

XML-RPC

This section demonstrates XML remote procedure calls.

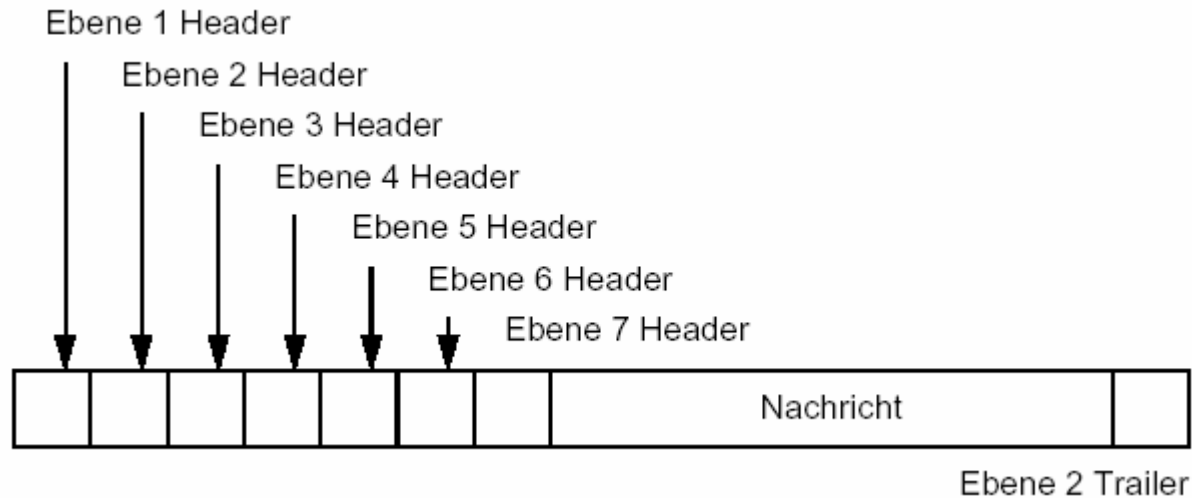
Inhalt

1.	Client-Server Modell	3
2.	Remote Procedure Call	6
2.1.	Basics.....	6
2.2.	XML-RPC Specification	9
2.2.1.	Requests.....	9
2.2.2.	Types	11
2.2.3.	Response	14
2.2.4.	Strategies/Goals	18
2.2.5.	FAQ.....	18
2.3.	Apache XML-RPC.....	21

2.3.1. Client classes.....	21
2.3.2. Server Side XML-PRC.....	22
2.3.3. Data types	25
2.4. XML-RPC Example	26

1. Client-Server Modell

Distributed systems could be realized using the OSI¹ reference model. Using this technique implies that a message has to be packed from each sender's layer and unpacked by the receiver.



This management overhead can be accepted in WAN (wide area network) environments but is unacceptable in LAN (local area network) environments.

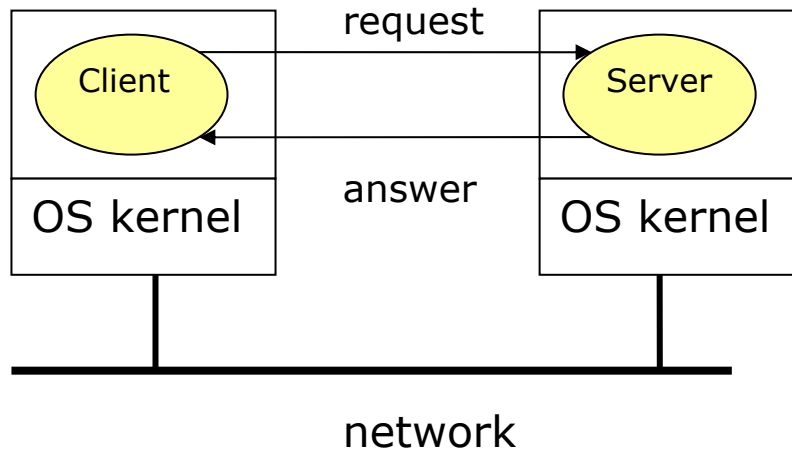
Here, the **client server model** is a better choice.

A distributed system based on the client server model consists of a set of cooperating processes (server) which offer services to clients.

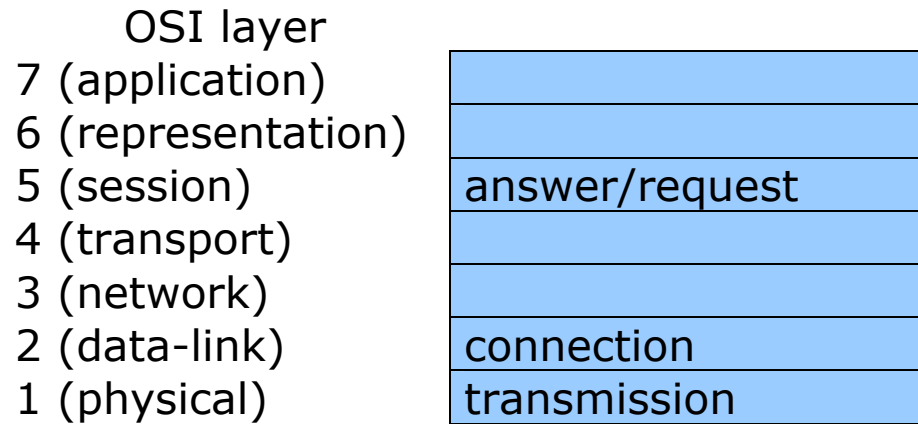
¹ Open Systems Interconnect model

Into a client server system a request/answer protocol is used between clients and a server:

A client sends a request (message) to a server which is responsible for the required service. The server sends the desired information (answer message) back to the client.



This kind of communication is more effective because only protocol layer 1, 2 and 5 are involved. Therefore client server communication can be represented into the OSI model as follows:



We will consider **remote procedure calls** to demonstrate, how Java threads can be used in client server environments.

2. Remote Procedure Call

Inside every computer, every time you click a key or the mouse, thousands of "**procedure calls**" are spawned, analyzing, computing and then acting on your gestures.

A procedure call is the **name of a procedure**, its **parameters**, and the **result** it returns.

A **remote procedure call** (RPC for short) is a very simple extension to the procedure call idea, it says let's create connections between procedures that are running in different applications, or on different machines.

2.1. Basics

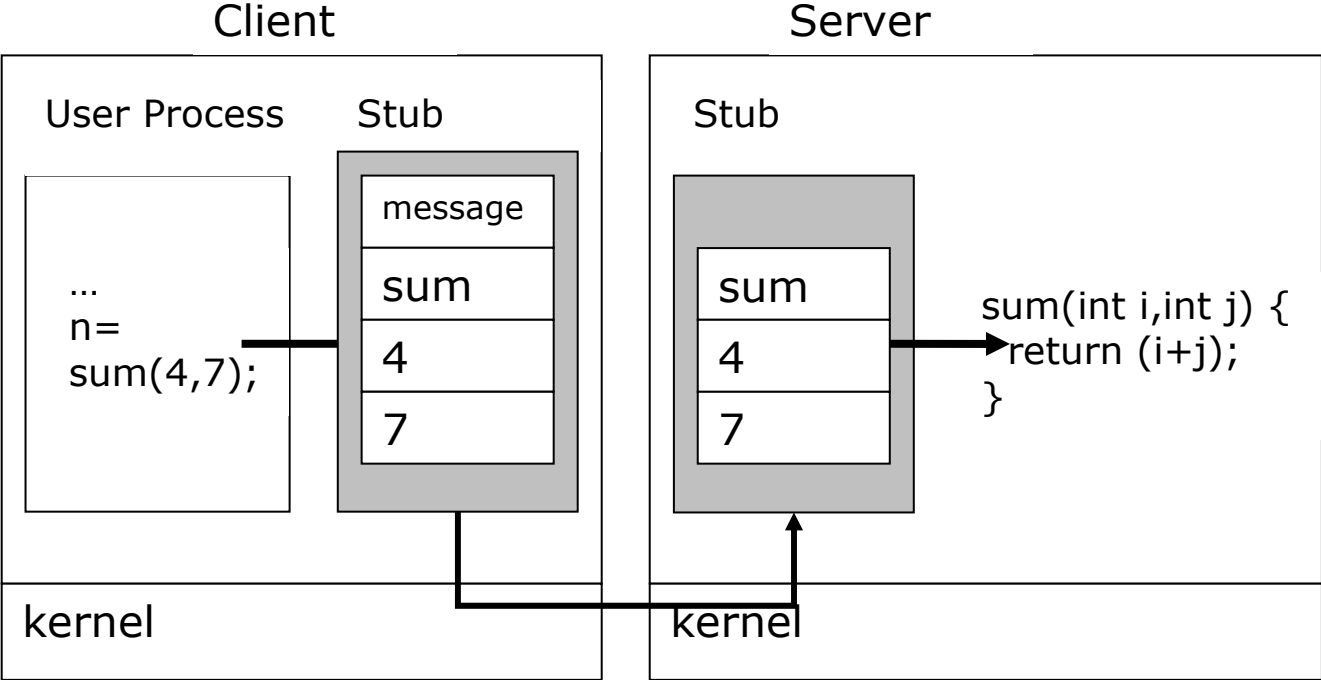
Conceptually, there's no difference between a local procedure call and a remote one, but they are implemented differently, perform differently (RPC is much slower) and therefore are used for different things.

Remote calls are "**marshalled**" by "stub procedures" into a format that can be understood on the other side of the connection; here "stub procedures" have to "unmarshall" the format.

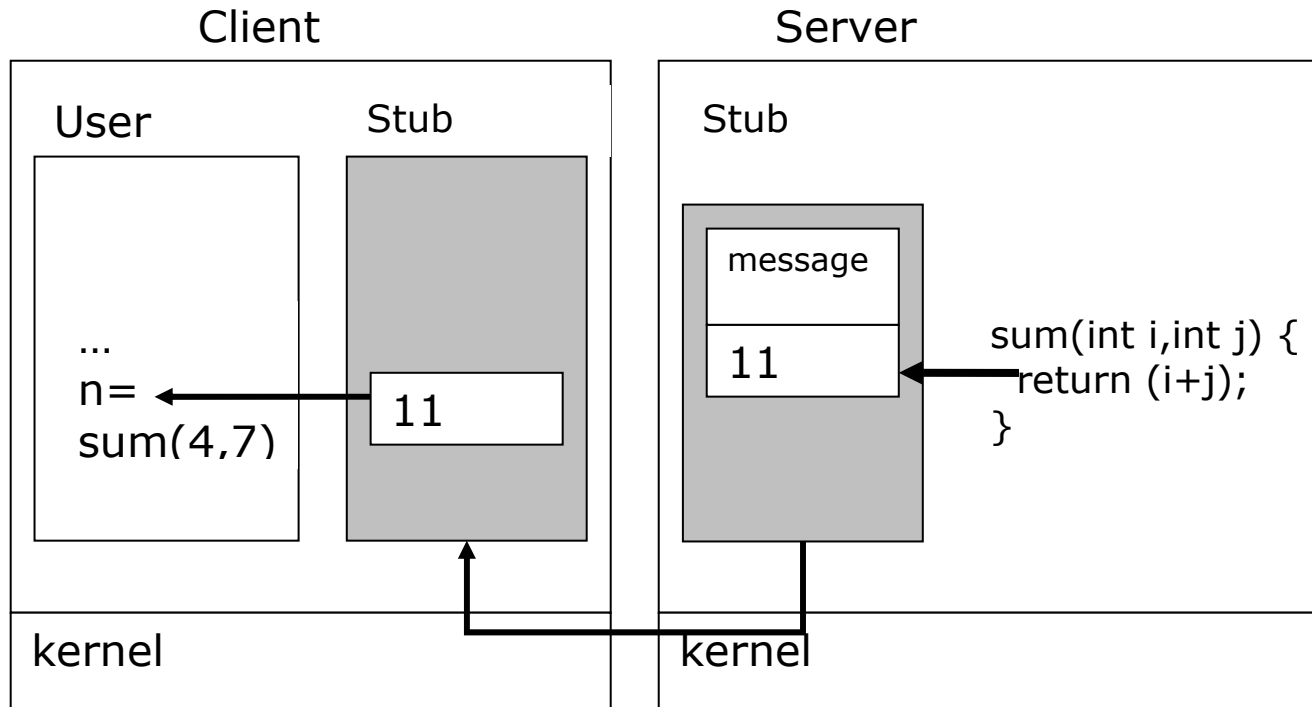
After the remote procedure has terminated its execution, the remote stub procedure marshals the result and transfers it to the caller, where it is made available to the caller of the remote procedure.

This idea is shown in the following picture:

1. procedure call



2. return result



There are an almost infinite number of formats possible.

To specify such a format, languages (interface definition languages) have been developed, i.e.:

- Suns ONC-RPC: XDR (eXternal Data Representation)
- OSF-RPC (DCE): IDL (Interface Definition Language)
- CORBA: IDL (Interface Definition Language)

One other possible format is **XML**, a new language that both humans and computers can read. XML-RPC uses XML as the marshalling format. It allows Macs to easily make procedure calls to software running on Windows machines and BeOS machines, as well as all flavours of Unix and Java, and IBM mainframes, and PDAs and mobile phones).

With XML it's easy to see what it's doing, and it's also relatively easy to marshal the internal procedure call format into a remote format.

2.2.XML-RPC Specification²

XML-RPC is a **Remote Procedure Calling protocol** that works over the **Internet** with fire-wall navigation.

An XML-RPC message is an **HTTP-POST request**. The **body** of the request is in **XML**. A procedure executes on the server and the **value** it **returns** is also formatted in **XML**.

Procedure **parameters** can be **scalars**, numbers, strings, dates, etc. and can also be complex **record** and **list structures**.

2.2.1. Requests

Here's an example of an XML-RPC request:

² © Copyright 1998-2002 [UserLand Software, Inc.](http://www.userland.com) (www.userland.com)
XML-RPC is a trademark of UserLand Software, Inc.

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181
```

header

```
<?xml version="1.0"?>
<methodCall>
  <methodName>
    examples.getStateName
  </methodName>
  <params>
    <param>
      <value>
        <i4>
          41
        </i4>
      </value>
    </param>
  </params>
</methodCall>
```

request as XML string

Header requirements

The **format** of the URI in the **first line** of the header is **not** specified. For example, it could be empty, a single slash, if the server is only handling XML-RPC calls. However, if the server is handling a mix of incoming HTTP requests, we allow the URI to help route

the request to the code that handles XML-RPC requests. (In the example, the URI is /RPC2, telling the server to route the request to the "RPC2" responder.)

A **User-Agent** and **Host must** be specified.

The **Content-Type** is **text/xml**.

The **Content-Length must** be specified and must be correct.

Payload format

The payload is in **XML**, a single **<methodCall>** structure.

The **<methodCall>** **must** contain a **<methodName>** sub-item, a string, containing the name of the method to be called. The string may only contain identifier characters, upper and lower-case A-Z, the numeric characters, 0-9, underscore, dot, colon and slash. It's entirely up to the server to decide how to interpret the characters in a methodName.

For example, the methodName could be the name of a file containing a script that executes on an incoming request. It could be the name of a cell in a database table. Or it could be a path to a file contained within a hierarchy of folders and files.

If the procedure call has parameters, the **<methodCall>** must contain a **<params>** sub-item. The **<params>** sub-item can contain any number of **<param>**s, each of which has a **<value>**.

2.2.2. Types

The following types are supported in XML-RPC:

Scalar <value>s

<value>s can be scalars, type is indicated by nesting the value inside one of the tags listed in this table:

Tag	Type	Example
<i4> or <int>	four-byte signed integer	-12
<boolean>	0 (false) or 1 (true)	1
<string>	ASCII string	hello world
<double>	double-precision signed floating point number	-12.214
<dateTime.iso8601>	date/time	19980717T14:08:55
<base64>	base64-encoded binary	eW91IGNhbid0IHJlYWQgdGhpcyE=

If no type is indicated, the type is string.

<struct>s

A value can also be of type <struct>.

A <struct> contains <member>s and each <member> contains a <name> and a <value>.

Here's an example of a two-element <struct>:

```
<struct>
  <member>
    <name>lowerBound</name>
    <value>
      <i4>18</i4>
    </value>
  </member>
  <member>
    <name>upperBound</name>
    <value>
      <i4>139</i4>
    </value>
  </member>
</struct>
```

<struct>s can be recursive, any <value> may contain a <struct> or any other type, including an <array>, described below.

<array>s

A value can also be of type <array>.

An <array> contains a single <data> element, which can contain any number of <value>s.

Here's an example of a four-element array:

```
<array>
  <data>
    <value><i4>12</i4></value>
    <value><string>Egypt</string></value>
    <value><boolean>0</boolean></value>
    <value><i4>-31</i4></value>
  </data>
</array>
```

`<array>` elements do not have names.

You can mix types as the example above illustrates.

`<arrays>`s can be recursive, any value may contain an `<array>` or any other type, including a `<struct>`, described above.

2.2.3. Response

Here's an example of a response to an XML-RPC request:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT
```

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>South Dakota</string></value>
    </param>
  </params>
</methodResponse>
```

Response format

Unless there's a lower-level error, **always** return **200 OK**.

The **Content-Type** is **text/xml**. **Content-Length** must be present and correct.

The **body** of the response is a **single XML** structure, a **<methodResponse>**, which can contain a single **<params>** which contains a single **<param>** which contains a single **<value>**.

The **<methodResponse>** could also contain a **<fault>** which contains a **<value>** which is a **<struct>** containing **two elements**, one named **<faultCode>**, an **<int>** and one named **<faultString>**, a **<string>**.

A <methodResponse> **can not** contain both a <fault> and a <params>.

Fault example

HTTP/1.1 200 OK

Connection: close
Content-Length: 426
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:02 GMT
Server: UserLand Frontier/5.1.2-WinNT

```
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>4</int></value>
        </member>
        <member>
          <name>faultString</name>
          <value><string>Too many parameters.</string></value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

2.2.4. Strategies/Goals

The following items influenced the design of XML-RPC:

- **Firewalls.** The goal of this protocol is to lay a compatible foundation across different environments, no new power is provided beyond the capabilities of the CGI interface. Firewall software can watch for POSTs whose Content-Type is text/xml.
- **Discoverability.** We wanted a clean, **extensible format** that's **very simple**. It should be possible for an **HTML coder** to be able to look at a file containing an XML-RPC procedure call, understand what it's doing, and be able to modify it and have it work on the first or second try.
- **Easy to implement.** We also wanted it to be an easy to implement protocol that could quickly be adapted to run in other environments or on other operating systems.

2.2.5. FAQ

The following questions came up on the UserLand [discussion group](#) as XML-RPC was being implemented in Python.

- The Response Format section says "The body of the response is a single XML structure, a <methodResponse>, which *can* contain a single <params>..." This is confusing. Can we leave out the <params>?

No you cannot leave it out if the procedure executed successfully. There are only two options, either a response contains a <params> structure or it contains a <fault> structure.

That's why we used the word "can" in that sentence.

- Is "boolean" a distinct data type, or can boolean values be interchanged with integers (e.g. zero=false, non-zero=true)?

Yes, boolean is a distinct data type. Some languages/environments allow for an easy coercion from zero to false and one to true, but if you mean true, send a boolean type with the value true, so your intent can't possibly be misunderstood.

- What is the legal syntax (and range) for integers? How to deal with leading zeros? Is a leading plus sign allowed? How to deal with whitespace?

An integer is a 32-bit signed number. You can include a plus or minus at the beginning of a string of numeric characters. Leading zeros are collapsed. Whitespace is not permitted. Just numeric characters preceded by a plus or minus.

- What is the legal syntax (and range) for floating point values (doubles)? How is the exponent represented? How to deal with whitespace? Can infinity and "not a number" be represented?

There is no representation for infinity or negative infinity or "not a number". At this time, only decimal point notation is allowed, a plus or a minus, followed by any number of numeric characters, followed by a period and any number of numeric characters. Whitespace is not allowed. The range of allowable values is implementation-dependent, is

not specified.

- What characters are allowed in strings? Non-printable characters? Null characters? Can a "string" be used to hold an arbitrary chunk of binary data?

Any characters are allowed in a string except < and &, which are encoded as < and &. A string can be used to encode binary data.

- Does the "struct" element keep the order of keys. Or in other words, is the struct "foo=1, bar=2" equivalent to "bar=2, foo=1" or not?

The struct element does not preserve the order of the keys. The two structs are equivalent.

- Can the <fault> struct contain other members than <faultCode> and <faultString>? Is there a global list of faultCodes? (so they can be mapped to distinct exceptions for languages like Python and Java)?

A <fault> struct **may not** contain members other than those specified. This is true for all other structures. We believe the specification is flexible enough so that all reasonable data-transfer needs can be accommodated within the specified structures. If you believe strongly that this is not true, please post a message on the discussion group. There is no global list of fault codes. It is up to the server implementer, or higher-level standards to

specify fault codes.

- What timezone should be assumed for the `dateTime.iso8601` type? UTC? localtime?

Don't assume a timezone. It should be specified by the server in its documentation what assumptions it makes about timezones.

2.3. Apache XML-RPC

To be able to concentrate to the application development, we use a Java implementation of the XML-RPC protocol. It is part of the Apache project.

[Apache XML-RPC](#) is a Java implementation of [XML-RPC](#), a popular protocol that uses XML over HTTP to implement remote procedure calls.

2.3.1. Client classes

Apache XML-RPC provides two client classes.

- [org.apache.xmlrpc.XmlRpcClient](#) uses `java.net.URLConnection`, the HTTP client that comes with the standard Java API
- [org.apache.xmlrpc.XmlRpcClientLite](#) provides its own lightweight HTTP client implementation.

XmlRpcClientLite is usually **faster**, but if you need **full HTTP support** (e.g. Proxies, Redirect etc), you should use **XmlRpcClient**.

Both client classes provide the same interface, which includes methods for synchronous and asynchronous calls. We will use the **synchronous** method.

Using the XML-RPC library on the **client side** is quite straightforward. Here is some sample code:

```
// create new client, parameter is the url of the server
XmlRpcClient xmlrpc = new XmlRpcClient ("http://rh:8080/RPC2");

// build parameter for the request
Vector params = new Vector ();
params.addElement ("some parameter");

// call the remote procedure
String result = (String) xmlrpc.execute ("method.name", params);
```

Note that `execute` can throw `XmlRpcException` and `IOException`, which must be caught or ignored by your code.

2.3.2. Server Side XML-RPC

On the server side, you can either embed the XML-RPC library into an existing server framework, or use the built-in special purpose HTTP server.

Let's first look at how to register handler objects to tell an XML-RPC server how to map incoming requests to actual methods.

XML-RPC Handler Objects

The org.apache.xmlrpc.XmlRpcServer and org.apache.xmlrpc.WebServer **classes** provide **methods** that let your **register** and unregister Java objects as XML-RPC **handlers**:

```
addHandler (String name, Object handler);
```

```
removeHandler (String name);
```

Depending on what kind of handler object you give to the server, it will do one of the following things:

1. If you pass the `XmlRpcServer` any **Java object**, the server will try to resolve incoming calls via **object introspection**, i.e. by **looking for public methods in the handler object corresponding to the method name** and the parameter types of incoming requests. The input parameters of incoming XML-RPC requests must match the argument types of the Java method (see [conversion table](#)), or otherwise the method won't be found. The return value of the Java method must be supported by XML-RPC. (We will use this kind of handler object.)
2. If you pass the `XmlRpcServer` an object that implements **interface** [org.apache.xmlrpc.XmlRpcHandler](#) or [org.apache.xmlrpc.AuthenticatedXmlRpcHandler](#) the `execute()` method will be called for every incoming request. **You are then in full control of how to process** the XML-RPC request, enabling you to perform input and output parameter checks and conversion, special error handling etc.

In both cases, incoming requests will be interpreted as **handlerName.methodName** with `handlerName` being the String that the handler has been registered with, and `methodName` being the name of the method to be invoked. You can work around this scheme by registering a handler with the name "\$default". In this case you can drop the `handlerName.` part from the method name.

Using the build-in HTTP-Server

The XML-RPC library comes with its own built-in HTTP server. This is **not** a general purpose web server, its only purpose is to handle XML-RPC requests. The HTTP server can be embedded in any Java application with a few simple lines:

```
// create Web-Server
WebServer webserver = new WebServer (port);
// add handler for a request
webserver.addHandler ("examples", someHandler);
```

A special bonus when using the built in Web server is that you can set the IP addresses of clients from which to accept or deny requests. This is done via the following methods:

```
webserver.setParanoid (true);           // deny all clients
webserver.acceptClient ("192.168.0.*"); // allow local access
webserver.denyClient ("192.168.0.3");   // except for this one
...
webserver.setParanoid (false);          // disable client filter
```

If the client filter is activated, entries to the deny list always override those in the accept list. Thus, `webserver.denyClient ("*.*.*.*")` would completely disable the web server.

Using XML-RPC within a Servlet environment (we do not use it in our example)

The XML-RPC library can be embedded into any Web server framework that supports reading HTTP POSTs from an `InputStream`. The typical code for processing an incoming XML-RPC request looks like this:

```
XmlRpcServer xmlrpc = new XmlRpcServer ();
xmlrpc.addHandler ("examples", new ExampleHandler ());
...
byte[] result = xmlrpc.execute (request.getInputStream ());
response.setContentType ("text/xml");
response.setContentLength (result.length);
OutputStream out = response.getOutputStream();
out.write (result);
out.flush ();
```

Note that the `execute` method does not throw any exception, since all errors are encoded into the XML result that will be sent back to the client.

A full example servlet is included in the package. There is a sample XML-RPC Servlet included in the library. You can use it as a starting point for your own needs.

2.3.3. Data types

The following table explains how data types are converted between their [XML-RPC representation](#) and Java.

Note that the automatic invocation mechanism expects your classes to take the primitive data types as input parameters. If your class defines any other types as input parameters (including `java.lang.Integer`, `long`, `float`), that method **won't** be usable from XML-RPC unless you write your own handler.

For return values, both the primitive types and their wrapper classes work fine.

XML-RPC data type	Java date type
<i4> or <int>	int
<boolean>	boolean
<string>	java.lang.String
<double>	double
<dateTime.iso8601>	java.util.Date
<struct>	java.util.Hashtable
<array>	java.util.Vector
<base64>	byte[]

2.4.XML-RPC Example

As an example of XML-PRC, we discuss a **calculator** with the basic operation “add”, “multiply”, “subtract” and “divide”.

First, we implement a **XML-RPC server**.

```
import java.util.Hashtable;
import org.apache.xmlrpc.*;

public class JavaServer {
    ...

    public Hashtable add (int x, int y) {
        ...
    }

    public static void main (String [] args) {
        try {
            // start XML-RPC server; Invoke me as <http://localhost:8080/RPC2>.
            System.out.println("Starting XML-RPC server ....");
            WebServer server = new WebServer(8080);

            // register our handler
            server.addHandler("calc", new JavaServer());

        } catch (Exception exception) {
            System.err.println("JavaServer: " + exception.toString());
        }
    } // main
}
```

A simple client could have the form:

```
import java.util.Vector;
import java.util.Hashtable;
import org.apache.xmlrpc.*;

public class JavaClient {
    // The location of our server.
    private final static String server_url =
        "http://localhost:8080/RPC2";
```

```

public static void main (String [] args) {
    try {
        // Create an object to represent our server.
        XmlRpcClient server = new XmlRpcClient(server_url);

        // Build our parameter list.
        Vector params = new Vector();
        params.addElement(new Integer(5));
        params.addElement(new Integer(2));

        // Call the server, and get our result.
        Hashtable result = (Hashtable) server.execute("calc.add", params);
        int sum = ((Integer) result.get("add")).intValue();

        // Print out our result.
        System.out.println("Sum: " + Integer.toString(sum));

    } catch (XmlRpcException exception) {
        System.err.println("JavaClient: XML-RPC Fault #" +
            Integer.toString(exception.code) + ": " + exception.toString());
    } catch (Exception exception) {
        System.err.println("JavaClient: " + exception.toString());
    }
}
}

```

The full server code is:

```
$ cat JavaServer.java
import java.util.Hashtable;
import org.apache.xmlrpc.*;

public class JavaServer {

    public JavaServer () {
        // Our handler is a regular Java object. It can have a
        // constructor and member variables in the ordinary fashion.
        // Public methods will be exposed to XML-RPC clients.
        System.out.println("Handler registered as 'calc'");
    }

    public Hashtable add (int x, int y) {
        Hashtable result = new Hashtable();
        result.put("add", new Integer(x + y));
        return result;
    }
}
```

```
public Hashtable sub (int x, int y) {
    Hashtable result = new Hashtable();
    result.put("sub", new Integer(x - y));
    return result;
}

public Hashtable mul (int x, int y) {
    Hashtable result = new Hashtable();
    result.put("mul", new Integer(x * y));
    return result;
}

public Hashtable div (int x, int y) {
    Hashtable result = new Hashtable();
    if ( y == 0 )
        result.put("div error", new Integer(0));
    else
        result.put("div", new Integer(x / y));
    return result;
}
```

```

public static void main (String [] args) {
    try {

        // start XML-RPC server; Invoke me as <http://localhost:8080/RPC2>.
        System.out.println("Starting XML-RPC server ....");
        WebServer server = new WebServer(8080);

        // register our handler
        server.addHandler("calc", new JavaServer());

    } catch (Exception exception) {
        System.err.println("JavaServer: " + exception.toString());
    }
}
}

```

\$