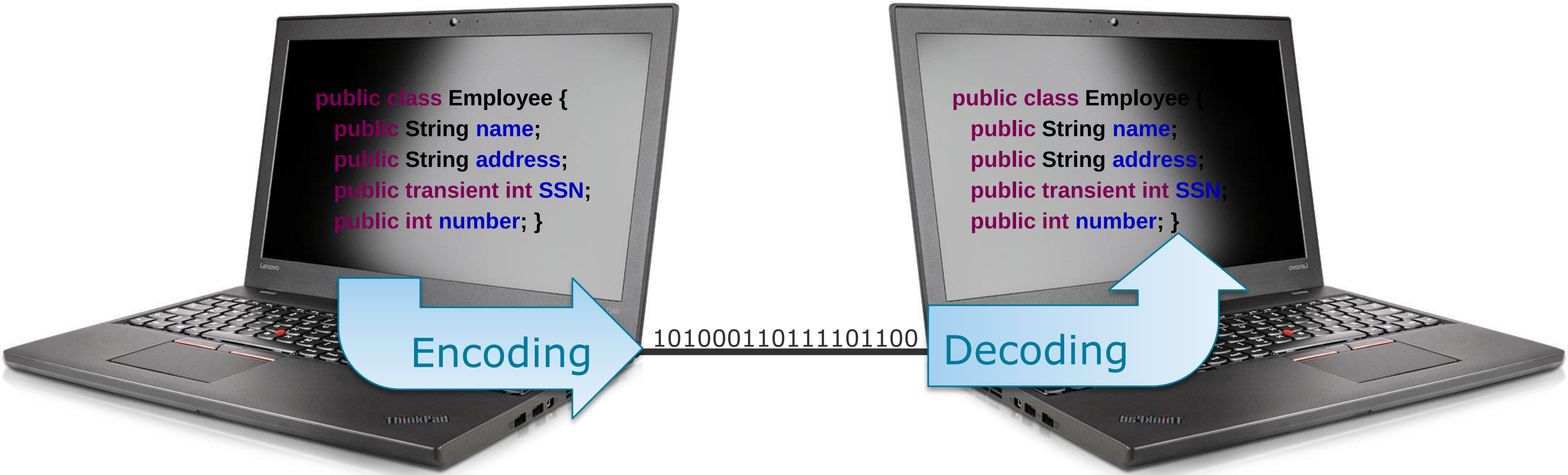
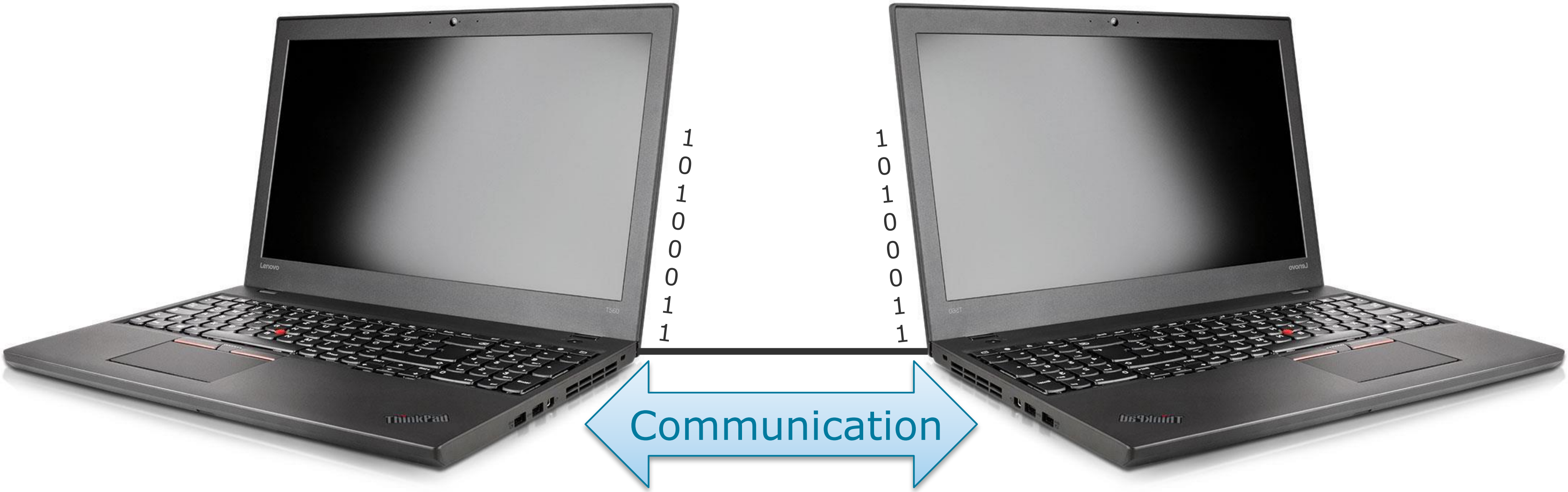
Distributed Data Management
Encoding and Communication

Thorsten Papenbrock

F-2.04, Campus II

Hasso Plattner Institut

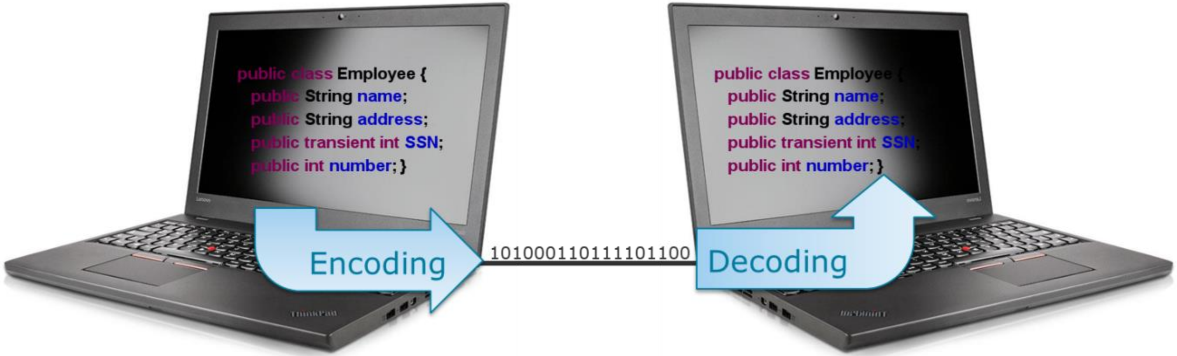

public class Employee {
public String name;
public String address;
public transient int SSN;
public int number; }
Encoding
1010001101111101100
Decoding
public class Employee {
public String name;
public String address;
public transient int SSN;
public int number; }


1 0 1 0 0 0 1 1
1 0 1 0 0 0 1 1
Communication

Overview
Encoding and Communication

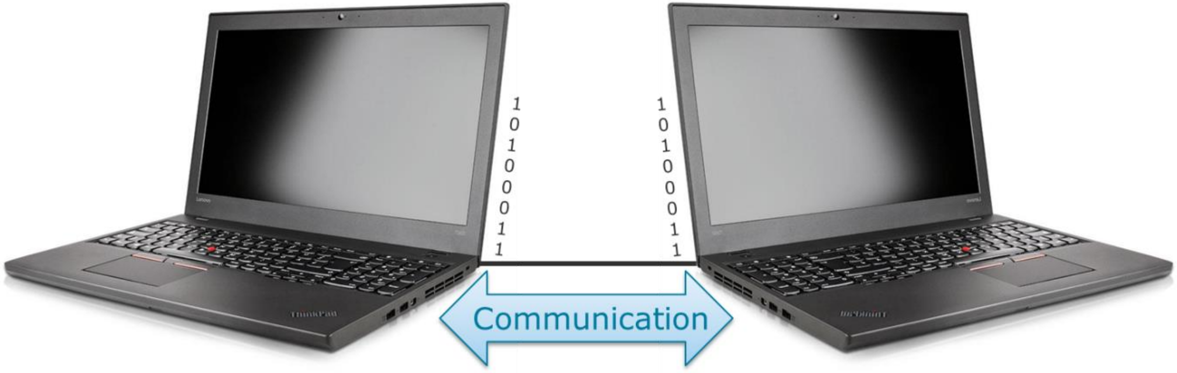
Encoding

- Language-Specific Encoding
- JSON/XML Encoding
- Binary Encoding



Communication

- Dataflow via Databases
- Dataflow via Services
- Dataflow via Message-Passing



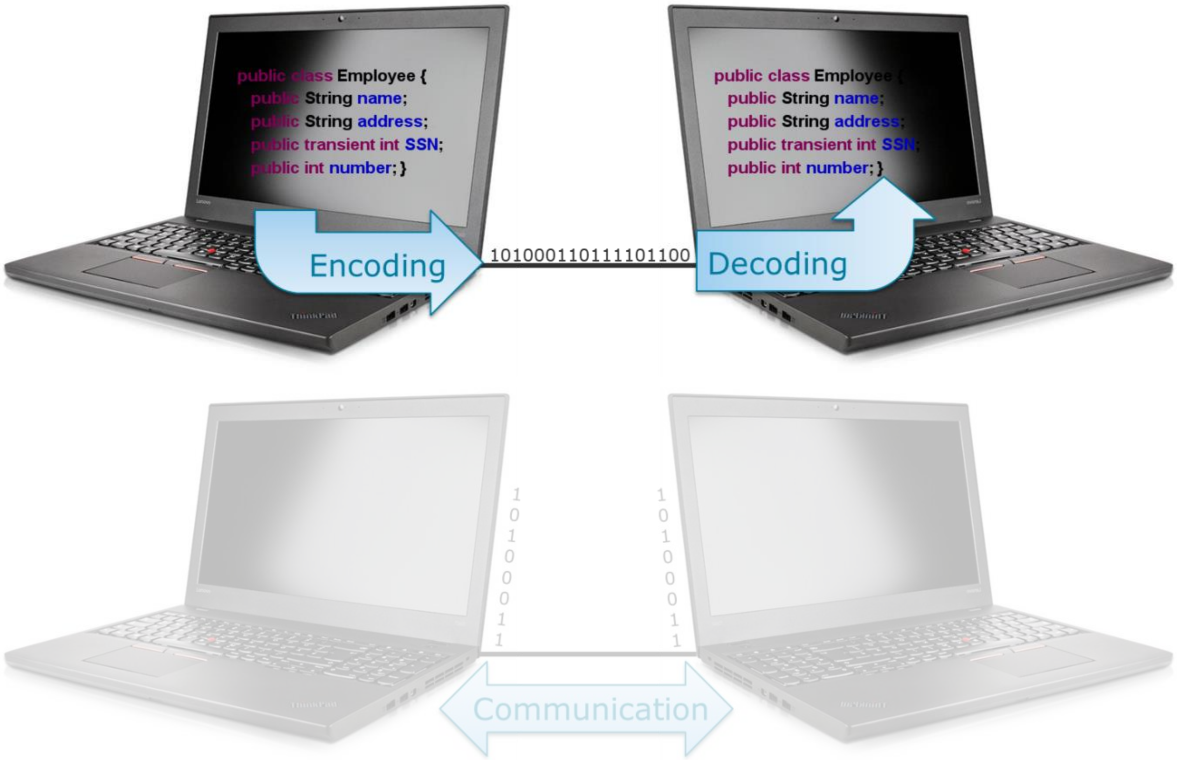
Encoding and Communication

Encoding

- Language-Specific Encoding
- JSON/XML Encoding
- Binary Encoding

Communication

- Dataflow via Databases
- Dataflow via Services
- Dataflow via Message-Passing



Layering Data Models

1. Conceptual layer

- Data structures, objects, modules, …
  - Application code

2. Logical layer

- Relational tables, JSON, XML, graphs, …
  - Database management system (DBMS) or storage engine

3. Representation layer

- Bytes in memory, on disk, on network, …
  - Database management system (DBMS) or storage engine

4. Physical layer

- Electrical currents, pulses of light, magnetic fields, …
  - Operating system and hardware drivers

```
class TestSerial {
  public byte version = 100;
  public byte count = 0;
}
```

```
{
  "class": TestSerial,
  "version": "100",
  "count": "0"
}
```

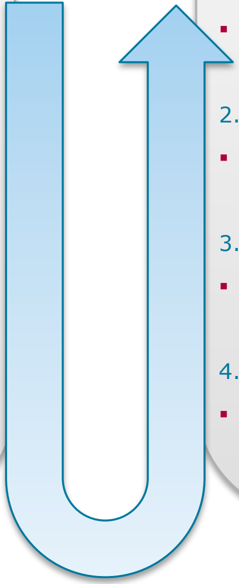
AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

Encoding

Network Connections are Physical

Node 1

1. Conceptual layer
   - Data structures, objects, modules, …
     - Application code
2. Logical layer
   - Relational tables, JSON, XML, graphs, …
     - Database management system (DBMS) or storage engine
3. Representation layer
   - Bytes in memory, on disk, on network, …
     - Database management system (DBMS) or storage engine
4. Physical layer
   - Electrical currents, pulses of light, magnetic fields, …
     - Operating system and hardware drivers



Node 2

1. Conceptual layer
   - Data structures, objects, modules, …
     - Application code
2. Logical layer
   - Relational tables, JSON, XML, graphs, …
     - Database management system (DBMS) or storage engine
3. Representation layer
   - Bytes in memory, on disk, on network, …
     - Database management system (DBMS) or storage engine
4. Physical layer
   - Electrical currents, pulses of light, magnetic fields, …
     - Operating system and hardware drivers

Network Connections are Physical

Encoding

- "Representation of the data"
  (or "Process of changing the representation")

Serialization

- "Serial encoding of the data"
  (or "Process of serializing a representation")
- i.e. special case of encoding
- Serial formats:
  - Char arrays in layer 2 (JSON, XML, …)
  - Byte arrays in layer 3

Decoding (and Deserialization)

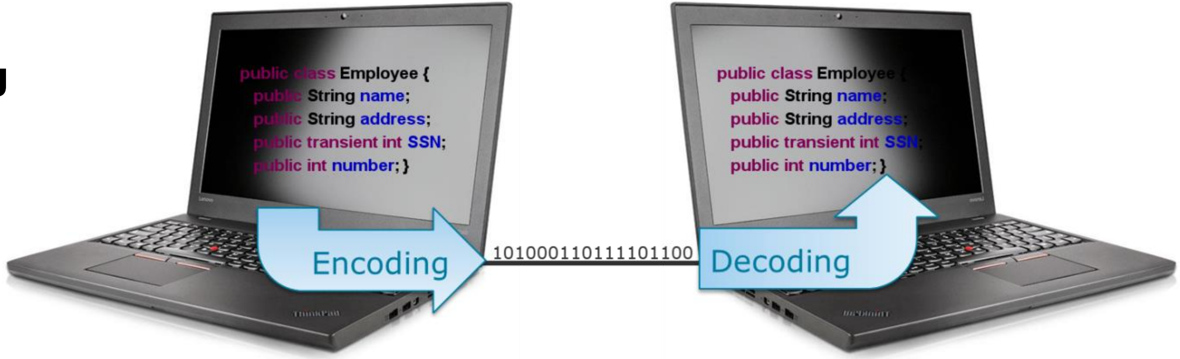
- The reverse of encoding (and serialization)

1. Conceptual layer
   - Data structures, objects, modules, …
     - ➢ Application code
2. Logical layer
   - Relational tables, JSON, XML, graphs, …
     - ➢ Database management system (DBMS) or storage engine
3. Representation layer
   - Bytes in memory, on disk, on network, …
     - ➢ Database management system (DBMS) or storage engine
4. Physical layer
   - Electrical currents, pulses of light, magnetic fields, …
     - ➢ Operating system and hardware drivers

Overview
Encoding and Communication

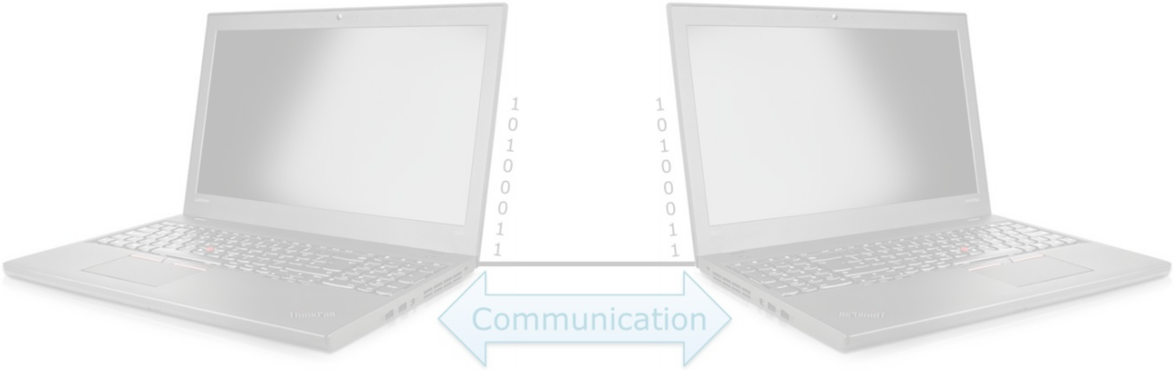
Encoding

- **Language-Specific Encoding**
- JSON/XML Encoding
- Binary Encoding



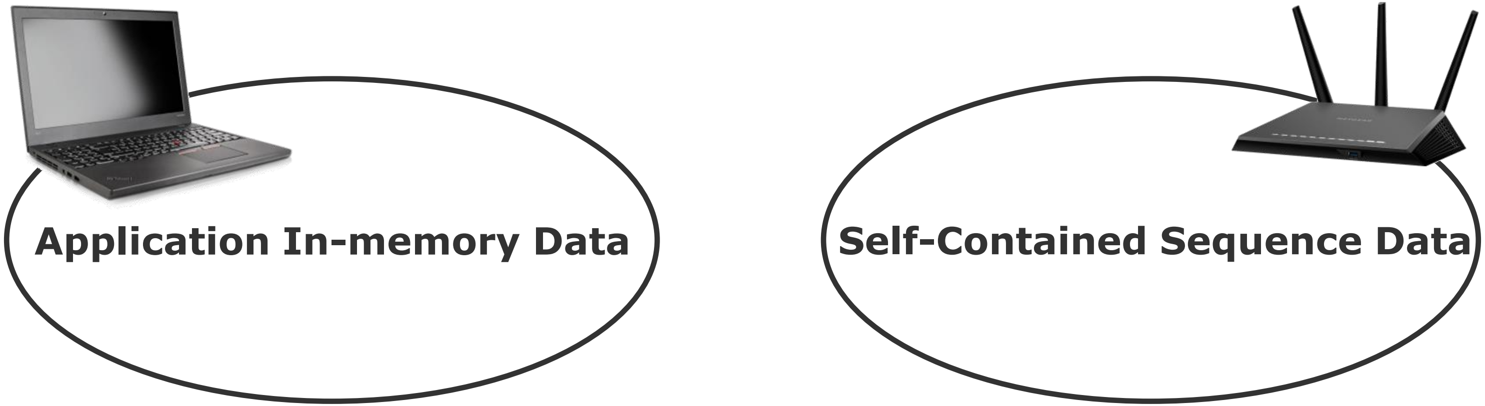
Communication

- Dataflow via Databases
- Dataflow via Services
- Dataflow via Message-Passing



Language-Specific Encoding

Two Different Representations

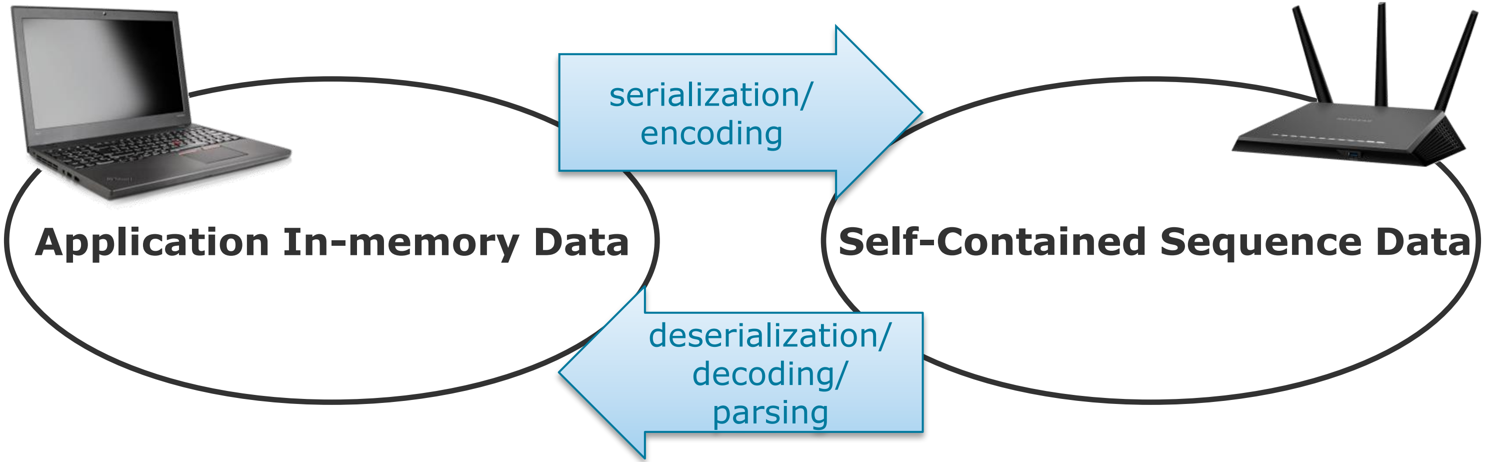


- Language specific formats
- Logical structures: objects, structs, lists, arrays, hash tables, trees, …
- Optimized for efficient manipulation by the CPU

- Standardized encoding formats
- Byte sequences: Native, JSON, XML, protocol buffers, Avro, …
- Optimized for disk persistence, network transmission, and inter-process communication

Language-Specific Encoding

Two Different Representations



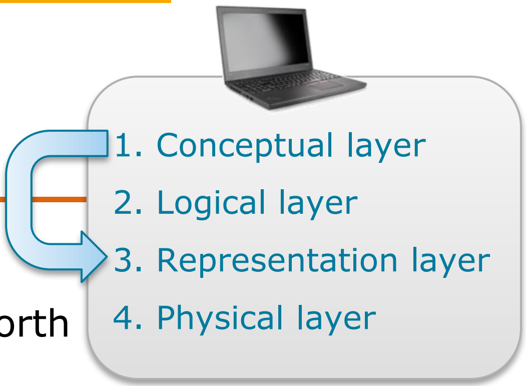
Problems:

- Tied to a programming language (language-specific data structures)
- Tied to an address space (process-specific pointers)

Problems:

- Inefficient and complicated access and manipulation operations due to lack of pointers and serial byte representation

Serialization/Encoding



Language-specific serialization formats

- Goal: convert in-memory data into byte sequence data back and forth
- Examples:
  `Serializable` (Java), `Kryo` (Java), `Marshal` (Ruby), `pickle` (Python), …

Advantages

- Native language support; easy to use
- Default implementation for intra-language (distributed) communication

Problems

- Serialized data is still tied to a programming language.
- Deserialization of arbitrary, byte-encoded objects can cause security issues.
- Data versioning is complicated, i.e., lack of forward/backward compatibility.
- Performance is often an issue, because arbitrary object serialization can be costly (e.g., Java `Serializable` is known to be inefficient).

Example: java.io.Serializable

```
import java.io.*;

public class Employee implements java.io.Serializable {
  public String name;
  public String address;
  public transient int SSN;
  public int number;

  public Employee(String name, String address, int SSN, int number) {
    this.name = name;
    this.address = address;
    this.SSN = SSN;
    this.number = number;
  }
}
```

Java can serialize any class that implement the Serializable interface (serialization via reflection)

All fields must also be serializable or explicitly marked as transient, i.e., non-serializable

https://www.tutorialspoint.com/java/java_serialization.htm

Example: java.io.Serializable

```
import java.io.*;
public class SerializeDemo {
  public static void main(String [ ] args) {
    Employee e = new Employee("Diana Brown", "Citystreet 8, Jamestown", 42, 123);
    try {
      FileOutputStream fileOut = new FileOutputStream("/tmp/employee.ser");
      ObjectOutputStream out = new ObjectOutputStream(fileOut);
      out.writeObject(e);
      out.close();
      fileOut.close();
    } catch(IOException i) {
      i.printStackTrace();
    }
  }
}
```

Can be any output stream; also to network etc.

Performs the actual serialization using reflection

https://www.tutorialspoint.com/java/java_serialization.htm

Example: java.io.Serializable

```
import java.io.*;
public class DeserializeDemo {
  public static void main(String [ ] args) {
    Employee e = null;
    try {
      FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
      ObjectInputStream in = new ObjectInputStream(fileIn);
      e = (Employee) in.readObject();
      in.close();
      fileIn.close();
    } catch(IOException | ClassNotFoundException i) {
      i.printStackTrace();
    }
  }
}
```

Performs the actual deserialization; result is an object

https://www.tutorialspoint.com/java/java_serialization.htm

Example: java.io.Serializable

Surprise!

- The serialized objects are much larger than expected:

```
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

serialization/ encoding →

Hexadecimal Code

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

2 bytes + header (?)

51 bytes

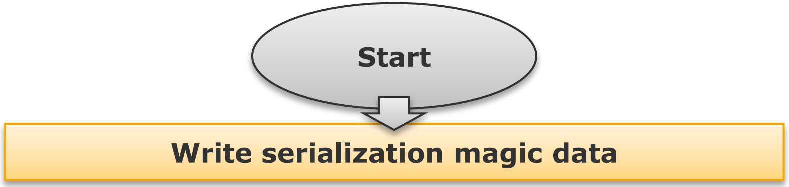
Why?

- Size scales linearly (but not favorably)

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:




```
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

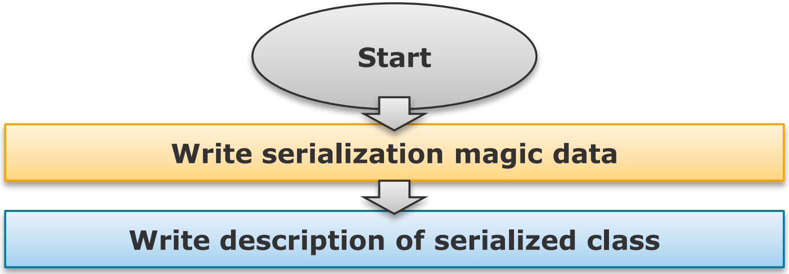
Serialization magic data specifies …

1. the serialization protocol (AC ED)
2. the serialization version (00 05)
3. the beginning of a new Object (0x73).

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:




```java
class TestSerial implements Serializable {
   public byte version = 100;
   public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

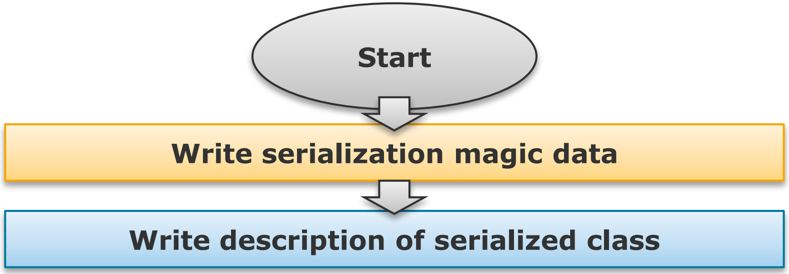
Description of serialized class specifies …

1. beginning of a new class (0x72)
2. length of the class name (00 0A)
3. name of the class (53 […] 74)
4. serial version identifier (A0 […] F9)
5. various flags (e.g. 0x02 = serialization support)
6. number of fields in this class (00 02)

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:




```
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

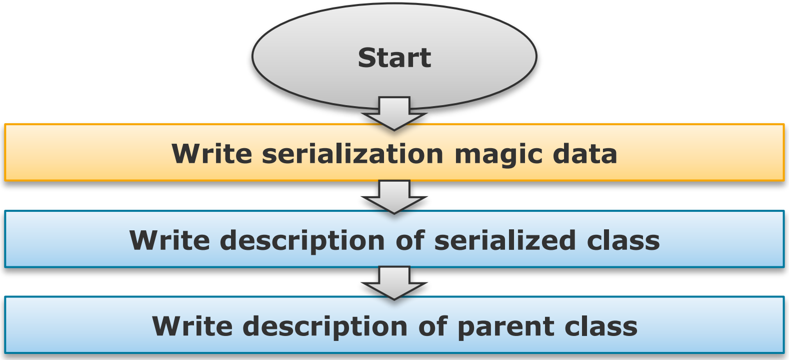
Description of serialized class specifies …

7. field code of “version” representing “byte” (0x42)
8. length of the field name (00 05)
9. name of the field (63 […] 74 which is “version”)
10. field code of “count” representing “byte” (0x42)
11. length of the field name (00 07)
12. name of the field (76 […] 6E which is “count”)

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:




```
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

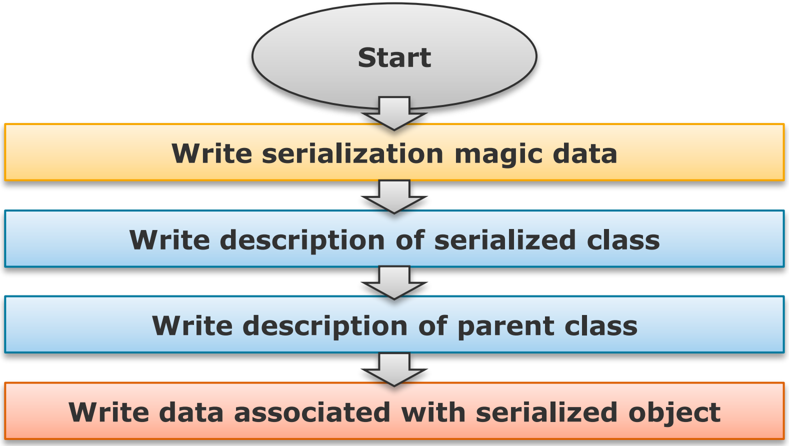
Description of parent class specification:

- Follows the same pattern as shown for the serialized class: (1) class definition and (2) field definitions
- Recursively adds the parent's parents until parent class is `Object`
- No parent here, because parent is already `Object`

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:




```
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

Data associated with serialized object …

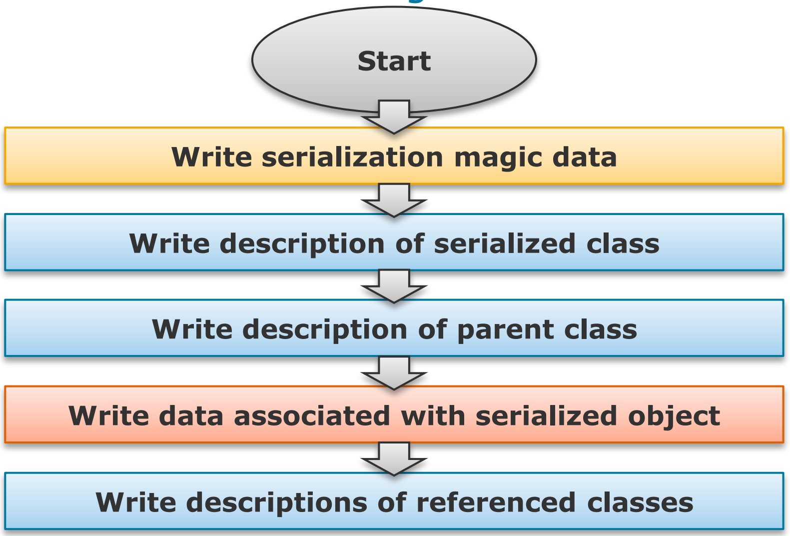
1. The first value (78 70 which is 100 for "version")
2. The second value (00 64 which is 0 for "count")

- Byte-length of the values is known from their types
- Fields are own and inherited fields

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:




```java
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

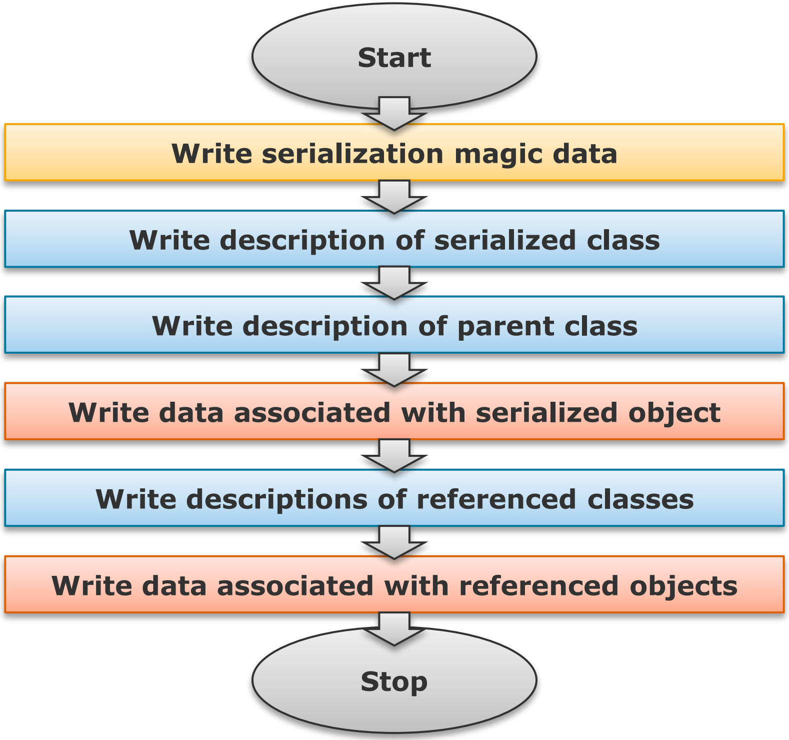
Description of referenced classes specification:

- Follow the same pattern as shown for the serialized class: (1) class definition and (2) field definitions
- No specifications here, because the class `TestSerial` has no referenced classes

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:




```java
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

Data associated with referenced objects …

- Follows the same pattern as shown for the serialized object: concatenation of byte encoded values
- No values here, because the class `TestSerial` has no referenced classes

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Hasso Plattner Institut

```
class Message implements Serializable {
  private static final long serialVersionUID = 6455048433435395034L;
  int[] data = {1,2,3};
  String name = "message42";
  boolean validity = true;
  Map<String, String> map = Stream.of(new String[][] {
       { "key1", "value1" },
       { "key2", "value2" },
    }).collect(Collectors.toMap(data -> data[0], data -> data[1]));
}
```

290 byte

ACED0005737200246465 2E6870692E6F63746F7075732E74657374696E672E5465737

Example: Kryo

Uses some optimizations that we will see for other serializer in a minute!

```
class Message implements Serializable {
  private static final long serialVersionUID = 6455048433435395034L;
  int[] data = {1,2,3};
  String name = "message42";
  boolean validity = true;
  Map<String, String> map = Stream.of(new String[][] {
        { "key1", "value1" },
        { "key2", "value2" },
    }).collect(Collectors.toMap(data -> data[0], data -> data[1]));
}
```

104 byte

010064652E6870692E6F63746F7075732E74657374696E672E546573742432 4D65737
36167E5010104020406010 16A6176612E7574696C2E486173684D61F0010203016B65
79B1030176616C7565B103016B6579B2030176616C7565B2016D6573736167653 4B2
01

Example: Kryo without Class-Serialization

```
class Message implements Serializable {
   private static final long serialVersionUID = 6455048433435395034L;
   int[] data = {1,2,3};
   String name = "message42";
   boolean validity = true;
   Map<String, String> map = Stream.of(new String[][] {
         { "key1", "value1" },
         { "key2", "value2" },
     }).collect(Collectors.toMap(data -> data[0], data -> data[1]));
}
```

66 byte

0101040204060100 6A6176612E7574696C2E486173684D61F0010203016B6579B1030
176616C7565B103016B6579B2030176616C7565B2016D6573736167653 4B201

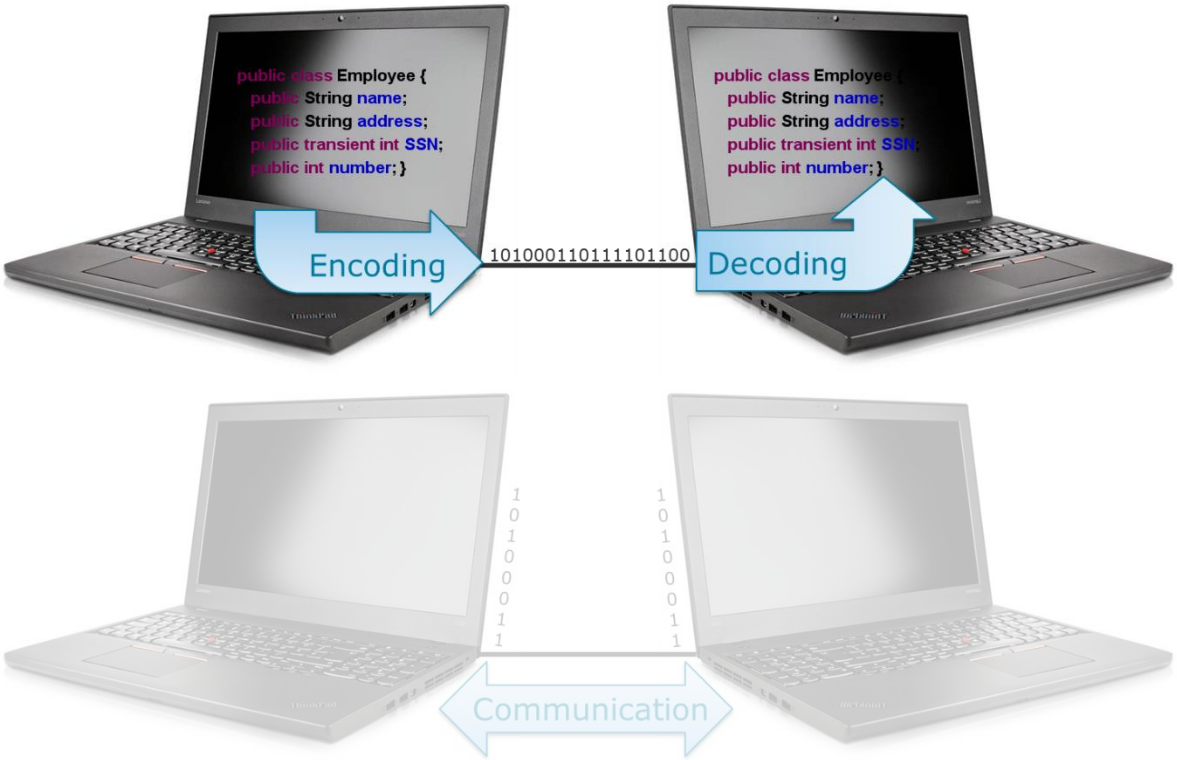
Overview
Encoding and Communication

Encoding

- Language-Specific Encoding
- **JSON/XML Encoding**
- Binary Encoding

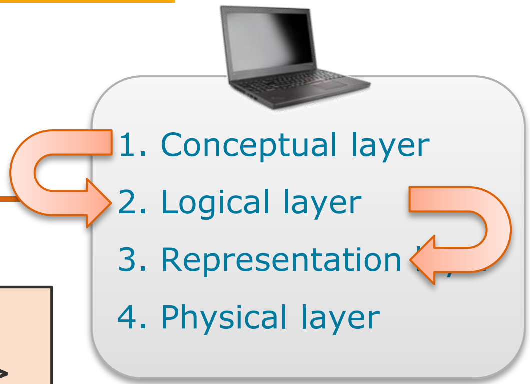
Communication

- Dataflow via Databases
- Dataflow via Services
- Dataflow via Message-Passing



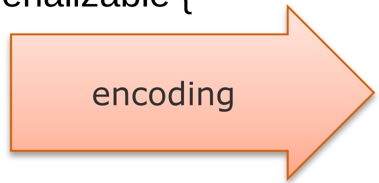
JSON/XML Encoding

Encoding Strategy

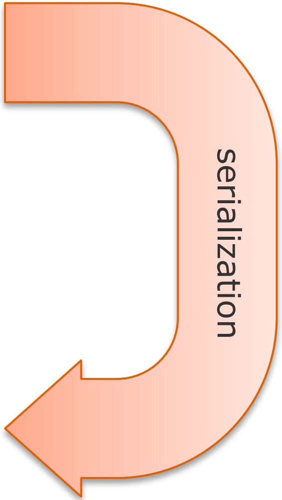


Example: XML

```java
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```



<TestSerial>
  <version>100</version>
  <count>0</count>
</TestSerial>



>51 bytes Java serialization
but language independent!

69 bytes

3C 54 65 73 74 53 65 72 69 61
6C 3E A2 02 03 C7 66 57 27 36
96 F6 E3 E3 13 03 03 C2 F7 66
57 27 36 96 F6 E3 EA 20 20 3C
63 6F 75 6E 74 3E 30 3C 2F 63
6F 75 6E 74 3E A3 C2 F5 46 57
37 45 36 57 26 96 16 C3 EA

JSON/XML Encoding

Structural Elements

```
{
  "_id": 1,
  "username": "ben",
  "password": "ughiwuv",
  "contact": {
                "phone": 0331-1781471,
                "email": "ben87@gmx.de",
                "skype": "benno.miller"
              },
  "access": {
               "level": 3,
               "group": "user"
             },
  "supervisor": {
                   "$ref": "AnnaMT",
                   "$id": 1,
                   "$db": "users"
                 }
}
```

JSON Format

Nested key-value pairs

```
<_id>Benno87</_id>
<username>ben</username>
<password>ughiwuv</password>
<contact>
    <phone>0331-1781254</phone>
    <email>ben87@gmx.de</email>
    <skype>benno.miller</skype>
</contact>
<access>
    <level>3</level>
    <group>user</group>
</access>
<supervisor>
    <ref>AnnaMT</ref>
    <id>1</id>
    <db>users</db>
</supervisor>
```

XML Format

Nested tagged values

Structural Elements

```
{
  "_id": 1,
  "username": "ben",
  "password": "ughiwuv",
  "contact": {
        "phone": 0331-1781471,
        "email": "ben87@gmx.de",
        "skype": "benno.miller"
      },
  "access": {
        "level": 3,
        "group": "user"
      },
  "supervisor": {
        "$ref": "AnnaMT",
        "$id": 1,
        "$db": "users"
      }
}
```

JSON Format

```
<_id>Benno87</_id>
<username>ben</username>
<password>ughiwuv</password>
<contact phone = "0331-1781254
         email = "ben87@gmx.de"
         skype = "benno.miller" />
<access level = "3"
        group = "user" />
<supervisor ref = "AnnaMT"
            id = "1"
            db = "users" />
```

XML Format

Using attributes makes XML much smaller, but the mix of tags and attributes is also harder to read.

Lists

```
{
  "employees": [
    {
      "firstName": "John",
      "lastName": "Doe"
    },
    {
      "firstName": "Anna",
      "lastName": "Smith"
    },
    {
      "firstName": "Peter",
      "lastName": "Jones"
    }]
}
```

JSON Format

```
<employees>
  <employee>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName>
    <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName>
    <lastName>Jones</lastName>
  </employee>
</employees>
```

XML Format

Both formats are similarly expressive.

Some Standardized Encodings

Hasso Plattner Institut

```xml
<UnitTemplate name="Tank" speed="0.5" health="100">
        <weapons>
                <weapon name="big cannon" />
                <weapon name="small turret" />
        </weapons>
        <abilities>
                <cloak cooldown="10" />
                <regenerate healthPerSecond="1" />
        </abilities>
</UnitTemplate>
```

e.g., for Web Services

XML

Extensible Markup Language

```
UnitTemplate name="Tank" speed=0.5 health=100 {
        weapons "big cannon" "small turret"
        abilities {
                "cloak" cooldown=10
                "regenerate" healthPerSecond=1
        }
}
```

SDL

Simple Declarative Language

```json
{ "UnitTemplate" : {
        "name" : "Tank",
        "speed" : 0.5,
        "health" : 100,
        "weapons" : [ "big cannon", "small turret" ],
        "abilities" : [
                { "cloak" : { "cooldown" : 10 } },
                { "regenerate" : { "healthPerSecond" : 1 } }
        ]
} }
```

e.g., for REST-based services

JSON

JavaScript Object Notation

```toml
[UnitTemplate]
name = "Tank"
speed = 0.5
health = 100
weapons = [ "big cannon", "small turret" ]

[[UnitTemplate.Ability]]
type = "cloak"
cooldown = 10

[[UnitTemplate.Ability]]
type = "regenerate"
healthPerSecond = 1
```

TOML

Tom's Obvious, Minimal Language

And many more: YAML, CSV, …

Standardized Encodings

Advantages

- Language and address-space independence
- Human readability (sometimes)
- Ability to query and store in a structured manner

Problems

- No or only weak typing
  - Number encoding is ambiguous and imprecise.
- No support for binary strings (only Unicode)
  - Storing binary strings in Unicode increases data size (>33%).
- Schemata, if needed, require optional (complicated) schema support (e.g., XML Schema)
  - Without explicit schema definition, applications must define schemata.
- Large binary representation (if String is directly serialized into binary)
  - Native encoding formats are typically more concise.

JSON distinguishes only strings and numbers, but not int, float, or double; XML sees all values as strings

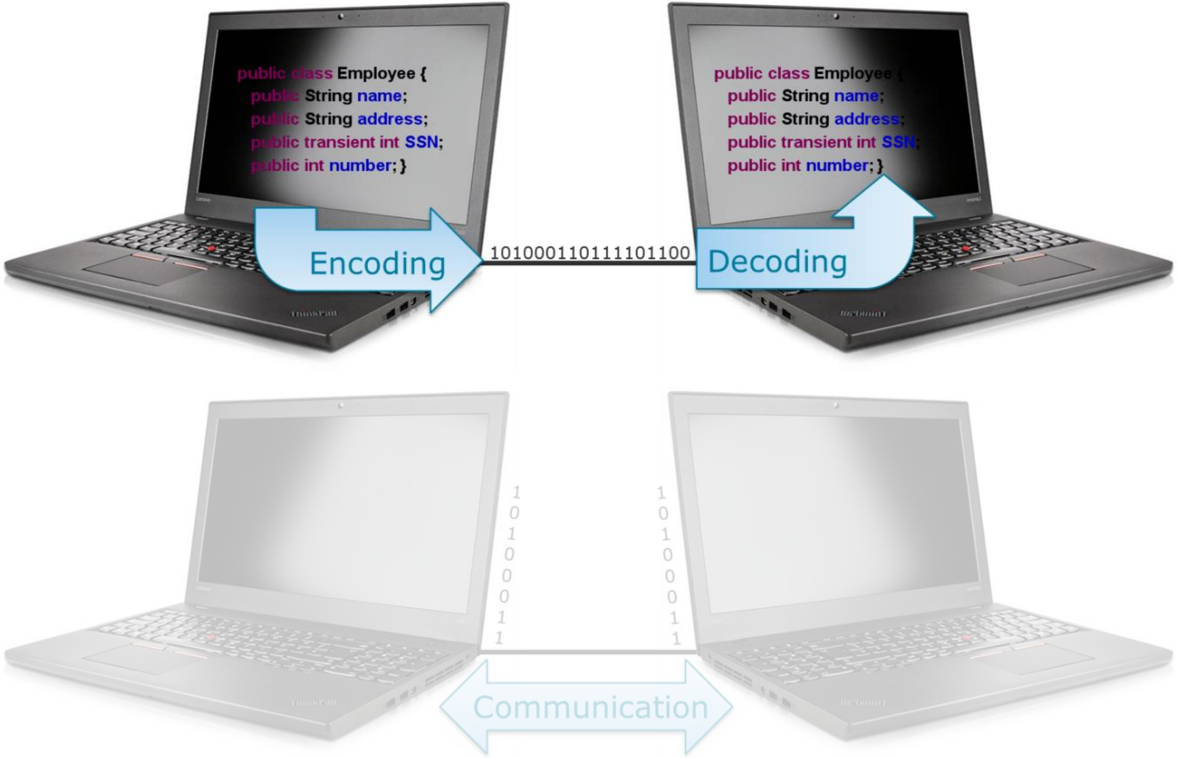
Encoding and Communication

Encoding

- Language-Specific Encoding
- JSON/XML Encoding
- **Binary Encoding**

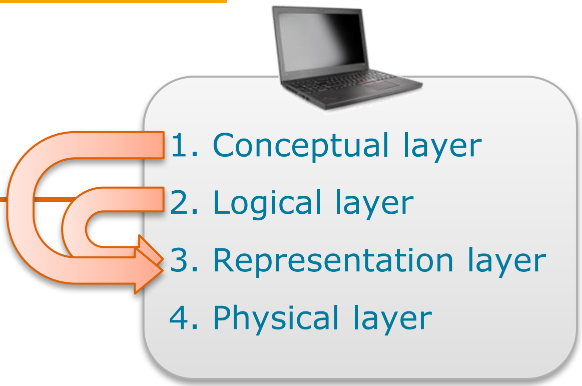
Communication

- Dataflow via Databases
- Dataflow via Services
- Dataflow via Message-Passing



Binary Encoding

Motivation



Problems

- Unicode formats and their naïve binary encodings are large.
- Data types are lost.

Idea

- Encode Unicode formats into binary strings with format-specific encodings.
- Keep the original structure (attribute names, nesting, …).

Binary Encodings

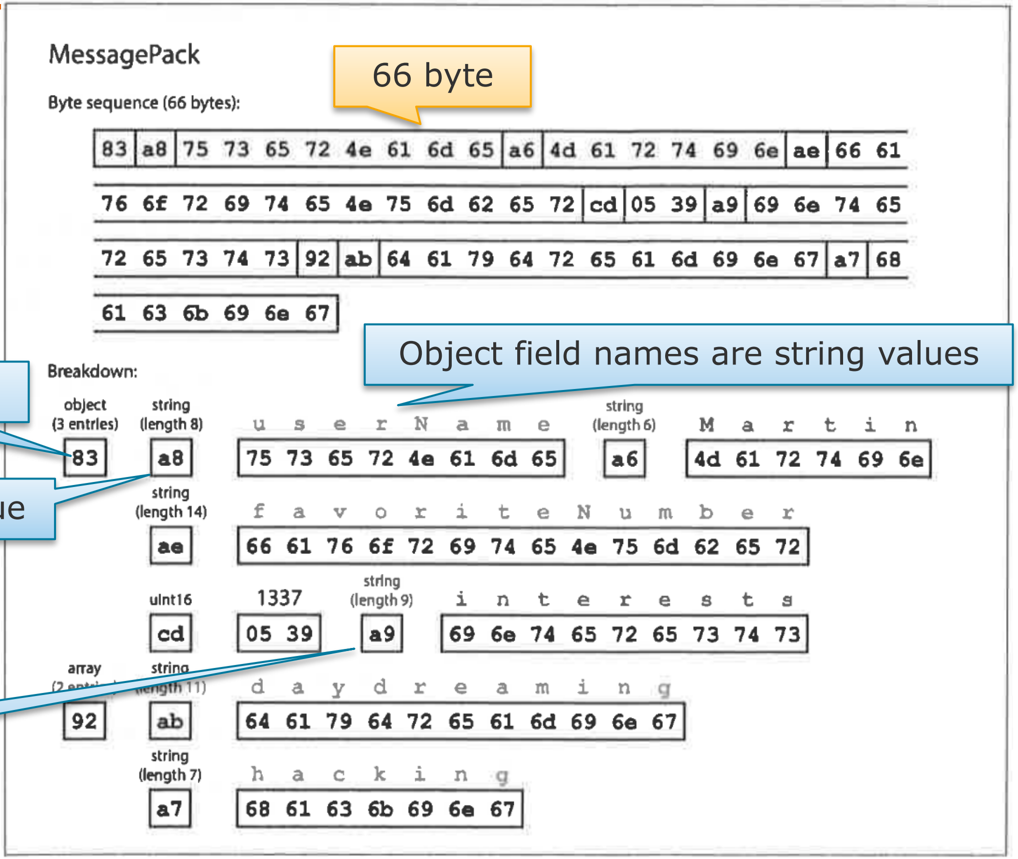
- For JSON: MessagePack, BSON, BJSON, UBJSON, BISON, Smile, …
- For XML: WBXML, Fast Infoset, …
- For Code: Apache Thrift, Protocol Buffers, Apache Avro

MessagePack

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

JSON Format

81 byte



66 byte

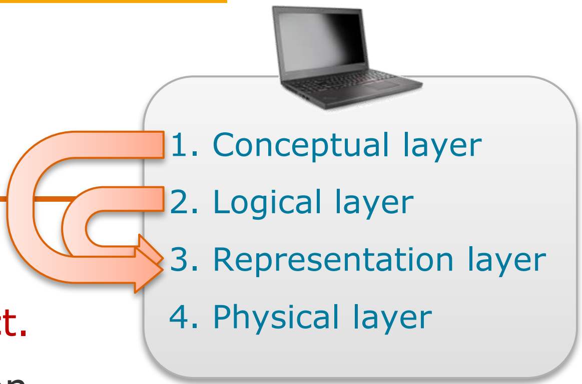
Object field names are string values

Object preamble

Alternating: data type (+ length) and value

Here: string-length < 16

Schema-based Binary Encoding



Motivation

- MessagePack stores attribute names (and types) for each object.
  - ➢ Redundant information that increases memory consumption

Idea

- Define the attributes (= fields) once for all objects.
  - ➢ Define a schema!
  - ➢ No need to encode the attributes and their size

Binary Encoding Libraries

- Apache Thrift (by Facebook)
  - https://thrift.apache.org/
- Protocol Buffers (by Google)
  - https://developers.google.com/protocol-buffers/
- …

} both open source since 2007

Binary Encoding
Thrift with BinaryProtocol

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

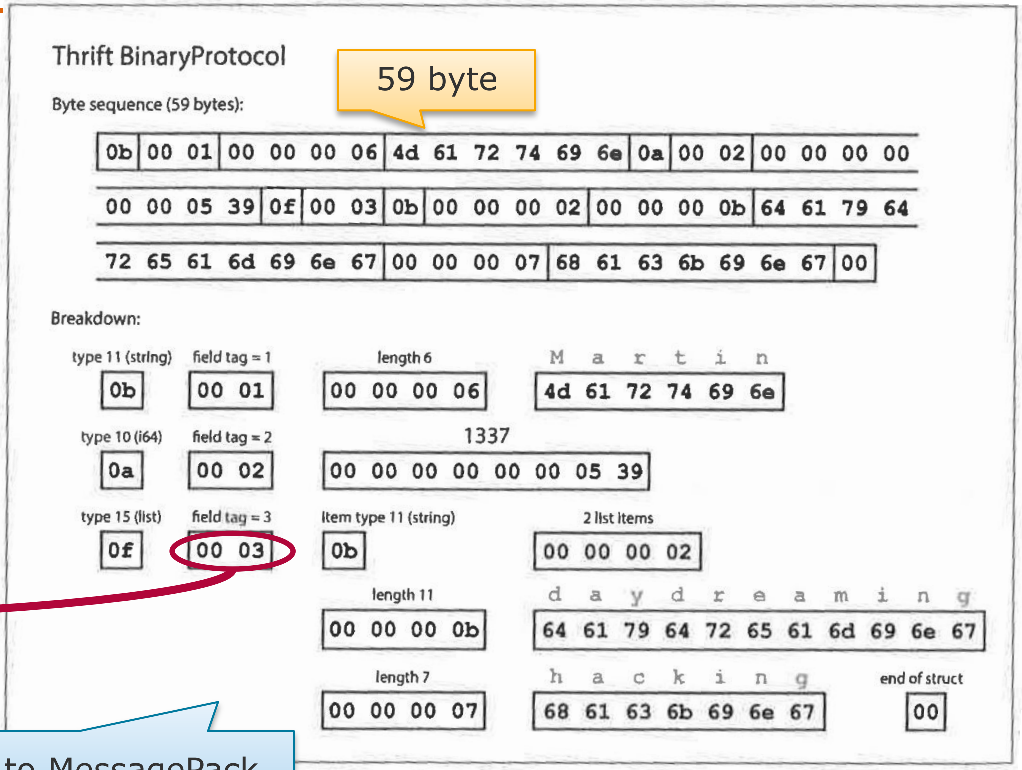
JSON Format

81 byte

```
struct Person {
  1: required string       username,
  2: optional i64          favoriteNumber,
  3: optional list<string> interests
}
```

Thrift Struct

Schema definition



59 byte

Backward compatibility

- Adding optional fields and changing field names possible
- Changing field tags (or types) breaks reading of old data

Very similar to MessagePack, but without field names

Thrift with CompactProtocol

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

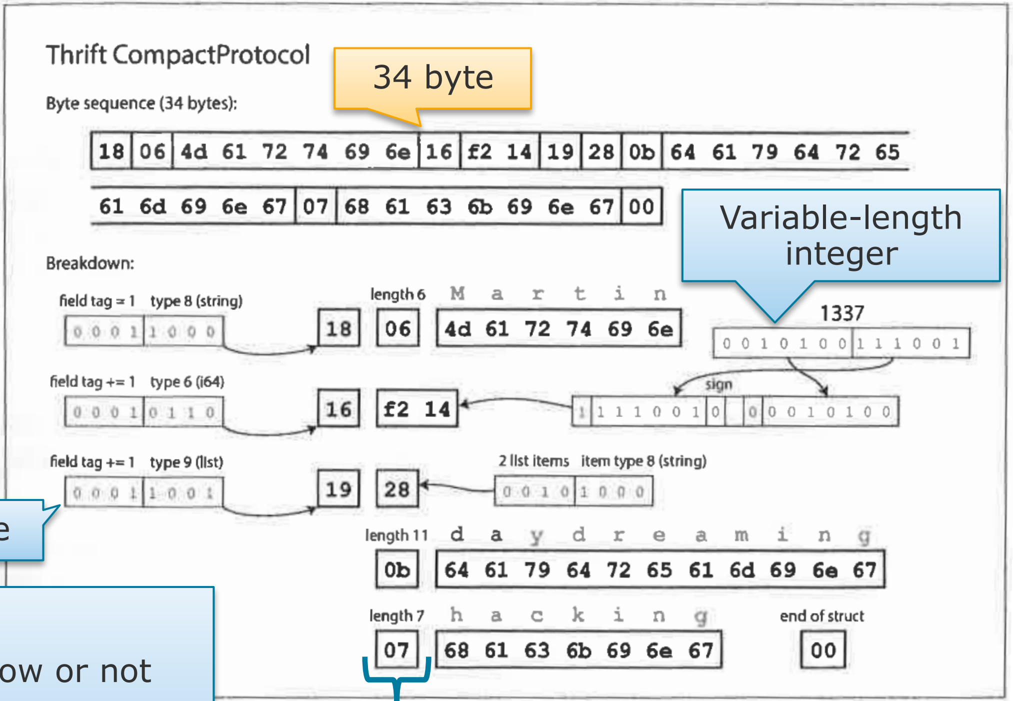
JSON Format

81 byte

```
struct Person {
  1: required  string        username,
  2: optional  i64           favoriteNumber,
  3: optional  list<string>  interests
}
```

Thrift Struct

Field tag + type in one byte



Variable-length integers

Variable-length integers

- First bit in each byte encodes if more bytes follow or not (0 = "last byte", 1 = "more bytes to come")
- Last bit of first byte encodes the integers sign (0 = "+", 1 = "–")

Protocol Buffers

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

JSON Format

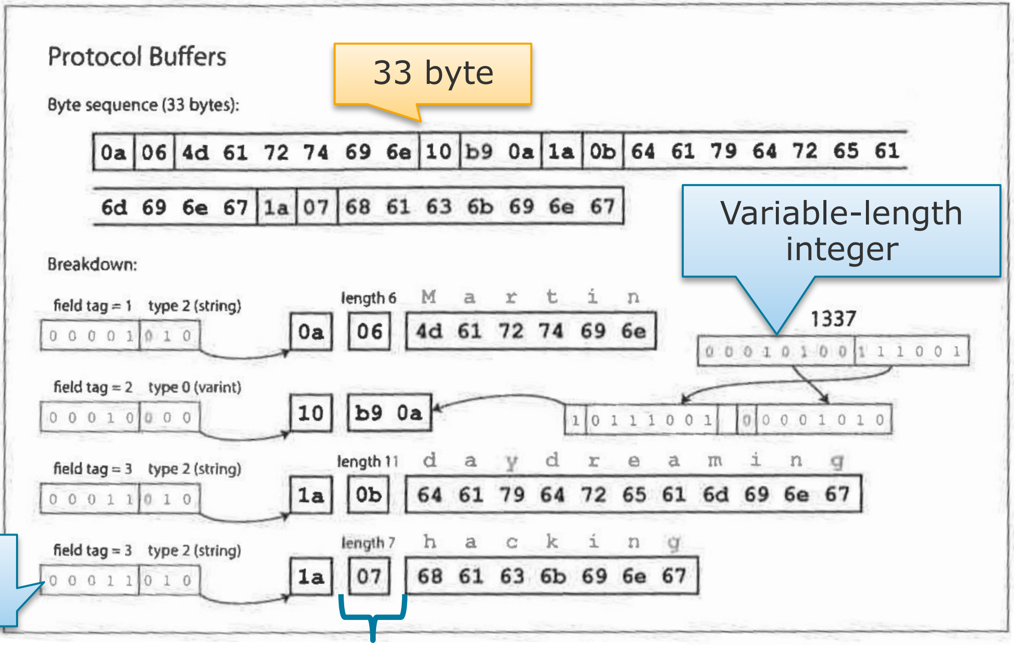
81 byte

```
message Person {
  required    string   username,          = 1;
  optional    i64      favoriteNumber,    = 2;
  repeated    string   interests          = 3;
}
```

P.B. message

Schema definition

Put values with same field tag in a list



Variable-length integers

Protocol Buffers are very similar to Thrift's CompactProtocol

Avro

Apache Avro

- A binary encoding format developed as a sub-project of Hadoop in 2009
  - https://avro.apache.org/
- Differences to Thrift and Protocol Buffers:
  - No tag numbers: fields are matched by order in schema and byte sequence
  - No field modifiers `optional` or `required`: optional fields have default values
  - Special data type `union`: specifies multiple data types (and `null` if allowed)
  - Nullable fields must have type `union`

```
record Person {
  string               username;
  union{null,long}     favoriteNumber=null;
  array<string>        interests;
}
```

Avro record

Schema definition

Uses of Avro

- Apache Pig (query engine for Hadoop)
- Espresso (database management system)
- Avro RPC (remote procedure call protocol)
- …

Avro

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

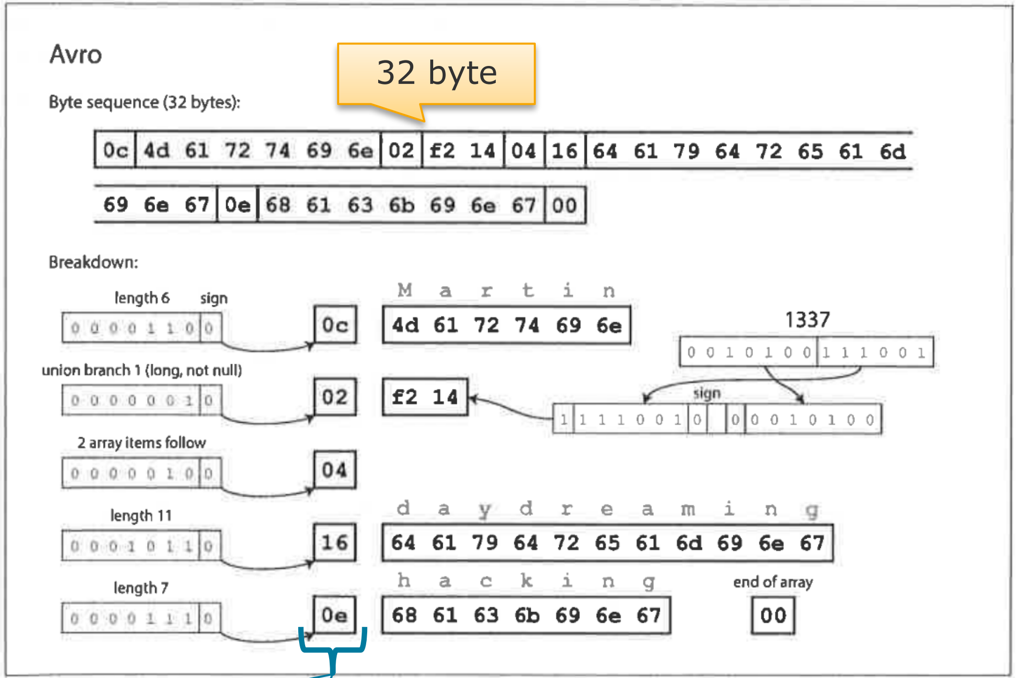
JSON Format

81 byte

```
record Person {
  string              username;
  union{null,long}    favoriteNumber=null;
  array<string>       interests;
}
```

Avro record

Schema definition



The data type is known;
the field is matched by sequence
→ we only need the length!

Avro: Writer's and Reader's schemata

- Avro associates data with two different schemata:
  - Writer's schema:
    - The schema with which the data was written
    - Fix for written data; might differ for other (newer/older) datasets
    - Stored with the data (in same file, database, or connection handshake)
  - "self-describing data"
  - Reader's schema:
    - The schema of the application reading the data
    - Might change with the version of the application
    - Stored in application
- When reading: Avro dynamically maps Reader's and Writer's schemata
- When writing: Avro uses the Reader's schema

Binary Encoding

Avro: W-R Mapping

Writer's schema

Data type	Field name
string	userName
union{null,long}	favoriteNumber
array<string>	interests
string	photoURL

Reader's schema

Data type	Field name
long	userID
union{null,int}	favoriteNumber
string	userName
array<string>	interests

Advantages

- Most compact binary encoding (compared with previous formats)
- Backward compatibility:
  - Avro dynamically maps schemata at read-time and resolves differences
  - Fields are mapped by name; no field tags that can break the encoding
  - Default values account for missing fields
  - Data types can change if conversion is possible (e.g. int → long, float → string)
- Schema generation:
  - Reader's schemata can be generated from existing data (no need to generate field tags that match a Writer's schema)

Encoding and Communication

Btw.: We did **not** discuss **encryption and authentication** here, which is also important but something I leave to

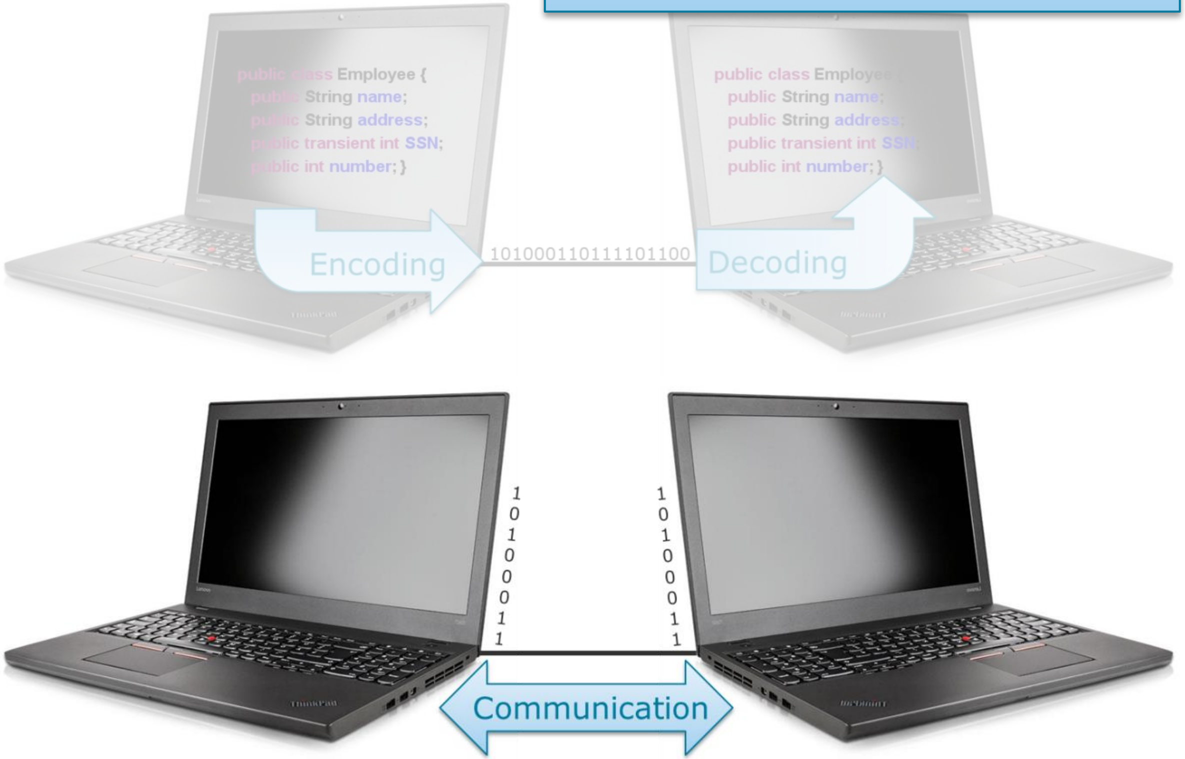
"Internet Security" by Prof. Meinel

Encoding

- Language-Specific Encoding
- JSON/XML Encoding
- Binary Encoding

Communication

- Dataflow via Databases
- Dataflow via Services
- Dataflow via Message-Passing



Motivation

The same also applies for **threads**!

Processes communicate

- With themselves
- With other processes on the same machine
- With processes on remote machines over the network
  - ➢ Data often needs to pass process boundaries!

Processes are heterogeneous

- Different languages, address spaces, access rights, hardware resources, complexities, interfaces, …
  - ➢ Communication models/protocols needed!

Process communication is expensive

- Communication channels (buses, network, memory, …) have limited speed, bandwidth and throughput
  - ➢ Number and size of messages matters!

Internet Protocol Suite

Internet protocol suite

Application layer

BGP · DHCP · DNS · FTP · HTTP · IMAP · LDAP · MGCP · NNTP · NTP · POP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

Transport layer

TCP · UDP · DCCP · SCTP · RSVP · *more...*

Internet layer

IP (IPv4 · IPv6) · ICMP · ICMPv6 · ECN · IGMP · IPsec · *more...*

Link layer

ARP · NDP · OSPF · Tunnels (L2TP) · PPP · MAC (Ethernet · DSL · ISDN · FDDI) · *more...*

V · T · E

- Sending of datagrams in local network
- Direct host-to-host messaging; no routing
- Addressing via MAC address
  - e.g. 34:f3:9a:fa:fb:59
- Hardware dependent (→ drivers needed)
- Abstracting hardware details to above layers

➢ Packetizing, (local) addressing, transmission and receiving of data

Internet Protocol Suite

Internet protocol suite

Application layer

BGP · DHCP · DNS · FTP · HTTP · IMAP · LDAP · MGCP · NNTP · NTP · POP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

Transport layer

TCP · UDP · DCCP · SCTP · RSVP · *more...*

Internet layer

IP (IPv4 · IPv6) · ICMP · ICMPv6 · ECN · IGMP · IPsec · *more...*

Link layer

ARP · NDP · OSPF · Tunnels (L2TP) · PPP · MAC (Ethernet · DSL · ISDN · FDDI) · *more...*

V · T · E

- Routing of datagrams across networks
- IP addresses
  - for addressing and routing
  - map to MAC addresses
  - e.g. 172.17.5.57
- Abstracting the actual network topology to above layers

➢ Packetizing, (global) addressing and routing of data

Internet Protocol Suite

TCP

- reliable (flow control)
- connection-based
- slow
- ➢ lost-message resents
- ➢ message ordering
- ➢ error correction
- ➢ duplicate removal
- ➢ congestion control

UDP

- unreliable
- connectionless
- fast

Internet protocol suite

Application layer

BGP · DHCP · DNS · FTP · HTTP · IMAP · LDAP · MGCP · NNTP · NTP · POP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

Transport layer

TCP · UDP · DCCP · SCTP · RSVP · *more...*

Internet layer

IP (IPv4 · IPv6) · ICMP · ICMPv6 · ECN · IGMP · IPsec · *more...*

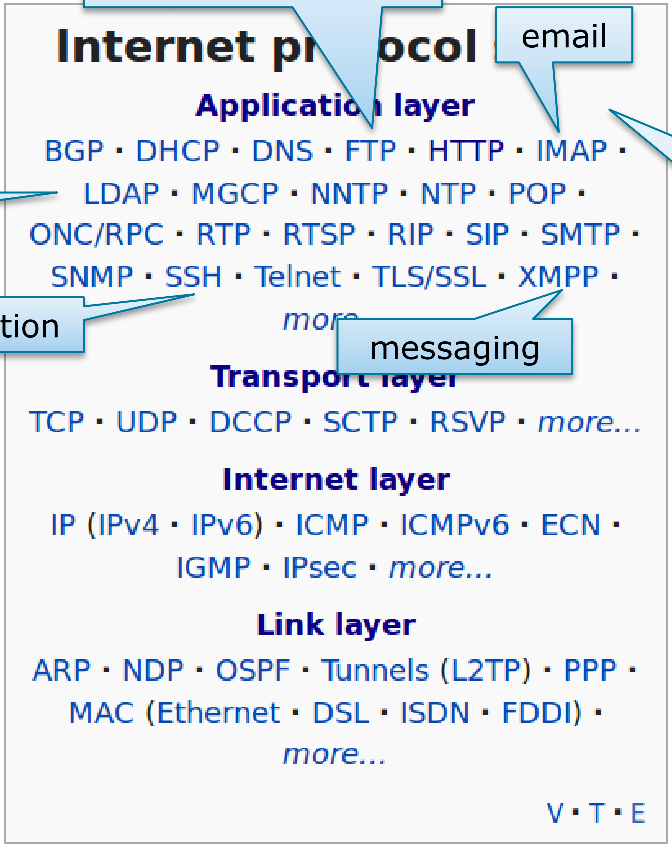
Link layer

ARP · NDP · OSPF · Tunnels (L2TP) · PPP · MAC (Ethernet · DSL · ISDN · FDDI) · *more...*

V · T · E

- Managing the datagram exchange
  - host-to-host (via arbitrary hops)
  - communication protocol
  - communication channel
- Port numbers for application addressing
  - e.g. 8080
- Abstracting communication details to above layers
- ➢ Packetizing of data

Internet Protocol Suite



Internet pr ocol

Applicatio layer

BGP · DHCP · DNS · FTP · HTTP · IMAP · LDAP · MGCP · NNTP · NTP · POP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

Transport layer

TCP · UDP · DCCP · SCTP · RSVP · *more...*

Internet layer

IP (IPv4 · IPv6) · ICMP · ICMPv6 · ECN · IGMP · IPsec · *more...*

Link layer

ARP · NDP · OSPF · Tunnels (L2TP) · PPP · MAC (Ethernet · DSL · ISDN · FDDI) · *more...*

V · T · E

large file transfer

email

access control

encrypted communication

messaging

- Creation and interpretation of data
- "Your application"
- Use the (reliable or unreliable) channels (identified by IP + Port) to send/receive data
- Higher level communication protocols
  - client-server
  - peer-to-peer
- "IP + Port + Applications" is a service

Internet Protocol Suite

Internet protocol suite

Application layer

BGP · DHCP · DNS · FTP · HTTP · IMAP · LDAP · MGCP · NNTP · NTP · POP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

Transport layer

TCP · UDP · DCCP · SCTP · RSVP · *more...*

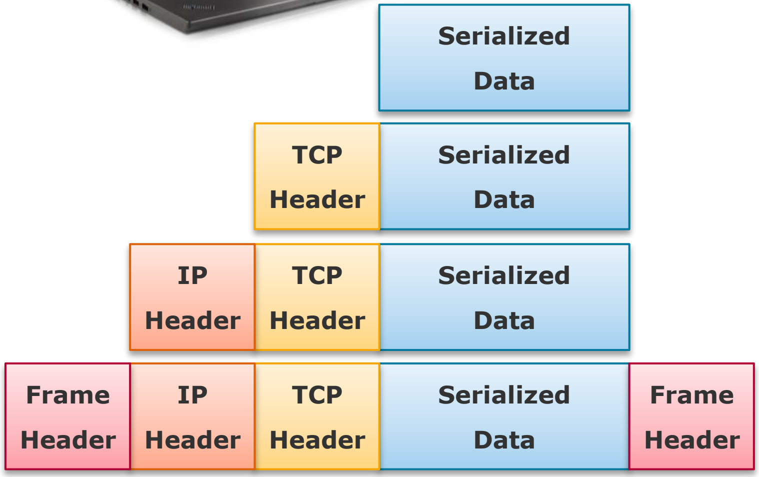
Internet layer

IP (IPv4 · IPv6) · ICMP · ICMPv6 · ECN · IGMP · IPsec · *more...*

Link layer

ARP · NDP · OSPF · Tunnels (L2TP) · PPP · MAC (Ethernet · DSL · ISDN · FDDI) · *more...*

V · T · E

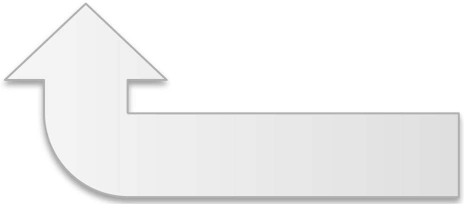


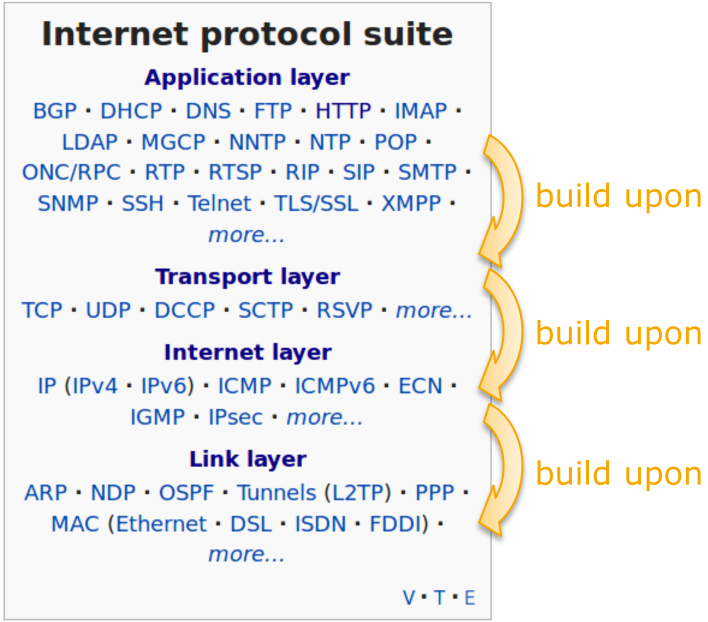
More in Prof. Meinel's lecture "Internet and WWW Technologies"

Communication Principle

Communication

- Message format: agreement on model, schema, and encoding of messages
- Protocol: agreement on how messages are exchanged





- Von Neumann architecture:
  - Messages may contain data and/or instructions.
  - Application needs to interpret the messages.

Models of Dataflow

Dataflow through Databases

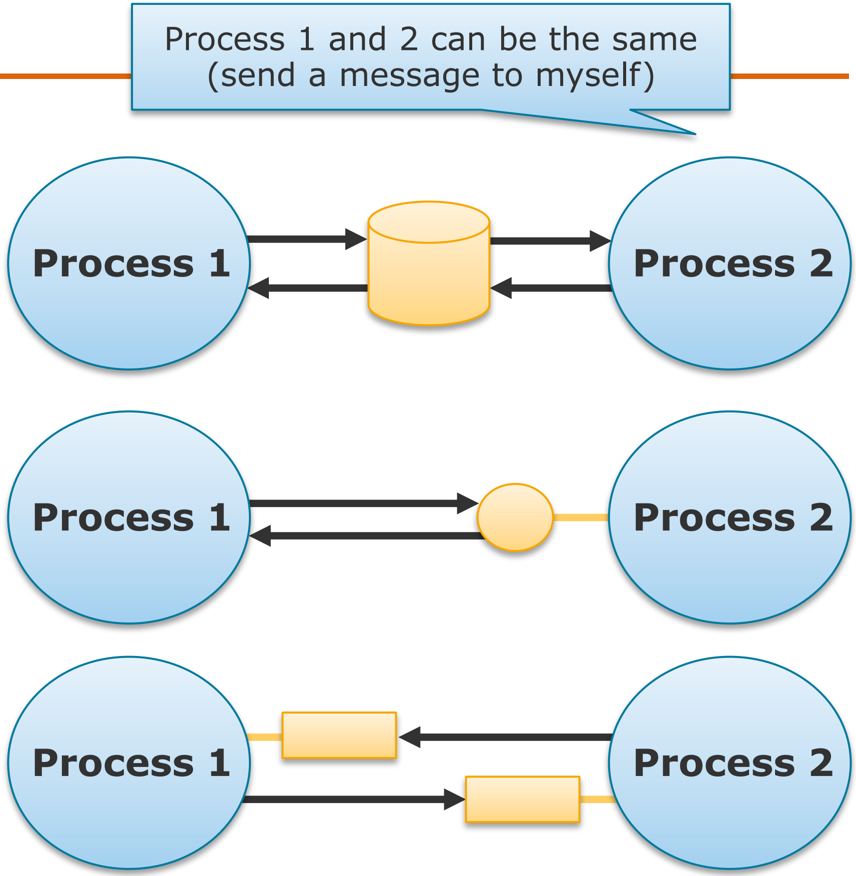
- information storage and retrieval

Dataflow through Services

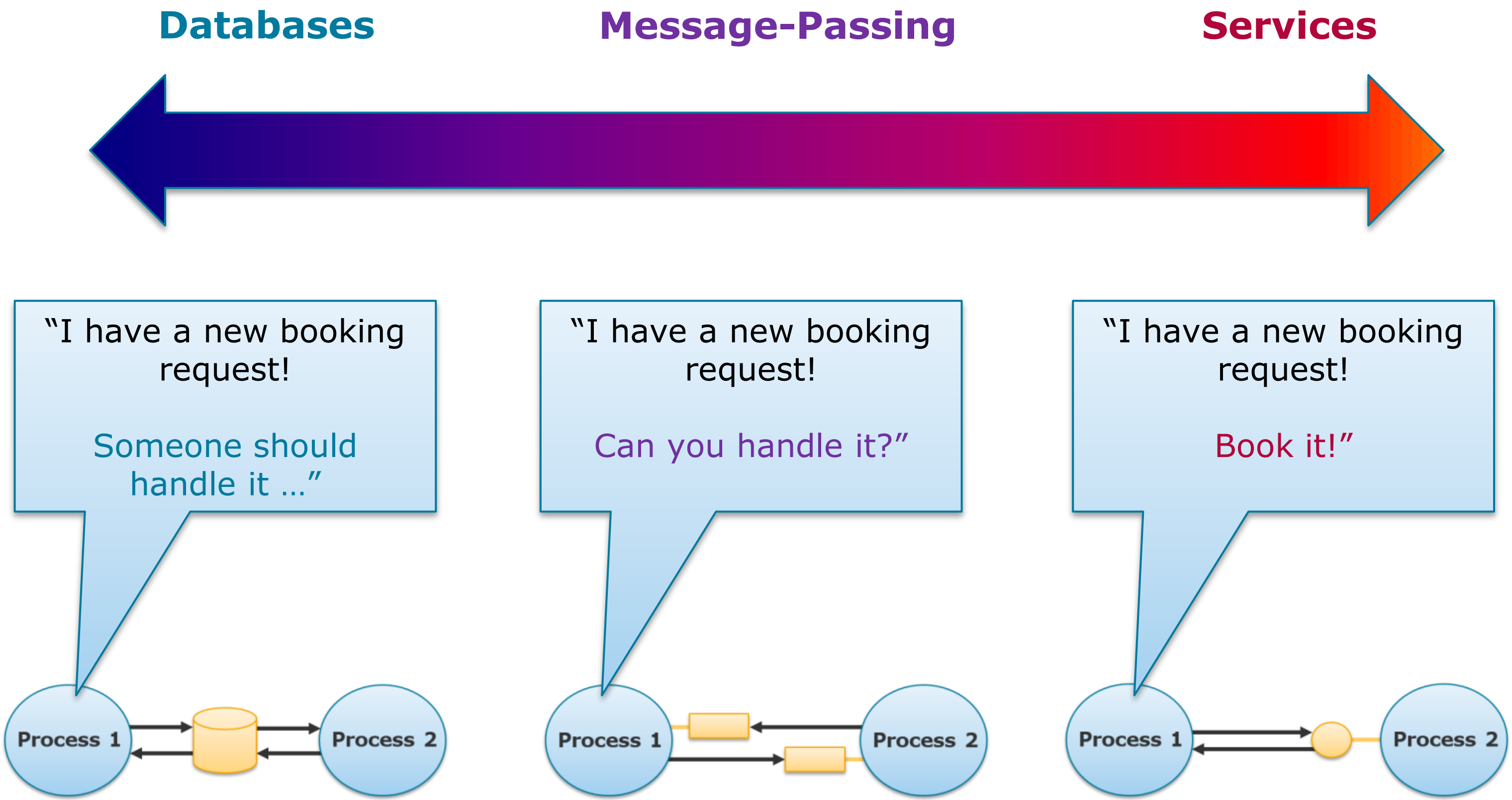
- service calls with responses

Message-Passing Dataflow

- asynchronous messages

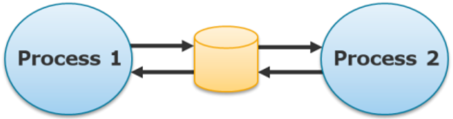


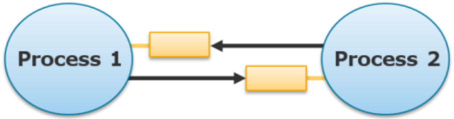
Models of Dataflow

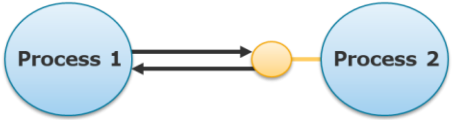


Models of Dataflow

Databases	Message-Passing	Services
Data	Messages	Function calls
No response	Maybe response	Response
Non-blocking	Usually non-blocking	Blocking
Asynchronous	Asynchronous	Synchronous
No addressing	Addressing recipient	Addressing recipient



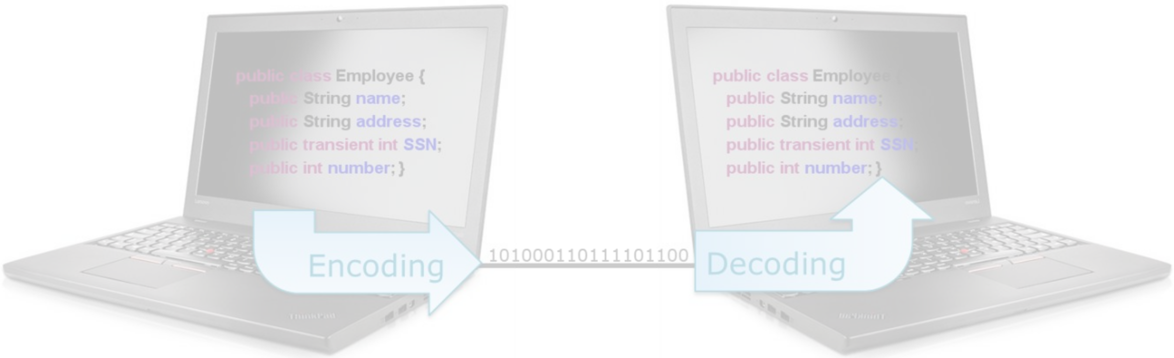




Encoding and Communication

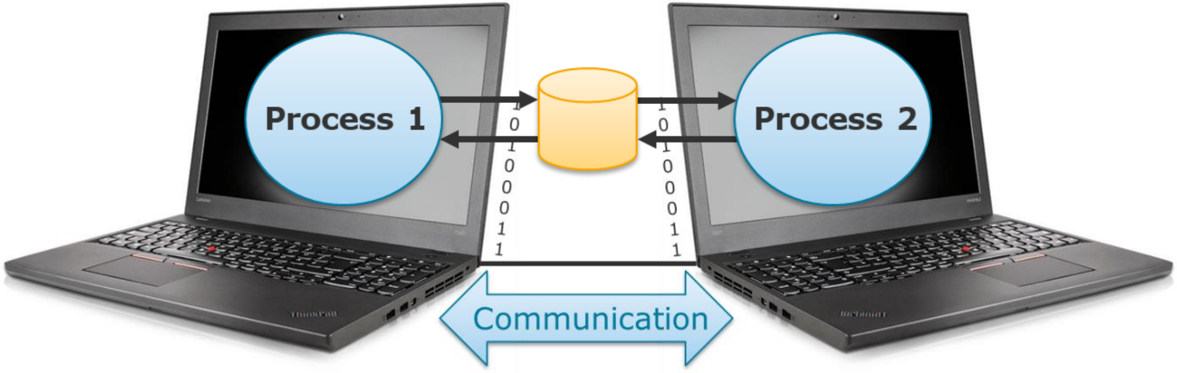
Encoding

- Language-Specific Encoding
- JSON/XML Encoding
- Binary Encoding

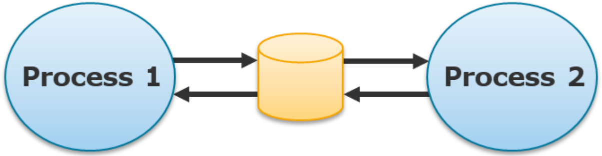


Communication

- **Dataflow via Databases**
- Dataflow via Services
- Dataflow via Message-Passing



Communication Principle



- Processes write data to and read data from a database:
  - Communication through manipulation of (persistent) global state
- Requires commonly understood model, schema, and encoding:
  - Model: relational, key-value, wide-column, document, graph, …
  - Schema: either schema-on-read or schema-on-write
  - Encoding: Unicode, binary, …
- Implicit message exchange:
  - No explicit sender or receiver (think of broadcast messages)
- Varying message lifetimes:
  - Data can quickly be overwritten (= overwritten message is lost)
  - Data can stay forever (known as: data outlives code)
- Shared memory parallel applications are very similar w.r.t. this model

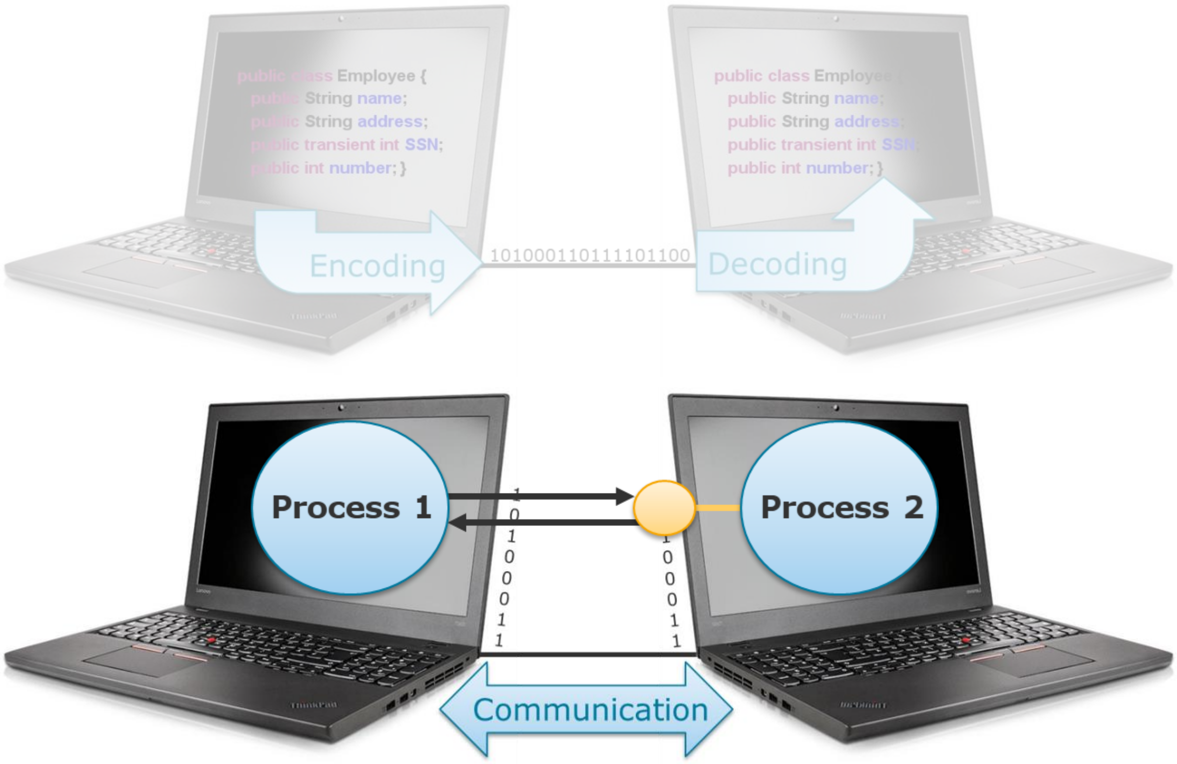
Encoding and Communication

Encoding

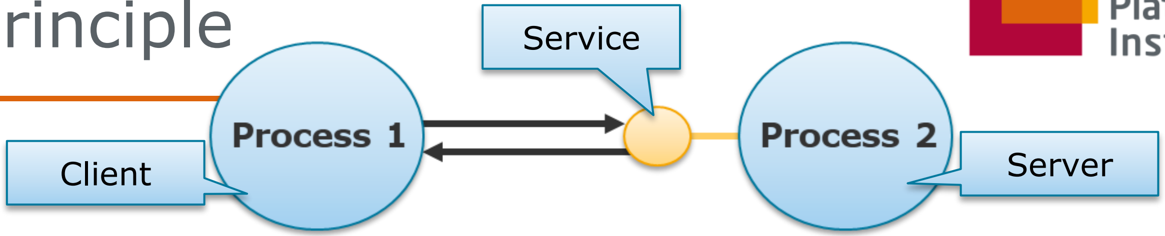
- Language-Specific Encoding
- JSON/XML Encoding
- Binary Encoding

Communication

- Dataflow via Databases
- **Dataflow via Services**
- Dataflow via Message-Passing



Communication Principle



Communication Structure

- Service:
  - An API that can be accessed by other (remote) processes
  - Identified by IP + Port
  - Offers functions that may take arguments (= a send message) and return values (= a receive message)
  - Offered functions define fine-grained restrictions on what can be communicated and what not (different from database APIs, which are more open)

Asymmetric Communication

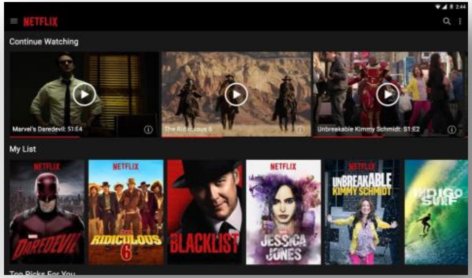
- Communicating processes have two roles:
  - Server: exposes a service that other processes can see and use
  - Client: connects to a server's service and calls functions

Service-Oriented Architecture (SOA)

- A server process can, again, become a client to some other server.
  - (Distributed) systems of interacting processes
- Services should be self-contained black box components that represent logical activities hiding lower-level services.
- Microservice architecture:
  - Variant of SOA where services are particularly fine-grained and the protocol is lightweight

Examples

Web Browser



Apps

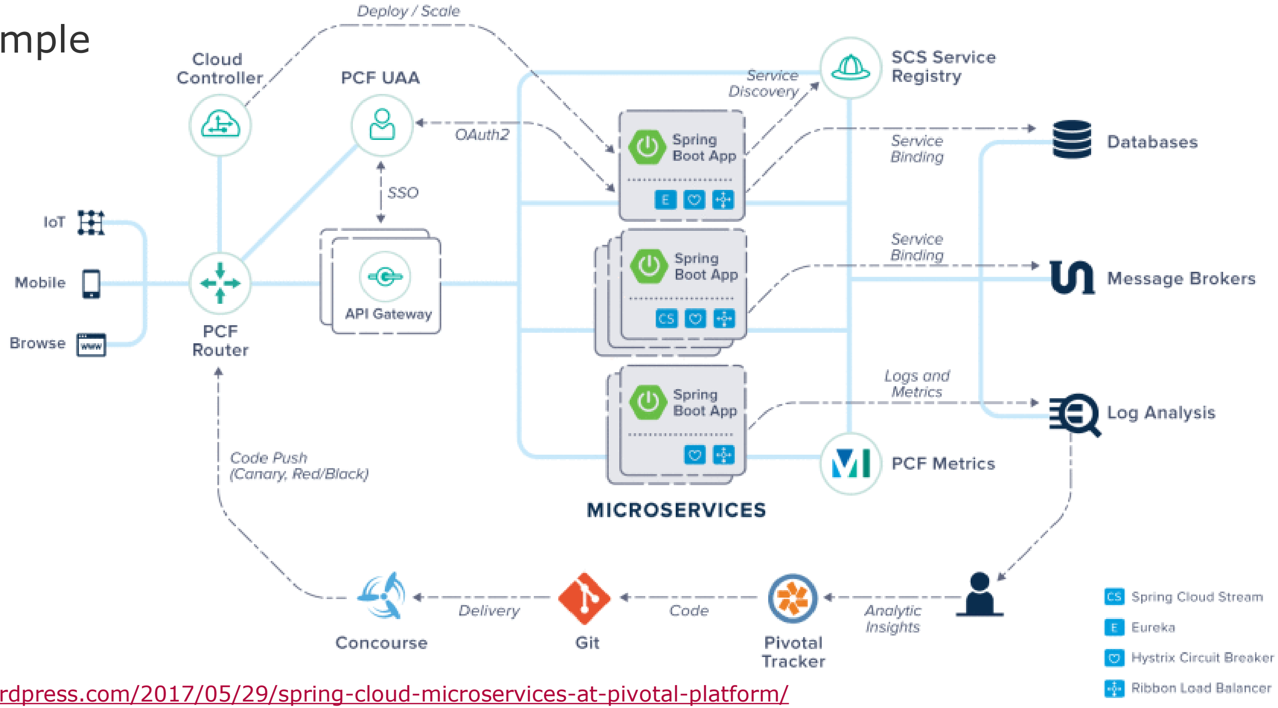
Online Games

Communication Principle

Service-Oriented Architecture (SOA)

- Microservice architecture
  - Example

➢ A light-weight application framework for Java with support for microservice development

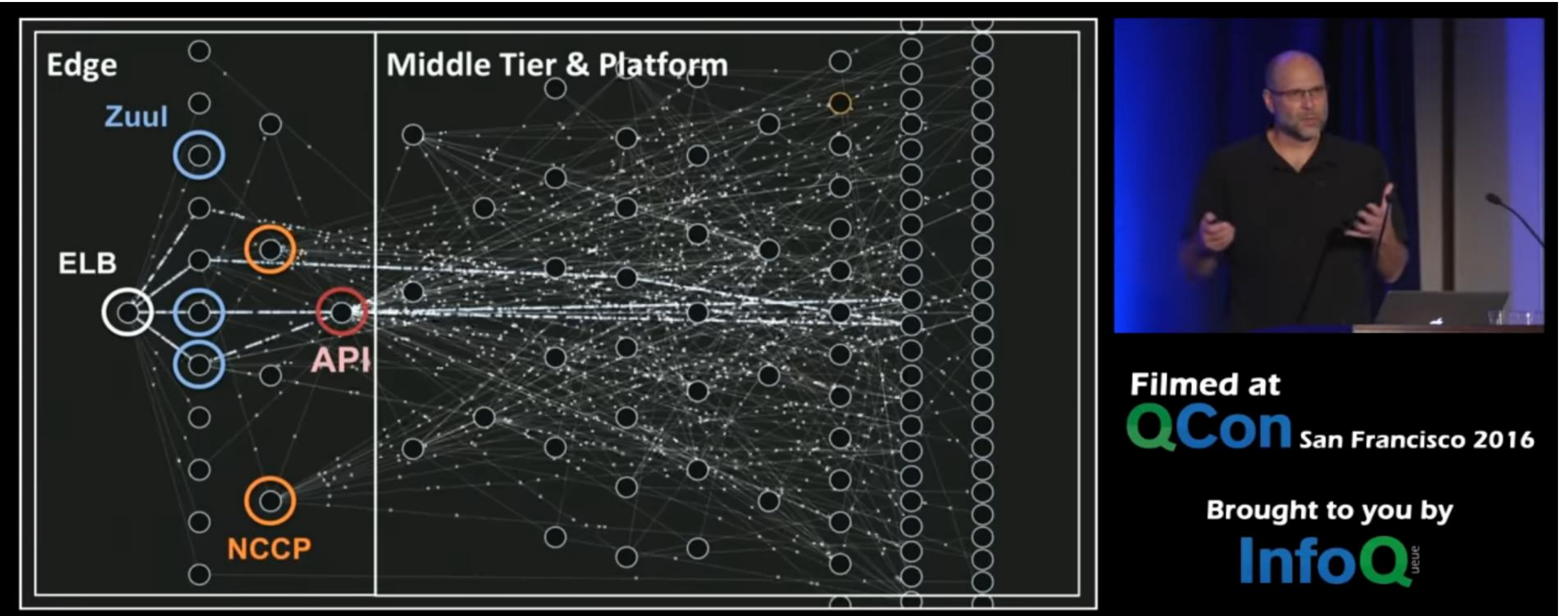


https://piotrminkowski.wordpress.com/2017/05/29/spring-cloud-microservices-at-pivotal-platform/

Communication Principle

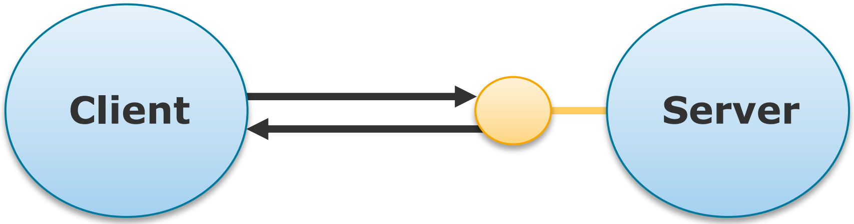
Service-Oriented Architecture (SOA)

- Microservice architecture
  - Example: Mastering Chaos - A Netflix Guide to Microservices



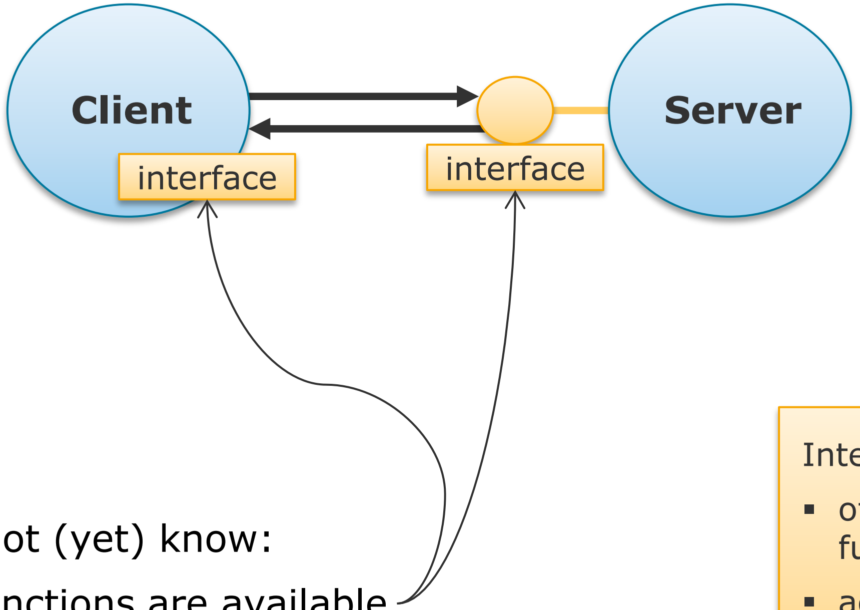
https://www.youtube.com/watch?v=CZ3wIuvmHeM

Motivation for Service Protocols



- Client knows:
  - How to address the server (IP + Port)
  - How to send data (serialization + packaging)
- Client does not (yet) know:
  - What functions are available
  - What data it needs to send to call a function

Motivation for Service Protocols

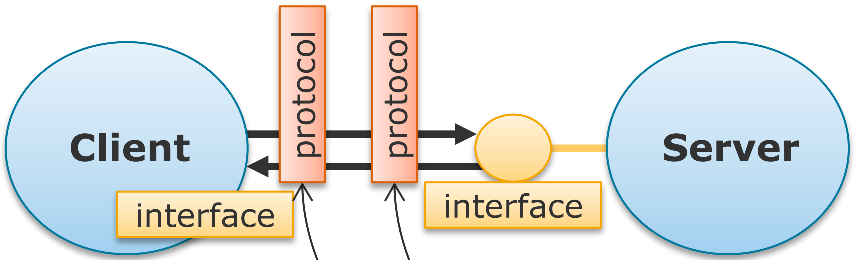


- Client does not (yet) know:
  - What functions are available
  - What data it needs to send to call a function

Interface:

- of the service functions
- added during client **implementation**

Motivation for Service Protocols



- Client does not (yet) know:
  - What functions are available
  - What data it needs to send to call a function

Protocol:
- function call → data
- data → function call

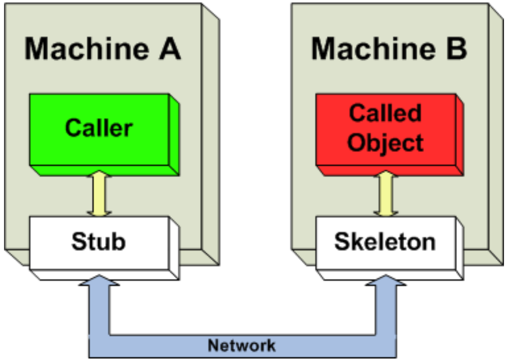
(w.r.t. given interface)

Remote Procedure Call (RPC)

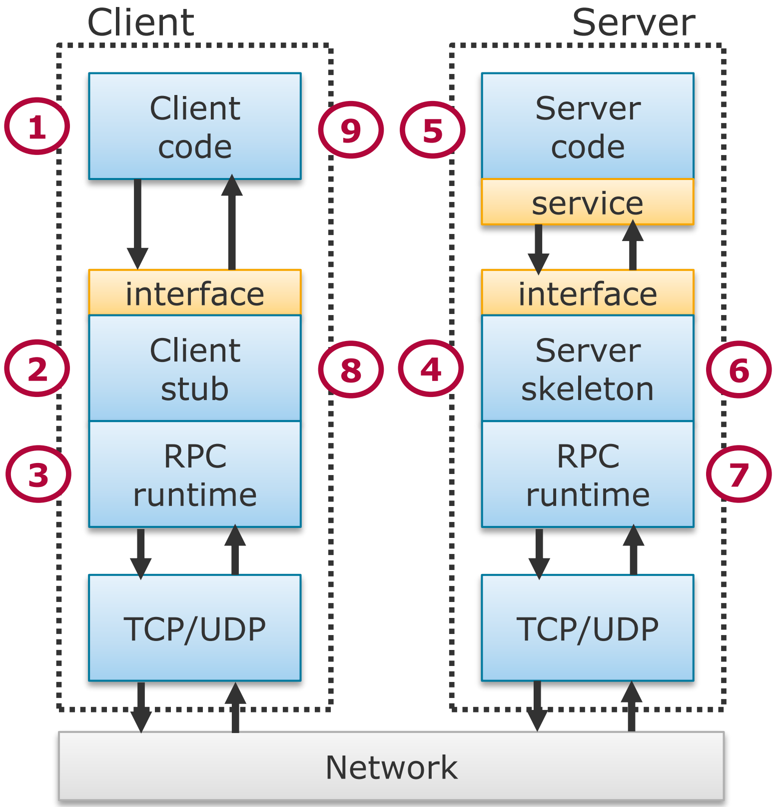
- A protocol that allows processes to directly call functions in remote processes (i.e., cause procedures to execute in different address spaces).
- The object-oriented equivalent is remote method invocation (RMI)

Both work very similarly.

- Remote procedures are called like normal (local) procedures.
  - ➢ Tight coupling between processes
- Requires the service's interface on server and client.
- Implements the protocol for transmitting a function call.
- The RPC framework use the interface to automatically generate two proxies.
  - Stub
    - function call → data
  - Skeleton
    - data → function call

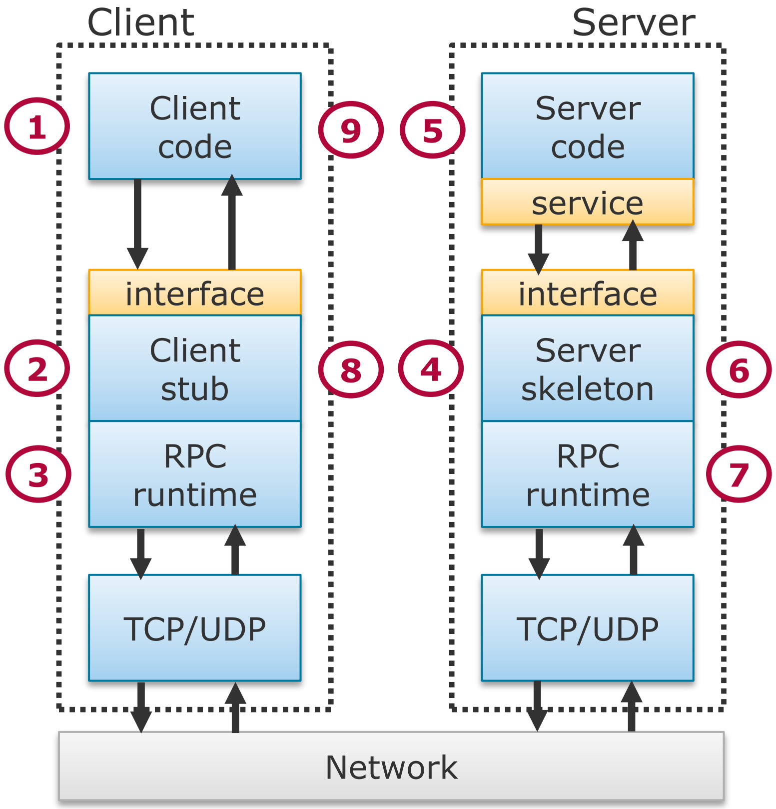


Remote Procedure Call (RPC)

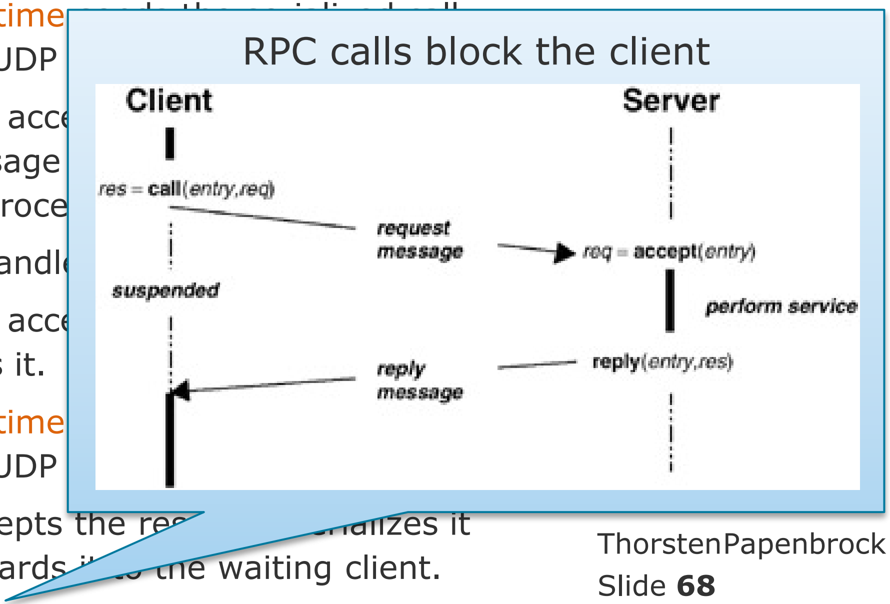


1. Client calls a remote procedure and waits.
2. Stub accepts the procedure call and serializes both the call and its parameters.
3. RPC Runtime sends the serialized call via TCP/UDP to the server.
4. Skeleton accepts procedure call, deserializes the message and calls the corresponding service procedure with the given parameters.
5. Server handles the call and returns a result.
6. Skeleton accepts the result and serializes it.
7. RPC Runtime sends the serialized result via TCP/UDP back to the client.
8. Stub accepts the result, deserializes it and forwards it to the waiting client.
9. Client awakes and accepts the result.

Remote Procedure Call (RPC)



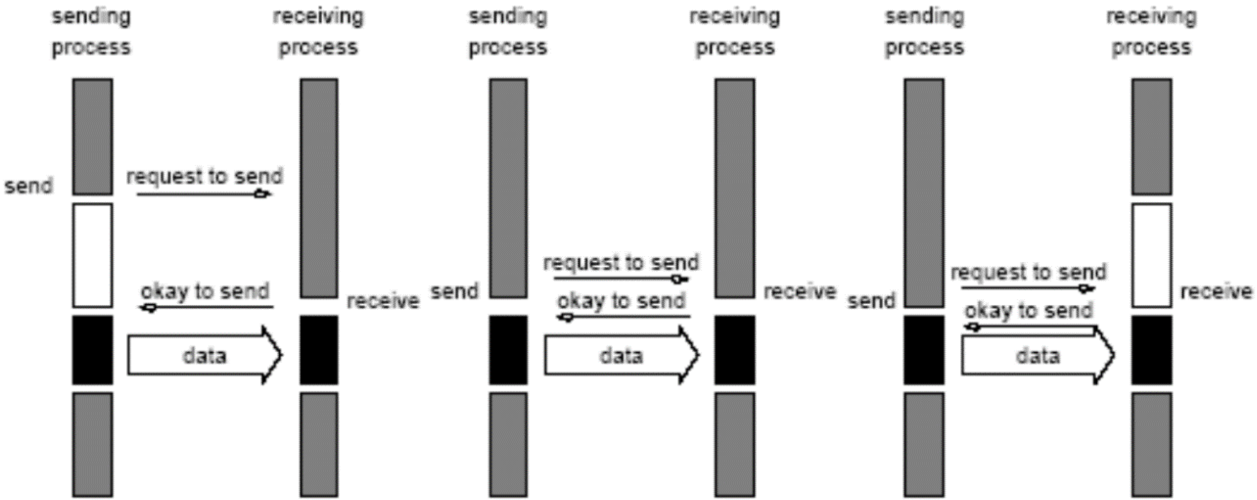
1. Client calls a remote procedure and waits.
2. Stub accepts the procedure call and serializes both the call and its parameters.
3. RPC Runtime … via TCP/UDP
4. Skeleton acc… the message … service proce…
5. Server handl…
6. Skeleton acc… serializes it.
7. RPC Runtime … via TCP/UDP
8. Stub accepts the re… …rializes it and forwards i… the waiting client.
9. Client awakes and accepts the result.



Remote Procedure Call (RPC)

Rendezvous protocol

- Handshake protocol for sending large blocks of data.
- Avoid sending data to processes that cannot accept them (at the moment).
- Before data is sent, the receiver needs to acknowledge that it is ready to accept data.



(a) Sender comes first; idling at sender (b) Sender and receiver come at about the same time; idling minimized (c) Receiver comes first; idling at receiver

Remote Procedure Call (RPC)

- RPC/RMI are protocols of which many framework implementations exist.

- RPC/RMI implementations can be language specific and language agnostic:
  - Language specific:
    interface is written in same language; often the programming language itself
  - Language agnostic:
    interface is written in some RPC/RMI dialect; compiles to different programming languages

- The RPC/RMI protocols propose blocking (= synchronous) communication
  but it is easy to turn the idea into non-blocking (= asynchronous) communication:
  - e.g. procedure calls may immediately return "Future" or "Promise" objects

- The concept of providing a communication interface in the programming language and hiding the communication protocol in a runtime is used by all messaging frameworks.
  - ➢ Understanding RPC/RMI provides a good understanding of any messaging system.

Remote Procedure Call (RPC)

Strengths of RPC/RMI

- RPC/RMI frameworks are well suited for machine to machine communication (remote calls appear like local calls; program does not leave its own language).
- RPC/RMI frameworks are easy to use (automatic code generation and abstraction of the messaging details).
- RPC/RMI frameworks are extensive (no restrictions other than those the interface language has).
- RPC/RMI frameworks offer good performance (highly optimized messaging, because the runtime controls both ends of the communication and no third party needs to understand the messages).

Remote Procedure Call (RPC)

Weaknesses of RPC/RMI

- RPC/RMI cause a tight coupling of server and client code.
  (interface changes always concern both)

- Local and remote function calls are, in fact, very different.
  - Local function calls are predictable: they succeed or fail, throw proper exceptions or starve processing; can handle same pointers and data types than caller
  - Remote function calls are unpredictable: they fail silently, succeed but responses get lost, are unavailable; cannot handle the caller's pointers (and all data types)

- RPC/RMI code may be hard to debug and test.
  (code generation; possibly hiding of network errors)

Good/modern RPC frameworks make differences explicit and forward errors transparently so that application code can (and should!) handle these issues.

Remote Procedure Call (RPC)

https://en.wikipedia.org/wiki/Remote_procedure_call

Language-specific [edit]

- Java's Java Remote Method Invocation (Java RMI) API provides similar functionality to standard Unix RPC methods.
- Modula-3's network objects, which were the basis for Java's RMI[10]
- RPyC implements RPC mechanisms in Python, with support for asynchronous calls.
- Distributed Ruby (DRb) allows Ruby programs to communicate with each other on the same machine or over a network. DRb uses remote method invocation (RMI) to pass commands and data between processes.
- Erlang is process oriented and natively supports distribution and RPCs via message passing between nodes and local processes alike.
- Elixir builds on top of the Erlang VM and allows process communication (Elixir/Erlang processes, not OS processes) of the same network out-of-the-box via Agents and message passing.

Application-specific [edit]

- Action Message Format (AMF) allows Adobe Flex applications to communicate with back-ends or other applications that support AMF.
- Remote Function Call is the standard SAP interface for communication between SAP systems. RFC calls a function to be executed in a remote system.

Actual RPC implementations

General [edit]

- NFS (Network File System) is one of the most prominent users of RPC
- Open Network Computing Remote Procedure Call, by Sun Microsystems
- D-Bus open source IPC program provides similar function to CORBA.
- SORCER provides the API and exertion-oriented language (EOL) for a federated method invocation
- XML-RPC is an RPC protocol that uses XML to encode its calls and HTTP as a transport mechanism.
- JSON-RPC is an RPC protocol that uses JSON-encoded messages
- JSON-WSP is an RPC protocol that uses JSON-encoded messages
- SOAP is a successor of XML-RPC and also uses XML to encode its HTTP-based calls.
- ZeroC's Internet Communications Engine (Ice) distributed computing platform.
- Etch framework for building network services.
- Apache Thrift protocol and framework.
- CORBA provides remote procedure invocation through an intermediate layer called the *object request broker*.
- Libevent provides a framework for creating RPC servers and clients.[11]
- Windows Communication Foundation is an application programming interface in the .NET framework for building connected, service-oriented applications.
- Microsoft .NET Remoting offers RPC facilities for distributed systems implemented on the Windows platform. It has been superseded by WCF.
- The Microsoft DCOM uses MSRPC which is based on DCE/RPC
- The Open Software Foundation DCE/RPC Distributed Computing Environment (also implemented by Microsoft).
- Google Protocol Buffers (protobufs) package includes an interface definition language used for its RPC protocols[12] open sourced in 2015 as gRPC.[13]
- Google Web Toolkit uses an asynchronous RPC to communicate to the server service.[14]
- Apache Avro provides RPC where client and server exchange schemas in the connection handshake and code generation is not required.
- Embedded RPC is lightweight RPC implementation developed by NXP, targeting primary CortexM cores

... Thrift-based

... Protocol Buffers-based

... Avro-based

Popular Service Protocols

HTTP

- Used by the largest SOA systems on the planet, e.g., the World Wide Web.

(HTTP) REST

- If you need clearer conventions for HTTP service APIs.
  (e.g. to make them easier to maintain and better machine consumable)
- Used by many Web applications to connect front- and backend systems.

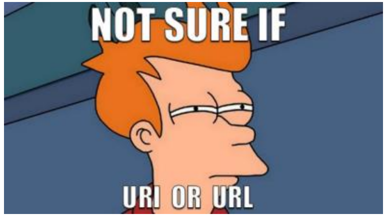
(HTTP + RPC) SOAP

- If you develop heterogeneous distributed systems that need to communicate not only data but also instructions (i.e., method calls).
- Used by many large scale, heterogeneous distributed systems.

Hypertext Transfer Protocol (HTTP)

Definition

- A stateless, synchronous request-response application protocol for distributed, collaborative, and hypermedia information systems
- The foundation for communication in the World Wide Web
- Hypertext: structured text that uses logical links (hyperlinks) between nodes containing text (usually HTML)



Technical Details

- Message format: designed for hypertext, but works for any text format
- Based on the TCP transport layer protocol
- Uniform Resource Locators (URLs) / Uniform Resource Identifier (URI) to find services and resources:

  scheme:[//[user[:password]@]host[:port]][/path]

  ➢ E.g.: http://hpi.de/naumann/people/thorsten-papenbrock

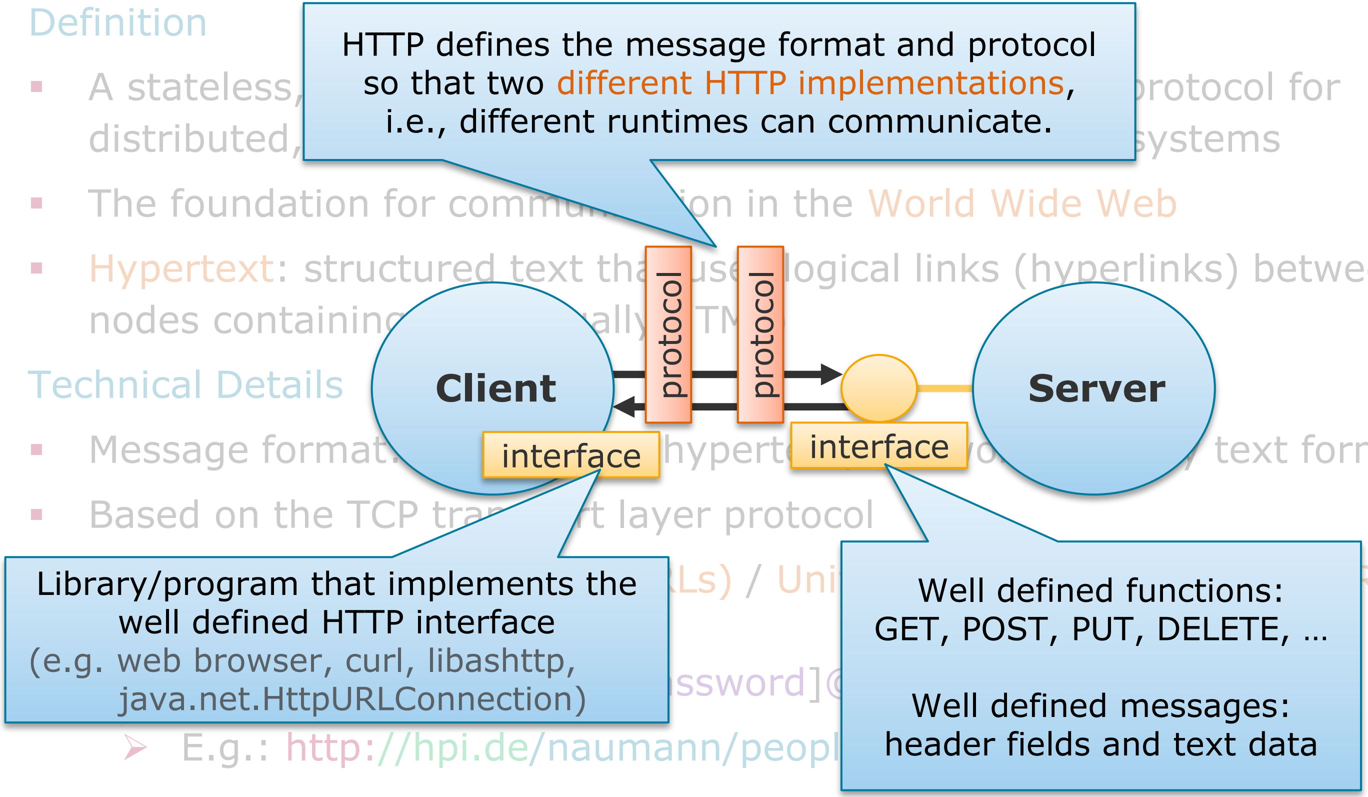
Hypertext Transfer Protocol (HTTP)

Definition

- A stateless, … protocol for distributed, … systems
- The foundation for commu…ion in the World Wide Web
- Hypertext: structured text tha… use… logical links (hyperlinks) between nodes containin… …ally …TM…

Technical Details

- Message format… hyperte… text format
- Based on the TCP tra…t layer protocol
- …Ls) / Uni… …RI)
- …ssword]@…
  - E.g.: http://hpi.de/naumann/peopl…



Hypertext Transfer Protocol (HTTP)

HTTPs

- HTTP over Transport Layer Security (TLS) / Secure Sockets Layer (SSL)
- Features:
  - Privacy through **symmetric encryption**
  - Authentication through **public-key cryptography**
  - Integrity through checking of **message authentication codes**

Session (HTTP/1.1)

- A sequence of network request-response transactions:
  1. Client establishes a TCP connection to server port (typically port 80).
  2. Client sends an HTTP message.
  3. Server sends back a status line with a message of its own.
  4. Client sends next HTTP message or closes the TCP connection.

Hypertext Transfer Protocol (HTTP)

Request Message Pattern

▪ A request-line:	<method> <resource identifier> <protocol version>
▪ Any header lines:	<header field>: <value>
▪ An empty line	
▪ A message-body:	<any text format> optional

Request Methods

- GET: Retrieve information from the target resource using a given URI (no side effects).
- HEAD: Like GET, but response contains only status line and header section (no content).
- POST: Send data to the target resource; the resource decides what to do with the data.
- PUT: Send data to the target resource; replace the content of the resource with that data.
- DELETE: Removes all content of the target resource.
- CONNECT: Establishes a tunnel to the server identified by a given URI.
- OPTIONS: Describe the communication options for the target resource.
- TRACE: Performs a message loop back test along with the path to the target resource.

Hypertext Transfer Protocol (HTTP)

Request Message Pattern

▪ A request-line:	<method> <resource identifier> <protocol version>
▪ Any header lines:	<header field>: <value>
▪ An empty line	
▪ A message-body:	<any text format> optional

Examples

- GET http://hpi.de/naumann/people/thorsten-papenbrock/publications HTTP/1.1
  → absolute URI: for requests to a proxy, which should forward the request
  → no additional header fields
- GET /naumann/people/thorsten-papenbrock/publications HTTP/1.1
  User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
  Host: www.hpi.de:80
  Accept-Language: en-us
  → relative URI: for request to origin server
  → some header fields as example

Hypertext Transfer Protocol (HTTP)

Request Message Pattern

- A request-line: <method> <resource identifier> <protocol version>
- Any header lines: <header field>: <value>
- An empty line
- A message-body: <any text format> optional

Examples

- POST /naumann/people/thorsten-papenbrock/publications HTTP/1.1
  Host: www.hpi.de:80
  Content-Type: text/xml; charset=utf-8
  Accept-Language: en-us
  Accept-Encoding: gzip, deflate
  Connection: Keep-Alive

  <publication>A Hybrid Approach to Functional Dependency Discovery</publication>
  → post a new publication entry to the publications resource (should be appended)
  → flags indicate utf-8 formatted xml content and ask to keep the connection open

PUT would replace all publi-cations with the new one

Hypertext Transfer Protocol (HTTP)

Response Message Pattern

▪ A status-line:	<protocol version> <status code> <reason-phrase>
▪ Any header lines:	<header field>: <value>
▪ An empty line	
▪ A message-body:	<any text format> optional

Status codes

- 1xx: Informational: the request was received and the process is continuing.
- 2xx: Success: the action was successfully received, understood, and accepted.
- 3xx: Redirection: further action must be taken in order to complete the request.
- 4xx: Client Error: the request contains incorrect syntax or cannot be fulfilled.
- 5xx: Server Error: the server failed to fulfill an apparently valid request.

Hypertext Transfer Protocol (HTTP)

Response Message Pattern

- A status-line: <protocol version> <status code> <reason-phrase>
- Any header lines: <header field>: <value>
- An empty line
- A message-body: <any text format> optional

Example

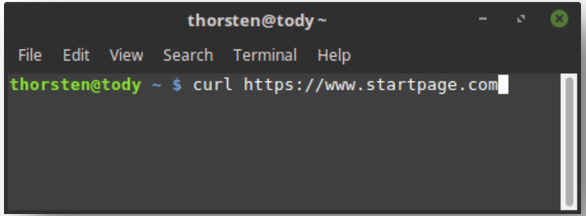
- GET http://www.my-host.com/my-new-homepage.html

```
HTTP/1.1 200 OK
Date: Mon, 24 Jul 2017 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Sat, 22 Jul 2017 13:15:56 GMT
Content-Length: 98
Content-Type: text/html
Connection: Closed

<html><body><h1>Welcome to my homepage!</h1></body></html>
```

Hypertext Transfer Protocol (HTTP)

The cURL Program

- Library and command-line tool for transferring data using various protocols
- Originally developed as "see url" in 1997
- Examples:
  - curl -i -X GET http://localhost:8080/datasets
  - curl -i -X GET http://localhost:8080/datasets/by/csv
  - curl -i -X POST -d '{"name":"Planets","ending":"csv","path":"datasets"}' -H 'Content-Type:application/json;charset=UTF-8‘ http://localhost:8080/datasets
  - curl -i -X DELETE http://localhost:8080/datasets/1
  - curl -i -X GET http://localhost:8080/datasets/1
  - curl -i -X PUT -d '{"name":"Planets","ending":"csv","path":"datasets"}' -H 'Content-Type:application/json;charset=UTF-8‘ http://localhost:8080/datasets/1



Representational State Transfer (REST)

- A design philosophy for HTTP services:
  - Resources are the main concept
  - CRUD (create, read, update, delete) operations on resources should use their corresponding HTTP methods
- Focus on simplicity

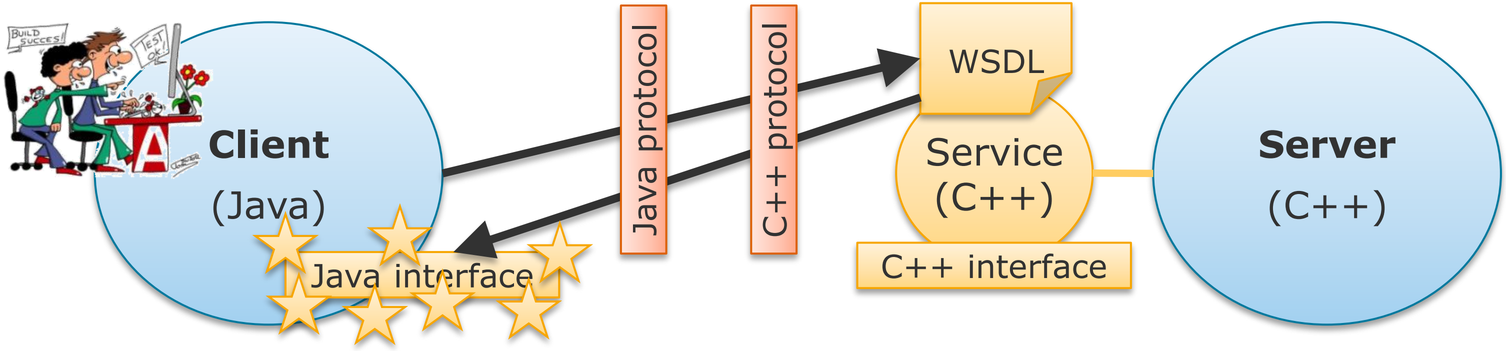
No method miss-use like
GET ...publications/?delete_id=42
which is typical for many HTTP services

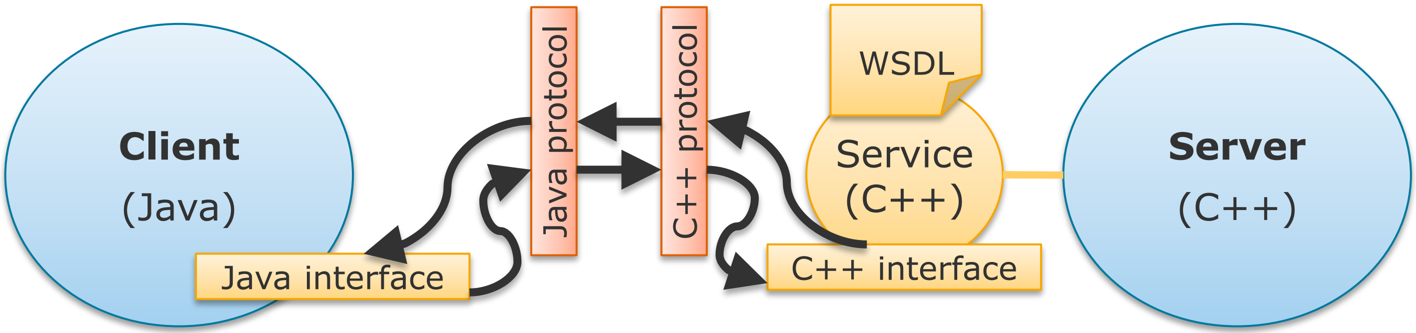
- OpenAPI Specification:
  - Creates the RESTful contract for your API.
  - RESTful contract describes all resources and their supported methods.
    - ➢ a language-agnostic interface description for the RESTful API
  - Implemented in, e.g., the Swagger framework (see https://swagger.io/)

Simple Object Access Protocol (SOAP)

- An XML-based RPC protocol for making network API requests
- Uses functions as main concepts (in contrast to resources in REST)
- Often implemented on top of HTTP but waiving most of its features
  - ➢ Comes with its own standards (the web service framework WS[…])
- Idea:
  - A server describes the API of its service in a WSDL document (Web Service Description Language; an XML dialect)
  - A client can use the WSDL document to generate the API code in its own programming language and then call the API functions
    - ➢ Both server and client can access the API in their own language
- Both programming languages and their IDEs must support SOAP for code and message generation
  - ➢ Interoperability without this support is difficult

Simple Object Access Protocol (SOAP)





Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP)

- Simple example:

```
<?xml version="1.0"?>
<definitions name="Booking">
  <message name="getBookingRequest">
    <part name="user" type="xs:string"/>
    <part name="house" type="xs:string"/>
  </message>
  <message name="getAvailabilityResponse">
    <part name="available" type="xs:boolean"/>
  </message>
  <portType name="BookingPort">
    <operation name="processBooking">
      <input message="getBookingRequest"/>
      <output message="getAvailability Response"/>
    </operation>
  </portType>
```

WSDL File

```
public interface BookingPort {
  public boolean processBooking(String user, String house);
}
```

A simple, language-agnostic interface definition

Simple Object Access Prot

Simple Object Access Protocol (SOAP)

- Simple example:

```
<?xml version="1.0"?>

<definitions name="Booking">

  <message name="getBookingRequest">
    <part name="user" type="xs:string"/>
    <part name="house" type="xs:string"/>
  </message>

  <message name="getAvailabilityResponse">
    <part name="available" type="xs:boolean"/>
  </message>

  <portType name="BookingPort">
    <operation name="processBooking">
      <input message="getBookingRequest"/>
      <output message="getAvailability Response"/>
    </operation>
  </portType>
```

WSDL File

Bundling of service calls to a SOAP service

Binding of an interface to concrete HTTP SOAP calls

```
<binding name="BookingBinding" type="BookingPort">
   <soap:binding
        style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="processBooking">
     <soap:operation
          soapAction="http://example.com/processBooking "/>
     <input>
       <soap:body use="literal"/>
     </input>
     <output>
       <soap:body use="literal"/>
     </output>
   </operation>
 </binding>

 <service name="BookingService">
   <documentation>A SOAP booking service</documentation>
   <port
        name="BookingPort"
        binding="BookingBinding">
     <soap:address location="http://example.com/booking"/>
   </port>
 </service>

</definitions>
```

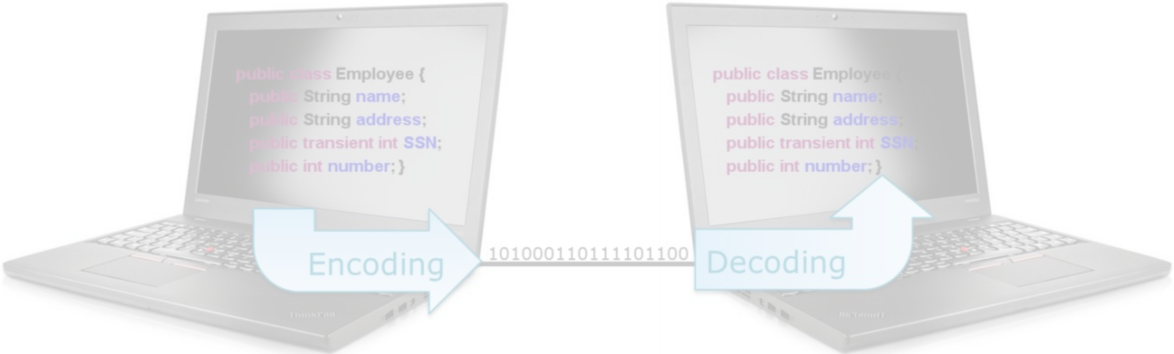
WSDL File (cont.)

Overview

Encoding and Communication

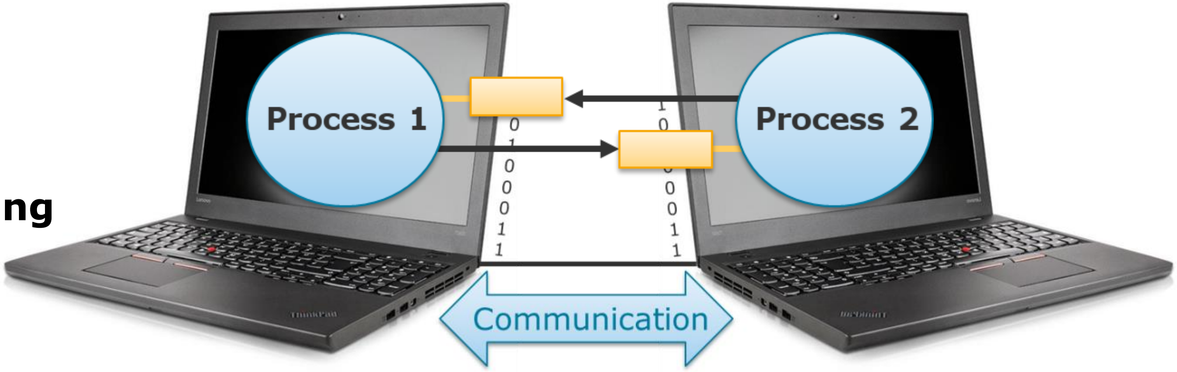
Encoding

- Language-Specific Encoding
- JSON/XML Encoding
- Binary Encoding

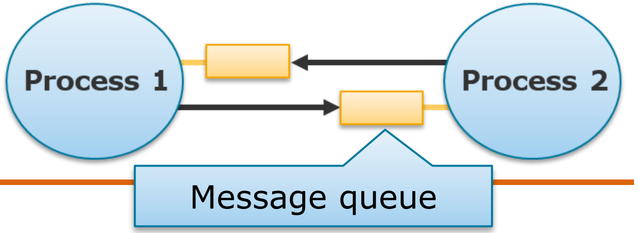


Communication

- Dataflow via Databases
- Dataflow via Services
- **Dataflow via Message-Passing**



Communication Principle



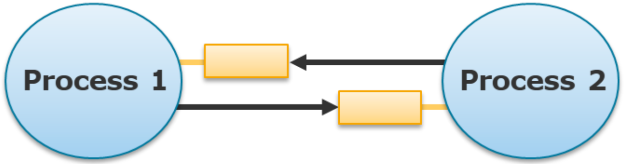
Communication

- An object-oriented paradigm that models all communication between objects (in different threads or processes) via asynchronous exchange of messages
  - ➢ Objects send messages to other objects via queues
- Message (also known as "mail"):
  - Container for data that implies information or commands
  - Often carries metadata, e.g., sender and receiver information
- The recipient decides how and if it handles a certain message
- Message queue (also known as "mailbox"):
  - Data structure (queue or list) assigned to communicating object(s)
  - Buffers incoming messages for being processed
  - Messages in the queue are (usually) ordered by time of arrival
- Messages in a queue are processed successively in their order

Messages can have any format understood by sender and receiver

to enable replies

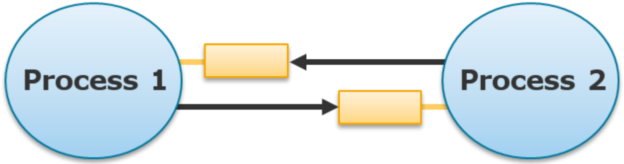
Communication Principle



Principles

- Encapsulation
  - Communicating objects have private state and private behavior.
  - Still objects communicate, i.e., cause other objects to react on their messages.
  - Communicate “what” is to be done not “how”.
- Distribution
  - Messages can pass through busses, channels, networks, …
  - Message-passing system resolves addresses and automatically routes messages from senders to receivers.
  - Allows objects to be transparently distributed, i.e., objects must not know where their communication partners actually are.

Communication Principle

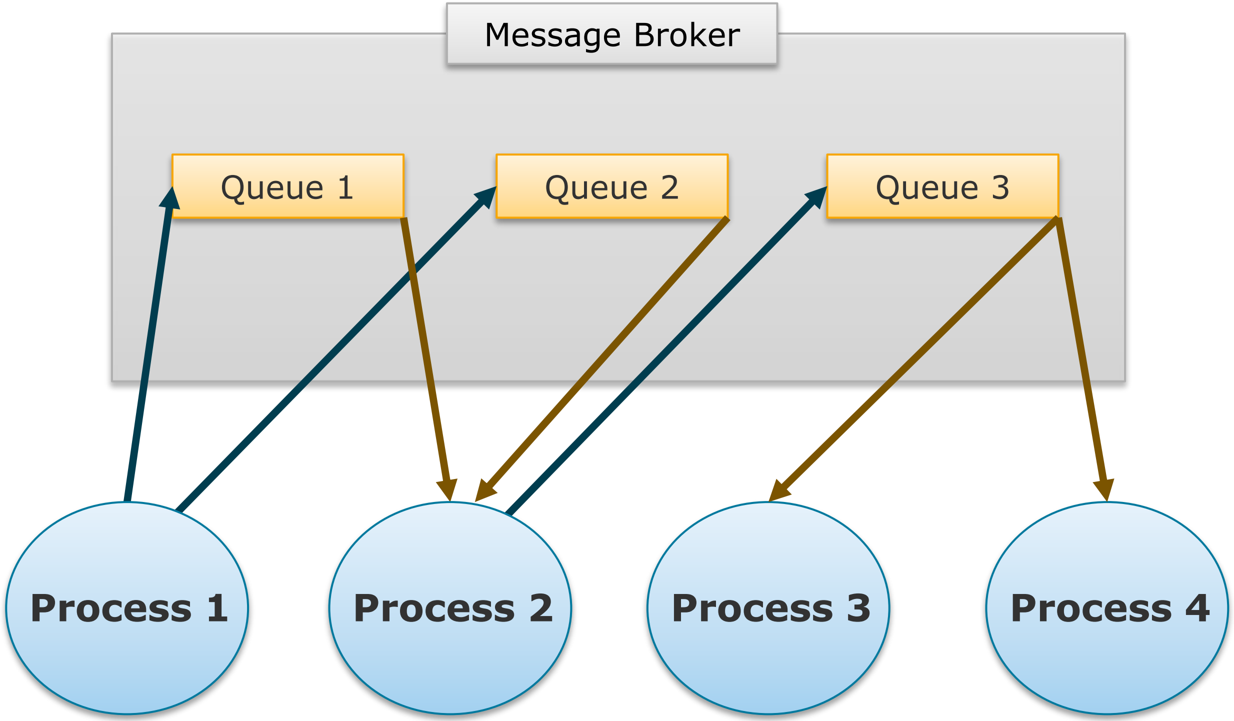


Principles

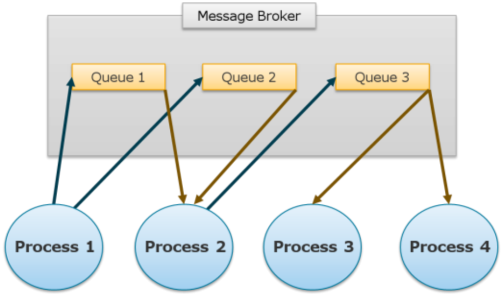
- Asynchronicity
  - Messaging is asynchronous, i.e., the sender does not wait for a reply.
  - Synchronous communication can be implemented on top of asynchronous messaging with certain protocols.
  - Enables reactive programming:
    - Instead of following a fixed calculation plan, the algorithm dynamically reacts to changes in the data.
    - Reaction = changing state (= variables) or behavior (= code)
    - Writing a reactive algorithm is more like declaring rules for how to react on certain input changes rather than defining a step-by-step execution plan.
    - Reactivity helps to optimize runtime, robustness and elasticity.

Dataflow via Message-Passing

Message Broker



Message Broker



Message Broker

- Also called message queue or message-oriented middleware
- Part of the message-passing framework that delivers messages from their senders to the appropriate queues and, finally, to the receiver(s).
- Resolves sender an receiver addresses (objects must not know ports/IPs).
- Can apply binary encoding on messages when delivered between processes.
- Routing:
  - One-to-one messages
  - One-to-many messages (broadcasting)
- Advantages:
  - Decouples sender and receiver objects (maintainability)
  - Buffers messages if receiver is unavailable or overloaded (reliability)
  - Redirects messages if receiver crashed (robustness)

Communication Principle

General message delivery

- Processes can …
  - create named message queues.
  - subscribe to existing message queues.
  - send messages to a queue.
- The message broker assures that send messages are delivered to some/all subscriber of a queue.

Message-passing frameworks

- Commercial:
  - TIBICO, IBM WebSphere, webMethods, …
- Open source:
  - Apache Kafka, RabbitMQ, ActiveMQ, HornetQ, NATS, …

Examples

Now

- RabbitMQ
- MPI

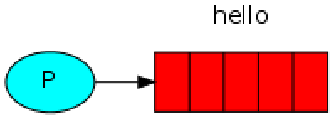
Later

- Kafka
- Akka
- Spark
- Flink

Dataflow via Message-Passing

Example: RabbitMQ – Sending a Message

https://www.rabbitmq.com/getstarted.html




```java
public class Send {

  private final static String QUEUE_NAME = "hello";

  public static void main(String[] argv) throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    String message = "Hello World!";
    channel.basicPublish("", QUEUE_NAME, null, message.getBytes("UTF-8"));
    System.out.println(" [x] Sent '" + message + "'");

    channel.close();
    connection.close();
  }
}
```

Create a connection to the message broker running on localhost.

Create a channel to a queue; the queue is created if it does not exist yet.

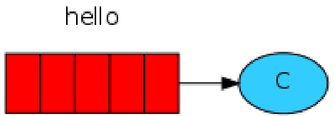
Send the message encoded as an array of bytes.

Close all channels and the connection.

Dataflow via Message-Passing

Example: RabbitMQ – Receiving a Message

https://www.rabbitmq.com/getstarted.html




```java
public class Recv {

  private final static String QUEUE_NAME = "hello";

  public static void main(String[] argv) throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

    Consumer consumer = new DefaultConsumer(channel) {
      @Override
      public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body)
          throws IOException {
        String message = new String(body, "UTF-8");
        System.out.println(" [x] Received '" + message + "'");
      }
    };
    channel.basicConsume(QUEUE_NAME, true, consumer);
  }
}
```

Create a connection to the message broker running on localhost.

Create a channel to a queue; the queue is created if it does not exist yet.

Create a callback object that can buffer and consume messages from a queue.

Decode any received byte message.

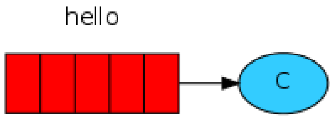
Subscribe the new consumer to the queue; the broker will call it with messages of that queue.

Dataflow via Message-Passing

Example: RabbitMQ – Receiving a Message

```java
public class Recv {

  private final static String QUEUE_NAME = "hello";

  public static void main(String[] argv) throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

    Consumer consumer = new DefaultConsumer(channel) {
      @Override
      public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body)
          throws IOException {
        String message = new String(body, "UTF-8");
        System.out.println(" [x] Received '" + message
      }
    };
    channel.basicConsume(QUEUE_NAME, true, consumer);
  }
}
```

https://www.rabbitmq.com/getstarted.html



This is metadata that pure RPCs are lacking:

- Encoding, timestamp, sender, priority, … of the message

Dataflow via Message-Passing

Example: RabbitMQ – Example in Python

```python
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(
        host='localhost'))
channel = connection.channel()


channel.queue_declare(queue='hello')

channel.basic_publish(exchange='',
                      routing_key='hello',
                      body='Hello World!')
print(" [x] Sent 'Hello World!'")
connection.close()
```


```python
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(
        host='localhost'))
channel = connection.channel()


channel.queue_declare(queue='hello')

def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)

channel.basic_consume(callback,
                      queue='hello',
                      no_ack=True)

print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

Further APIs:

Ruby, PHP, C#, JavaScript, Go, Elixir, Objective-C, Swift, …

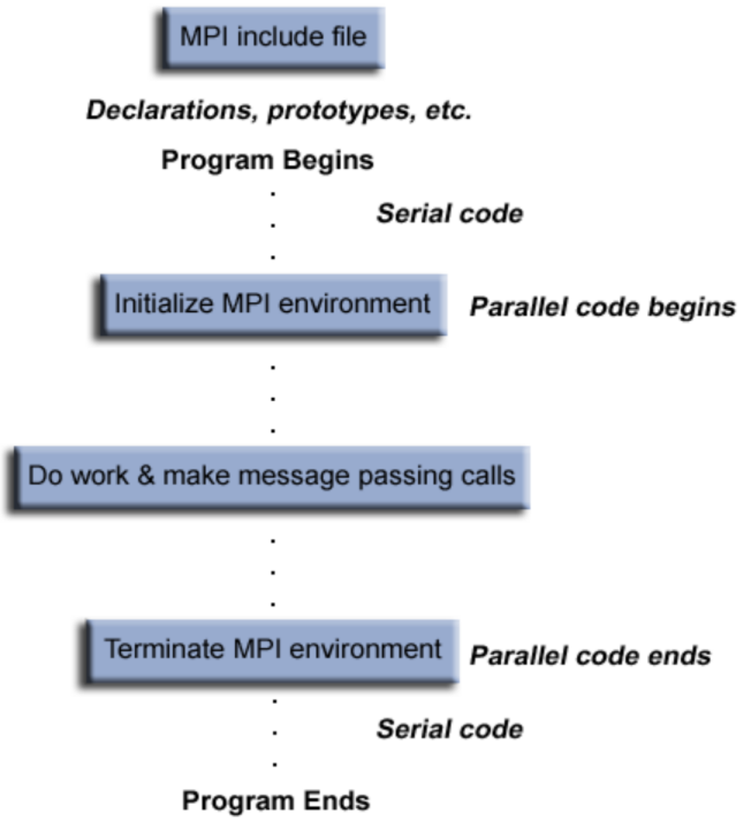
https://www.rabbitmq.com/getstarted.html

Example: MPI

- Message Passing Interface
- A specification for a family of message-passing libraries
  (MVAPICH, Open MPI, Intel MPI, IBM Spectrum MPI, …)
- Popular implementations for C, C++ and Fortran
- Platform dependent (message broker is usually Linux-based)
- Strength:
  - Performance
    (highly optimized messaging; can exploit special hardware features)
- Weakness:
  - Complexity
    (many low-level messaging functions; developer needs to ensure correct parallelism)

https://computing.llnl.gov/tutorials/mpi/

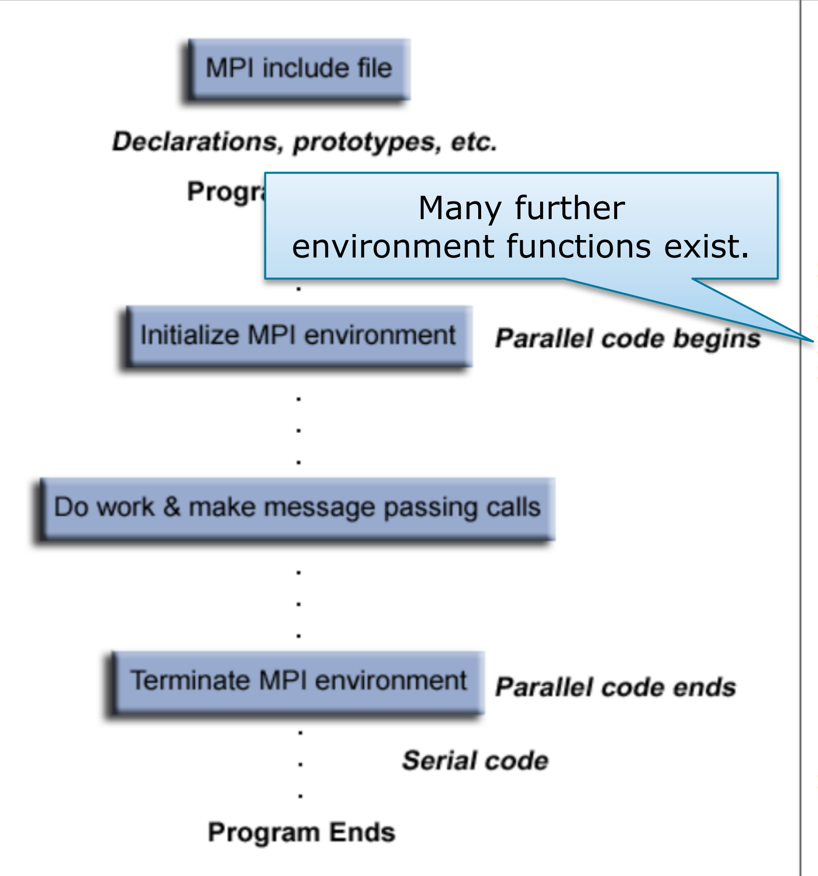
Example: MPI – Send and Receive




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

int main (int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

https://computing.llnl.gov/tutorials/mpi/

Example: MPI – Send and Receive



Many further environment functions exist.

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

int main (int argc, char *argv[])
{
int numtasks, rank, dest, s
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

= all known processes of the cluster

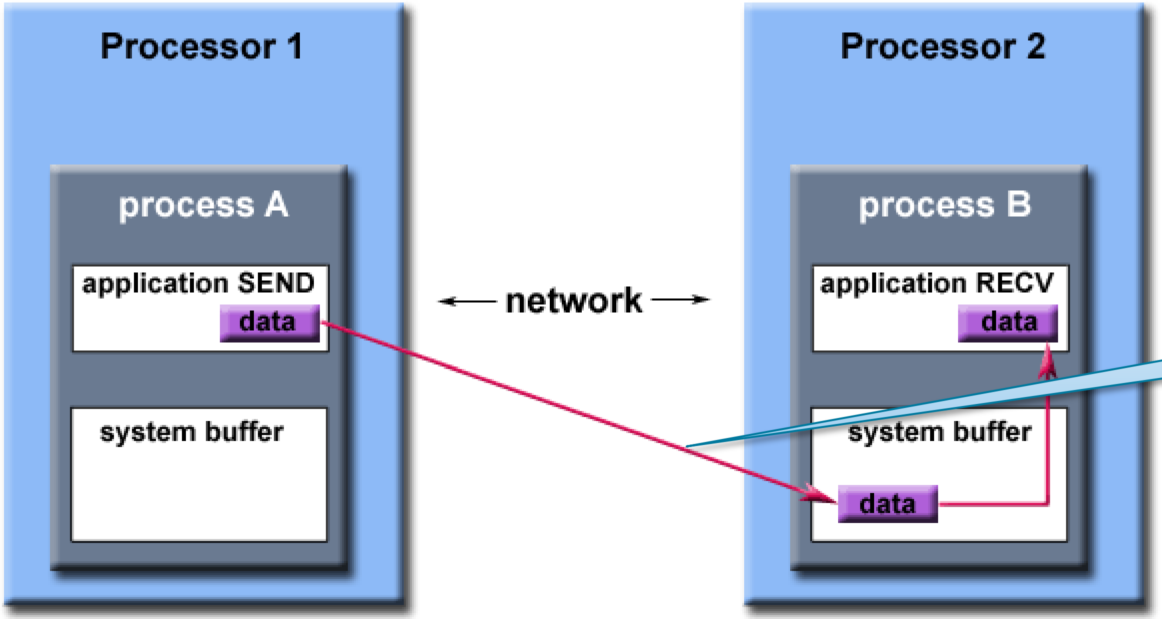
= our process ID in the cluster

If this is process 0, first send and then receive a message.

If this is process 1, first receive and then send a message.

https://computing.llnl.gov/tutorials/mpi/

Example: MPI – Send and Receive



MPI library = message broker
MPI system buffer = message queue

MPI_Send() → blocking
MPI_ISend() → non-blocking

Path of a message buffered at the receiving process

https://computing.llnl.gov/tutorials/mpi/

Example: MPI – blocking & synchronous

In this terminology,
RabbitMQ, Kafka, Akka and other
JVM-based message broker are
non-blocking + asynchronous
(messages should not be modified
but arrived in the message broker)

Blocking

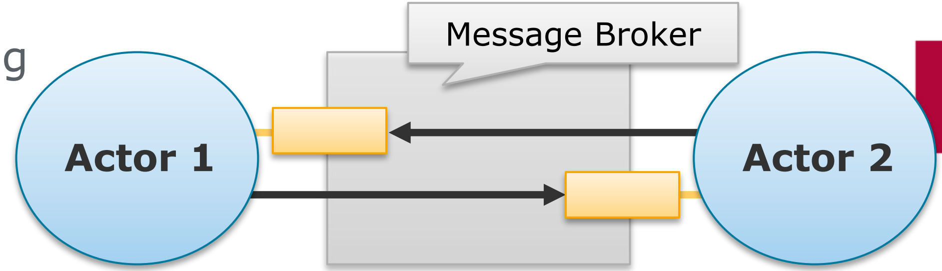
- Send() call returns if the data was send.
- The message in the send buffer can safely be modified.
- Synchronous
  - The receiving side acknowledged having received the data.
- Asynchronous
  - The system buffer acknowledged having received the data
    (the system buffer copied the data and will make sure it gets send).

Non-Blocking

- Send() call returns immediately.
- The message in the send buffer should not be modified.

https://computing.llnl.gov/tutorials/mpi/

Actor Model

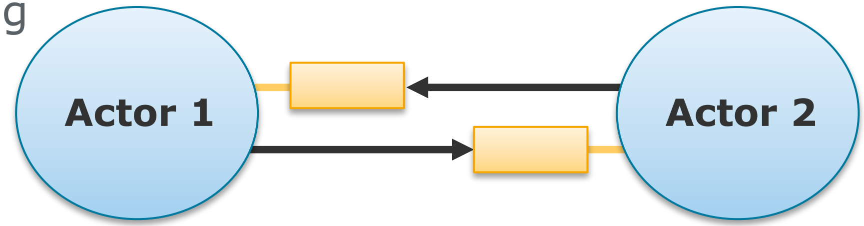


Actor Model

- A stricter message-passing model that treats actors as the universal primitives of concurrent computation.
- Actor:
  - Computational entity (private state/behavior)
  - Owns exactly one mailbox (cannot subscribe to more or less queues)
  - Reacts on messages it receives (one message at a time)
- Actor reactions:
  - Send a finite number of messages to other actors
  - Create a finite number of new actors
  - Change own state, i.e., behavior for next message
- Actor model prevents many parallel programming issues (race conditions, locking, deadlocks, …)

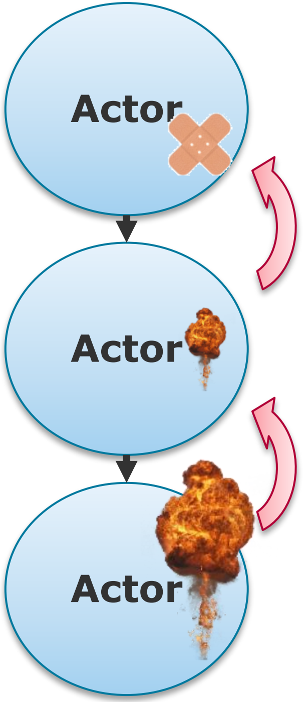
"The actor model retained more of what I thought were good features of the object idea"
Alan Kay, pioneer of object orientation

Actor Model

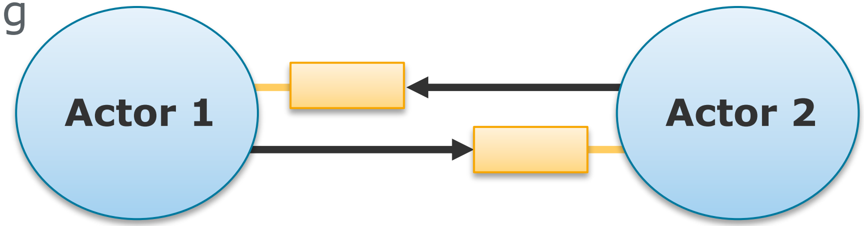


"Let it crash" philosophy

- Distributed systems are inherently prone to errors (because there is simply more to go wrong/break)
  - Message loss, unreachable mailboxes, crashing actors …
- Make sure that critical code is supervised by some entity that knows how errors can be handled
- Then, if an error occurs, do not (desperately) try to fix it: let it crash!
  - Errors are propagated to supervisors that can deal better with them
- Example: Actor discovers a parsing error and crashes
  - Maybe message was corrupted in message transfer
    - Its supervisor restarts the actor and resends the message



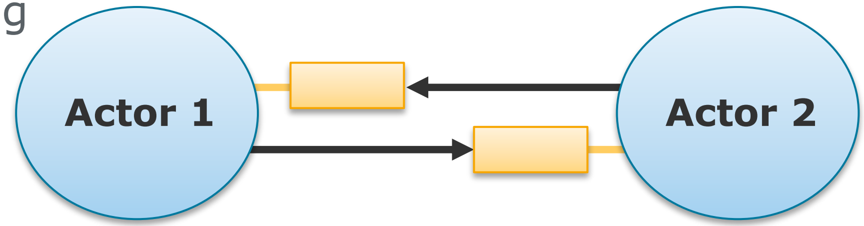
Actor Model



Advantages over pure RPC

- Fault-tolerance
  - "Let it crash!" philosophy to heal from unexpected errors
  - Automatic restart of failed actors; resent/re-route of failed messages
    - ➢ Errors are expected to happen and implemented into the model:
- Deadlock/starvation prevention
  - Asynchronous messaging and private state actors prevent many parallelization issues
- Parallelization
  - Actors process one message at a time but different actors operate independently (parallelization between actors not within an actor)
  - Actors may spawn new actors if needed (dynamic parallelization)

Actor Model



Popular Actor Frameworks

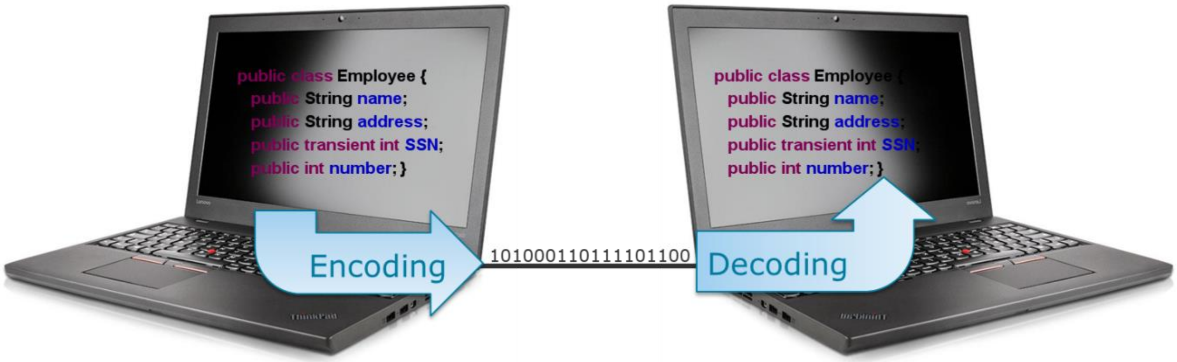
- Erlang
  - Actor framework already included in the language
  - First popular actor implementation
  - Most consistent actor implementation
    (best native support and strongest actor isolation)
- Akka
  - Actor framework for the JVM (Java and Scala)
  - Most popular actor implementation (at the moment)
- Orleans
  - Actor framework for Microsoft .NET
  - E.g. Halo (PC game)

A lot more on Akka in our upcoming hands-on!

Summary

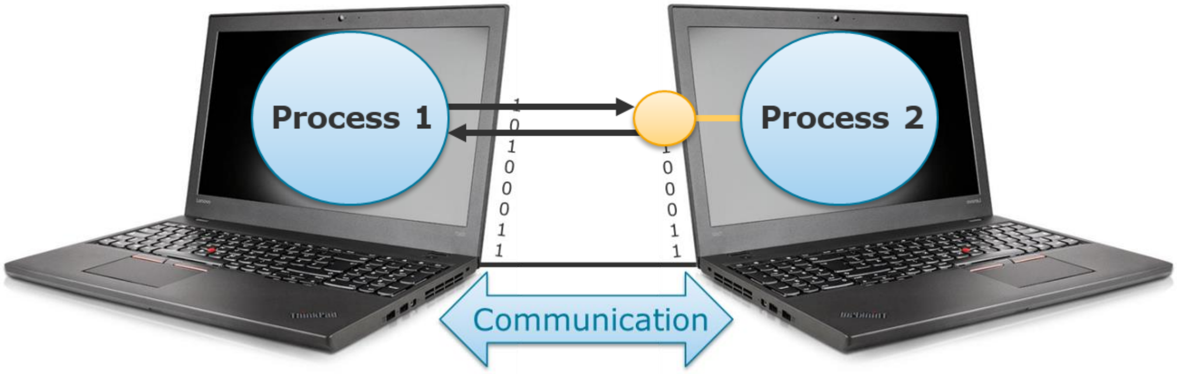
Encoding

- Language-Specific Encoding
- JSON/XML Encoding
- Binary Encoding



Communication

- Dataflow via Databases
- Dataflow via Services
- Dataflow via Message-Passing



Summary

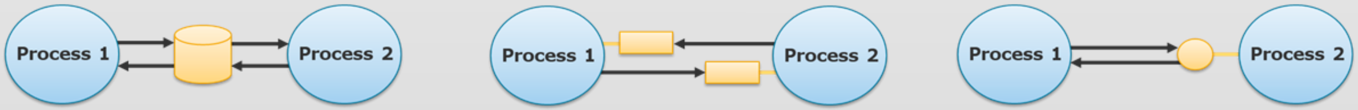
Encodin

- Lang
- JSON
- Binar

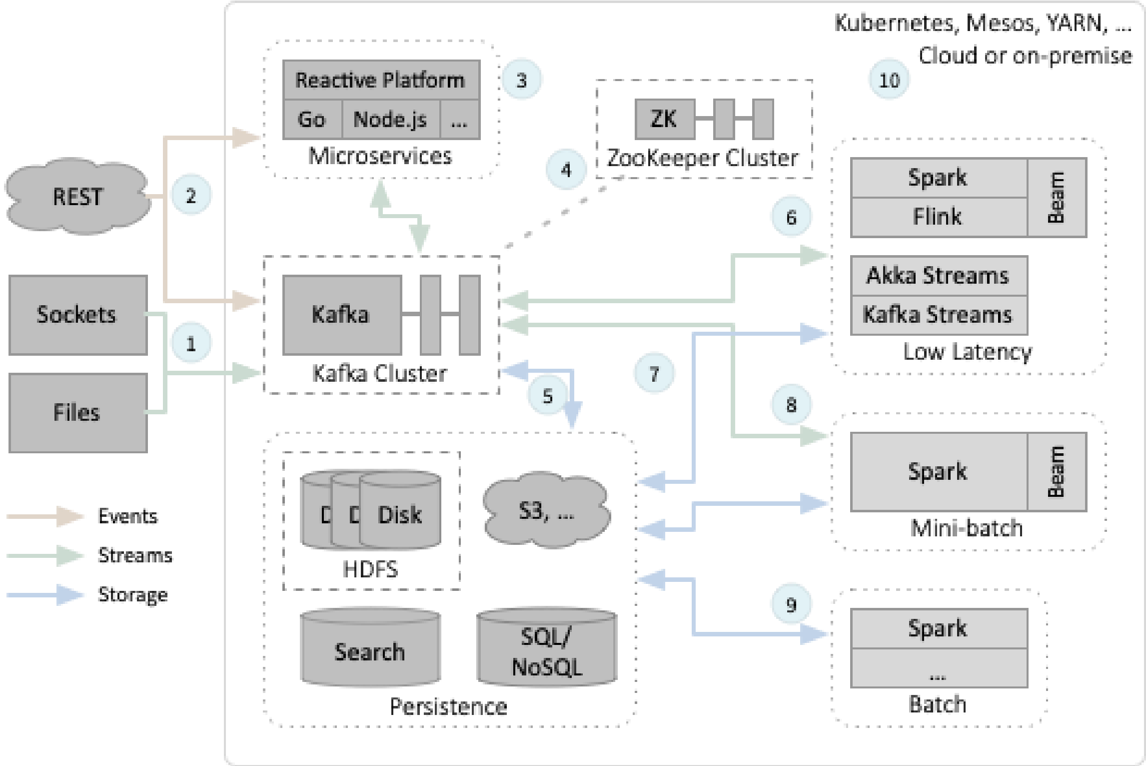
Commu

- Data
- Data
- Data

Databases	Message-Passing	Services
Data	Messages	Function calls
No response	Maybe response	Response
Non-blocking	Usually non-blocking	Blocking
Asynchronous	Asynchronous	Synchronous
No addressing	Addressing recipient	Addressing recipient



Outlook



https://www.lightbend.com/blog/cloud-native-streaming-data-with-akka-streams-kafka-steams

Check yourself

Suppose we have a linked list that is implemented as shown in the following code snippet:

```
public class IntLinkedList {
    int size;
    IntNode first;
    IntNode last;
    …

    private static class IntNode {
        int item;
        IntNode next;
        IntNode prev;
    }
    …
}
```

Question 1:

Give reasons why the default Java serializer should not be used here.

Question 2:

How would a more reasonable serialization look like?

Chapter 4. Encoding and Evolution