Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Algorithms and Programming IV

Remote Invocation: Remote Procedure Call

Summer Term 2021 | 14.06.2021

Barry Linnert

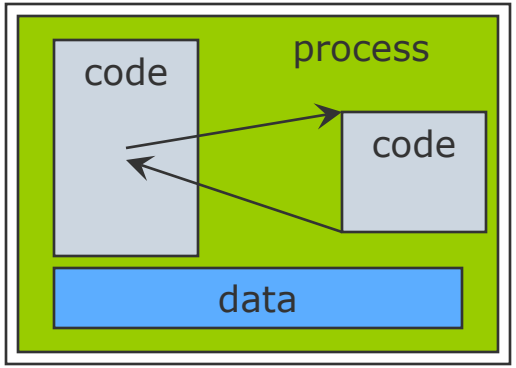
Control Flow and Data Flow

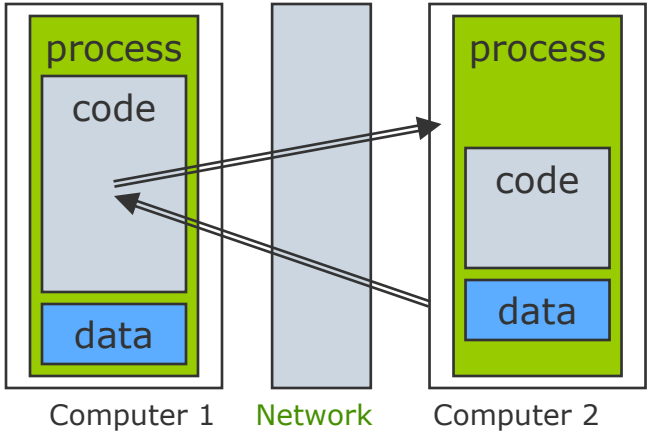
Local call:

- 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 the called process is executed by another 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:

- Here there is only one question/answer message pair, not a longer 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.

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

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.

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.

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

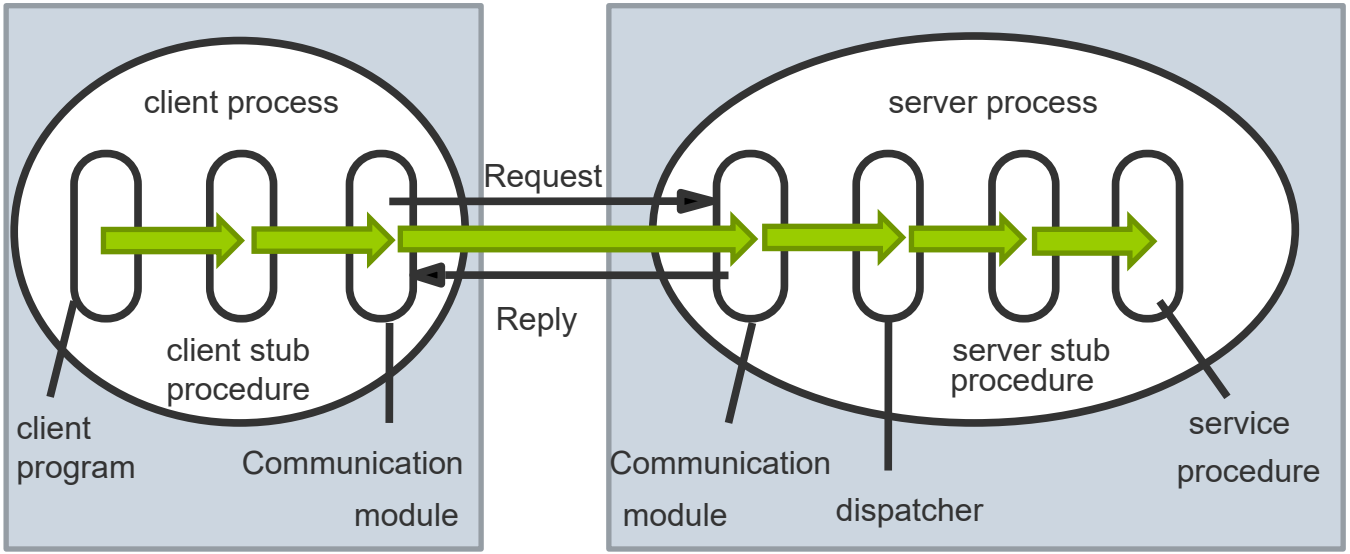
At-least-once semantics

- 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

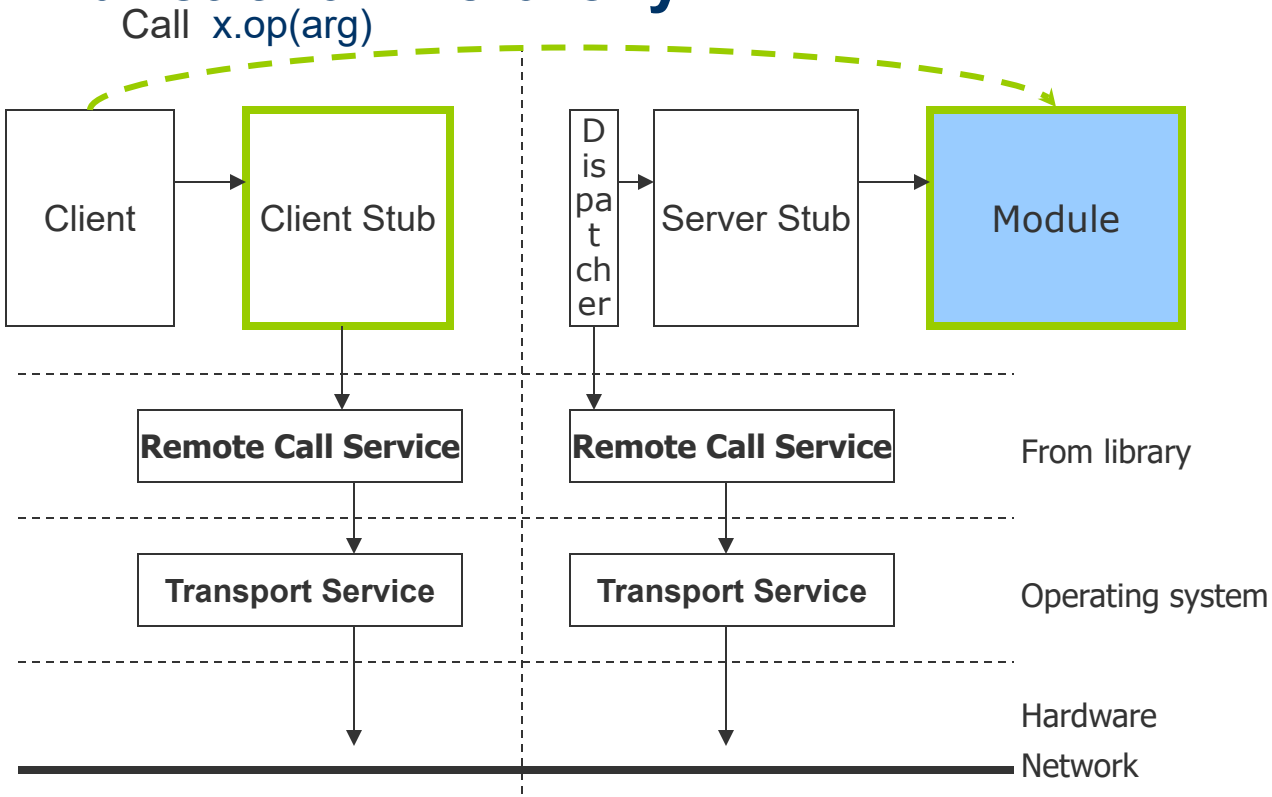
At-most-once semantics

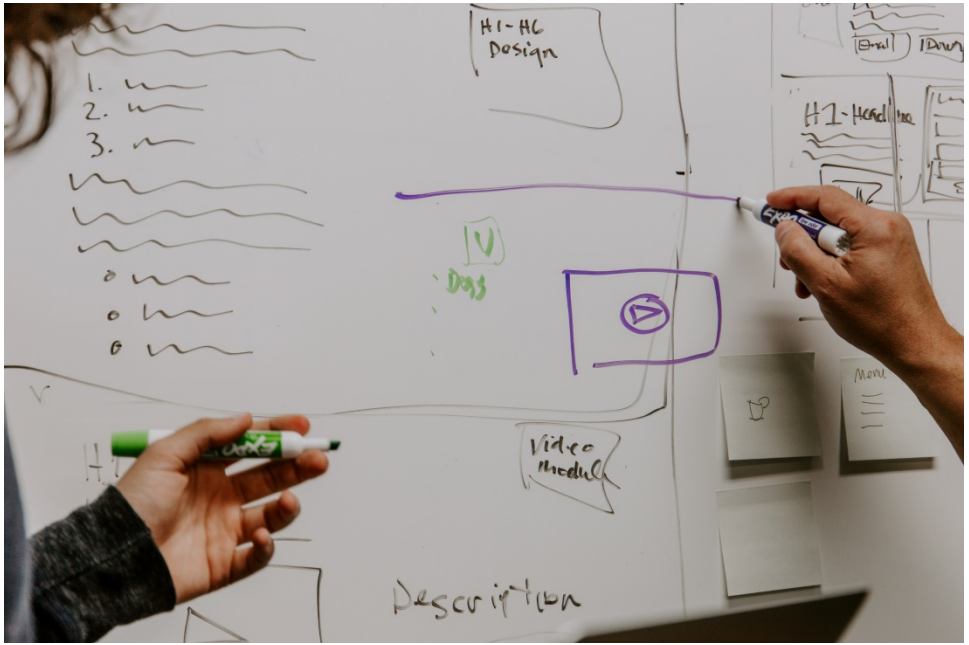
- Caller receives either a result, then the procedure was executed once, or an exception that no results has been received

Implementation of RPC



Remote Call: Functional Hierarchy





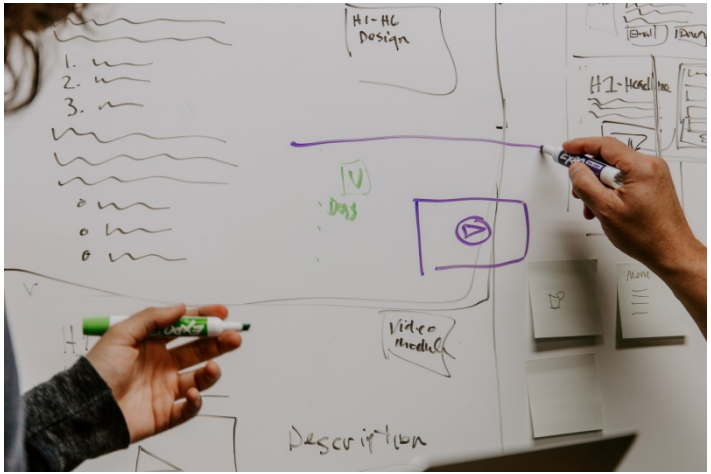
APPLICATION CASE: WHITEBOARD

Photo by Kaleidico on Unsplash

Collaborative Whiteboard

Our aim is to create prototypes 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 shape of drawing area
- Retrieve shapes from drawing area



SimpleServer

```
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 prot 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