Communication in Distributed Systems

Tran Giang Son, tran-giang.son@usth.edu.vn

ICT Department, USTH

Commmunication

What?

> *The concept or state of exchanging data or information between entities.*
>
> - Wikipedia

- Local communication
- Network communication

Why?

- Local scale needs communication
- Distribution needs a means to communicate

How: Local communication

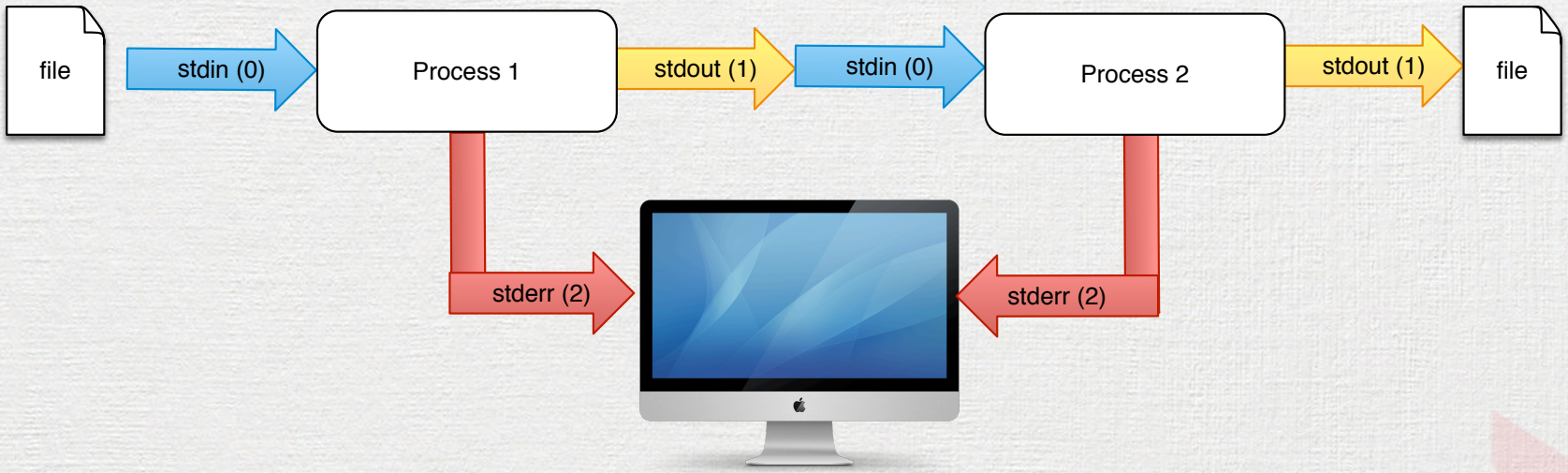
- Remind: UNIX philosophy

How: Local communication

- Remind: UNIX philosophy
- Remind: IO Redirection

How: Local communication

- Remind: UNIX philosophy
- Remind: IO Redirection



How: Local communication

- Inter-Process Communication (local)
  - Signal
  - UNIX domain socket
  - Shared memory
  - Pipe

How: Local communication - Signal

- What?
  - Software generated interrupts
    - Sent to a process when an event happens
    - e.g. `SIGSTOP`, `SIGCONT`, `SIGSEGV`, `SIGINFO`
  - Asynchronous
  - Limited (31 signals only)
- Why?
  - Standard in UNIX
  - Early form of IPC

How: Local communication - Signal (cont.)

- How?
  - Implement signal handler

```
void handler(int sig) {}
```

  - Register

```
void (*signal(int sig, void (*func)(int)))(int);
```

How: Local communication - Signal (cont.)

- How?
  - Implement signal handler

```
void handler(int sig) {}
```

  - Register

```
void (*signal(int sig, void (*func)(int)))(int);
```

- See TinySU for a more detailed example ☺

How: Local communication - UNIX Domain Socket

- What?
  - Local socket
    - Bi/uni-directional
    - Similar to network sockets
  - Seen as files in VFS, but not really files
  - "bind" the pathname to socket
- Why?
  - Faster than TCP loopback [1]
  - Less overhead

[1] Benchmark tool is called ipc-bench

How: Local communication - UNIX Domain Socket

- How? Similar to network socket...
  - `socket(AF_UNIX, SOCK_STREAM, 0);`
  - `struct sockaddr_un addr;`
  - `addr.sun_path` points to a path in VFS
  - `bind()`
  - `listen()`
  - `connect()`
  - `accept()`

How: Local communication - UNIX Domain Socket

- How? Similar to network socket...
  - `socket(AF_UNIX, SOCK_STREAM, 0);`
  - `struct sockaddr_un addr;`
  - `addr.sun_path` points to a path in VFS
  - `bind()`
  - `listen()`
  - `connect()`
  - `accept()`
- See TinySU for a more detailed example ☺

How: Local communication - Pipe

- What?
  - FIFO mechanism to pass data
  - Output of one is input of another
  - Unidirectional
- Why?
  - Simple to use for simple communication
    - no `socket()`, `bind()`, `listen()`, `connect()`, `accept()`
  - Just like sockets...

How: Local communication - Pipe (cont.)

- How?
  - `int mypipe[2];`
  - `pipe(mypipe);`
  - `mypipe[0]` is the read end
  - `mypipe[1]` is the write end
  - Data written to `mypipe[1]` can be read from `mypipe[0]`
  - Use **before** `fork()`

How: Local communication - Pipe (cont.)

- How?
  - `int mypipe[2];`
  - `pipe(mypipe);`
  - `mypipe[0]` is the read end
  - `mypipe[1]` is the write end
  - Data written to `mypipe[1]` can be read from `mypipe[0]`
  - Use **before** `fork()`
- See TinySU for a more detailed example ☺

How: Local communication - Shared memory

- What?
  - A memory region that can be accessed by different local processes
  - Permission support
- Why?
  - Fast
  - Large
  - Structured

How: Local communication - Shared memory (cont.)

- How?
  - `shmget(key, size, flag)`: create/obtain access to a shared memory segment
  - `shmctl(id, cmd, data)`: control a shared memory segment (perm., lock...)
  - `shmat(id, addr, flag)`: attach a shared memory segment to a process
  - `shmdt(addr)`: detach a shared memory segment from a process

How: Local communication - Shared memory (cont.)

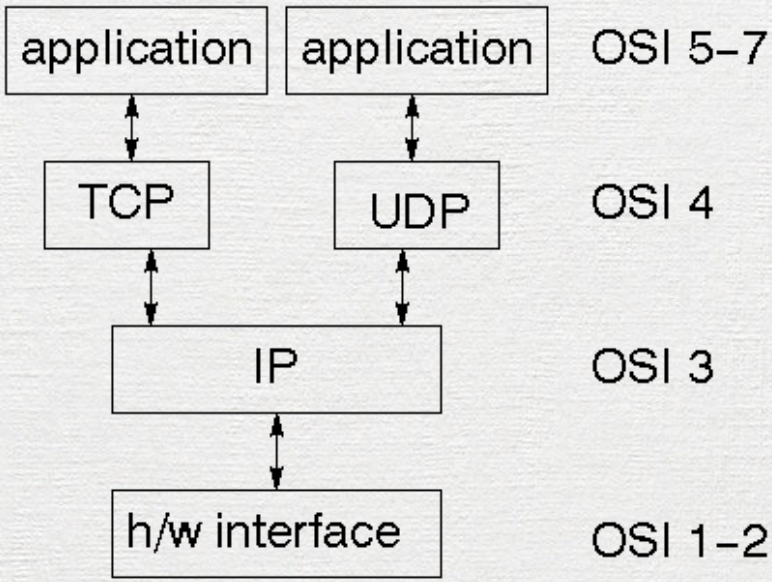
- How?
  - `shmget(key, size, flag)`: create/obtain access to a shared memory segment
  - `shmctl(id, cmd, data)`: control a shared memory segment (perm., lock...)
  - `shmat(id, addr, flag)`: attach a shared memory segment to a process
  - `shmdt(addr)`: detach a shared memory segment from a process
- Unfortunately, TinySU doesn't use shared memory yet ☺

How: Inter-Node Communication

- Remind: OSI model

How: Inter-Node Communication

- Remind: OSI model



How: OSI model

1. Physical: mechanical & electrical details
2. Data link: group bits into frames
3. Network: routing packets
4. Transport: transfer messages, breaking to packets, connection-oriented or connectionless
5. Session: dialog control & synchronization
6. Presentation: data format diferrence solutions
7. Application: well - what you make

How?

- Inter-Node Communication on top of IP
  - Socket (Ref. Practical work 1)
  - Remote Procedure Call
  - Message Passing

RPC

What?

- **R**emote **P**rocedure **C**all
- Mechanism to call procedures on other machines
- Widely used in distributed computing

Why?

- Socket is used mainly for data transfer
  - `connect()`
  - `read()` / `write()`
  - `close()`
- App
  - Procedure calls
  - e.g. `checkLogin()`, `verifyCardNumber()`
- RPC makes distributed computing look like centralized system

How?

- Local procedure call

```
x = f(a, "test", 5);
```

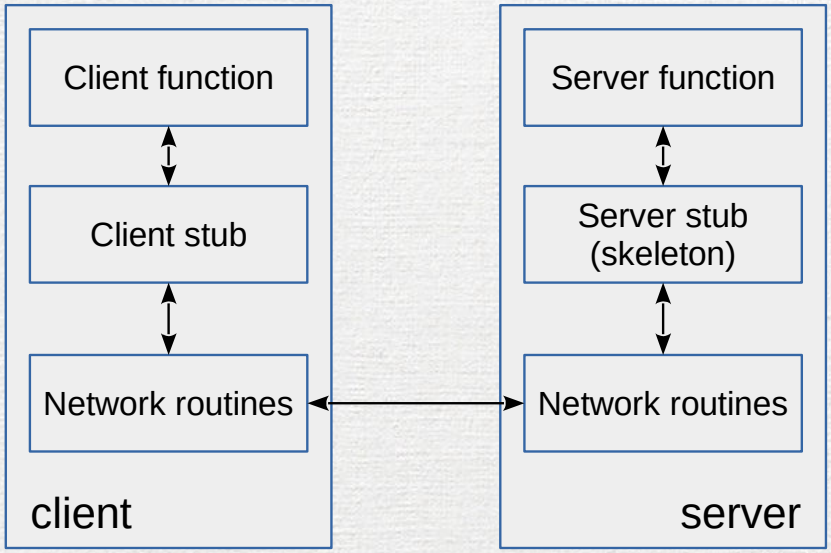
- Generated code:
  - Push the value 5 on the stack – Push the address of the string “test” on the stack – Push the current value of a on the stack – Generate a call to the function `f`
  - Execute `f`
  - Assign the return value of `f` back to `x`
- RPC: No architecture support

$\Rightarrow$ Simulate it

How?

- Create stub functions
  - Name only
  - No implementation
  - Appear to the user as local

How?



- Function to/from stub: params on stack
- Stub to/from network routine: "marshals"/"unmarshals" [2] the params

[2] Previously known as "(de)serialize" in Mobile Dev

How?

- Marshaling / Unmarshaling
  - Scalar values: easy
  - Pass by reference: no shared memory
    - Objects
    - Strings
    - Requires object/pointer reconstructions
- Not all languages support reflection for automatic serialization
  - e.g. C/C++

How?

- Runtime problems
  - Client cannot locate server
  - Request (call) is lost
    - Network down
    - Timed out
  - Server crash
  - Interface is incompatible
  - Security
  - Performance

How?

- Other consideration
  - Byte ordering
  - 16/32/64 bit ints
  - Float/double
  - Text encoding

How?

- Interface Definition Language
  - Define remote procedure interfaces
  - Names, parameters, return values
- IDL syntax
  - Looks like function prototypes
  - Varies with RPC compiler

How?

Example: CORBA IDL

```
interface CaesarAlgorithm {
    typedef sequence<char> charsequence;
    charsequence encrypt(in string info,
            in unsigned long k,
            in unsigned long shift);
    string decrypt(in charsequence info,
            in unsigned long k,
            in unsigned long shift);
    boolean shutdown();
};
```

How?

- RPC compiler
  - Parses IDL
  - Generate stubs
    - Both client and server
    - (Un)marshaling code
    - Network transport routines
    - Required headers

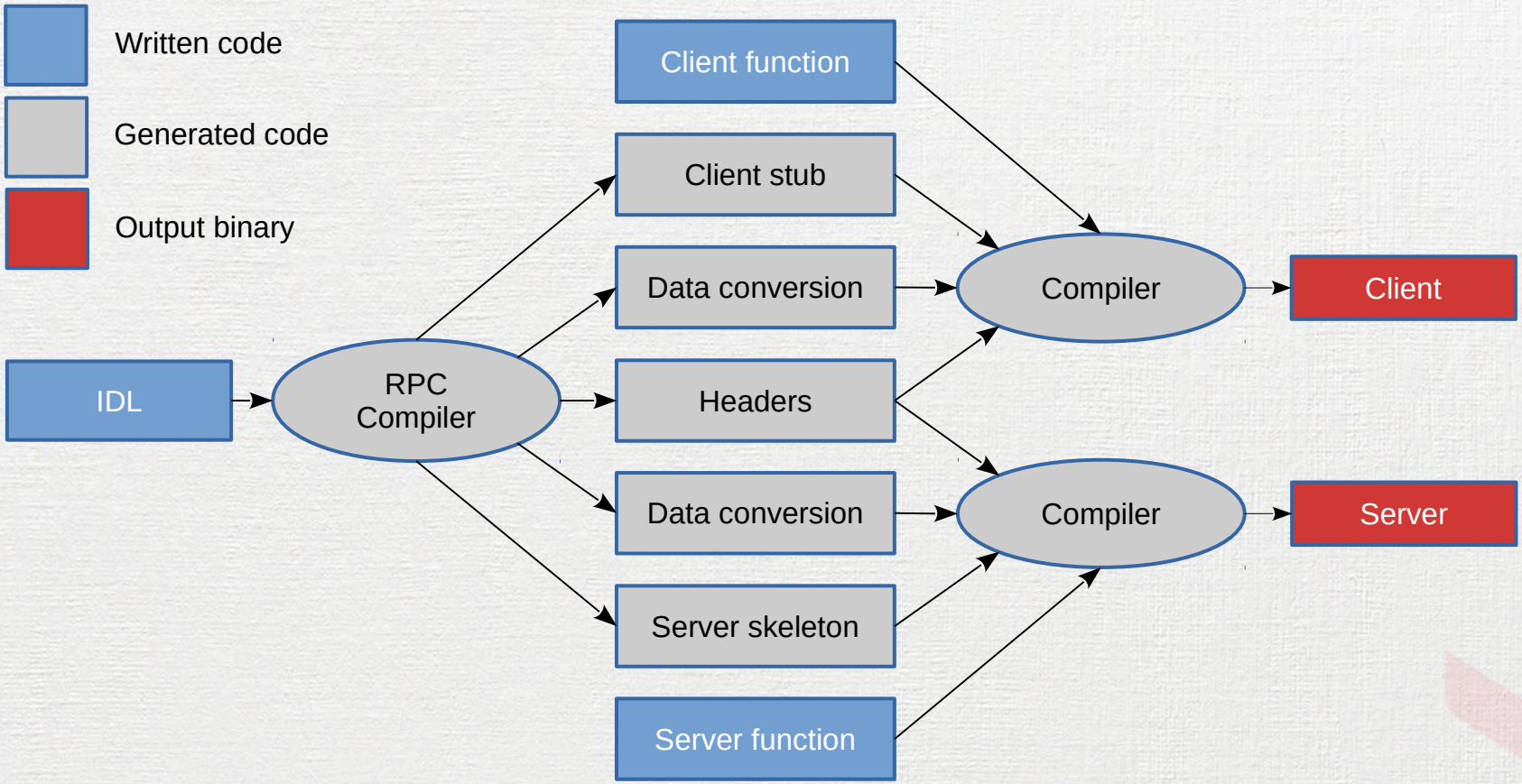
How?

- RPC compiler examples
  - MIDL
  - OmniORB omniidl (CORBA based)
  - Integrated JAVA `Remote` interface

How?

- Naming service
  - Optional
  - The central point of servers and procedures
  - RPC servers register their addresses and procedures
  - RPC clients connect to the "Registry" and ask for procedures with server addresses
- Example
  - Java RMI
  - CORBA

How?


Written code
Generated code
Output binary
IDL
RPC Compiler
Client function
Client stub
Data conversion
Headers
Data conversion
Server skeleton
Server function
Compiler
Compiler
Client
Server

Practical work 2: RPC File transfer

- Copy your TCP file transfer system to a new directory called RPC
- Upgrade it to a file transfer system using RPC
  - Use any kind of RPC service
- Write a short report in LaTeX:
  - Name it « 02.rpc.file.transfer.tex »
  - How you design your RPC service. Figure.
  - How you organize your system. Figure.
  - How you implement the file transfer. Code snippet.
  - Who does what
- Work in your group, in parallel
- Push your report to corresponding forked Github repository

MPI

What?

- **M**essage **P**assing **I**nterface
  - A standard
  - Not a language
  - Not a compiler
  - Not a specific implementation
- For parallel computers, clusters, grids
- C/C++/Fortran

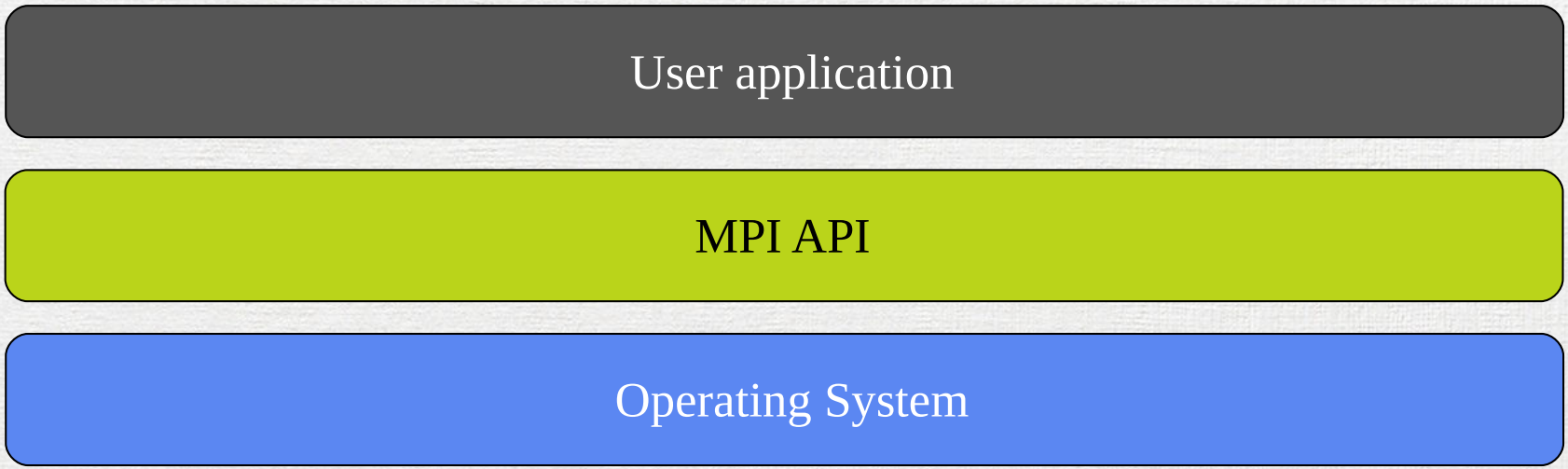
What?

- Software implementations of specs
  - HP MPI
  - Intel MPI
  - Scali MPI
  - OpenMPI
  - MPICH

What?

- "Middleware"
  - Sits between the application and network
  - Simplifies network activity to the application
- Source code portability
  - Run apps on commodity clusters and supercomputers

What?



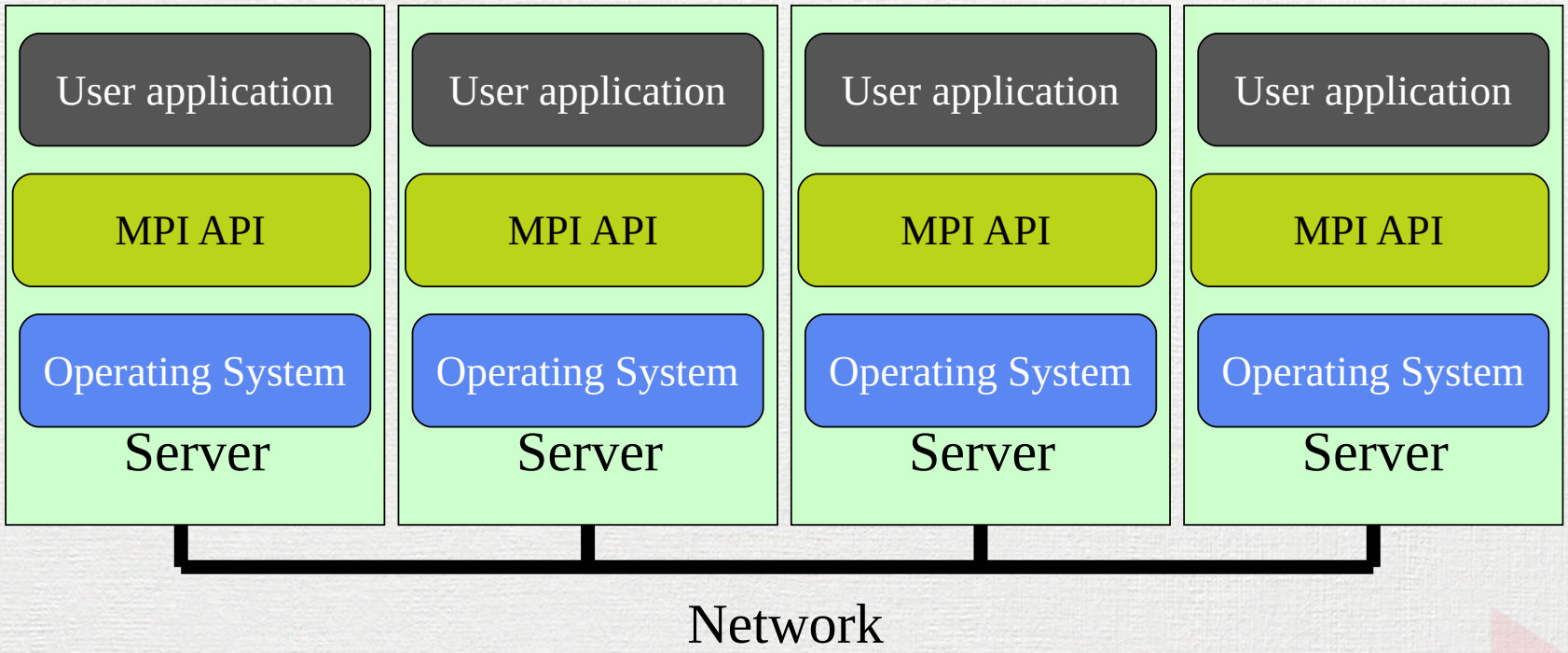
Why?

- Powerful, efficient
  - Low latency
  - High bandwidth
- **Scalable** (!)
- Portable
  - Different OS
  - Different network backends
    - Regular TCP/IP
    - Infiniband
    - ...

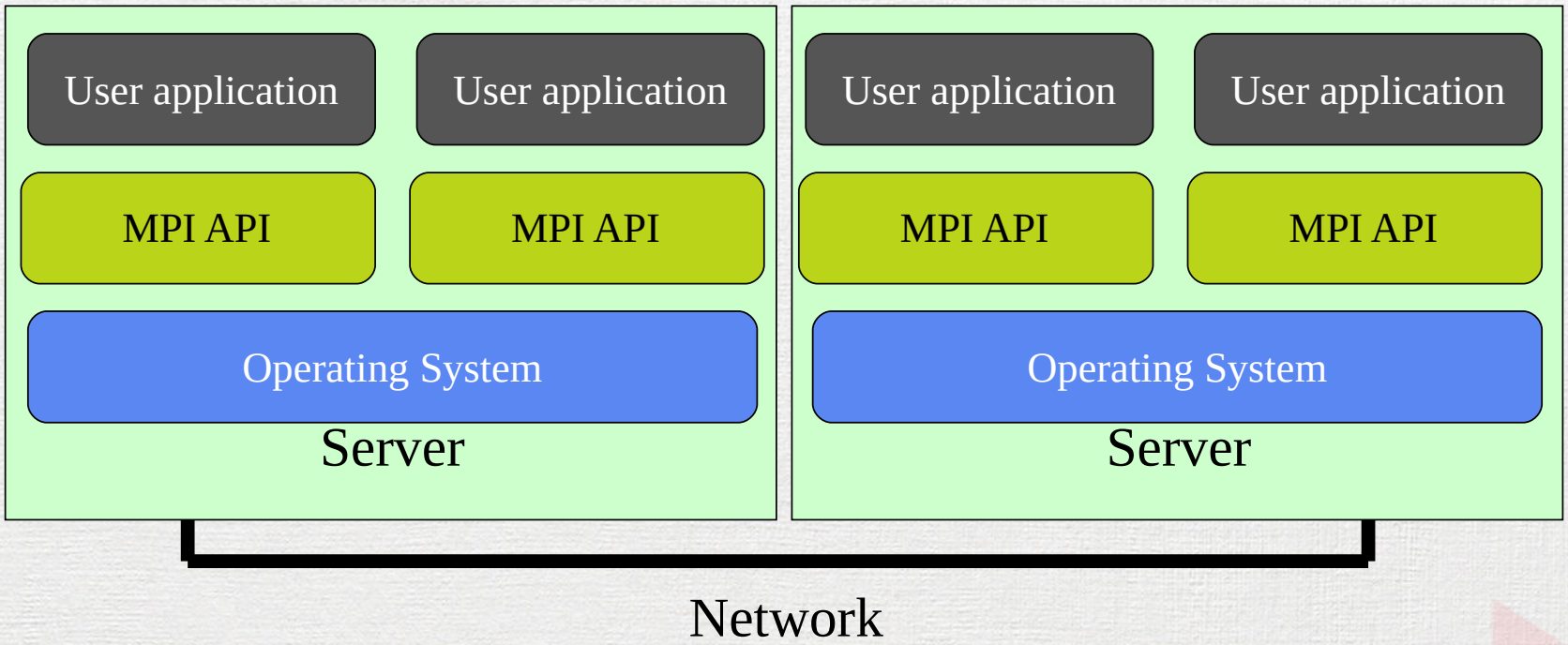
Why?

- Abstraction of the network
  - Sockets
  - Shared memory
  - Ethernet
  - Infiniband

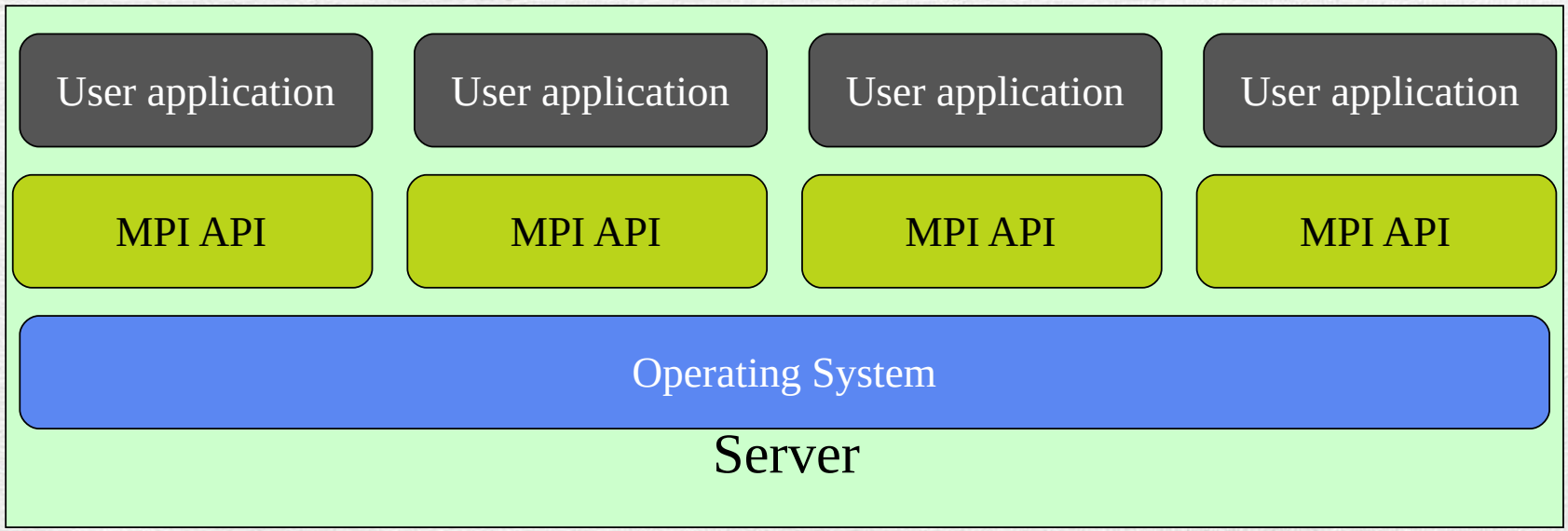
Why: Abstraction of the network



Why: Abstraction of the network



Why: Abstraction of the network



Why: MPI vs Socket?

Criteria	MPI	Socket
Abstraction	High level	Low level
Complexity	Low	High
Network backend	Infiniband, TCP	TCP/UDP
Connection	Automatic	Manual
Commmunication	Point-to-point, Collective	Point-to-point

How?

- Group: set of processes
- Process ID: "rank" in the group
- Communicator: set of processes that can communicate with each other
- Default communicator
  - Group contains all initial processes
  - `MPI_COMM_WORLD`

How: Hello world!

```c
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    printf("Hello, world!\n");
    MPI_Finalize();
    return 0;
}
```

How: Hello world!

```
$ gcc mpi_example.c -o mpi_example
$ mpiexec mpi_example
```

How: What next?

- Datatypes
- Identify sender/receiver
- Send/Receive
- Synchronization

How: Datatypes

- Data is sent with a triple (target, count, datatype)
- Datatypes:
  - Predefined (e.g. `MPI_INT`, `MPI_DOUBLE_PRECISION`)
  - Contiguous array of MPI datatypes
  - Strided block of datatypes
  - Indexed array of blocks of datatypes
  - Structure of datatypes

How: Datatype

- Messages can be attached with "Tags"
- Identify the message type
- `MPI_ANY_TAG`

How: Sender/receiver

- `rank` / `comm` (`MPI_COMM_WORLD`)
- How many processes are there?

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- Who am I?

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

How? Send/Receive

```
int MPI_Send(const void *buf, int count,
    MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm);
```

- `buf`: data to send
- `count`: number of element to send
- `datatype`: `MPI_SHORT`, `MPI_INT`, `MPI_LONG`...
- `dest`: which rank to send to
- `tag`: any value that can be attached to this message
- `comm`: the communicator

How? Send/Receive

Example:

```
int buf = 3875;
int dest = 2;
int tag = 1;
MPI_Send(&buf, 1, MPI_INT,
     dest, tag, MPI_COMM_WORLD);
```

How? Send/Receive

```
int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype,
             int source, int tag,
             MPI_Comm comm,
             MPI_Status *status);
```

- `buf`: buffer to receive
- `count`: maximum number of element to receive
- `datatype`: `MPI_SHORT`, `MPI_INT`, `MPI_LONG`...
- `source`: which rank to receive from
- `tag`: any value that can be attached to this message
- `comm`: the communicator

How? Send/Receive

Example:

```
int buf[10];
int src = 1;
int tag = 1;
MPI_Send(&buf, sizeof(buf)/sizeof(int),
    MPI_INT, src, tag, MPI_COMM_WORLD);
```

How? Send/Receive

- Broadcast: `MPI_Bcast()`

How? Send/Receive

- Broadcast: `MPI_Bcast()`
- Should this be true?

$$MPI\_Bcast_k = \sum_{i=1}^{n} MPI\_Send_{k \to i}$$

How? Send/Receive

- Broadcast: `MPI_Bcast()`
- Should this be true?

$$MPI\_Bcast_k = \sum_{i=1}^{n} MPI\_Send_{k \to i}$$

NOPE

How? Send/Receive

- Broadcast: `MPI_Bcast()`
- Should this be true?

$$MPI\_Bcast_k = \sum_{i=1}^{n} MPI\_Send_{k \to i}$$

NOPE

Why? 

How? Synchronization

```
int MPI_Barrier(MPI_Comm comm);
```

- Block the caller
- Wait until **all** members call it

Practical work 3: MPI File transfer

- Copy your TCP file transfer system to a new directory called MPI
- Upgrade it to a file transfer system using MPI
  - Use any MPI implementation of your choice
- Write a short report in LaTeX:
  - Name it « 03.mpi.file.transfer.tex »
  - Why you chose your specific MPI implementation
  - How you design your MPI service. Figure.
  - How you organize your system. Figure.
  - How you implement the file transfer. Code snippet.
  - Who does what.
- Work in your group, in parallel