Communication in Distributed Systems II

Course: Robotic Programming Environments

Michał Drwięga
michal.drwiega@pwr.edu.pl
www.mdrwiega.com/edu/rpe

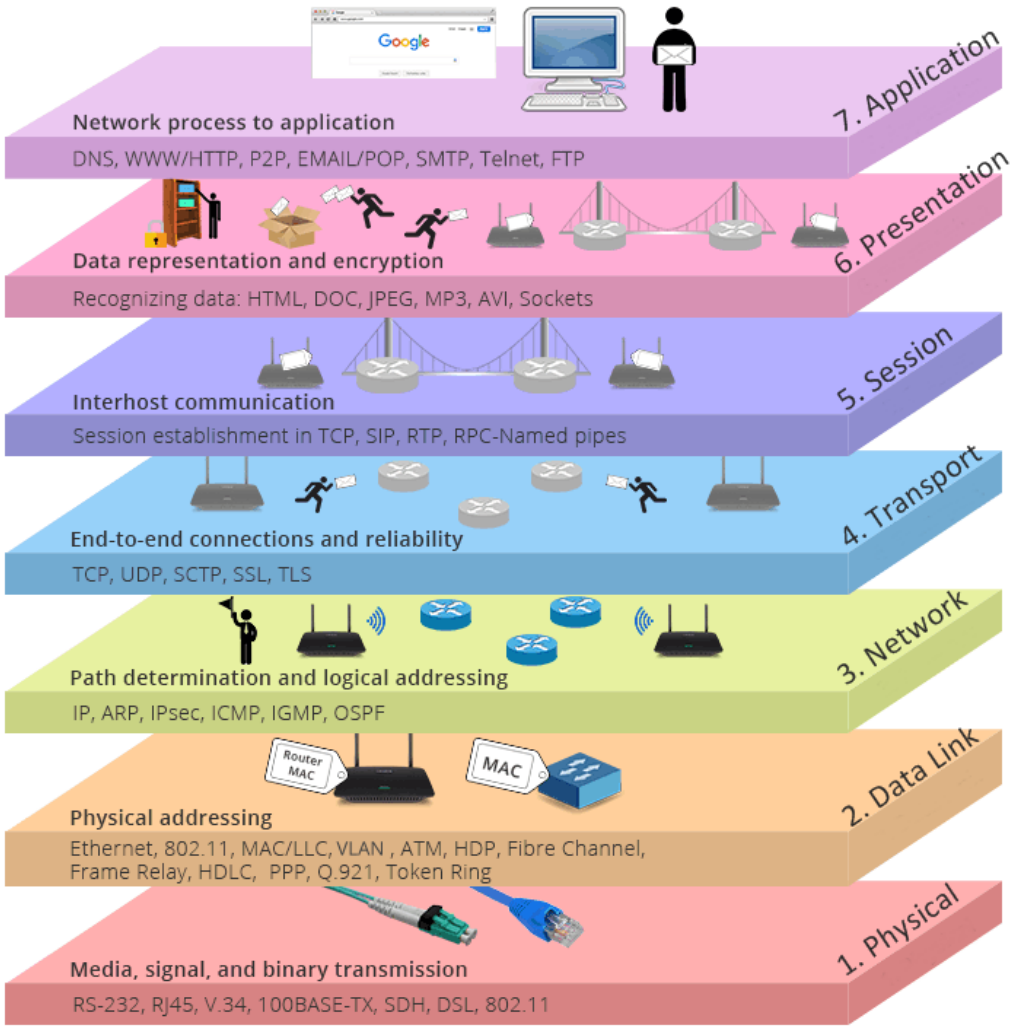
Department of Cybernetics and Robotics
Wrocław University of Science and Technology

Wrocław University
of Science and Technology

Content

- TCP vs UDP
- Inter-Process Communication (IPC)
- ZeroMQ
- Data representation, serialization
- Protobuf, gRPC
- Streaming protocols
- DDS Introduction

OSI Model

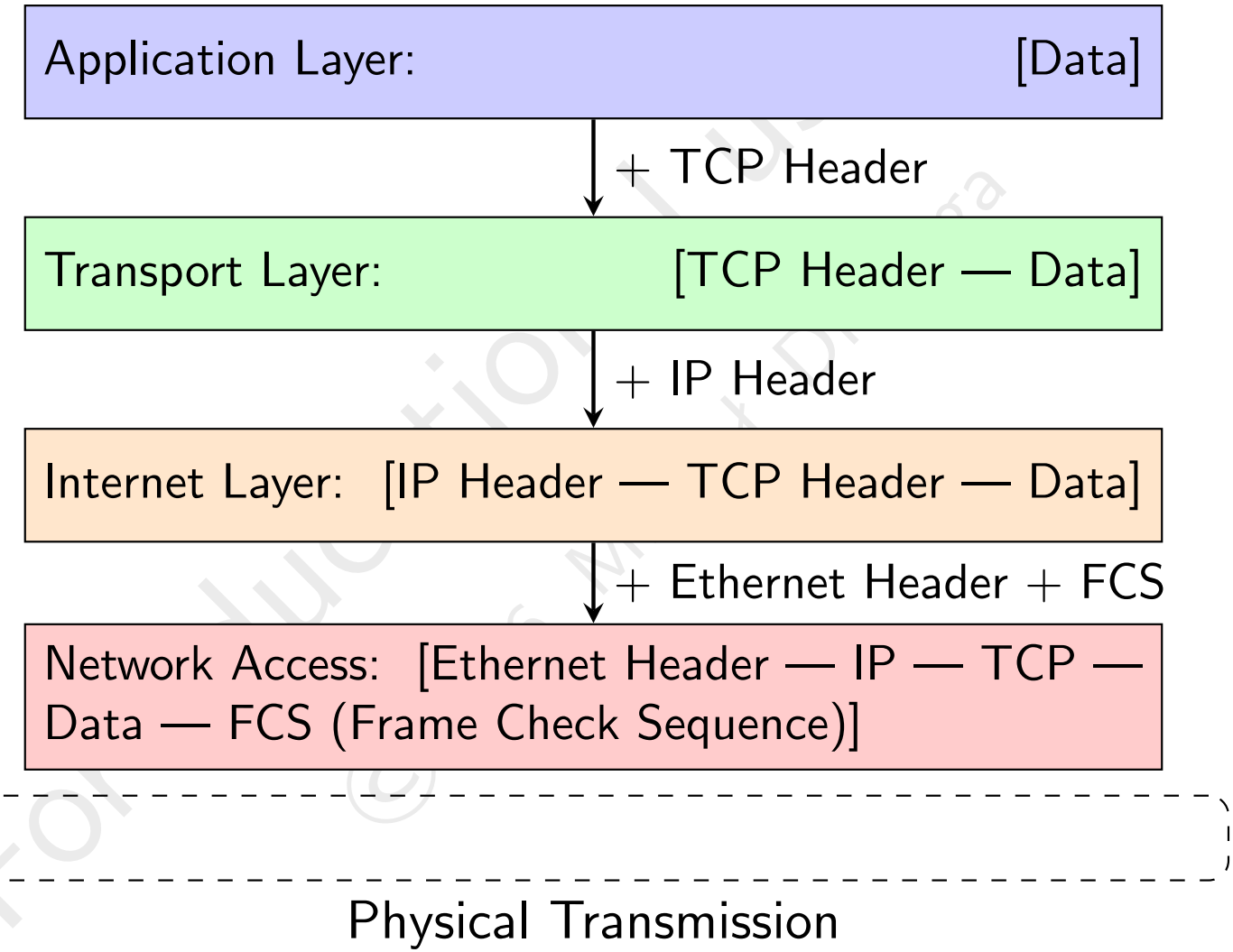


1

[1] https://community.fs.com/

TCP/IP Stack

Data Encapsulation Example



TCP vs UDP

- Two main transport protocols in the Internet
- Both operate at the **Transport Layer (Layer 4)** of the OSI model
- Built on top of **IP**

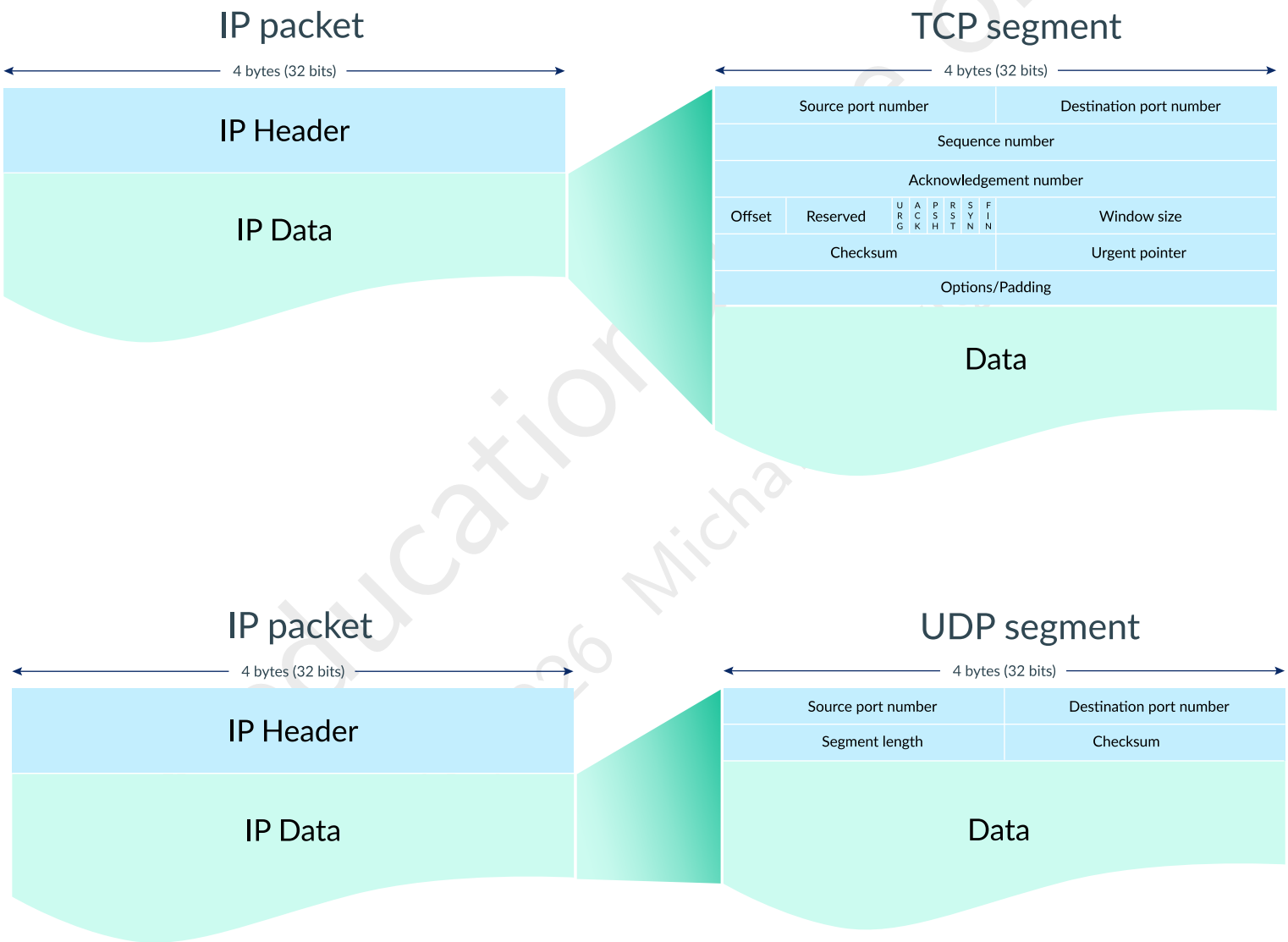
Transmission Control Protocol (TCP)

  - Connection-oriented
  - Reliable data transmission
  - Ordered delivery

User Datagram Protocol (UDP)

  - Connectionless
  - No delivery guarantees
  - Low latency and low overhead

TCP vs UDP


IP packet
4 bytes (32 bits)
IP Header
IP Data
TCP segment
4 bytes (32 bits)
Source port number
Destination port number
Sequence number
Acknowledgement number
Offset
Reserved
URG
ACK
PSH
RST
SYN
FIN
Window size
Checksum
Urgent pointer
Options/Padding
Data
IP packet
4 bytes (32 bits)
IP Header
IP Data
UDP segment
4 bytes (32 bits)
Source port number
Destination port number
Segment length
Checksum
Data

TCP Connection Establishment

TCP is a connection-oriented protocol

Before data transfer begins, a connection must be established using the **three-way handshake**.

1. Client sends **SYN** (synchronize) packet
2. Server responds with **SYN-ACK**
3. Client replies with **ACK**

Purpose

- Synchronize sequence numbers
- Confirm both hosts are reachable
- Initialize connection parameters

TCP Reliability Mechanisms

TCP ensures reliable data transmission through several mechanisms.

Sequence Numbers

- Every byte in the stream is numbered
- Enables ordered delivery

Acknowledgements (ACK)

- Receiver confirms successful reception
- Missing packets trigger retransmission

Retransmission

- Lost packets are resent
- Retransmission timeout (RTO)

TCP Flow Control

Problem

Fast sender may overwhelm a slow receiver.

Solution: Sliding Window

- Receiver advertises available buffer space
- Sender limits the amount of unacknowledged data

Window Size

- Determines how much data can be sent without ACK
- Adjusted dynamically

Result

Prevents receiver buffer overflow.

TCP Congestion Control

TCP adapts transmission rate to network conditions.

Main mechanisms

- Slow Start
- Congestion Avoidance
- Fast Retransmit
- Fast Recovery

Slow Start

- Start with small congestion window
- Increase exponentially

Goal

Prevent network congestion and packet loss.

Transport Layer Protocols

TCP vs UDP Comparison

Feature	**TCP**	**UDP**
Connection	Connection-oriented	Connectionless
Reliability	Guaranteed delivery	Best effort
Ordering	Maintains order	No ordering
Flow Control	Yes	No
Speed	Slower	Faster
Header Size	20-60 bytes	8 bytes
Use Cases	Web, Email, FTP	Streaming, DNS, VoIP

When to use which?

- **TCP:** Need reliability (file transfer, web browsing)
- **UDP:** Need speed, can tolerate loss (video streaming, gaming)

lwIP - Lightweight IP Stack in C

An Open Source TCP/IP Stack for Embedded Systems

Key Features

- TCP/IP stack implementation
- Small memory footprint (tens of KB RAM + 40KB ROM)
- No OS dependency

Architecture

- Dual API interface:
  - Raw API (callback-based)
  - Sequential API (socket-like)
- Zero-copy operation
- Network interface abstraction

Protocols [2]

- IP: IPv4 and IPv6
- TCP, UDP
- ICMP (Internet Control Message Protocol)
- DHCP (Dynamic Host Configuration Protocol)
- PPPoE (Point-to-Point Protocol over Ethernet)
- DNS (Domain Name Resolver)

[2] https://github.com/frida/lwip

Why Middleware Is Needed?

Transport protocols such as TCP and UDP provide only **basic data transport**.

They do not provide

- Message discovery
- Data serialization
- Publish–subscribe communication
- Quality of Service control

Middleware

Middleware frameworks add a higher abstraction layer for building distributed systems.

Examples:

- MQTT
- ZeroMQ
- gRPC
- DDS

What about communication within a single machine?

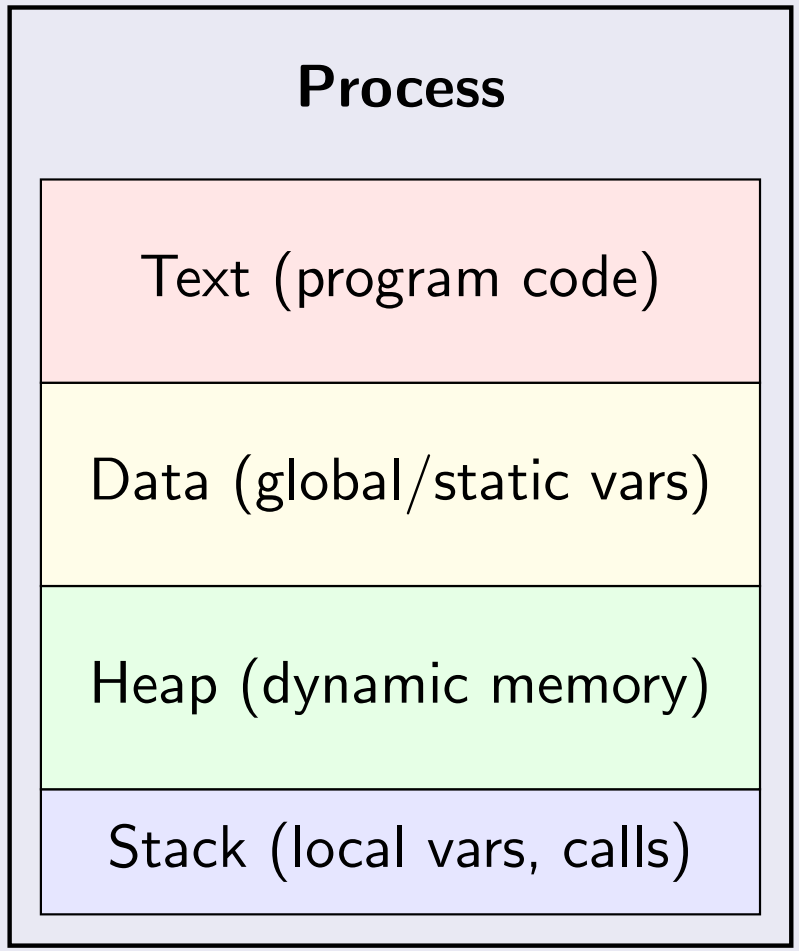
Process — Program in Execution

What is a process?

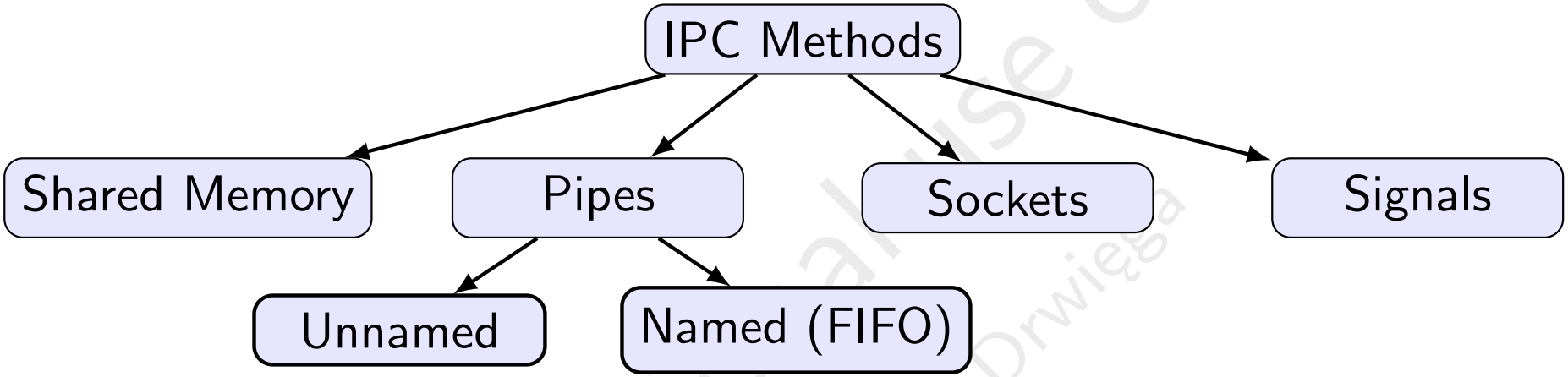
A **process** is an instance of a program currently running.

- Each process has its own **isolated address space**
- OS manages processes via the **process control block (PCB)**
- Multiple processes can run the same program

Process Memory Layout



Inter-Process Communication (IPC) Methods



- **Shared memory:** Fastest IPC. Processes read/write to a common memory region
- **Pipes:** Unidirectional data flow. *unnamed* (between related processes) vs. *named* (FIFOs, any processes)
- **Sockets:** Bidirectional, can also be used over a network.
- **Signals:** Asynchronous notifications, simple events only.

ZeroMQ: High-Performance Messaging Library

- **Asynchronous messaging**: Designed for efficient communication in distributed or concurrent applications.
- **Why "zero"?**
  - **Zero brokers** — No central message broker
  - **Zero latency** — Minimal overhead
  - **Zero administration** — No configuration
  - **Zero waste** — Optimized for performance
- **Multiple messaging patterns**:
  - Request-Reply
  - Publish-Subscribe
  - Push-Pull

[3]

[3] https://zeromq.org/

ZeroMQ: Transport Protocols

tcp://	Network communication (host:port)
ipc://	Inter-process on same machine (Unix sockets)
inproc://	In-process threading (fastest)
pgm://	Pragmatic General Multicast
ws://	WebSockets for browser clients

ZeroMQ: Minimal Example

Publisher (Python)

```python
import zmq
import time

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

topic = "sensor/temp"
while True:
    message = f"{topic} 23.5"
    socket.send_string(message)
    print(f"Sent: {message}")
    time.sleep(1)
```

Subscriber (Python)

```python
import zmq

context = zmq.Context()
socket = context.socket(zmq.SUB)
socket.connect("tcp://localhost:5556")
socket.setsockopt_string(zmq.SUBSCRIBE, "
    sensor/temp")

while True:
    message = socket.recv_string()
    print(f"Received: {message}")
```

Install: `pip install pyzmq`
C++: `apt install libzmq3-dev`
Docs: `https://zeromq.org/`

Data Representation

Data Representation in Different Systems

Data Representation Differences

- **Data sizes vary:** int (16, 32, 64 bits), float precision
- **Character encoding:** ASCII vs. Unicode (UTF-8, UTF-16)
  - "A" = 65 (ASCII), but UTF-16 uses 2 bytes
  - "€" not representable in ASCII
- **Endianness:** Big-endian vs. Little-endian
  - 0x12345678 stored as:
  - LE: 78 56 34 12
  - BE: 12 34 56 78

Complex Data Challenges

- **Pointers:** Address representation differs
  - Address 0x7ffe... meaningless on another machine
- **Complex structures:** Trees, lists
  - Linked list → must send values, not pointers
- **Memory layout:** Structure padding, alignment

Padding and Memory Alignment

What is Padding?

- Extra bytes added to ensure proper memory alignment
- Improves CPU efficiency
- Prevents misaligned access errors

Example Struct

```
struct Example {
    char a;    // 1 byte
    int b;     // 4 bytes
};
```

Memory Layout

Without padding:

```
| a | b b b b |
```

With padding:

```
| a | P P P | b b b b |
```

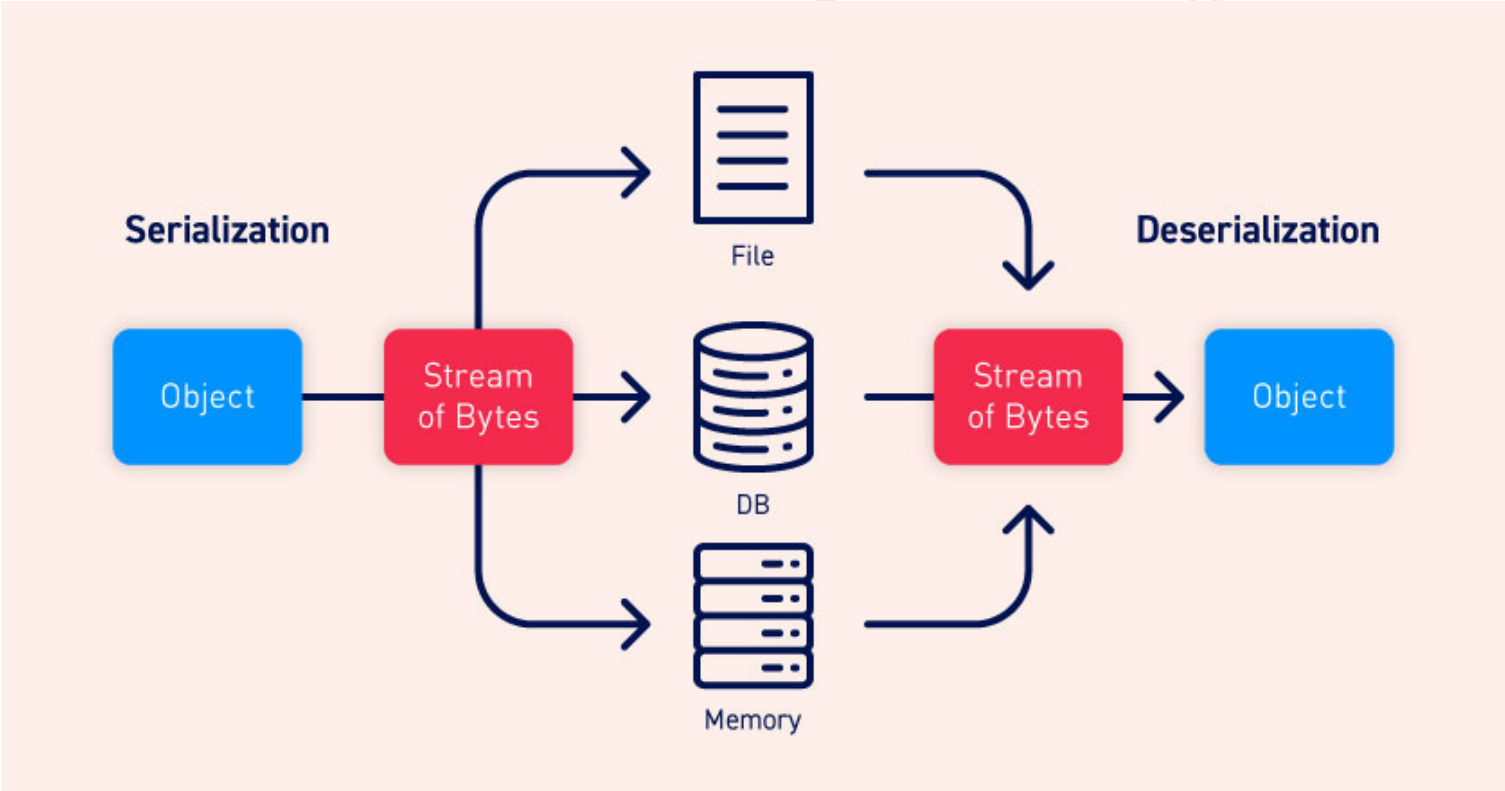
- P = padding bytes (unused)
- sizeof(struct Example) = 8 bytes

Implication: Different systems may use different padding → must not transmit raw memory!

How can we reliably transmit data between heterogeneous platforms without misinterpretation?

Serialization/deserialization

- **Serialization** is the process of translating a data structure or object state into a format that can be stored or transmitted and reconstructed later

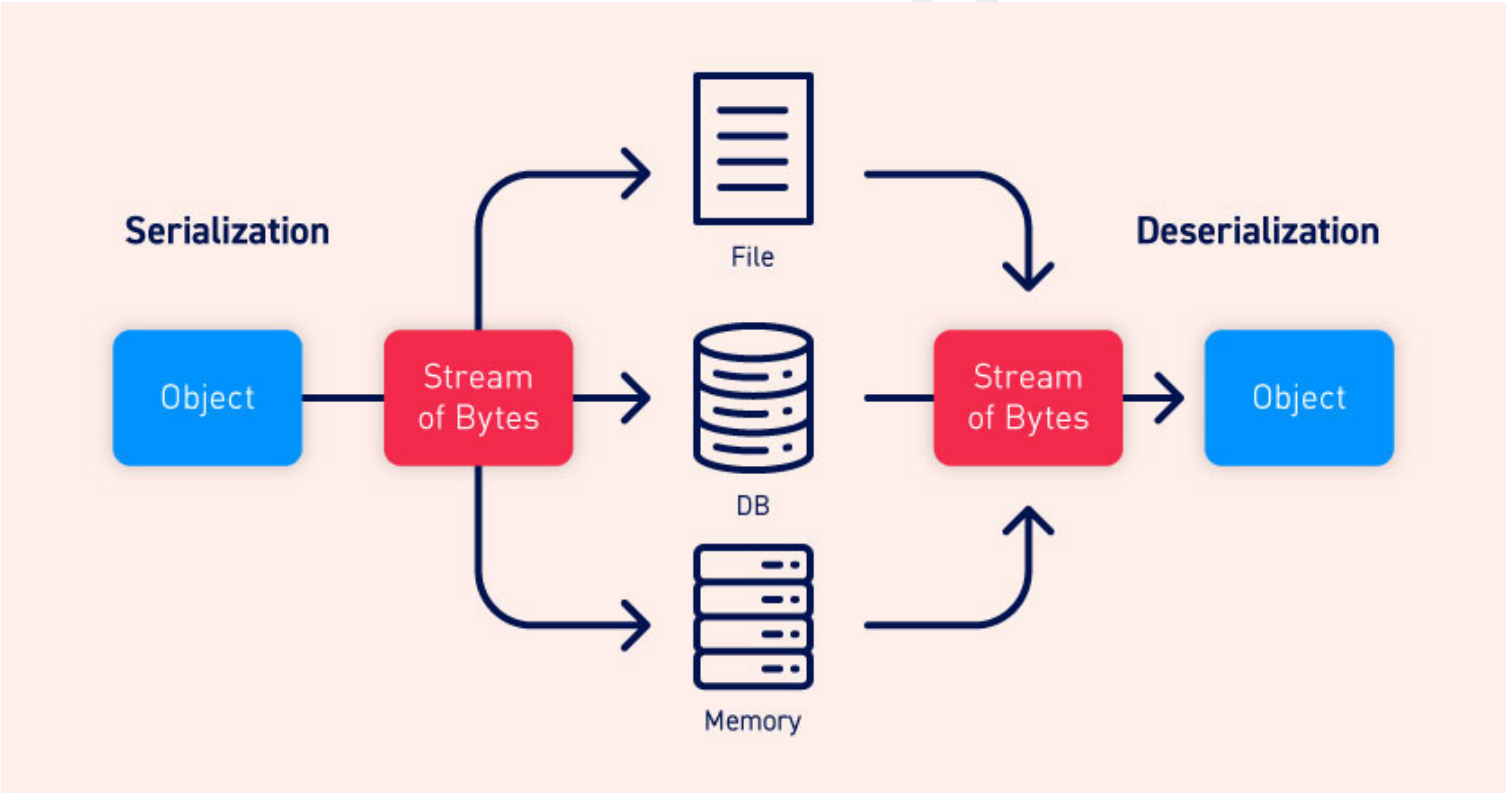


4

[4] https://portswigger.net/

Serialization/deserialization

- **Deserialization** is the opposite operation to serialization. It's extracting a data structure from a series of bytes.



4

[4] https://portswigger.net/

Serialization/deserialization - data schema

Explicit typing

- Self-describing data (tags)
- Additional information added to the message
- Examples:
  - XML: <temperature unit="Celsius">25.3 </temperature>
  - JSON: {"temperature": 25.3, "unit": "Celsius"}

Implicit typing

- Devices at both ends know how to encode/decode messages
- Depends on a predefined protocol specification
- **Examples:**
  - Legacy serial communication where field order and size are fixed
  - Custom binary protocol: first 2 bytes = temperature (int16)

Protocol Buffers (protobuf)

- Protocol Buffers are language-neutral, platform-neutral extensible mechanisms for serializing structured data [5]
- Open-source
- Designed to be smaller and faster than XML
- Data structure schemas (messages) described in a *proto* file
- Messages compilers for multiple languages (for example *protoc* for C/C++)

[5] https://protobuf.dev/

Protocol Buffers - example message

```
1  syntax = "proto3";
2
3  message Person {
4    string name = 1;
5    int32 id = 2;
6    message PhoneNumber {
7      optional string number = 1;
8    }
9    repeated PhoneNumber phones = 3;
10 }
11
12 message AddressBook {
13   repeated Person people = 1;
14 }
```

Protocol Buffers - example

- Generate Python code for the message

```
protoc -I=. --python_out=. ./person.proto
```

- Import and use message in Python

```
import person.person_pb2 as person_pb2

person_msg = person_pb2.Person()
person_msg.name = "John"
person_msg.id = 5

# Write message to binary file
with open("person.bin", "wb") as f:
    data = person_msg.SerializeToString()
    f.write(data)

# Read message from binary file
with open("person.bin", "rb") as f:
    msg = person_pb2.Person().FromString(f.read())
```

gRPC (Google Remote Procedure Call)

What is gRPC? A high-performance Remote Procedure Call (RPC) framework for communication between distributed services.

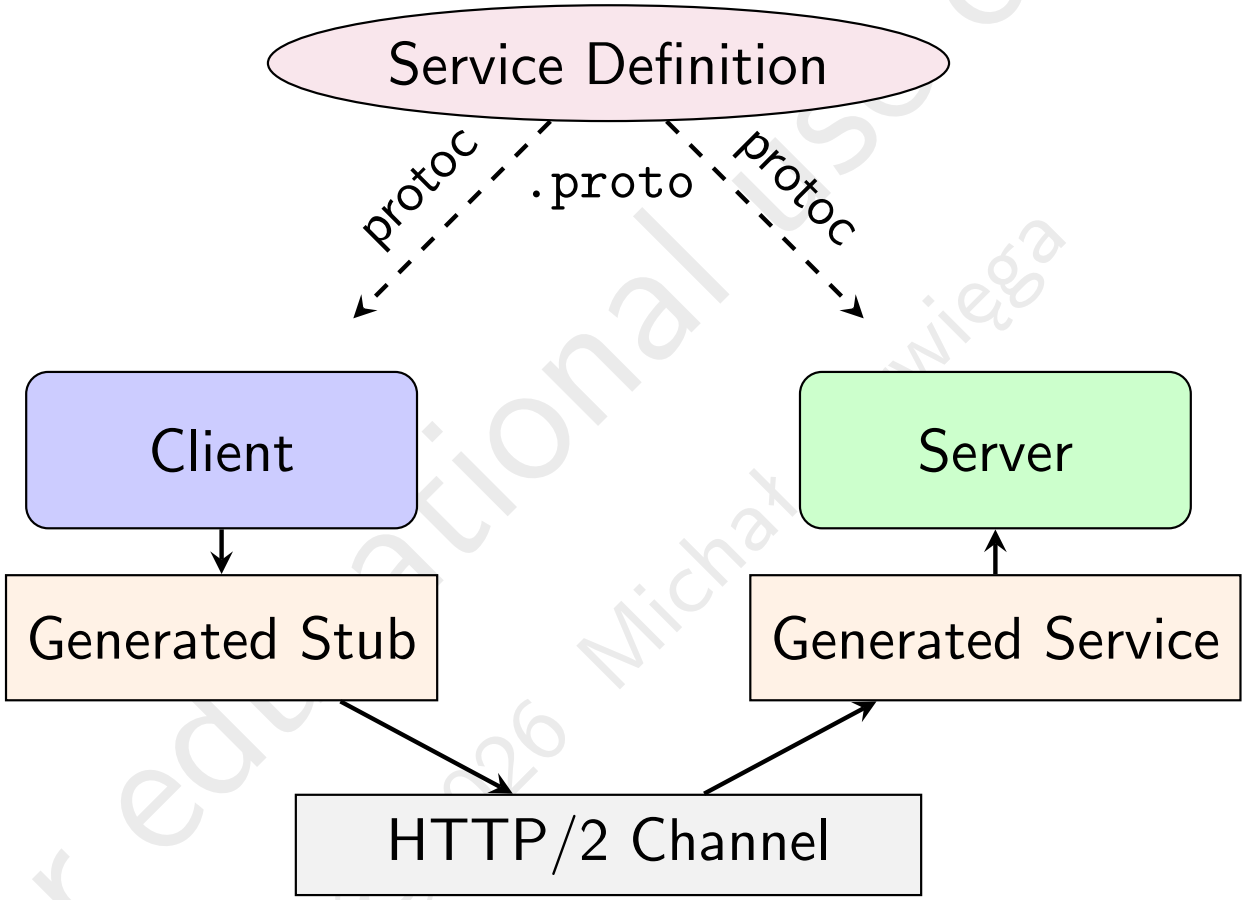
Features:

- Strongly typed APIs using **Protocol Buffers**
- Runs over **HTTP/2** (multiplexing, flow control)
- Supports unary, streaming, and bidirectional RPCs
- Language- and platform-independent

Use Cases:

- Robot–backend communication (cloud / edge)
- Configuration and command interfaces
- Status, diagnostics, and service calls

gRPC Architecture



Language-neutral: Generate clients/servers in C++, Java, Python, Go, etc.

Robot Sensor Service - Protocol Definition

Service Definition

```
service RobotSensorService {
  rpc GetLidarData(LidarRequest)
      returns (LidarResponse);
}
```

Message Types

```
message LidarRequest {
  string robot_id = 1;
}

message LidarResponse {
  repeated float distances =
      1;
  string unit = 2;
  int32 timestamp = 3;
}
```

- ✓ **Explicit typing:** Field tags (1, 2, 3) ensure correct interpretation
- ✓ **Cross-platform:** Same protocol works on C++ (ROS), Python, Java
- ✓ **Extensible:** Can add fields (e.g., `timestamp`) without breaking old clients

Robot LIDAR Server Example

Key Server Logic

```
class RobotSensorServicer(...):
  def GetLidarData(self, request, context):
      distances = [rand.uniform(0.5,10) for _ in range(360)]
      return LidarResponse(distances=distances, unit="meters")
```

Server Setup

```
server = grpc.server(...)
add_RobotSensorServiceServicer_to_server(...)
server.add_insecure_port('[::]:50051')
server.start()
```

- Provides simulated LIDAR data for other robots
- Runs independently on each robot or edge node

Robot Client Example

Client Request

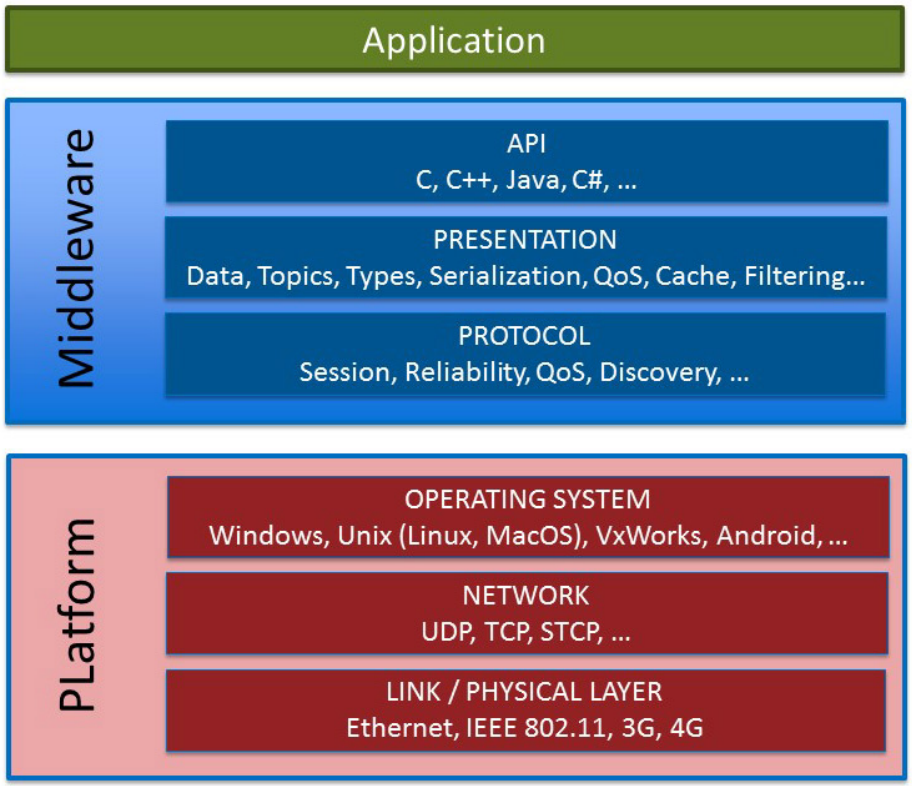
```
channel = grpc.insecure_channel('localhost:50051')
stub = RobotSensorServiceStub(channel)
request = LidarRequest(robot_id="RobotB")
response = stub.GetLidarData(request)

print(f"Received {len(response.distances)} points")
```

- Client can run on different hardware or language
- Data received is already serialized/deserialized automatically

Data Distribution Service (DDS)

- The DDS Middleware is a software layer that abstracts the Application from the details of the operating system, network transport, and low-level data formats.



6

- *We will come back to this topic!*

[6] https://www.dds-foundation.org/

MQTT vs Brokerless

MQTT (Broker)

+ Simple clients
+ NAT/firewall friendly
+ Central management
- Single point of failure
- Scalability bottleneck

Brokerless (ZeroMQ, DDS, etc.)

+ No SPOF
+ Lower latency
+ Better scalability
- Complex discovery
- NAT traversal issues

Data Streaming Protocols

Sensor Data Streaming in Robotics

Why streaming is needed:

- Real-time perception for navigation and obstacle avoidance
- Remote monitoring of cameras, sonar, LiDAR, or IMUs
- Integration with distributed compute nodes (edge/cloud)

Key challenges:

- Low-latency delivery
- Synchronization across multiple sensors
- Bandwidth and packet loss management

Real-Time Streaming Protocol

RTSP (Real-Time Streaming Protocol)

- Protocol for controlling streaming media sessions over IP
- Relationship with **RTP (Real-time Protocol):**
  - RTSP controls the session
  - RTP carries the actual media frames (video, sonar, etc.)

How RTSP works?

1. **OPTIONS / DESCRIBE**: client queries server capabilities
2. **SETUP**: initialize a session, choose transport (UDP/TCP)
3. **PLAY**: start streaming
4. **PAUSE / TEARDOWN**: pause or stop the session

RTP (Real-time Protocol)

RTP Packet Structure

- **Header (12 bytes fixed):**
  - Version, Padding, Extension, CSRC count
  - Marker bit (frame boundary)
  - Payload type (e.g., H.264, custom sensor)
  - Sequence number (detect packet loss, reorder)
  - Timestamp (synchronize playback / sensor timing)
  - SSRC (synchronization source identifier)
- **Payload:** actual data (video frame, audio sample, sensor readings)

Video Compression in Robotics

Why compress video?

- **Reduce bandwidth:** Raw video consumes huge network resources
- **Reduce storage:** High-resolution video quickly fills memory
- **Enable real-time transmission:** Compressed streams travel faster
- **Optimize resources:** Balance compression ratio vs. visual fidelity

Notes:

- Common codecs: H.264, H.265 (HEVC), VP8/VP9
- Compression/decompression requires encoding CPU/GPU resources
- Embedded boards (Raspberry Pi, Jetson Nano) may limit codec choice due to real-time constraints
- Trade-offs: latency, energy consumption, and hardware load vs. bandwidth savings

Summary

- TCP vs UDP,
- ZeroMQ
- Inter-Process Communication (IPC)
- Data representation, serialization
- Protobuf, gRPC
- Streaming protocols

The end

Thank you for your attention!