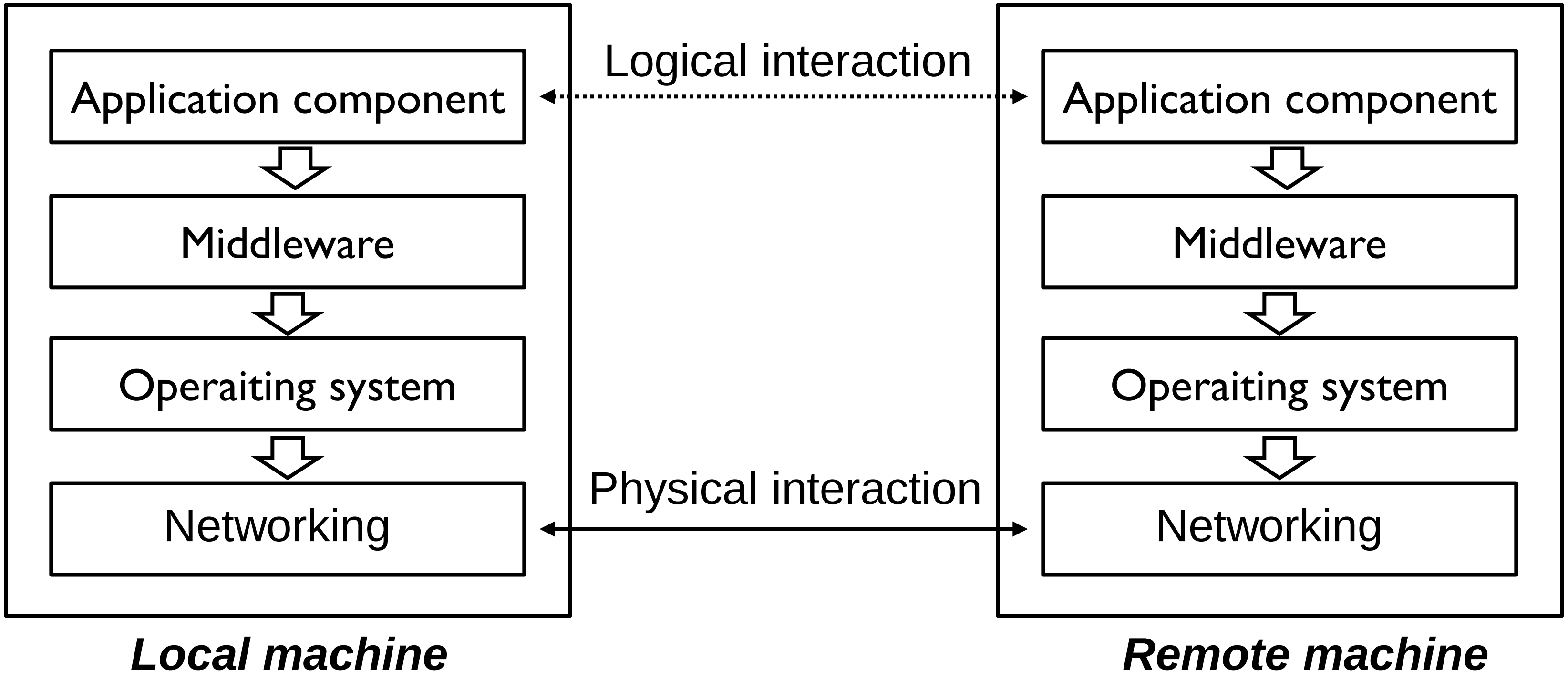
Development of Large Systems (RPC: remote procedure call)

Andrea Corradini
aco@cphnusinnes.dk

Remote procedure call (RPC)

- RPC supports client-server computing
  - servers offer a set of operations through a service interface
  - clients call these operations as if they were local

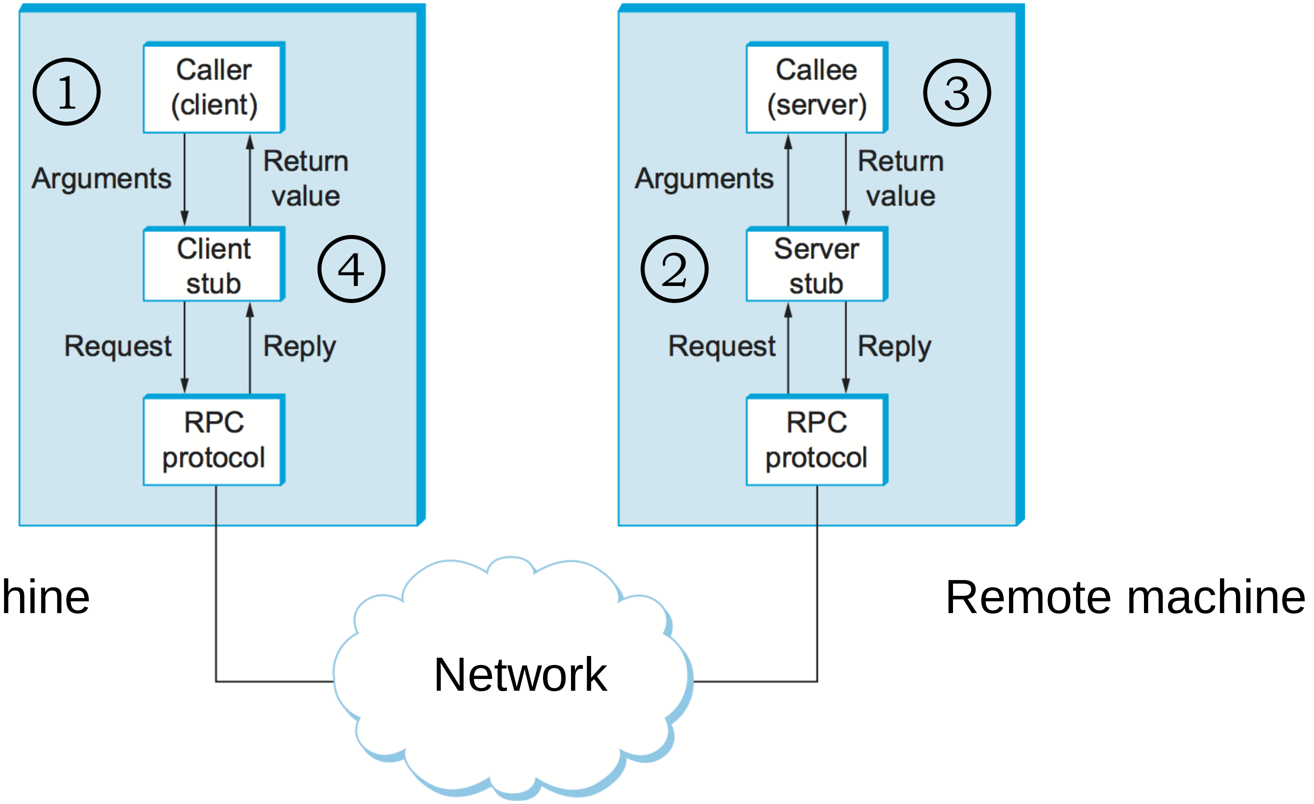


RPC features

- Allows a process executing on a machine to invoke a procedure in a process executing on another machine on a network
- Is a form of synchronous inter-process communication
  - requesting program is suspended until the result of the remote procedure is calculated and returned
- Based on request-response protocol (i.e. client-server)
  - initiated by the client sending a request message to a remote server to execute a given procedure with supplied parameters
- Abstracts procedure calls between processes and simplifies the process of writing distributed applications
  - preserving the syntax of a local procedure call while transparently initiating network communication
- Requires an IDL to describe the procedures that can be called over the network as a standard vocabulary for exchanging information

RPC flow

- Programs using RPC are compiled into executables that include a stub acting as representative of the remote procedure code



RPC semantics in case of failures

- Failure types
  - the client is unable to locate the server
  - the request message from the client to the server is lost
  - the reply message from the server to the client is lost
  - the server crashes after receiving a request
  - *t*he client crashes after sending a request

Failure 1: client cannot locate the server

- We distinguish between five different classes of failures that can occur in RPC systems
  - the client is unable to locate the server
  - the request message from the client to the server is lost
  - the reply message from the server to the client is lost
  - the server crashes after receiving a request
  - *t*he client crashes after sending a request
- Common solutions: report failure, signal handlers, exception handling, etc.

Failure 2: lost request messages

- We distinguish between five different classes of failures that can occur in RPC systems
  - the client is unable to locate the server
  - the request message from the client to the server is lost
  - the reply message from the server to the client is lost
  - the server crashes after receiving a request
  - *t*he client crashes after sending a request

Failure 3: lost reply messages

- We distinguish between five different classes of failures that can occur in RPC systems
  - the client is unable to locate the server
  - the request message from the client to the server is lost
  - the reply message from the server to the client is lost
  - the server crashes after receiving a request
  - *t*he client crashes after sending a request

Failure 2 and 3: treatment

- Client waits for reply message, resends request upon timeout
  - client cannot tell whether the request or reply was lost
- Client can safely resend a request for idempotent operations
  - idempotent operations produce the same result when executed repeatedly
- For non idempotent operations, client can add sequence numbers to requests for the server to tell a retransmitted request from the original one
  - server need keep track of the most recently received sequence number from each client
  - server will not carry out a retransmitted request, but sends a reply to the client

Failure 4: server crash after receiving request

- We distinguish between five different classes of failures that can occur in RPC systems
  - the client is unable to locate the server
  - the request message from the client to the server is lost
  - the reply message from the server to the client is lost
  - the server crashes after receiving a request
  - *t*he client crashes after sending a request

Failure 4: treatments

- The client cannot tell if the crash occurred before or after the request is carried out
  - request arrives, but the server crashes before sending the reply

    → client must retransmit the request
  - request arrives and is carried out, but the server crashes before sending the reply

    → server somehow has to report back to the client e.g. by raising an exception

Failure 4: approaches

- Give up immediately and report back failure (*at most once semantics*)
  - RPC is carried out at most one time, but possibly none at all
- Keep trying until a reply has been received then give it to the client (*at least once semantics*)
  - RPC is carried out at least one time, or possibly more
- Do not guarantee anything
  - when a server crashes, the client gets no help/promises
  - the RPC may have been carried out any number of times
- Ideal is *exactly once semantics* but usually no way to arrange it

Failure 5: client crashes

- We distinguish between five different classes of failures that can occur in RPC systems
  - the client is unable to locate the server
  - the request message from the client to the server is lost
  - the reply message from the server to the client is lost
  - the server crashes after receiving a request
  - *the client crashes after sending a request*

Failure 5: dealing with orphans

- Extermination: client keeps log entry before sending an RPC and kills off the orphan when it comes back up
  - expensive for memory storage
  - orphans may make RPC i.e., grand orphans are created
- Reincarnation: when a client reboots, it broadcasts a new epoch number; when server receives the broadcast, it kills all the computations running on behalf of the client
- Gentle reincarnation: variant of reincarnation, when computations killed only for clients that cannot be located
- Expiration: each RPC is given an amount of time T to do job
  - if it cannot finish, it must explicitly ask for another quantum T but right amount of time T is unknown

RPC protocols

- Connection-oriented protocol or connectionless protocol?
- Standard general-purpose protocol or one specifically designed for RPC?

XML-RPC

- Protocol uses a subset of XML vocabulary as encoding mechanism and interface definition language
  - client specifies a procedure name and parameters in the XML request
  - server returns a response or a failure in the XML response format
- Protocol uses the HTTP protocol to pass information from client to a server
  - XML messages are the payload of the HTTP response and request

XML-RPC data: basic types

- XML-RPC specification defines six basic data types
  - used in passing parameters, return values, and error messages
  - can be combined to create two more compound data types to support almost any data types in any programming language
- Basic data type:
  - 32-bit integer like e.g. *<int>23097</int>*
  - 64-bit double like e.g. *<double>-23.097</double>*
  - boolean like e.g. *<boolean>1</boolean>*
  - string like e.g. *<string>Hello World!</string>*
  - date and time like e.g. *<dateTime.iso8601>20210321T13:08:29 </dateTime.iso8601>*
  - binary values like e.g. *<base64>GpyBpcyBhHNob3J0Ncm3k0luZy </base64>*

XML-RPC compound data: array

- Multidimensional arrays can also be created by defining arrays inside <*value*> </*value*> tags

```
<value>
       <array>
            <data>
                 <value><boolean>1</boolean></value>
                 <value><string>It is true</string></value>
                 <value><int>23097</int></value>
            </data>
        </array>
 </value>
```

XML-RPC compound data: struct

- Structs are not objects

```
<value>
    <struct>
        <member>
            <name>givenName</name>
            <value><string>Andrea</string></value>
        </member>
        <member>
            <name>familyName</name>
            <value><string>Corradini</string></value>
        </member>
    </struct>
</value>
```

Correspondence with types in Java

XML-RPC Type	Type	Java Type
<i4> or <int>	Four byte signed integer	int
<boolean>	0 (false) or 1 (true)	boolean
<string>	string	String
<double>	double-precision signed floating point number	double
<dateTime.iso8601>	date/time	N/A
<base64>	base64-encoded binary	byte[] or String
<array>	array of elements	array object
<struct>	record of elements	object

Coding example: client request

- Method call encoded as message sent from client to server look like:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
    <methodName> sumhandler.sum</methodName>
    <params>
         <param> <value><int>23</int></value> </param>
         <param> <value><int>9</int></value> </param>
    </params>
</methodCall>
```

Coding example I: server response

- Messages returned from server to client look like:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodResponse>
    <params>
         <param> <value><int>32</int></value> </param>
    </params>
</methodResponse>
```

Coding example II: server error response

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodResponse>
    <fault>
        <value>
            <struct>
                <member>
                    <name>fault</name>
                    <value><int>99</int></value>
                </member>
                <member>
                    <name>faultString</name>
                    <value><string>No such method</string></value>
                </member>
            </struct>
        <value>
    </fault>
</methodResponse>
```

RPC limitations

- Limited number of data types
- No provision for passing objects as it is not compatible with objects
- No type checking of array values; mixed type not forbidden
- Strings allow only ASCII characters
- No check for duplicate names in struct
- No representation of NaN for double
- Limited security since firewalls are bypassed using HTTP

Coding exercise: XML-RPC

- Do the tutorial on XML-RPC at

  https://www.tutorialspoint.com/xml-rpc/xml_rpc_examples.htm
  - it is a remote call for a simple addition operation
  - add more operations
  - make the entire project more OOP (i.e., split responsibilities between objects)
  - display XML messages exchanged between client and server
  - display XML error messages exchanged between client and server
  - display the time it takes to run a method remotely and compare it with the time it take to run it locally
- (elective) Do your own client and server using Java sockets to send back and forth messages in HTTP

Headers for HTTP (for elective part)

- When method calls and method responses are sent using HTTP, messages require certain specific headers

Method call (from client)

POST / HTTP 1.1

Host: server host name

User-Agent: software making the request

Content-Type: text/xml

Content-Length: nr of bytes in payload

… here the payload …

Method response (from server)

HTTP 200 OK

Content-Type: text/xml

Content-Length: nr of bytes in payload

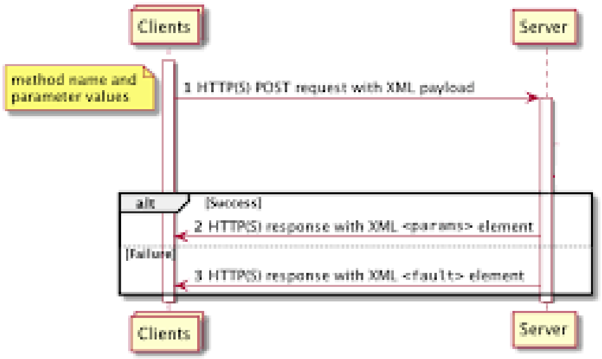
… here the payload…

RPC-XML: how to code it in Java (for elective part)

CLIENT

1 - Build an XML element *methodCall* that names the method to call and provides its parameters

2 - Send a POST request whose payload (content) is the XML element just built over a socket connection


Clients
Server
method name and
parameter values
1 HTTP(S) POST request with XML payload
alt
[Success]
2 HTTP(S) response with XML <params> element
[Failure]
3 HTTP(S) response with XML <fault> element
Clients
Server

7 - Receive the response message, parse the XML element returned, and use the outcome in the client code

SERVER

3 - Receive the request and use HTTP header Content-Length to read the XML payload

4 - Parse the XML element, extract method name, and retrieve the its parameters

5 - Search for the desired method and, if found, invoke it with the given parameters using reflection

6 - If method executes successfully, package the return value in *methodResponse* XML element (otherwise package a fault message), and send that message back to the client over the socket