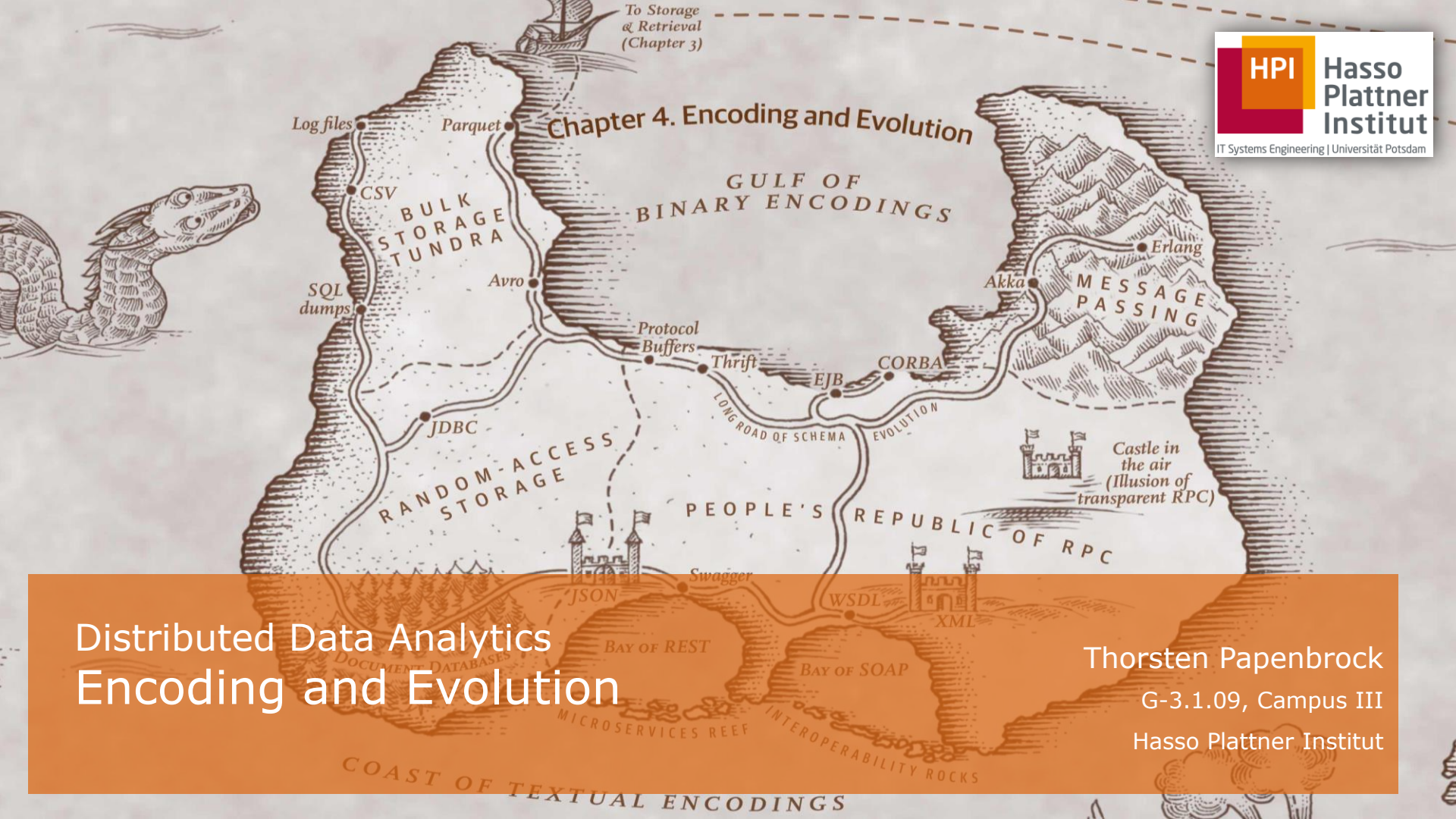


HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Distributed Data Analytics
Encoding and Evolution

Thorsten Papenbrock
G-3.1.09, Campus III
Hasso Plattner Institut

Layering Data Models

1. Conceptual layer
   - Data structures, objects, modules, …
     - Application code
2. Logical layer

   our focus now
   - 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 Evolution

Formats for Encoding Data

- Language-Specific Formats
- JSON, XML, and Binary Variants
- Thrift and Protocol Buffers
- Avro

Models of Dataflow

- Dataflow Through Databases
- Dataflow Through Services
- Message-Passing Dataflow

Overview

Encoding and Evolution

Formats for Encoding Data

- Language-Specific Formats
- JSON, XML, and Binary Variants
- Thrift and Protocol Buffers
- Avro

Models of Dataflow

- Dataflow Through Databases
- Dataflow Through Services
- Message-Passing Dataflow

Two Different Representations

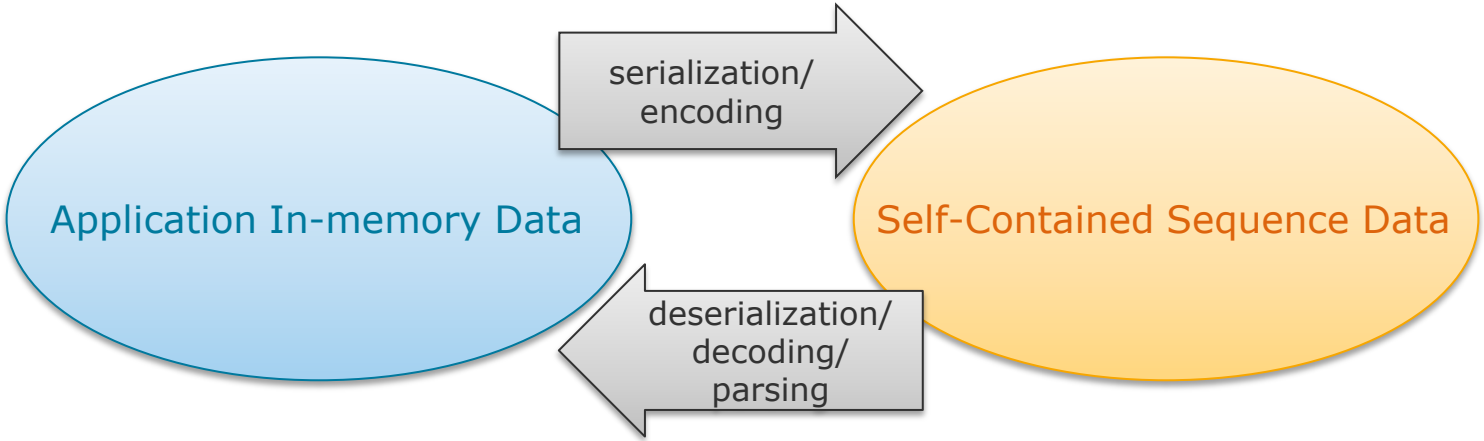
Application In-memory Data

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

Self-Contained Sequence Data

- Standardized encoding formats
- Byte sequences: JSON, XML, Protocol Buffers, Avro, …
- Optimized for disk persistence, network transmission, inter-process communication, …

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

Overview

Encoding and Evolution

Formats for Encoding Data

- **Language-Specific Formats**
- JSON, XML, and Binary Variants
- Thrift and Protocol Buffers
- Avro

Models of Dataflow

- Dataflow Through Databases
- Dataflow Through Services
- Message-Passing Dataflow

Serialization/Encoding

Language-specific serialization formats

- Goal: convert in-memory data into 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

Haxadecimal 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

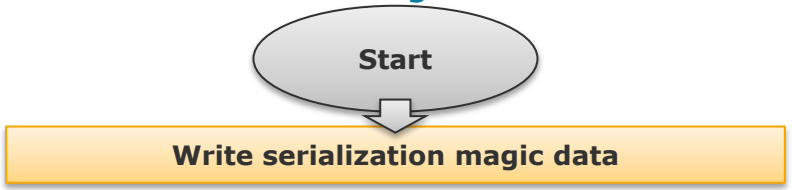
51 bytes

Why?

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

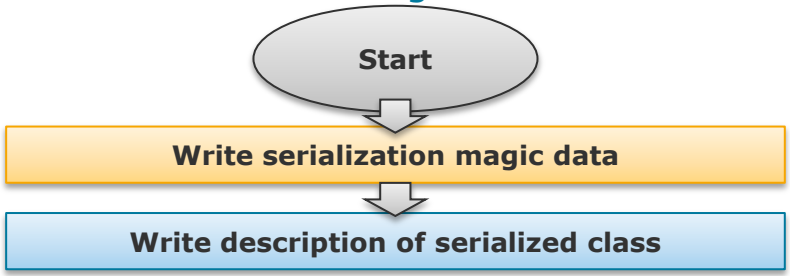
Serialization magic data specifies …

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:




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

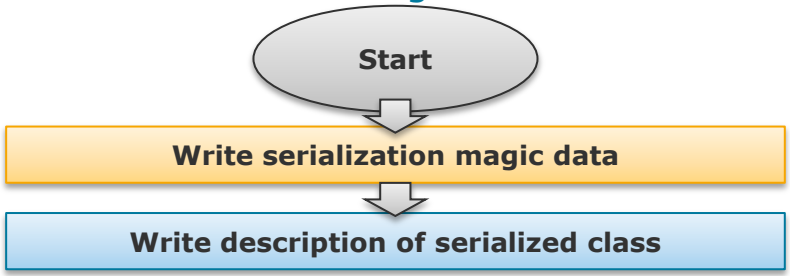
AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

Description of serialized class specifies …
1. the beginning of a new class (0x72)
2. the length of the class name (00 0A)
3. the name of the class (53 […] 74)
4. the serial version identifier (A0 […] F9)
5. various flags (e.g. 0x02 = serialization support)
6. the 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;
}
```

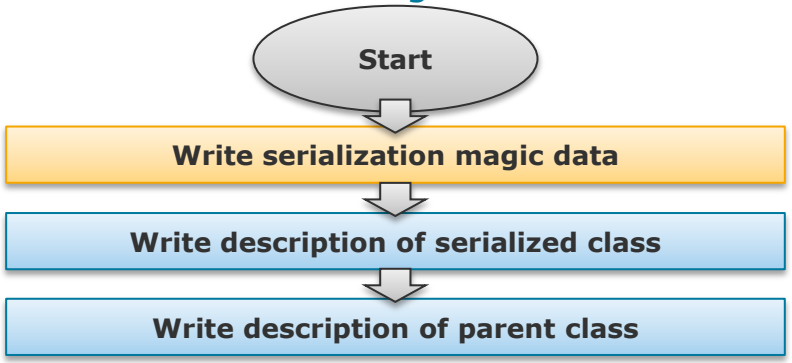
AC ED 00 05 73 72 00 0A 53 65 72 69 61 6C 54 65 73 74 A0 0C 34 00 FE B1 DD F9 02 00 02 42 00 05 63 6F 75 6E 74 42 00 07 76 65 72 73 69 6F 6E 78 70 00 64

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

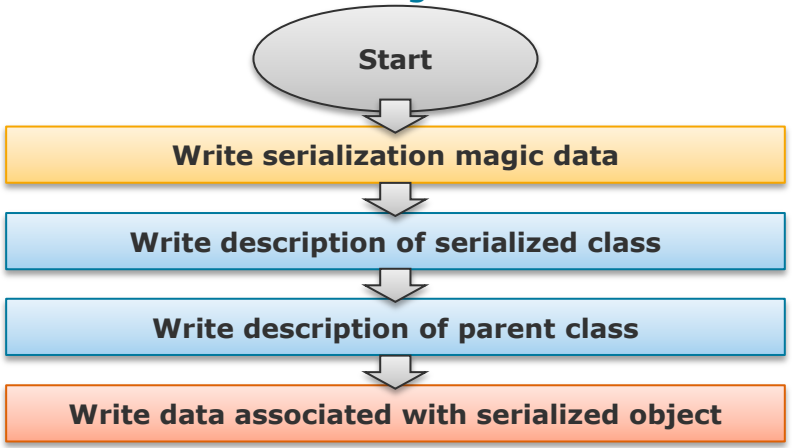
Description of parent class specification:

- 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 …

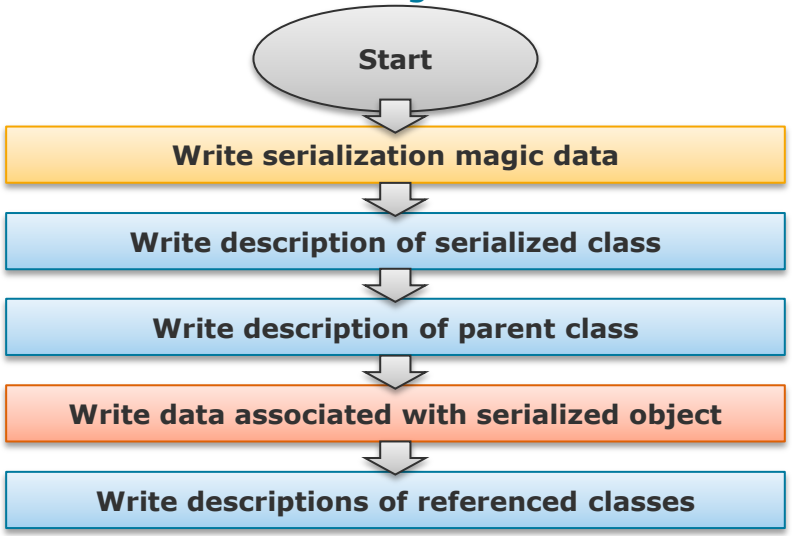
1. The first value (78 70 which is 100 for “version”)
2. The second value (00 64 which is 0 for “count”)

- 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:




```
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

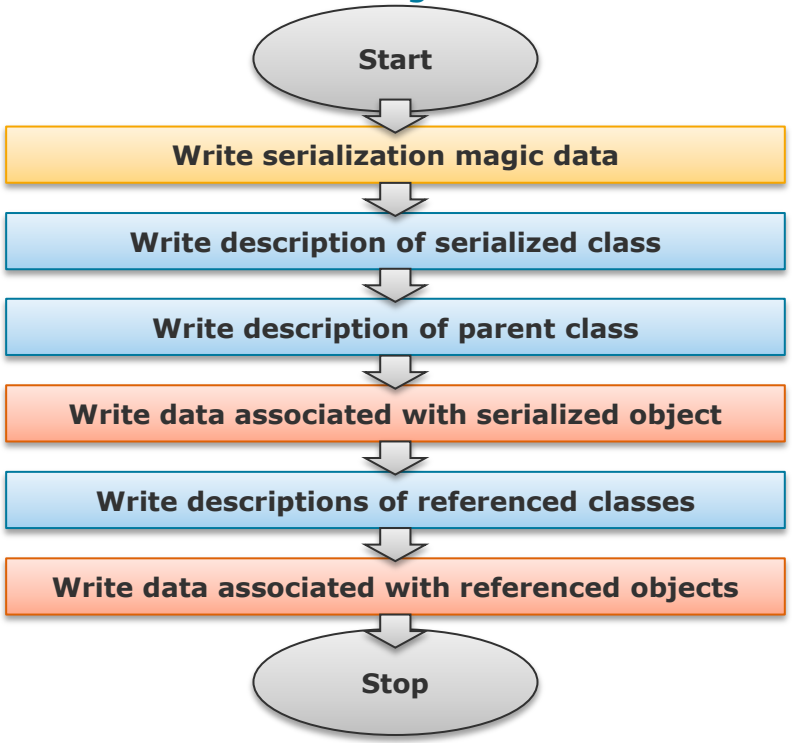
Description of referenced classes specification:

- 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:




```
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

Overview

Encoding and Evolution

Formats for Encoding Data

- Language-Specific Formats
- **JSON, XML, and Binary Variants**
- Thrift and Protocol Buffers
- Avro

Models of Dataflow

- Dataflow Through Databases
- Dataflow Through Services
- Message-Passing Dataflow

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

```
<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>
```

XML

Extensible Markup Language

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

JSON

JavaScript Object Notation

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

SDL

Simple Declarative Language

```
[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

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
  - Without explicit schema definition applications must define schemata

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

Binary Encoding

Problem

- Unicode formats are large

Idea

- Encode Unicode formats into binary strings to reduce their size
- Keep the original structure (attribute names, nesting, …)

Binary Encodings

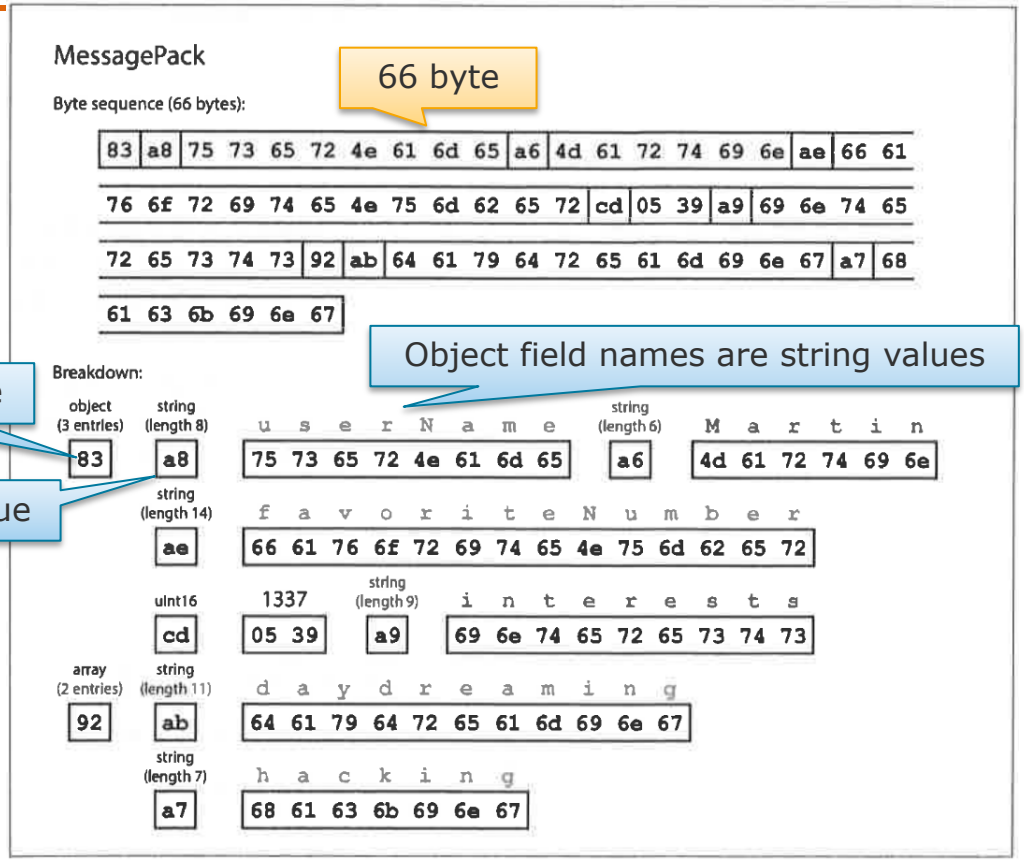
- For JSON: MassagePack, BSON, BJSON, UBJSON, BISON, Smile, …
- For XML: WBXML, Fast Infoset, …

MessagePack

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

JSON Format

81 byte



Overview

Encoding and Evolution

Formats for Encoding Data

- Language-Specific Formats
- JSON, XML, and Binary Variants
- **Thrift and Protocol Buffers**
- Avro

Models of Dataflow

- Dataflow Through Databases
- Dataflow Through Services
- Message-Passing Dataflow

Schema-based Binary Encoding

Motivation

- Binary JSON/XML encodings store 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)
- Protocol Buffers (by Google)

} both open source since 2007

- …

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

Backward compatibility

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

Thrift BinaryProtocol

59 byte

Byte sequence (59 bytes):

```
0b 00 01 00 00 00 06 4d 61 72 74 69 6e 0a 00 02 00 00 00 00
00 00 05 39 0f 00 03 0b 00 00 00 02 00 00 00 0b 64 61 79 64
72 65 61 6d 69 6e 67 00 00 00 07 68 61 63 6b 69 6e 67 00
```

Breakdown:

type 11 (string)	field tag = 1	length 6	M a r t i n
0b	00 01	00 00 00 06	4d 61 72 74 69 6e
type 10 (i64)	field tag = 2	1337	
0a	00 02	00 00 00 00 00 00 05 39	
type 15 (list)	field tag = 3	Item type 11 (string)	2 list items
0f	00 03	0b	00 00 00 02
		length 11	d a y d r e a m i n g
		00 00 00 0b	64 61 79 64 72 65 61 6d 69 6e 67
		length 7	h a c k i n g
		00 00 00 07	68 61 63 6b 69 6e 67
			end of struct
			00

Very similar to MessagePack but without field names

Thrift with CompactProtocol

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

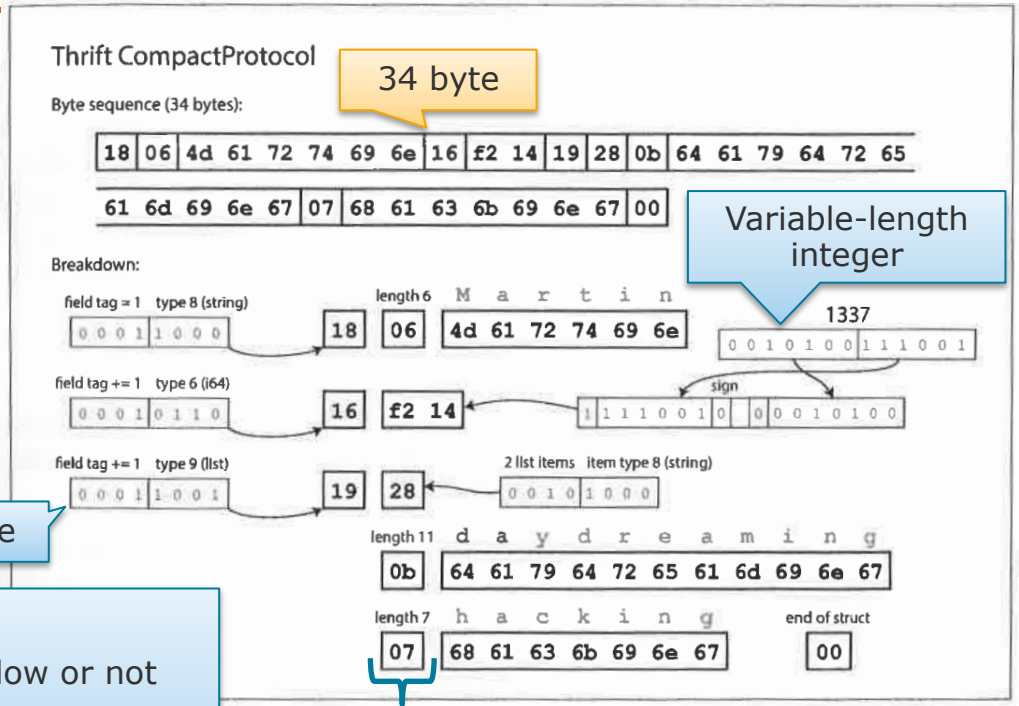
JSON Format

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

- 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

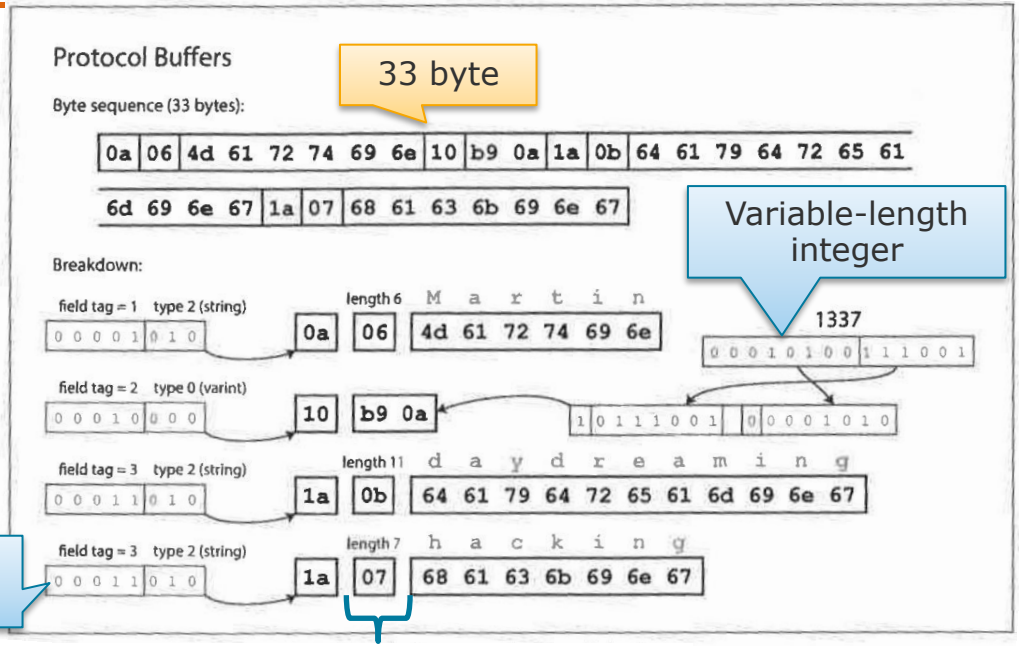
81 byte

```
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

Overview

Encoding and Evolution

Formats for Encoding Data

- Language-Specific Formats
- JSON, XML, and Binary Variants
- Thrift and Protocol Buffers
- **Avro**

Models of Dataflow

- Dataflow Through Databases
- Dataflow Through Services
- Message-Passing Dataflow

Avro

Apache Avro

- A binary encoding format developed as a sub-project of Hadoop in 2009
- 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"]
}
```

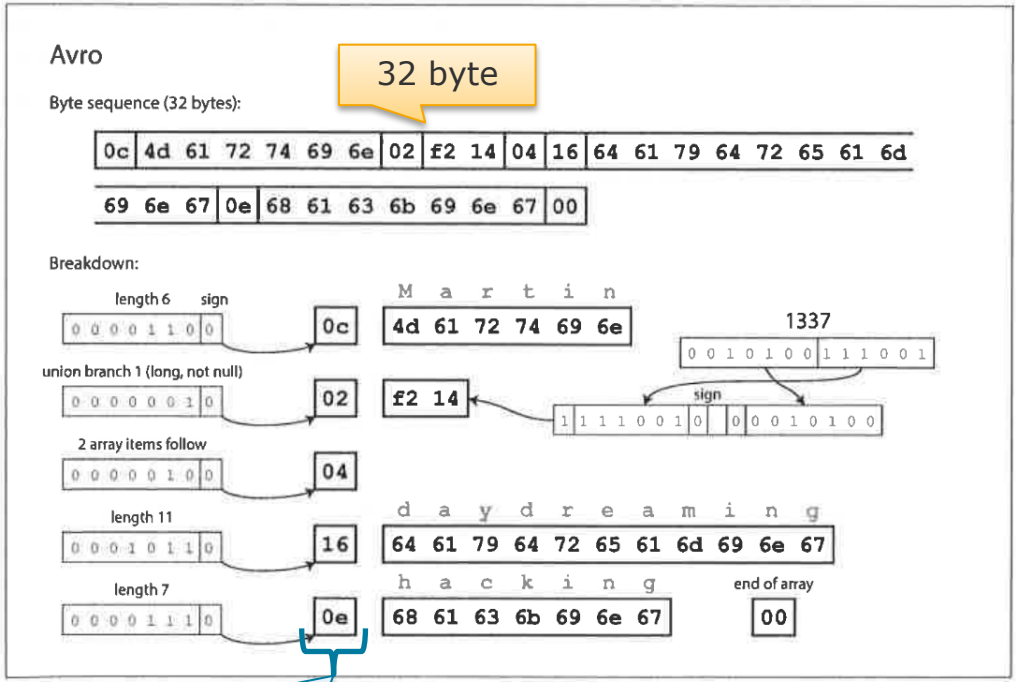
JSON Format

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!

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)
  - 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

"self-describing data"

Avro

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 Evolution

Formats for Encoding Data

- Language-Specific Formats
- JSON, XML, and Binary Variants
- Thrift and Protocol Buffers
- Avro

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

"Internet Security" by Prof. Meinel

Models of Dataflow

- Dataflow Through Databases
- Dataflow Through Services
- Message-Passing Dataflow

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

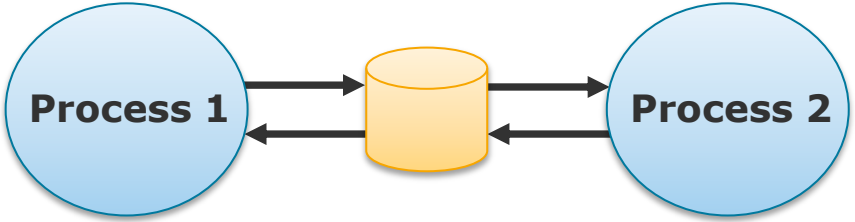
- Communication channels (buses, network, memory, …) have limited speed, bandwidth and throughput
  - Number and size of messages matters!

Overview

Process 1 and 2 can be the same (send a message to myself)

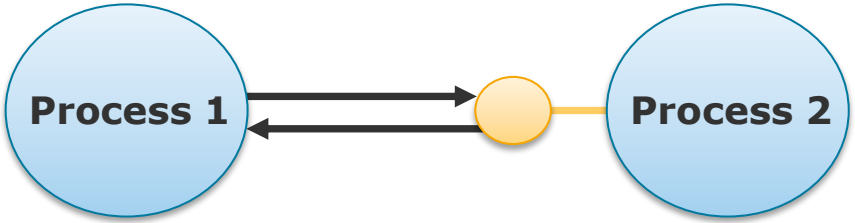
Dataflow Through Databases

- information storage and retrieval



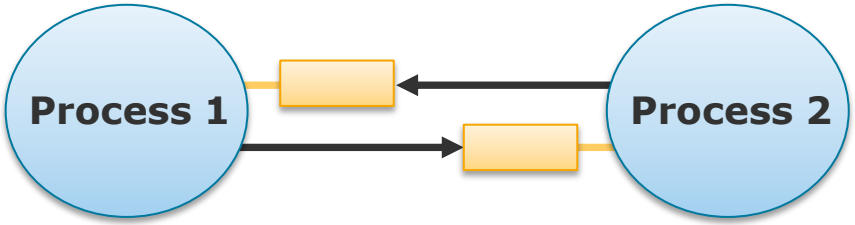
Dataflow Through Services

- service calls with responses



Message-Passing Dataflow

- asynchronous messages



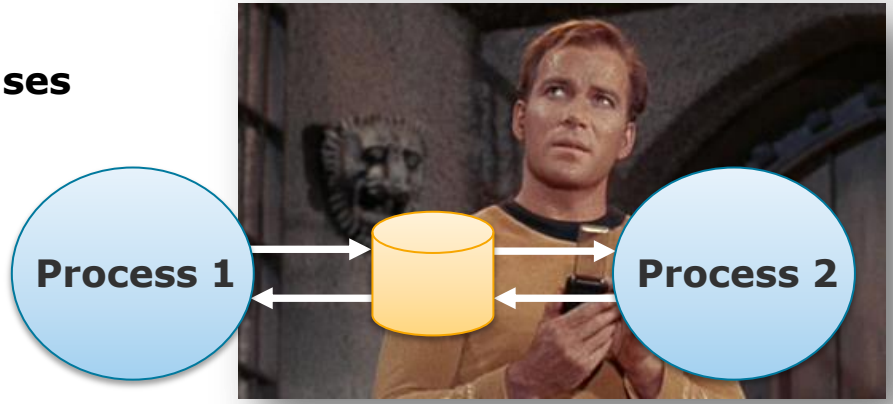
Encoding and Evolution

Formats for Encoding Data

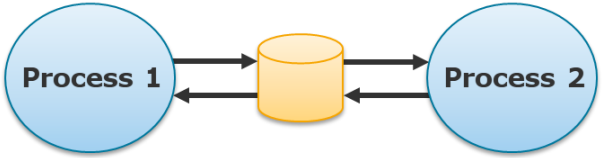
- Language-Specific Formats
- JSON, XML, and Binary Variants
- Thrift and Protocol Buffers
- Avro

Models of Dataflow

- **Dataflow Through Databases**
- Dataflow Through Services
- Message-Passing Dataflow



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

Every data value is a message

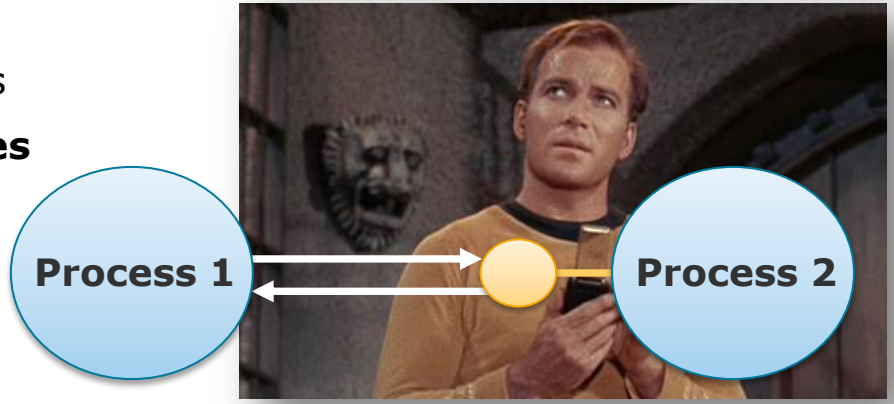
Encoding and Evolution

Formats for Encoding Data

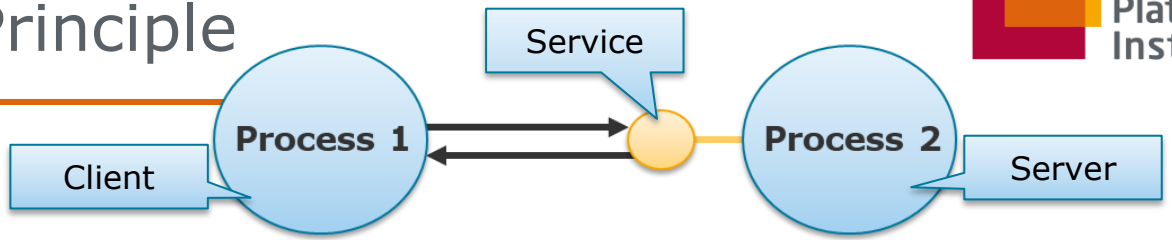
- Language-Specific Formats
- JSON, XML, and Binary Variants
- Thrift and Protocol Buffers
- Avro

Models of Dataflow

- Dataflow Through Databases
- **Dataflow Through Services**
- Message-Passing Dataflow



Communication Principle



Communication Structure

- Service:
  - An API that can be accessed by other (remote) processes
  - 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
    (other than database APIs that are more open)

Asymmetric Communication

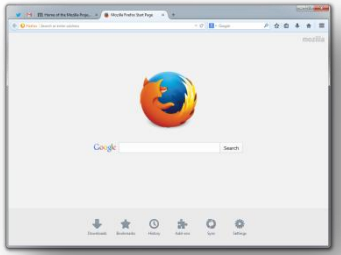
- Communicating processes have two roles:
  - Server: exposes a service that other processes can see and use
  - Client: connects to a server's service and calls functions

Communication Principle

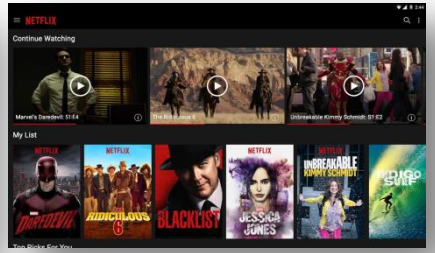
Service-Oriented Architecture (SOA)

- 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



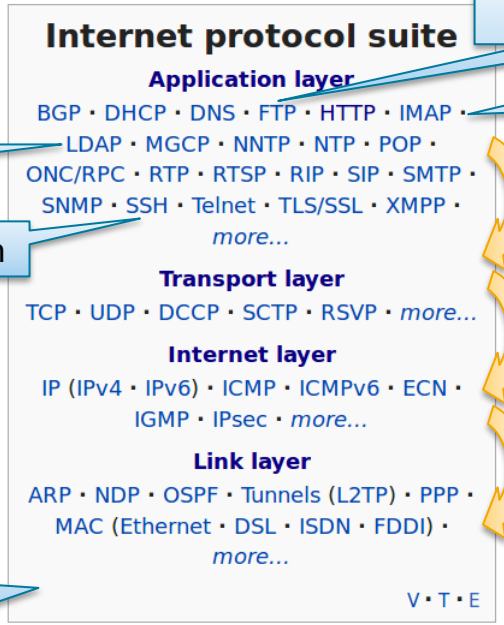
Apps

Online Games

Communication Principle

Communication

- Message format: agreement on model, schema, and encoding of messages
- Protocol: agreement on how messages are exchanged
  - Plethora of implementations
  - Example: internet protocols



large file transfer

email

access control

encrypted communication

build upon

build upon

build upon

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

Hypertext Transfer Protocol (HTTP)

Definition

- A stateless 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 data format
- Based on the TCP transport layer protocol
- Uses 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)

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

- A sequence of network request-response transactions:
  1. client establishes a TCP connection to server port (typically port 80)
  2. client sends a client's a HTTP message
  3. server sends back a status line with a message of its own
  4. server 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

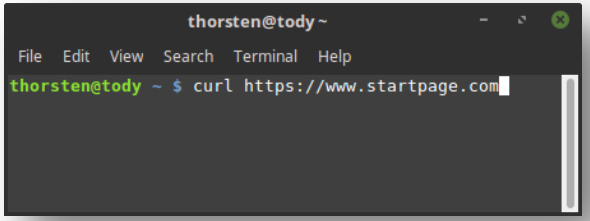
- GET http://www.my-host.com/my-new-homepage.html

```
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



REST and SOAP Services

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
- OpenAPI Specification:
  - Creates the RESTful contract for your API, i.e., all resources and their supported methods
  - Implemented in the Swagger framework (see https://swagger.io/)

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

REST and SOAP Services

Functions are main concepts in SOAP (in contrast to **resources** in REST)

Simple Object Access Protocol (SOAP)

- An XML-based protocol for making network API requests
- 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

REST and SOAP Services

Simple Object Access Protocol (SOAP)

- Simple example:

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

```
<?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

A simple, language-agnostic interface definition

REST and SOAP Services

Binding of an interface to concrete HTTP SOAP calls

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 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.)

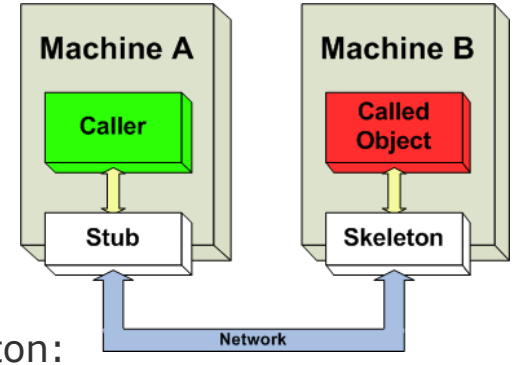
RPC Services

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)
- Remote procedures are called like normal (local) procedures
  - Tight coupling between processes

Example for OO languages:

- **Both** machines hold an interface containing the remote procedures
- The RPC framework uses this interface to generate Stub and Skeleton:
  - Stub: accepts procedure calls, serializes them, sends the serialized messages to the Skeleton, awaits results, deserializes them, and passes them to the caller
  - Skeleton: accepts procedure call, deserializes the message, calls the corresponding method with the given parameters, serializes results, and sends them back to stub

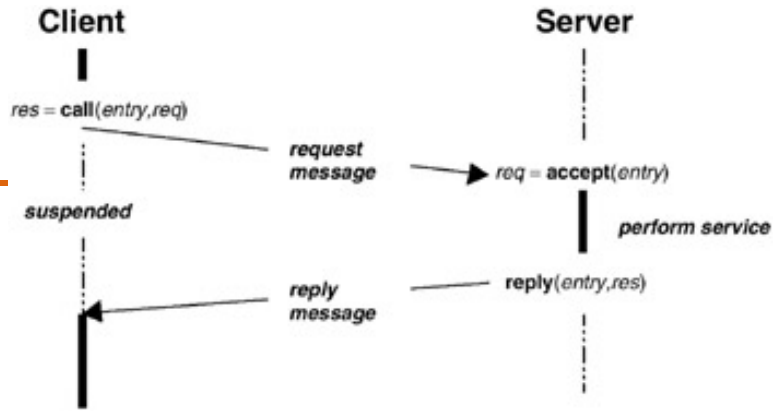


Dataflow Through Services

RPC Services

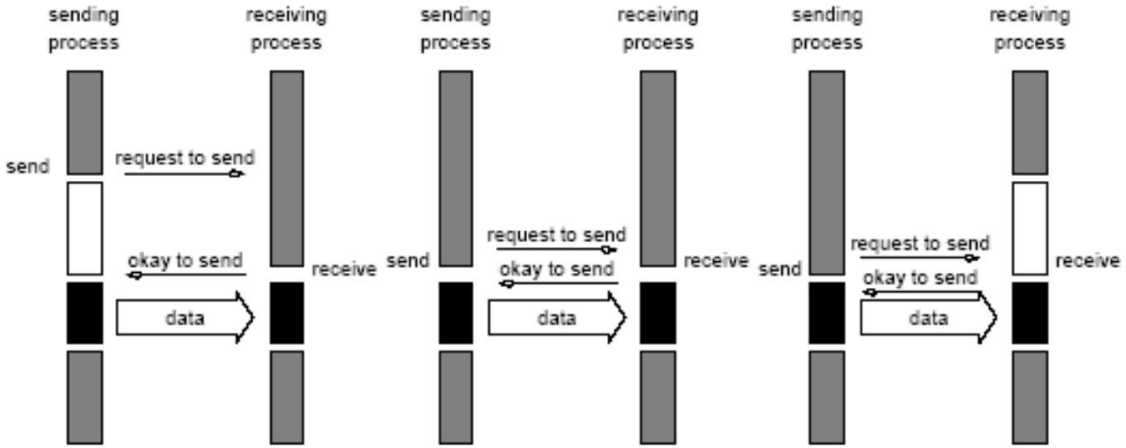
Blocking communication

- RPC calls block the program flow until the call finished



Rendezvous protocol

- Handshake protocol for sending data
- Avoid sending data to processes that can (at the moment) not accept them
- Before data is send, 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

RPC Services

Advantages

- RPC communication appears to be easy (no protocol specific coding) and extensive (no restrictions other than those the current language already has)
- RPC protocols often perform better than REST services

Disadvantages

- Tight coupling of server and client code
  - ➢ Interface changes always concern both
- Remote and local 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)
- Hard to debug and test due to code generation (even worse if the RPC framework hides network errors)

Conclusion: **REST for public APIs; RPC (maybe) for private APIs**

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

Dataflow Through Services

RPC Services

Lots of RPC implementations

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.

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

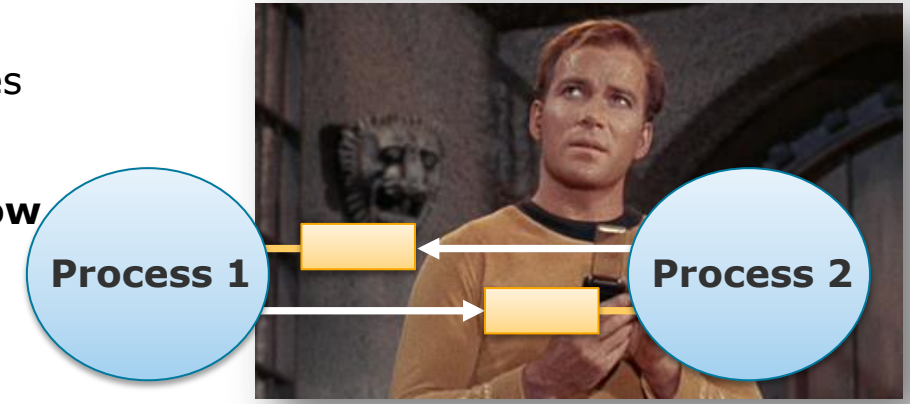
Encoding and Evolution

Formats for Encoding Data

- Language-Specific Formats
- JSON, XML, and Binary Variants
- Thrift and Protocol Buffers
- Avro

Models of Dataflow

- Dataflow Through Databases
- Dataflow Through Services
- **Message-Passing Dataflow**



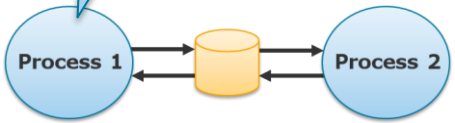
Message-Passing Dataflow

Communication Principles

Databases **Message-Passing** **Services**

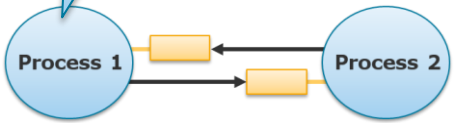
"I have a new booking request!

Someone should handle it …"



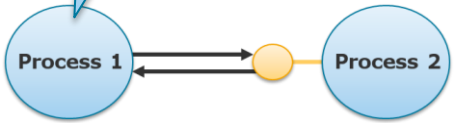
"I have a new booking request!

Can you handle it?"



"I have a new booking request!

Book it!"



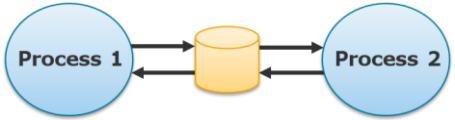
Communication Principles

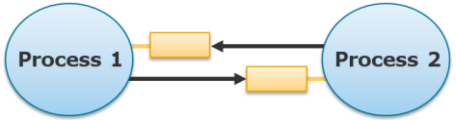
Databases | **Message-Passing** | **Services**

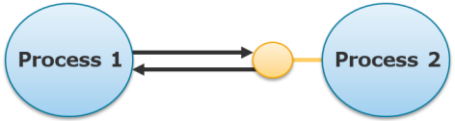
- Data
- No response
- Non-blocking
- Asynchronous
- No addressing

- Messages
- Maybe response
- Usually non-blocking
- Asynchronous
- Addressing recipient

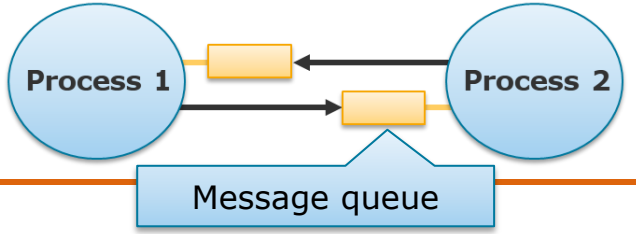
- Function calls
- Response
- Blocking
- Synchronous
- Addressing recipient







Communication Principle

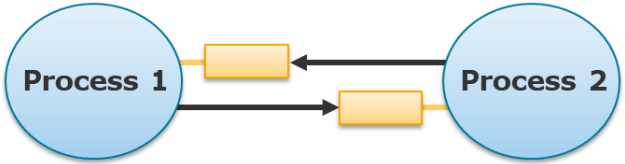


Communication

- An object-oriented paradigm that models all communication between objects (in different threads or processes) via 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 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

Communication Principle



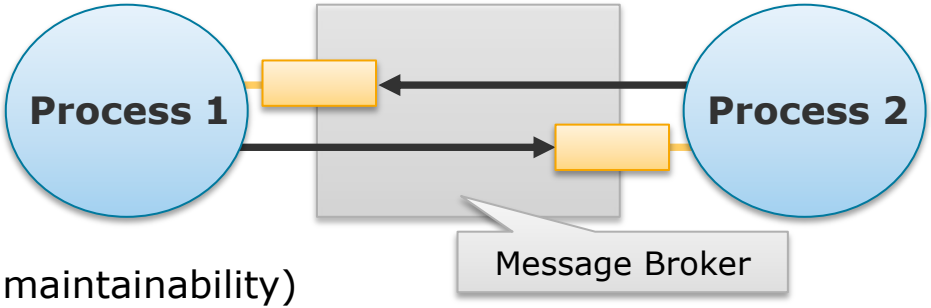
Principles

- 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 partner actually are

Communication Principle

Message Broker

- Also called message queue or message-oriented middleware
- Part of the message-passing framework that delivers messages from their sender 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

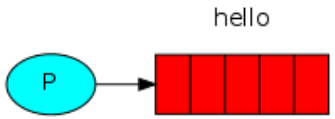
Message-passing frameworks

- Commercial:
  - TIBICO, IBM WebSphere, webMethods, …
- Open source:
  - RabbitMQ, ActiveMQ, HornetQ, NATS, Apache Kafka, …
- Most massage-passing frameworks are actually implemented using **RPC**

Example: RabbitMQ – Sending a Message

```
5   public class Send {
6
7     private final static String QUEUE_NAME = "hello";
8
9     public static void main(String[] argv) throws Exception {
10      ConnectionFactory factory = new ConnectionFactory();
11      factory.setHost("localhost");
12      Connection connection = factory.newConnection();
13      Channel channel = connection.createChannel();
14
15      channel.queueDeclare(QUEUE_NAME, false, false, false, null);
16      String message = "Hello World!";
17      channel.basicPublish("", QUEUE_NAME, null, message.getBytes("UTF-8"));
18      System.out.println(" [x] Sent '" + message + "'");
19
20      channel.close();
21      connection.close();
22    }
23  }
```

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



Create a connection to the message broker running on localhost.

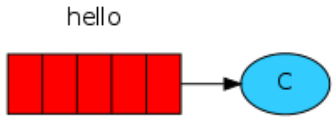
Create a channel to a queue; the queue is created if it does not exist yet.

Send the message encoded as an array of bytes.

Close all channels and the connection.

Example: RabbitMQ – Receiving a Message

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




```
5  public class Recv {
6
7    private final static String QUEUE_NAME = "hello";
8
9    public static void main(String[] argv) throws Exception {
10     ConnectionFactory factory = new ConnectionFactory();
11     factory.setHost("localhost");
12     Connection connection = factory.newConnection();
13     Channel channel = connection.createChannel();
14
15     channel.queueDeclare(QUEUE_NAME, false, false, false, null);
16     System.out.println(" [*] Waiting for messages. To exit press CTRL+C");
17
18     Consumer consumer = new DefaultConsumer(channel) {
19       @Override
20       public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body)
21           throws IOException {
22         String message = new String(body, "UTF-8");
23         System.out.println(" [x] Received '" + message + "'");
24       }
25     };
26     channel.basicConsume(QUEUE_NAME, true, consumer);
27   }
28 }
```

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.

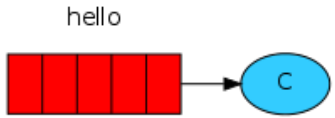
Decode any received byte message.

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

Example: RabbitMQ – Receiving a Message

```
5   public class Recv {
6
7     private final static String QUEUE_NAME = "hello";
8
9     public static void main(String[] argv) throws Exception {
10      ConnectionFactory factory = new ConnectionFactory();
11      factory.setHost("localhost");
12      Connection connection = factory.newConnection();
13      Channel channel = connection.createChannel();
14
15      channel.queueDeclare(QUEUE_NAME, false, false, false, null);
16      System.out.println(" [*] Waiting for messages. To exit press CTRL+C");
17
18      Consumer consumer = new DefaultConsumer(channel) {
19        @Override
20        public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body)
21            throws IOException {
22          String message = new String(body, "UTF-8");
23          System.out.println(" [x] Received '" + message
24        }
25      };
26      channel.basicConsume(QUEUE_NAME, true, consumer);
27    }
28  }
```

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



This is the metadata that pure RPCs are lacking:

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

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()
```

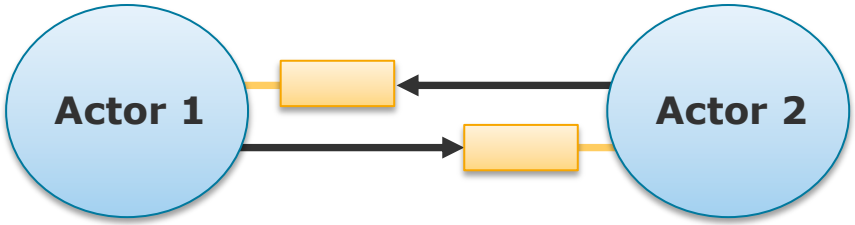
```python
#!/usr/bin/env python
import pika

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

Actor Model

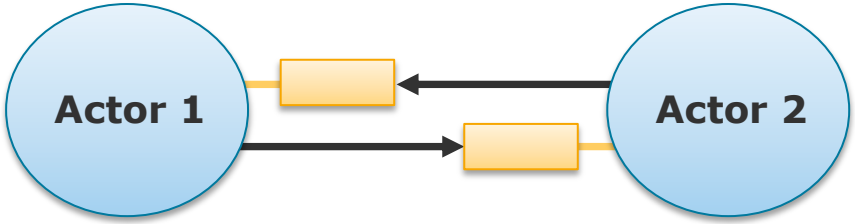


Actor Model

- A stricter message-passing model that treats actors as the universal primitives of concurrent computation
- Actor:
  - Computational entity (message-passing object; private state/behavior)
  - Owns exactly one mailbox (cannot subscribe to more or less queues)
  - Reacts on messages it receives
- 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, …)

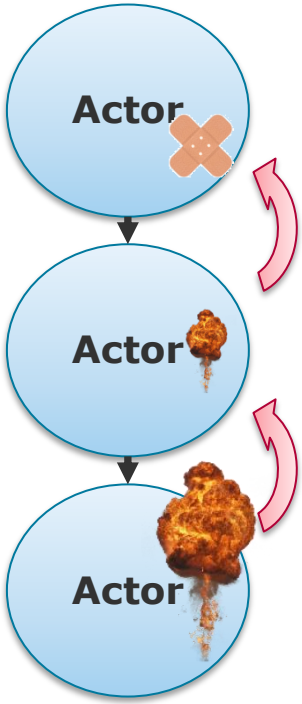
"The actor model retained more of what I thought were good features of the object idea"
Alan Kay, pioneer of object orientation

Actor Model

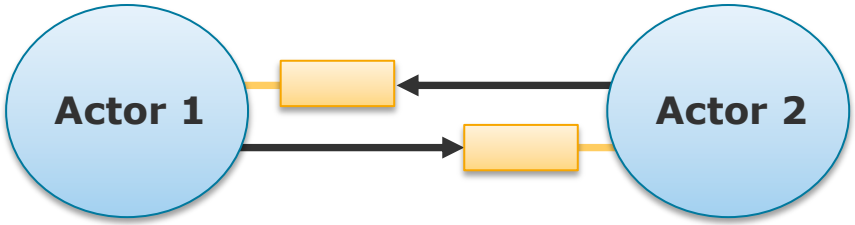


"Let it crash" philosophy

- 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
  - Its supervisor restarts the actor and resends the corrupted message



Actor Model



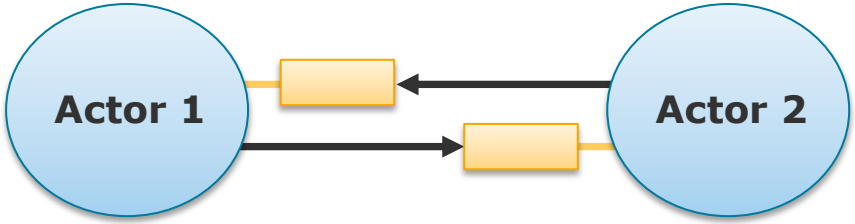
Advantages over pure RPC

- Errors are expected to happen and implemented into the model:
  - "Let it crash!" philosophy for unexpected errors
  - Message loss does not starve sender, because messaging is asynchronous
  - Crashing actors are detected and (depending on configuration) automatically restarted
  - Undeliverable messages are resend/re-routed
  - …

Dynamic Parallelization

- Actors process one message at a time
  - Parallelization between actors not within an actor
  - ➢ Spawn new (child) actors if needed

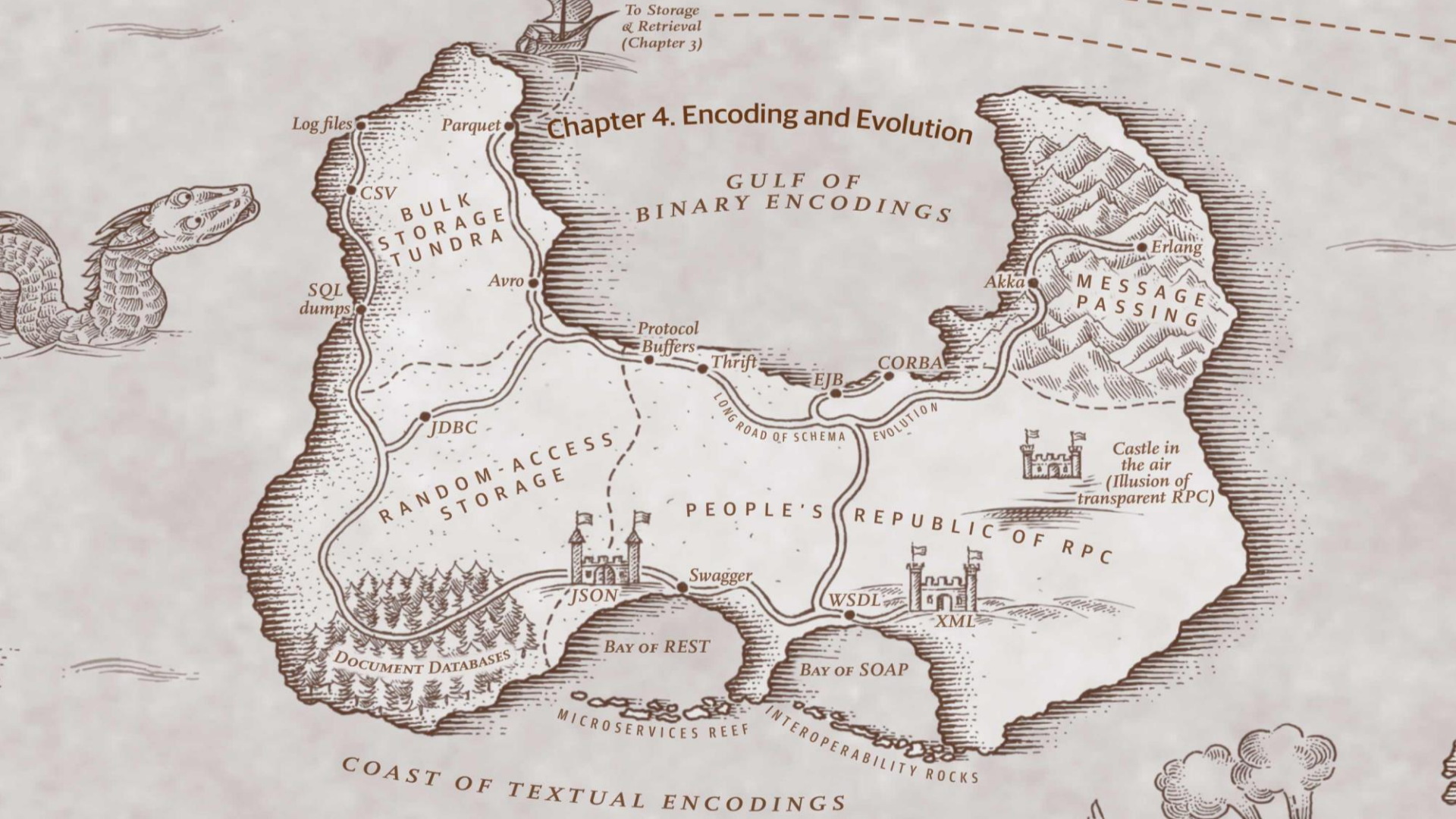
Actor Model



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

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


To Storage & Retrieval (Chapter 3)
Chapter 4. Encoding and Evolution
Log files
Parquet
CSV
BULK STORAGE TUNDRA
GULF OF BINARY ENCODINGS
SQL dumps
Avro
Protocol Buffers
Thrift
EJB
CORBA
Akka
Erlang
MESSAGE PASSING
LONG ROAD OF SCHEMA EVOLUTION
JDBC
RANDOM-ACCESS STORAGE
Castle in the air (Illusion of transparent RPC)
PEOPLE'S REPUBLIC OF RPC
JSON
Swagger
WSDL
XML
DOCUMENT DATABASES
BAY OF REST
BAY OF SOAP
MICROSERVICES REEF
INTEROPERABILITY ROCKS
COAST OF TEXTUAL ENCODINGS