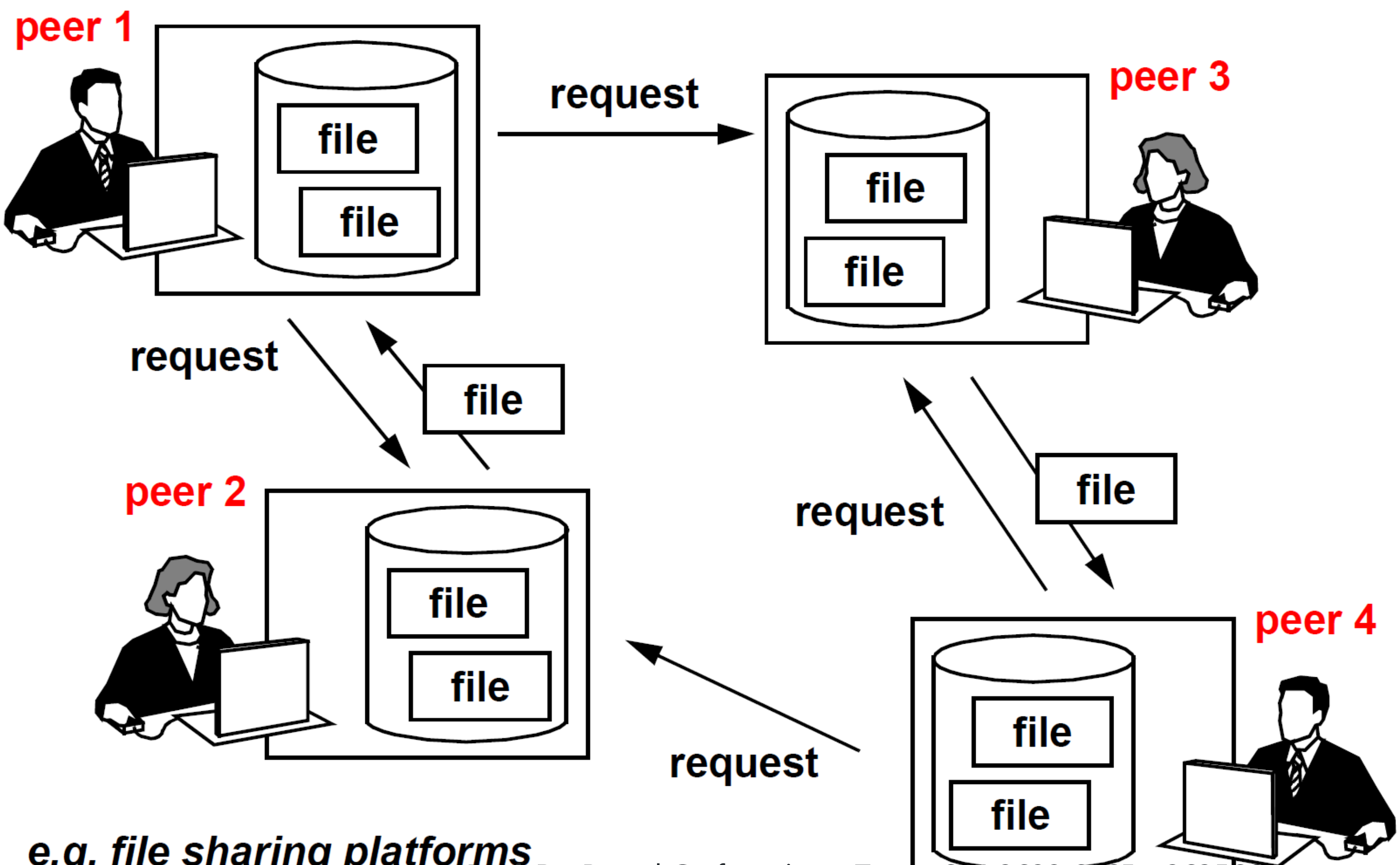
- A request is processed **at least once**
- **Mechanism**: Acknowledgments
  - Server acknowledges receipt of a request
  - Client resends requests not acknowledged within a timeout
  - May resend to a different server

"Exactly Once" Guarantee

- A request is processed **exactly once**
- **Mechanism**: Combination of the above two

3.2 Alternative: Peer-to-Peer Cooperation (P2P)

- Files / services are distributed over „peer nodes“
- Each node can use services of the other nodes



e.g. file sharing platforms

Alternative: Peer-to-Peer Cooperation (P2P)

P2P: Decentralized Organization

- Client-server: centralized organization with a few servers
- P2P: decentralized, all participants can act as both client and server

Advantages

- Many resources available
- Load distribution and fault tolerance
  - Achieved by replicating resources
- Self-organization

Drawbacks

- Higher costs for searches and communication
- Security and data protection are harder to enforce

Summary

- **Stream Sockets (TCP):**
  - Connection-oriented: server listens and accepts, client connects.
  - Communication via `sendall()` and `recv()`.
- **Datagram Sockets (UDP):**
  - Connectionless: each `sendto()`/`recvfrom()` is a discrete message.
  - No guaranteed delivery/order—requires manual error handling.
- **Client-Server Model:**
  - Server = service provider; Client = service consumer.
  - Cooperation sequence: request → processing → response.
- **Communication Guarantees:**
  - *At most once*: sequence numbers. *At least once*: acknowledgments + retries.
  - *Exactly once*: combine both mechanisms.
- **P2P Alternative:**
  - Fully decentralized, each node is both a client and a server.
  - Benefits: redundancy, scalability. Challenges: discovery, security, consistency.

In two weeks

- More on the client-server model
- Remote Procedure Call