Programming is understanding.
- Kristen Nygaard (1926-2002)

Programming Project #3: Building and Optimizing a Simple XML-RPC System

Assigned: April 14
Due: April 28 by class time (i.e., 12:30pm)
The project should be accomplished in a group of up to three students.

Overview

Remote Procedure Call (RPC) is one of the most widely used programming paradigms for building distributed systems. Although the basic principles and architecture of RPC have been introduced several decades ago, RPC implementations have gone through multiple evolutionary steps. For example, in Programming Project #2, you had an opportunity to experiment with RMI, an object-oriented implementation of the RPC mechanism for Java. The emergence of Service Oriented Architecture and Web Services has fostered further developments in RPC evolution. To meet the demands of interoperability, some Web Services utilize RPC communication style to encode their messages in XML and use HTTP for network communication. Such XML- and HTTP-based RPC mechanisms provide convenient building blocks for several major Web services standards such as SOAP.

Despite the wide use of RPC in building a variety of distributed systems, many find various aspects of this technology somewhat mysterious. This project aims at helping you gain a solid understanding of several major design and implementation principles of RPC and Web services. You will build a simple but fully functional XML-based RPC system with a mapping to the Java language. Through this system, one should be able to invoke remotely any *public static* method of a Java class. We will call this system VT RPC.

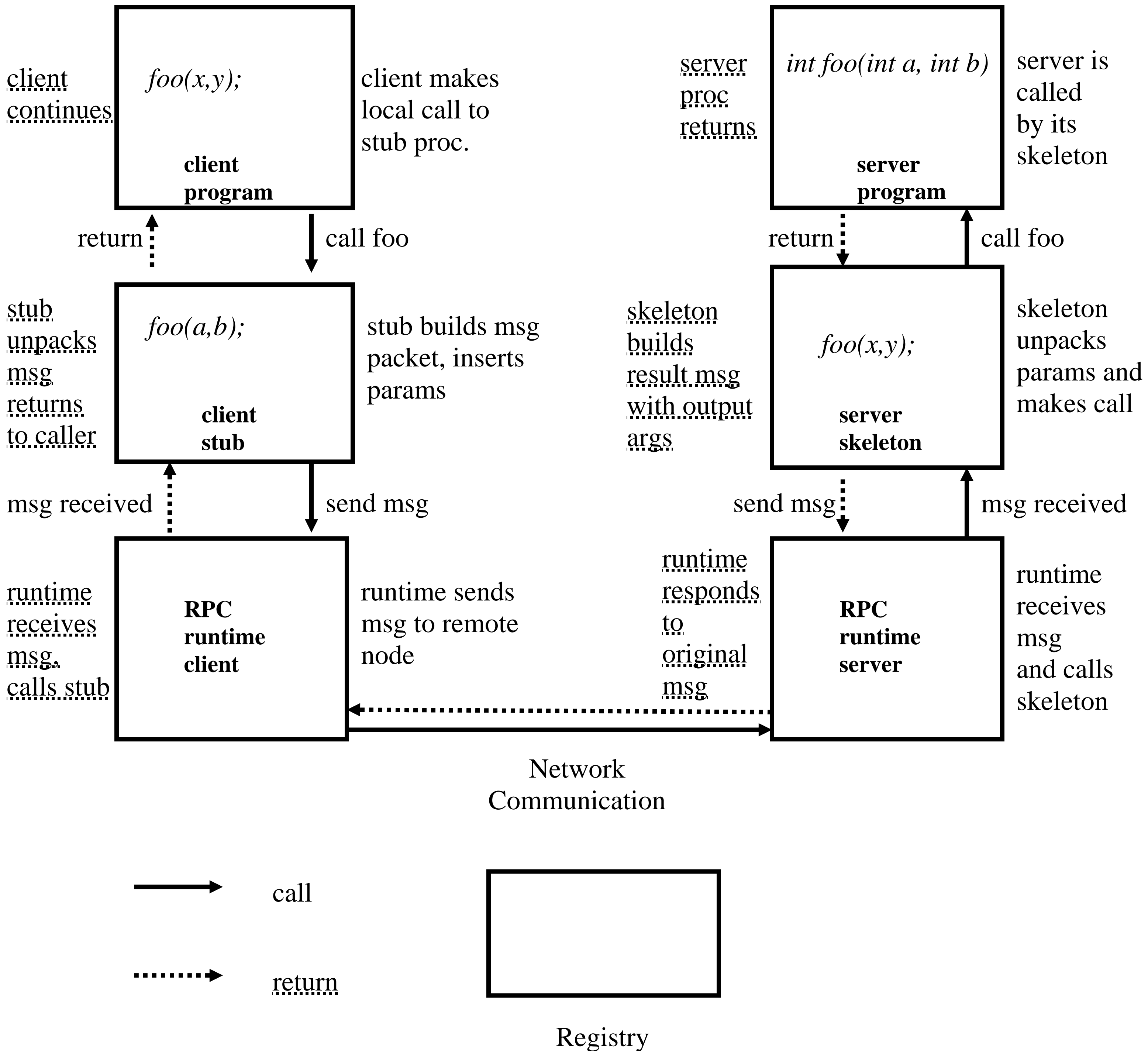
You are encouraged to reuse as much as possible the code that you wrote for your first and second programming projects. If the composition of your team has changed for this project, you can reuse the code written by any member of your group.

General Description

Consider a standard RPC architecture that we have reviewed extensively in class. This architecture has the following layers:

1. Client program (Programming API or language mapping)
2. Client Stub
3. Client Runtime
4. Server Runtime
5. Server skeleton
6. Server Program
7. Registry

The following diagram shows these layers schematically:



VT RPC will be simpler than that having fewer layers. Specifically, it won't have client stub and registry. The following diagram shows the layers of VT RPC:

client continues

foo(x,y);

client program

client makes local call to Transport

call foo

msg received

server proc returns

int foo(int a, int b)

server program

server is called by its skeleton

return

call foo

skeleton builds result msg with output args

foo(x,y);

server skeleton

skeleton unpacks params and makes call

send msg

msg received

Transport receives reply, unpacks XML, returns to caller

Transport

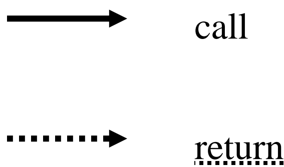
Transport builds XML message that represents the call; sends it to the server

runtime responds to original msg

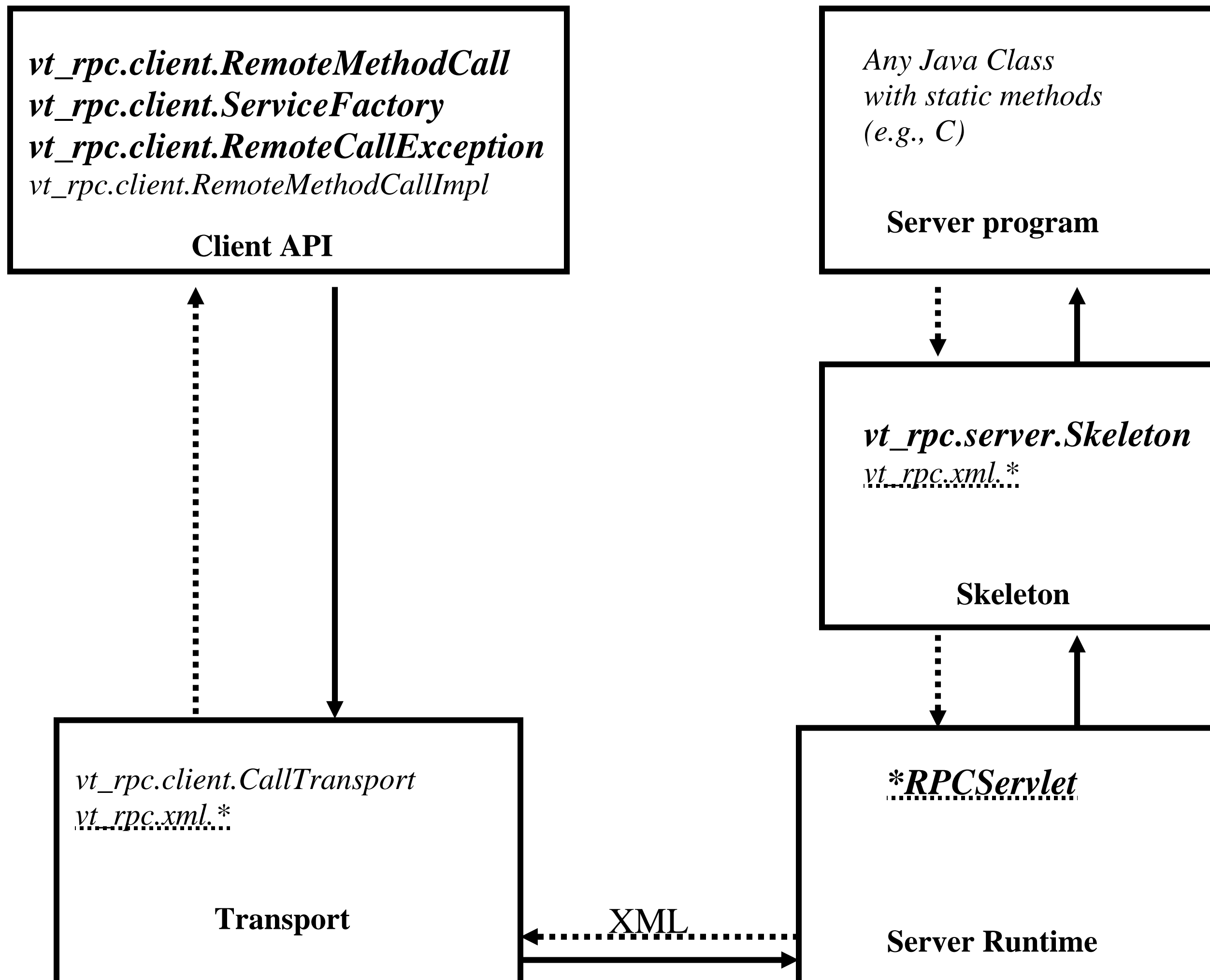
Server runtime

runtime receives msg and calls skeleton

HTTP Communication



Rather than implementing all the functionality from scratch, you'll be provided with some code and would have to fill in the missing pieces. The following diagram shows the Java classes of VT RPC:



Legend:

Class is fully implemented

Class is partially implemented

Class is not implemented

Class has to be automatically generated (using Castor)

You can find all the project files in **VT_RPC.JAR**
Save it to disk and uncompress its content.

The *src* directory contains VT RPC Java source files.
The *xml* directory contains XML grammar files.
The *castor* directory contains all the Castor framework files required for XML code generation and at runtime. (All the Castor and other 3rd party API jars that the generated classes reference at runtime can be found in this directory).

Implementation Details

You can follow the following step-by-step directions to complete the project:

1. Generate Java code for marshaling and unmarshaling XML data. VT RPC uses XML as a format for sending parameters to the server and returning the result back. Castor provides an open-source code generator that, given an XML grammar, produces XML processing code and eliminates the need for parsing XML documents explicitly. (See the notes from the XML lecture for details).
   The Castor code generator takes an XML grammar file as input. This grammar must be in XML Schema (.xsd) format. For this project, we will need three schema files: *vt_rpc_call.xsd, vt_rpc_reply.xsd, and vt_rpc_values.xsd*. The first two files share common content described in *vt_rpc_values.xsd*, which describes the values that could be taken by the parameters and the return value of a remote call.
   Among the files provided for this assignment, you will find three grammar files in .dtd format:
   a. *vt_rpc_call.dtd*
   b. *vt_rpc.reply.dtd*
   c. *vt_rpc_values.dtd*

   In addition, you will find *vt_rpc_values.xsd* file, which is an already translated version of *vt_rpc_values.dtd*. These two files are provided to help you understand the differences between .dtd. and .xsd formats so that you'd be able to translate *vt_rpc_call.dtd* and *vt_rpc_reply.dtd* to .xsd format as well.
   Hint: you should use the statement
   "<xs:include schemaLocation="vt_rpc_values.xsd"/>" to include *vt_rpc_values.xsd* into both *vt_rpc_call.xsd* and *vt_rpc_reply.xsd*.
   The resulting .xsd files can be used as input to the Castor code generator, and the syntax for running this generator is
   java org.exolab.castor.builder.SourceGenerator -i grammar_file.xsd -f -package somePackage
   The xml directory contains several batch scripts that you might find useful. However, first you'd have to adjust some settings in these scripts for your system. The suggested package name for the generated Java files is *vt_rpc.xml*.
2. Complete the code for *vt_rpc.client.CallTransport* and *vt_rpc.client.RemoteMethodCallImpl* using the comments in the code for directions. Notice that at this point, you can directly call *vt_rpc.server.Skeleton* from *vt_rpc.client.CallTransport* to test the functionality of your system without any network communication by using method *sendRequestAndReceiveReplyTest.*
3. Write a servlet capable of processing HTTP POST requests that contain XML documents representing VT RPC calls.
4. (Optional—your Servlet Container might be fine as it is) Fix your Servlet Container implementation to handle your VT RPC servlet. Specifically, you'll need to provide a proper implementation for *HTTPServletRequest.getInputStream()* method. Your servlet will use this method to read the body of the POST request. If you read the body of the post request and store it locally, you should make this method return a stream that would read the body from your stored copy. You might want to implement *HTTPServletResponse.getOutputStream()* method, but this is not really necessary. One can use the *OutputWriter* returned by

HTTPServletResponse.getWriter() and use its *println* method to send back the reply serialized in XML.

5. Change the code *vt_rpc.client.CallTransport* to properly form a POST request, send it to the servlet, and receive a reply back.
6. Test your implementation with *test.TestC* class and build your own tests if you like. You can use JUnit for your tests.
7. Find some ways to optimize your implementation.
   In your report you have to describe at least three different ways you have found to optimize your implementation, but you should implement only *one* of these optimizations.
   In designing and implementing your optimizations, you might want to apply some of the lessons you've learned in implementing different caching schemes in Programming Project #2.
   Warning: one optimization that is not valid is to assume that any remote method you invoke through VT RPC is idempotent. In layman terms, idempotency with respect to method calls implies that a method will return the same results if invoked multiple times with the same arguments. For example, the *factorial* method taking the integer argument 3, will always return 6:
   int res1 = factorial (3);
   int res2 = factorial (3);
   Idempotency is a common feature in functional languages. However, in a language such as Java, idempotency can never be assumed. So even if you know that the method will return the same result if given the same arguments, it is not valid to cache the result to avoid performing the actual remote method invocation. Idempotency is neither a property of the majority of Web Services.

How to turn in

- As for the previous projects, we'll use the Curator system to submit the project. The submission instructions are identical to the ones used in the first two programming projects.
- Your submission must be originated before the assignment submission deadline: no extensions will be granted.

Grading

- 50% your code; 50% your report that describes your implementation, testing strategies, and optimizations.
- Total: 100 points.

One more point: if you are still interviewing for a job or an internship, do not forget to mention during your interviews that in one of your classes you are working on a programming project in which you are building and optimizing your own Web services infrastructure (based on the RPC model), using Castor to automate XML data binding, and your own Servlet engine for deployment. ☺