Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Algorithms and Programming IV

From IPC to RPC

Summer Term 2023 | 21.06.2023

Claudia Müller-Birn, Barry Linnert

Our topics today

Interprocess Communication

- Multicast Communication

Remote Invocation

- Remote Procedure Call
- External Data Representation and Marshalling

RECAP

Recap: Architectural Model

Architectural elements

- **Communicating entities**
  - Processes
  - Objects
  - Components
  - Web Services
- **Communication paradigm**
  - Inter-process communication
    - **UDP sockets**
    - **TCP sockets**
    - Multi-cast
  - Indirect communication
  - Remote invocation
- **Roles and res-ponsibilities**
  - Architectural styles
    - Client-server
    - Peer-to-peer
- **Placement**
  - Multiple server
  - Proxy/Cache
  - Mobile code

Architectural styles

Architectural Styles in Distributed Systems

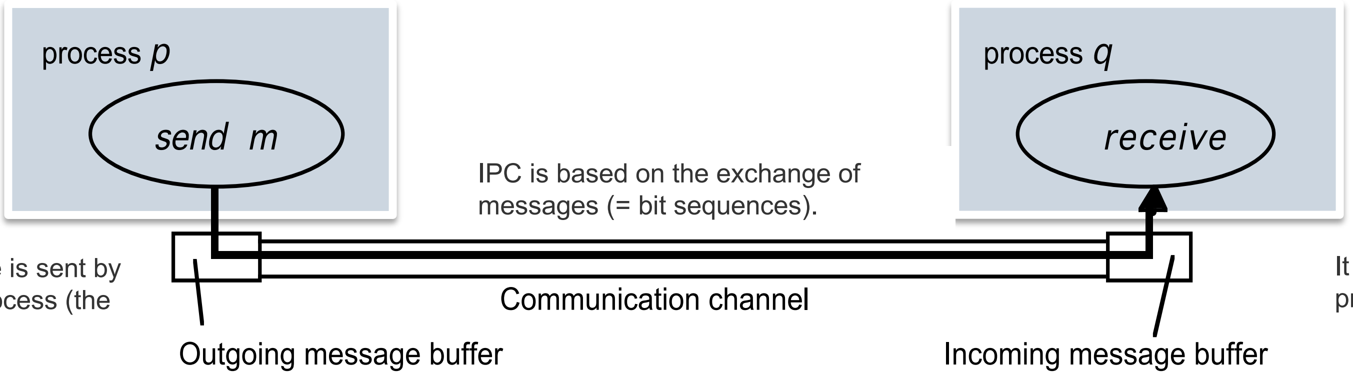
- Layered architectures
- Service-oriented architectures
- Publish-subscribe architectures

Interprocess Communication

Interprocess Communication (IPC) mechanisms provide a low-level support to enable processes from different address spaces to connect and exchange information.

A process is an object of the operating system through which applications gain secure access to computer resources. Individual processes are isolated from each other for this purpose.

A message is sent by the one process (the sender).



It is received by another process (the receiver).

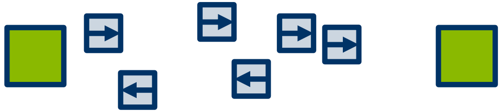
Freie Universität Berlin

Layers ISO Model vs. TCP/IP Model

Application	Application
Presentation	
Session	
Transport	Transport (UDP, TCP)
Network	Internet
Data link	Network Access
Physical	

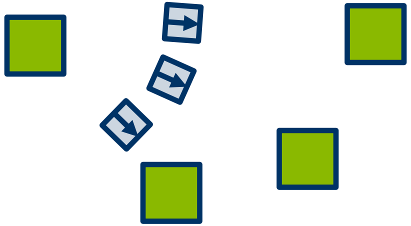
Possiblities to Communicate

- Connectionless 1:1
  UDP (unicast, datagram communication)



- Connection-oriented 1:1
  TCP (unicast, stream communication)

- Connectionless 1:n
  Multicast



Freie Universität Berlin

Interprocess Communication

MULTICAST COMMUNICATION

Multicast Communication

Efficient group communication has become important in applications such as video conferencing or joint editing of documents.

The standard solution is called multicast and provides 1-to-n communication:

- The application only needs to manage one connection per group.
- The resources in the network are used more efficiently.

Using Multicast for building Distributed Systems

- Fault tolerance based on replicated services
- Discovering services in spontaneous networking
- Better performance through replicated data
- Propagation of event notifications

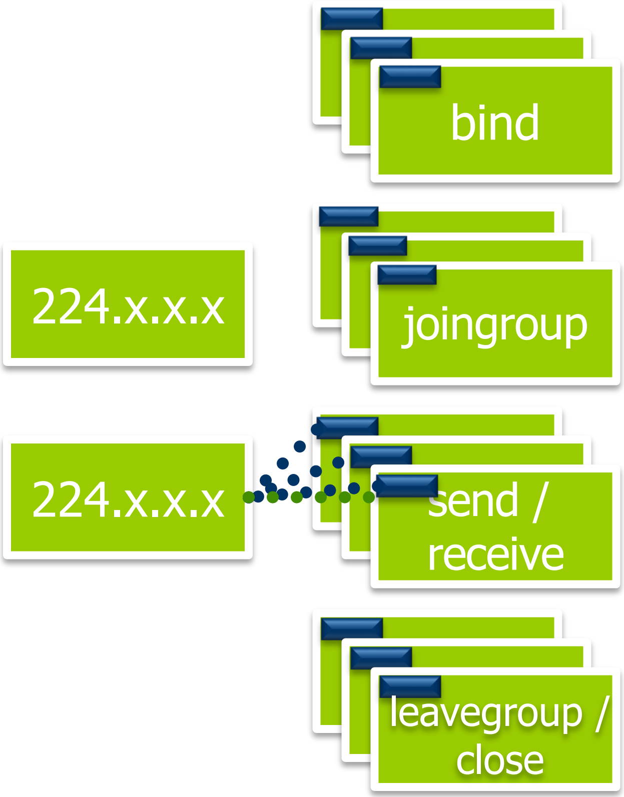
Multicast Sockets

1. Participants bind socket

2. Participants join group

3. Particpants receive messages from sender

4. Partcipants leave group and release socket



IP Multicast

- Is built on top of the Internet Protocol (IP) and allow the sender to transmit a single IP packet to a set of computers that form a multicast group.
- Multicast group is specified by a Class D Internet Address. Every IP datagram whose destination address starts with "1110" (in IPv4) is an IP Multicast datagram.
- IP packets can be multicast on a local and wider network. In order to limit the distance of operation, the sender can specify the number of routers that can be passed (i.e. time to live, or TTL)
- Multicast addresses can be permanent (e.g. 224.0.1.1 is reserved for the Network Time Protocol (NTP))

Java API: java.net.MulticastSocket

```
public class MulticastSocket extends DatagramSocket {
        public MulticastSocket()...
        public MulticastSocket(int port)...
        // create socket and select port number explicitely or implicitely

        public void joinGroup(InetAddress mcastaddr) throws ...
        // join group under the address mcastaddr
        public void leaveGroup(InetAddress mcastaddr) throws ...
        // leave group
        public void setTimeToLive(int ttl) ...
        // define Time to Live – default is 1 !
}
```

Please note: send, receive, ... are inherited from class DatagramSocket

Issue of Multicast

- A significant issue in applying multicast was setting up reliable communication paths for information dissemination, which involved a huge management effort.

- With the advent of peer-to-peer technology, and, notably structured overlay management, it became easier to set up communication paths.

- As peer-to-peer solutions are typically deployed at the application layer, various application-level multicasting techniques have been introduced. – we talk about it ☺

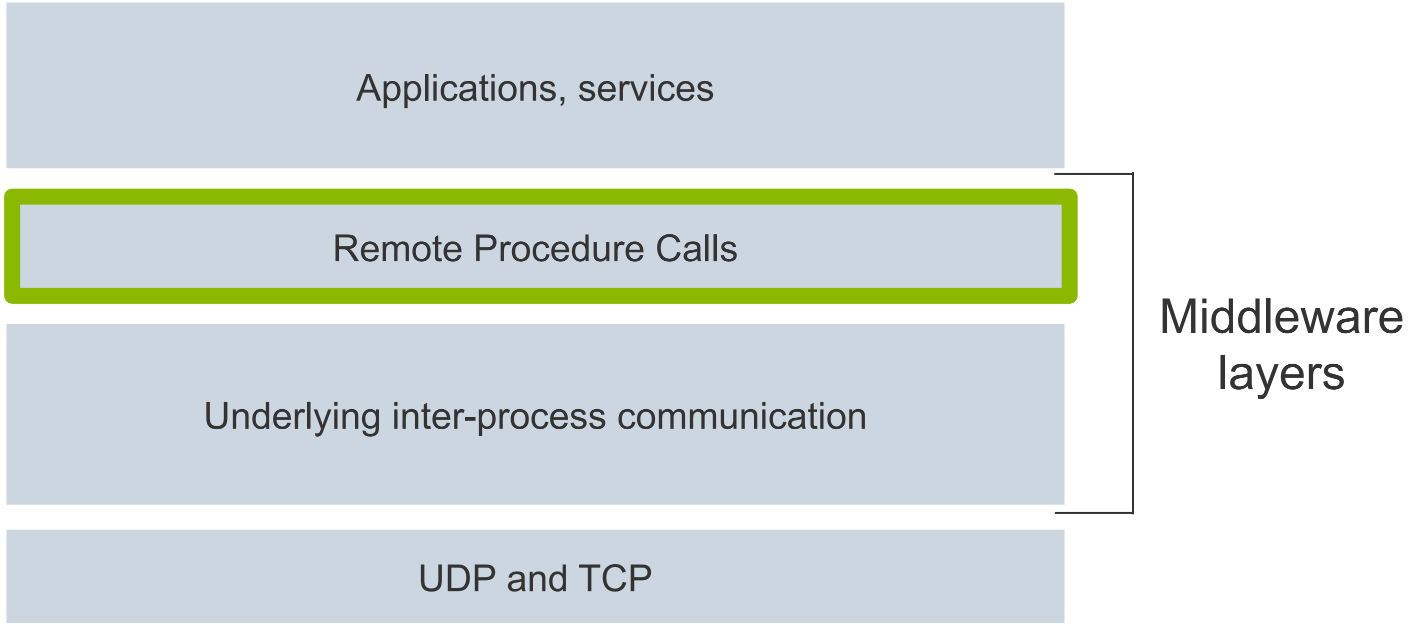
Observations

Observation 1:
Message-based interaction between processes over sockets in distributed software is cumbersome, untyped, error-prone.

Observation 2:
The service-oriented question/answer pattern is similar to the call-based interaction pattern between procedures, methods, ... for non-distributed software.

Conclusion:
Design a question/answer message pair as a programming-language call – and thus, develop distributed software similar to a non-distributed software!

Middleware Layers



Freie Universität Berlin

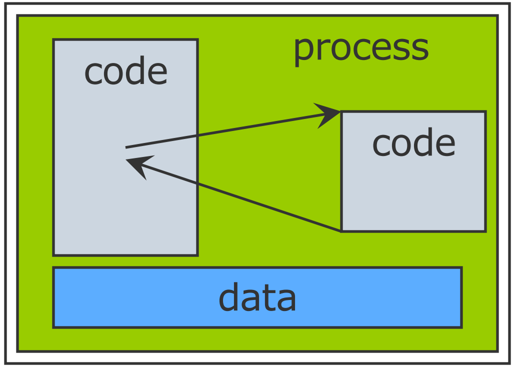
From IPC to RPC

REMOTE PROCEDURE CALL

Control Flow and Data Flow

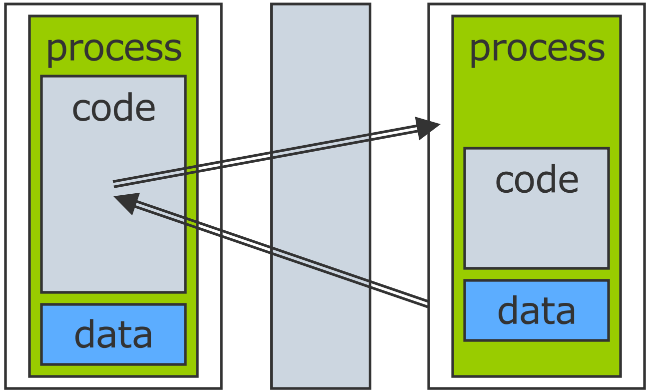
Local call:

- Provide arguments (stack)
- Jump to called code
- Provide results (stack)
- Return to caller



Remote call:

- Pack arguments in message
- Message from client to service provider
- Provider provides results
- Pack results in response
- Response from provider to client



Defining a Remote Call

A call is implemented as a remote call if another process executes the called process in another address space - and possibly in another computer - than that of the caller.

Implementation:

- The caller sends a message as a client that identifies the called party and contains the arguments to be passed.
- The called party replies as a service provider with a message containing the results to be transferred.

Attention:

- There is only one question/answer message pair, not a more extended dialog, as it is possible over TCP connections.

Issues that are important to understand the concept

The style of programming promoted by RPC – programming with interfaces.

The call semantics associated with RPC.

The key issue of transparency and how it relates to remote procedure calls.

Issues that are important to understand the concept

The style of programming promoted by RPC – programming with interfaces.

The call semantics associated with RPC.

The key issue of transparency and how it relates to remote procedure calls.

Programming with Interfaces

- Modern programming languages provide a means of organizing a program as a set of modules that can communicate with one another.

- Communication between modules can be by means of procedure calls between modules or by direct access to the variables in another module

- In order to control possible interactions between modules, an interface is defined for each module which specifies the procedures and variables that can be assessed.

Freie Universität Berlin

Advantages of using Interfaces in Distributed Systems

- Modular programming allows programmers to be concerned only with the abstraction offered by the service interface and they need not be aware of implementation details.

- Extrapolating to (potentially heterogeneous) distributed systems, programmers also do not need to know the programming language or underlying platform used to implement the services.

- Approach provides the natural support for software evolution in that implementations can change as long as the interface (the external view) remains the same.

Issues that are important to understand the concept

The style of programming promoted by RPC – programming with interfaces.

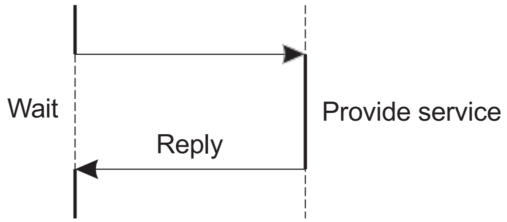
The call semantics associated with RPC.

The key issue of transparency and how it relates to remote procedure calls.

Basic Client–Server Model

Characteristics:

- There are processes offering services (servers)
- There are processes that use services (clients)
- Clients and servers can be on different machines
- Clients follow request/reply model with respect to using services



RPC Call Semantics

Fault tolerance measures			*Call semantics*
Retransmit request message	***Duplicate filtering***	***Re-execute procedure or retransmit reply***	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

RPC Call Semantics (*cont.*)

Maybe semantics

- RPC may be executed once or not at all, it means that faults are not tolerated
- Can suffer from omission and crash failures

At-least-once semantics

- Invoker receives either a result, in which case the procedure was executed at least once, or an exception informing that no result was received
- Can suffer from crash failures and arbitrary failures

At-most-once semantics

- Caller receives either a result, then the procedure was executed once, or an exception that no results has been received

Issues that are important to understand the concept

The style of programming promoted by RPC – programming with interfaces.

The call semantics associated with RPC.

The key issue of transparency and how it relates to remote procedure calls.

Distribution Transparency

Goal of a good remote access system
is the attainment of the highest possible degree of
Distribution Transparency.

Distribution Transparency has several facets:

- Access Transparency
- Location Transparency
- Migration Transparency
- Replication Transparency

Freie Universität Berlin

Remote Procedure Call

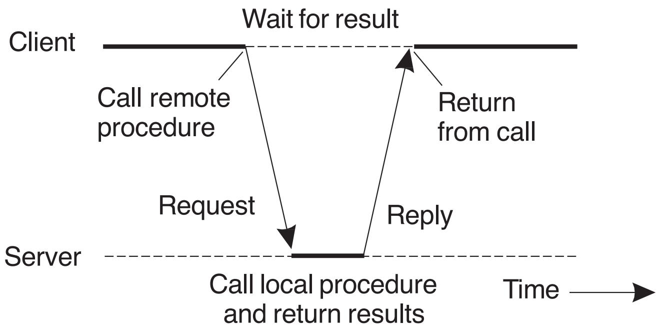
BASIC CONCEPT

Basic RPC operation

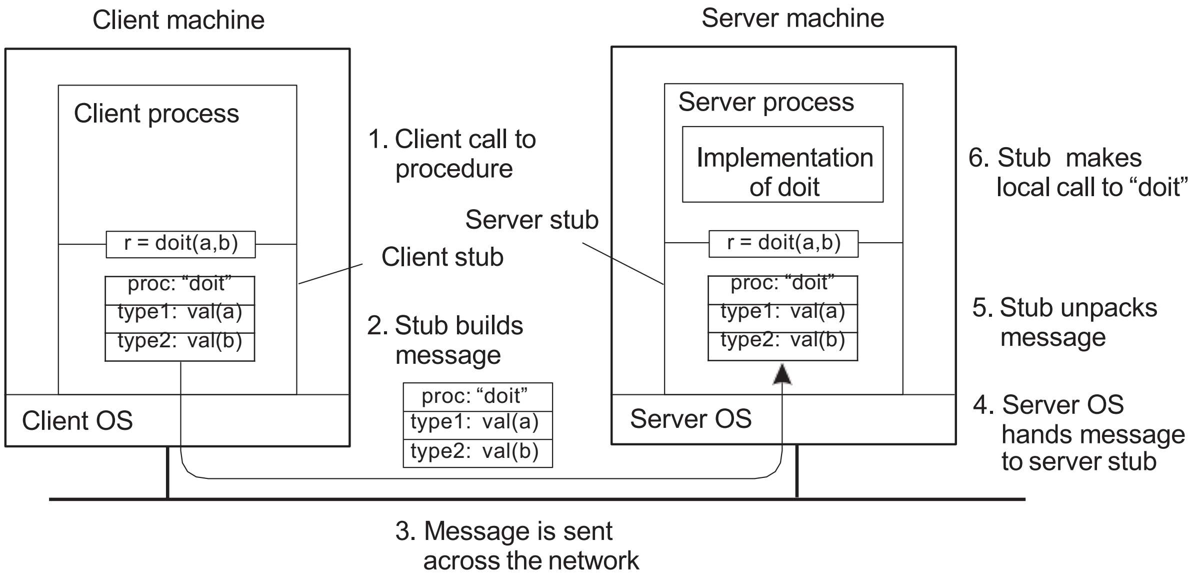
Observations

- Application developers are familiar with simple procedure model
- Well-engineered procedures operate in isolation (black box)
- There is no fundamental reason not to execute procedures on separate machine

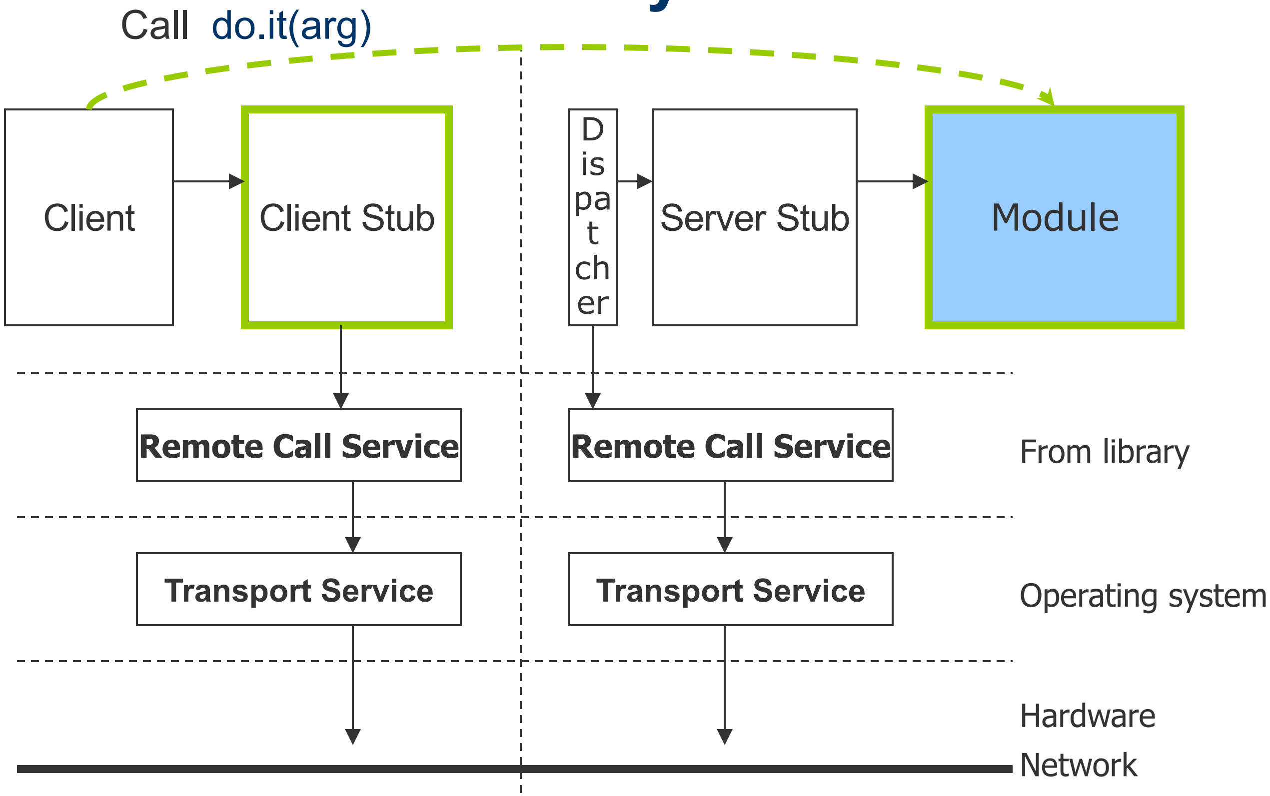
Conclusion
Communication between caller & callee can be hidden by using procedure-call mechanism.



Basic RPC Operation



Remote Call: Functional Hierarchy



Consideration

The function of the client stub is to take its parameters, pack them into a message, and send them to the server stub.

Why is it not to simple as it at first appears?

What is the Challenge?

Messages consist of sequences of bytes.

Some Interoperability problems

- Big-endian, little-endian byte ordering
- Character encodings (ASCII, UTF-8, Unicode)

So, we must either:

- Have both sides agree on an external representation or
- transmit in the sender's format along with an indication of the format used. The receiver converts to its form.

Remote Procedure Calls

EXTERNAL DATA REPRESENTATION AND MARSHALLING

External Data Representation and Marshalling

External data representation

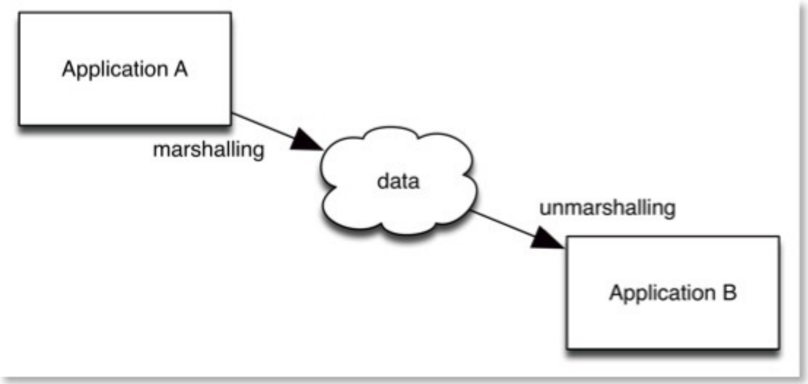
- An agreed standard for the representation of data structures and primitive values.

Marshalling

- The process of taking a collection of data structures into an external data representation type appropriate for transmission in a message.

Unmarshalling

- The converse of this process is unmarshalling, which involves reformatting the transferred data upon arrival to recreate the original data structures at the destination.



http://www.breti.org/tech/files/b400feb80f01f69e5cafca5160be5d65-67.html

Approaches for External Data Representation

XML (Extensible Markup Language)

Protocol buffer (protobuf)

JSON (JavaScript Object Notation)

Java’s object serialization

Java Object Serialization

```
public class Person implements Serializable {
        private String name;
        private String place;
        private int year;
        public Person(String aName, String aPlace, int aYear) {
                name = aName;
                place = aPlace;
                year = aYear;
        }
        // followed by methods for accessing the instance variables
}
```

A class implements the Serializable interface (which is provided in the java.io package) has the effect of allowing its instances to be serialized.

Extensible Markup Language (XML)

- XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use for writing structured documents for the Web.

- XML data items are tagged with ‘markup’ strings. The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures. For a specification of XML, see the pages on XML provided by W3C [www.w3.org VI].

- XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services.

Example: XML definition with namespace

```
<person pers:id="123456789" xmlns:pers = "http://www.nonsense.net/person">
        <pers:name> Smith </pers:name>
        <pers:place> London </pers:place >
        <pers:year> 1984 </pers:year>
        <!-- a comment -->
</person>
```

Example: XML schema

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions     >
        <xsd:element name= "person" type ="personType" />
            <xsd:complexType name="personType">
                <xsd:sequence>
                        <xsd:element name = "name"  type="xs:string"/>
                        <xsd:element name = "place"  type="xs:string"/>
                        <xsd:element name = "year"  type="xs:positiveInteger"/>
                </xsd:sequence>
                <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
            </xsd:complexType>
</xsd:schema>
```

Google Protocol Buffer

- Google Protocol Buffer (protobuf) is a common serialization format for storing and interchanging all kinds of structured information. It serves as a basis for a remote procedure call (RPC) system that is used for nearly all inter-machine communication at Google.

- The goal of Protocol Buffer is to provide a language- and platform-neutral way to specify and serialize data, it has been released as open source.

- Protobuf is 3-10 times smaller than an XML and 10-100 times faster than an XML.

http://code.google.com/apis/protocolbuffers

JSON (JavaScript Object Notation)

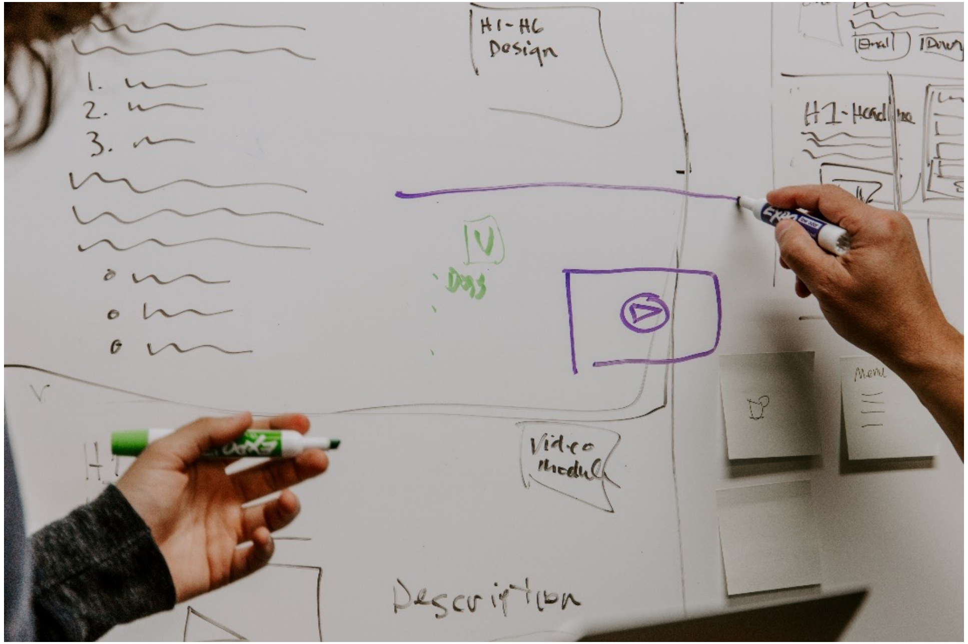
- JavaScript Object Notation (JSON) is a language-independent data format.
- It was derived from JavaScript, but many modern programming languages include code to generate and parse JSON-format data.
- Example:

```
{
    "firstName": "John",
    “lastName": "Smith",
    “birthyear”: “1984”,
    "address": {"city": "New York", "state": "NY},
}
```

Comparison of Data-Serialization Formats

	Standardized	Binary	Human-Readable	Standard-API
Java	Yes	Yes	No	Yes
XML	Yes	Partial	Yes	Yes
protobuf	No	Yes	Partial	For example, C++, Java, C#, Python, Ruby, C, PHP, R
JSON	Yes	No	Yes	Partial (JSON-LD)

Source: https://en.wikipedia.org/wiki/Comparison_of_data-serialization_formats



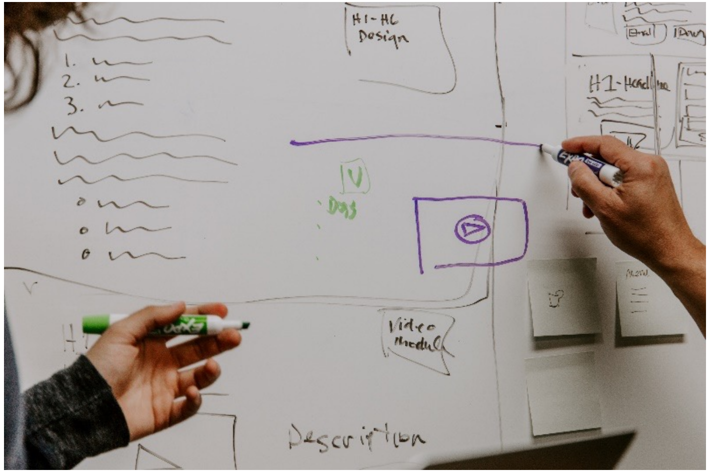
APPLICATION CASE: WHITEBOARD

Photo by [Kaleidico](https://unsplash.com) on [Unsplash](https://unsplash.com)

Collaborative Whiteboard

We aim to create a prototype for a "collaborative whiteboard“ which allows for the following activities:

- Select a shape (available shapes: triangle, rectangle, circle)
- Place shape on the drawing area
- Delete the shape of the drawing area
- Retrieve shapes from the drawing area



SimpleServer

```java
public class SimpleServer {

    private ServerSocket serverListen;
    private WhiteBoard whiteBoard;


    public SimpleServer(int port) throws IOException {
        this.serverListen = new ServerSocket(port);
        this.whiteBoard = new WhiteBoard();
    }
    public void startServer() throws IOException{
        while (true) {
            System.out.println("Server is Listening......");
            Socket socket=serverListen.accept();
            new WhiteBoardHandler(socket, this.whiteBoard).startCommunicationHandler();
            System.out.println("Connection closed");
        }
    }

. . .
```

https://github.com/FUB-HCC/WhiteBoard-Implementation-Examples/tree/master/RPCExampleSimple

SimpleServer (*cont.*)

```
. . .

    public static void main(String[] args) throws IOException{
        SimpleServer server = new SimpleServer(12345);
        try {
            server.startServer();
        } catch (Exception e) {
            System.err.println("Server couldn't be started");
e.printStackTrace();
System.exit(1);
        }

    }
}
```

https://github.com/FUB-HCC/WhiteBoard-Implementation-Examples/tree/master/RPCExampleSimple

Client

```
public class Client {

    static final int PORT = 12345;
    static final String HOST = "127.0.0.1";

    public static void main(String[] args) {

            BufferedReader bufferReader = new BufferedReader(new InputStreamReader(System.in));
            Socket socket = null;


            try {
            socket = new Socket(HOST, PORT); // connect to the server on port 6066 localhost
. . .
            try {
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintStream out = new PrintStream(socket.getOutputStream());
            System.out.println("write Commands here: ");
            System.out.println(in.readLine());

. . .
```

https://github.com/FUB-HCC/WhiteBoard-Implementation-Examples/tree/master/RPCExampleSimple

Recap: Architectural Model