University of Illinois at Chicago
EECS 573 - Distributed computing systems
Prof. Shatz

A complete implementation of a fault-tolerant RPC library respecting the "exactly once" call semantics

FIRST DRAFT (November 7th, 2000)

By Daniele Paolo Scarpazza

≪Söta mìa a turà, làatìff,
ta vèdet mìa che so bèa a dré ?≫.
A typical reply by a RPC Server stub from Crema.

1 Introduction

This paper is the description of both the results and the process of writing a simple but complete RPC implementation which enforces the "exactly once" semantics. As we seen during the lectures, an RPC implementation based on a unreliable message passing mechanism can exhibit different type of behaviours and guarantee different objectives, depending on which type of call semantics was chosen by the designer:

- the *maybe* call semantics does not offer any guarantee about the result of the call; if the request message from the client or the reply message from the server should get lost, there is no way to successfully complete the call;

- the *at least once* call semantics guarantees about that, upon return from call, at least one reply was received from the server and that the remote called routine was executed at least once. To enforce this semantics some additional modifications are required, we have to provide a limited amount of intelligence on board of the client, in order to keep it sending (in association with a timeout) request messages until a reply message manages to successfully reach the client. The main advantage of this call semantics is that, when control is yielded back to the caller on the client side, at least one successful execution is guaranteed to have taken place; the main disadvantage, on the opposite side, is that the called server routine could have been executed many times, thus rendering this semantics safe only for *idempotent* procedures.

  One could argue that by using a pure functional programming style, everything could be implemented by using idempotent procedures only (the procedures are thought as stateless machines that, given an input, yield an output and have *no side-effects* at all) but I would like instead to put emphasis on the fact that when implementing a client-server database application, the side-effects are the core of what we are interested in and the functional aspect of the remote procedures is almost unimportant. Let us consider as an example the case of "deposit", a remote procedure that allows to deposit a given amount of money on a given bank account: The actual update of the account record in the database is the side-effect and practically there is no output, if we do not consider some completion status as output. The database itself is a result of side-effects, therefore refusing the possibility of side-effects means in this case refusing the feasibility of a database application.

Once stated that we indeed need non-idempotent procedures, we also need to find a suitable call semantics that allows us to execute them remotely in full safety;

- the *exactly once* call semantics guarantees that, upon return, the called procedure was executed once and once only. I will not need to explain why it is not desirable that functions like the "deposit" are executed multiple times when their execution is desired only once, it seems to me quite self-evident. To enforce the *exactly once* call semantics, additional intelligence is required on both the client and the server side, basically the client has got to number each request message with a progressive identifier so that the server has the ability to recognize multiple repeated messages that correspond to the same original request (and therefore have the same identifier): In this case the server avoids repeating the execution and instead replies to the client with the result of the procedure unique execution which was stored.

This paper will deal with this last type of call semantics and will explore with all the details that are required to have a functioning implementation enforcing it.

2 Objective

The resulting program wants to be both a working implementation (something that really functions and, at least in principle, could be actually used in a real-world application) and, at the same time, a sort of teaching aid, that could be used to help students understand the basic RPC concepts and implementation details.

3 Design Choices

- Since the program is supposed to be used both by instructors on PCs in lecture rooms and by students at home or in the labs, I decided to write the programs under the Microsoft Windows operating system. Another choice could have been to code them for a UNIX-like operating system, but I decided to reject this possibility (even if the development would have been more straightforward in this case) because the typical student users could not have a working installation of a UNIX-like OS at home (Linux, FreeBSD, ...) and/or the skills required to compile the programs there.
- Due to the "didactic" orientation of the application, I want to provide the user with the ability to interactively monitor and control the program's execution on a step-by-step basis. This could be useful both during a

lecture, when the instructor needs to interleave the program use with explanations and during individual use, when these feature allow a self-pace learning.

- I chose the Microsoft Foundation Classes as a framework to implement the applications' GUI because it provides a look and feel which is well known by the user, letting him focus on the conceptual part of the application without wasting time in understanding detailed command-line options or the intricacies to be provided on a command line-based interface.

- Tough programming for Windows, I want a clean architecture, where most of the code is portable to any other socket-based networked operating systems. Therefore most of the code shall be OS-independent, while the OS-dependent sections will be in separate layers. It should be easy to port the application by isolating and rewriting the only OS-dependent layers.

- A complete RPC implementation requires some sort of a language to define the signatures of the remote procedures, and usually this language is translated in to a number of skeleton implementation files which it is the programmer's responsibility to finish or to include and use in the final application. We will not be exception: I will provide a formal definition for a simple interface definition language and I will make available a corresponding compiler. Of course, since this is a distributed systems course and not a course on languages and compilers, we are not going to put extreme emphasis on the IDL compiler design or on language issues.

- As a network programming API, we will use the version of the sockets available under Windows (the so-called *winsock*), trying to avoid any call that has been provided by Microsoft and was not present in the original API from Berkeley.

- As a protocol suite we will use TCP/IP, but we will not use the TCP in fact since it already provides reliable, error-free, stream-oriented communications. I will use instead the UDP protocol, a connectionless, unreliable, message-based protocol. Using this protocol I have a complete control over what is actually transmitted and my application has to embed all the "intelligence" required to deal with lost packets, packets received in a random order and with an arbitrary delay. Using the TCP protocol I wouldn't have the possibility to deal with these issues, since they are resolved *a priori* by the protocol implementation itself, and the implementation would have been much poorer, from a didactical point of view.

- As a solution for the client to find the server I will use of broadcasting. It is possibly a less powerful solution with respect to its most common counterpart, the *binding server*, but it is also extremely simple to develop and relieves the programmer (me!) from the time-consuming activity of writing a separated application. Furthermore, this solution is inherently distributed and is free from single-point failure problems.

4 A detailed look over our RPC architecture

The following figure depicts the typical structure of a RPC-based system

1. the **user application** is an autonomously executable program which invokes a set of predefined functions which the designer decided to implement implemented via RPC. It is obtained compiling together the application programmer source files and the client stub source files, and linking the result to the RPC client libraries. Since the client stub exports a set of calls which are exactly the ones required by the application, and it exports them as the real implementation of that call would be local, there's no way for the application developer to realize that he is in fact calling a remote procedure and not a local one.

   Our implementation is accompanied by an example application which wants to be a very limited abstraction of a bank accounting system. As you can see the only allowed type is the "int", we will better discuss in the next chapters this and the other limitations present both in our IDL compiler and in our RPC runtime libraries; at the moment what you may want to know is that –for completeness purposes– the list of exported functions for the `account` service (which is here reported) has been prepared trying to exploit all the degrees of freedom allowed by the IDL/RPC implementation; you will notice that there are functions with and without input parameters and with and without return values:

The `account` service:

- `void create_account    (int account_number)`
  This function will create a new account with the given number and set its balance to zero. Note that this function takes a single parameter as input and has no return value.

- `int get_balance (int account_number)`
  This function will return the current balance for the account corresponding to the given number. Note that this function takes one argument and has a return value.
- `int withdraw (int account_number, int amount)`
  `int deposit (int account_number, int amount)`
  This two functions have similar behaviour, they have two input parameters, the account number and the amount, they perform a deposit or withdraw operation on the account identified by that number and return its new balance after the operation has been performed.
- `int get_customer_count(void)`
  This function takes no input parameters and returns a number which tells the number of accounts currently open on the system. This function has been included just to have also a representative of the functions which take no arguments.

---

2. the **client stub** will be basically constituted by a set of functions whose signature is exactly the same chosen by the application designer and whose body is constituted by a call to our RPC client libraries. In our implementation, which takes into account only the 'C' language, the client stub is constituted by two files:

- `account_header.h` – the common header file
  This file contains no more than the prototypes for the defined functions, like in the following way:

  ```
  void create_account(int account_number);
  int get_balance();
  int withdraw(int account_number, int amount);
  int deposit(int account_number, int amount);
  ```

  This file will be automatically written by the IDL compiler
- `account_client_stub.c` – the client stub implementation file
  This file will contain, for each defined remote procedure, the actual code required to perform the remote call, like in the following example. Note that we introduced some sort of a standard way to pack the input arguments together and to pass the reference for the output value; the example shows how this way works for the `deposit` procedure, that code is what we would like the IDL compiler to generate for the given procedure: all the parameters are put in an array whose pointer is passed to the `rpc_call` library function, which takes care of the actual execution of the remote call and of the the output value, which is put it in the location whose pointer is provided.

```
int deposit(int account_number, int amount)
{
 int return_value;
 int parameter_array[2];
 parameter_array[0] = account_number;
 parameter_array[1] = amount;
 rpc_call("account","deposit", 2, parameter_array,& return_value);
 return return_value;
}
```

---

3. the **RPC client libraries** will contain all the code required, at run time, to initialize the RPC communication, to find out the appropriate server able to carry out the desired calls, to take care of all the client-side parameter marshalling activities, to actually send to the server the requests for every call and wait for the responses, possibly repeating requests if some internal timeout is elapsed and there's the reasonable belief that some packets were lost by the network. After a careful simplifying work, we decided to export to the upper layers one single function, whose prototype is the following one:

   `rpc_call(`$\langle Service \rangle$,$\langle Procedure \rangle$, $\langle ParamCount \rangle$,$\langle Parameters \rangle$,$\langle Return \rangle$`);`

   where:

   - $\langle Service \rangle$ is a string containing the name of the service which provides the call implementation; this name is required because a server could implement a different number of services and two service could provide procedures with the same name; in our example the service name will be `account`.
   - $\langle Procedure \rangle$ is a string containing the name of the procedure that we want to call, for example `deposit`;
   - $\langle ParamCount \rangle$ is the number of input parameters that the client stub would like to pass: note that the RPC client library does not know which is the signature of the called function until the call is tried for the first time, so it should rely on the one provided by the client stub. If both the client stub and the server skeleton have been compiled from the same version of the IDL definition file, a mismatch could not happen, otherwise an error message will be issued by an appropriate error handling manager that we will implement.
   - $\langle Parameters \rangle$ this is an array of integer values which contain the parameters; passing a NULL pointer and zero as the $\langle ParamCount \rangle$, means that the function doesn't take any input parameters;
   - $\langle Return \rangle$ this is a pointer to the return value; passing a NULL pointer, means that the called function is not supposed to return any output value;

We decided not to provide other functions, in particular initialization or final destruction functions in order to keep the IDL compiler as simple and modular as possible and, most important to preserve **access transparency**: since the application programmer is not asked to call any initialization function before calling any local procedure, we will not ask him to do that before calling the remote ones.

4. the **application server** is an autonomously executable program which is obtained compiling together the server skeleton, the server servant and linking the result to the RPC server libraries. Its role will be to actually carry out the requests of the different clients at run time.

5. the **server skeleton** is a file initially generated by the IDL compiler but that should be completed by the programmer; in our example it will be called `account_server_skeleton.c`.

   The initial contents of the file is a list of empty procedure bodied, like the following:

   ```
   int get_balance(int account_number)
   {
    /* Put here your implementation code */
   }
   ```

   The programmer is supposed to "fill in the blanks", that is to say provide the actual code required to carry out the actions that the procedures are required to do.

6. the **server servant** is another file automatically generated by the IDL compiler – in our example it will be called `account_server_servant.c` – which needs not to be modified, unless special features are required. It contains:

   - the **procedure wrappers**, one for each defined remote procedure, which are functions that receive their parameters in a standard way, with a fixed signature, so that they can be called by the server procedure in the same standard manner, even if the desired remote procedures to be called have different signatures; here is an example of the function wrapper for the `get_balance` procedure:

     ```
     unsigned int get_balance_wrapper(void * pCallData)
     {
      int * parameter_array = (int*) pCallData;
      int account_number = parameter_array[0];
      return get_balance(account_number);
     }
     ```

   - the **server main function**, which does nothing more than registering all the procedures that the server itself will make available to the clients and entering the main server loop. Both these activities are performed with the help of functions provided by the RPC server library. Here is reported a sample server main:

```
int main(int argc, char ** argv)
{
 rpc_register_procedure("account","create_account",create_account_wrapper, 1, 0);
 rpc_register_procedure("account","get_balance",get_balance_wrapper, 1, 1);
 rpc_register_procedure("account","withdraw",withdraw_wrapper, 2, 1);
 rpc_register_procedure("account","deposit",deposit_wrapper, 2, 1);
 rpc_register_procedure("account","get_customer_count",
                                   get_customer_count_wrapper, 0, 1);
 return rpc_server_loop();
}
```

7. the **RPC server libraries**, which take care of all the server-side parameter marshalling, performing also all the bookkeeping activities required to provide a reasonable amount of fault-tolerance features. After a deep abstraction work, analog to the one made for the client-side –as you can see from the previous example– we decided to ask to the libraries to provide only two functions:

   - `rpc_register_procedure(`$\langle$*Service*$\rangle$,$\langle$*Procedure*$\rangle$, $\langle$*Wrapper*$\rangle$, $\langle$*ParamCount*$\rangle$,$\langle$*ReturnCount*$\rangle$`);`

     where:

     - $\langle$*Service*$\rangle$ is a string containing the name of the service;
     - $\langle$*Procedure*$\rangle$ is a string containing the name of the procedure that we want to register;
     - $\langle$*Wrapper*$\rangle$ is a pointer to the function which implements the wrapper of the function we want to register;
     - $\langle$*ParamCount*$\rangle$ is the number of input parameters that the procedure accepts. This is the number against which the incoming requests are verified to make sure they are providing the right number of parameters;
     - $\langle$*ReturnCount*$\rangle$ this is a number which indicates how many values the procedure returns, in our implementation this parameter can only assume the 0 or 1 values;

     This function is used to register into the server all the exported procedures, and to declare how many input and output arguments they accept and return. The RPC server libraries, as a consequence of a call to `rpc_register_procedure`, adds a new entry to its current table of procedures and prepares to allow requests from clients that want to invoke that procedure;

   - `rpc_server_loop();`
     after the procedure registration it is time to actually start up the service, accepting incoming requests and responding to them. This is accomplished by calling the `rpc_server_loop`. This function is **not** supposed to return, since the server, once started, should not

interrupt its operation unless it is externally stopped (for example with a kill signal).

5 My Interface Definition Language

In order to make our RPC implementation complete, we will not forget to provide an interface definition language and its compiler. The particular dialect of IDL we are providing will be called "ScarpazIDL", and will be a very simplified version of IDL if compared to languages already available for real-world applications (e.g. the CORBA IDL by the Object Management Group, the MIDL by Microsoft).

Since we are not preparing a project for a "Languages and compilers" course, we are not focusing on the language expressing power or the technology of the compiler, nevertheless we will give a formal definition of our simple language:

$$
\begin{array}{rcl}
\langle DefinitionFile \rangle & ::= & \langle ServiceDefinition \rangle \hookleftarrow \langle ProcedureDefSeq \rangle \\
\langle ServiceDefinition \rangle & ::= & \texttt{interface}\ \langle CIdentifier \rangle \\
\langle ProcedureDefSeq \rangle & ::= & \epsilon \mid \\
 & & \langle ProcedureDefinition \rangle \hookleftarrow \langle ProcedureDefSeq \rangle \\
\langle ProcedureDefinition \rangle & ::= & \langle Type \rangle\, \langle CIdentifier \rangle\, (\, \texttt{void}\,) \hookleftarrow \mid \\
 & & \langle Type \rangle\, \langle CIdentifier \rangle\, (\, \langle ParamList \rangle\,) \hookleftarrow \\
\langle ParamList \rangle & ::= & \langle NonVoidType \rangle\, \langle CIdentifier \rangle \mid \\
 & & \langle NonVoidType \rangle\, \langle CIdentifier \rangle\, ,\, \langle ParamList \rangle \\
\langle Type \rangle & ::= & \texttt{void} \mid \langle NonVoidType \rangle \\
\langle NonVoidType \rangle & ::= & \texttt{int} \\
\langle CIdentifier \rangle & ::= & \text{“any C programming language identifier”}
\end{array}
$$

where $\hookleftarrow$ represent the carriage return character.

In the following listing is reported the document that describes the functions of the `account` interface that we have already introduced:

```
interface account

void create_account      (int account_number)
int  get_balance         (int account_number)

int  withdraw            (int account_number, int amount)
int  deposit             (int account_number, int amount)

int  get_customer_count  (void)
```

The language is very limited, since functions can have only integer type input and output parameters, and they can have only one zero or one return parameters, anyway these limitations were introduced to save time only, and do not cause any loss of generality: my ScarpazIDL compiler could have been easily written in order to accept procedures with an arbitrary type and number of input and output parameters, the compiler would have been only much longer to write.

6 Development phases

The first piece of software to be developed was the ScarpazIDL compiler. I decided to finish it first because it was needed to prepare include and implementation files needed by the applications. Its implementation, once it was well defined by the input and the output files, was rather straightforward.

Then I went to the client and the server applications. I would like to emphasize the similarities between the development style I employed and the biological development of an organism: I started from a single, very simple, indifferentiated "organism" and then I made it more and more complex; then I decided to split it in the two client and server separate "cells". Finally the RPC library code will be brought out from both the programs and two dynamic link libraries (the client service library and the server service library) will be deployed. I will finish the two parts separately.

1. Simple single program, all the code in the same place
   Description: A single very simple program is created, whose abilities are simply to send an example message and (at the same time) listen for incoming messages. At this point there is no difference between client and server, they bind to different ports (known in advance), both the programs provide a button to send the message and display an information window when the message is received. To test the program two instances of it are executed at the same time.
   Objectives:
   (a) Provide a correct linkage with the winsock libraries and a correct networking environment initialization for the UDP protocol;
   (b) Providing the program with the basic communication capabilities (binding, sending and receiving point-to-point and broadcast messages);
   (c) Providing the program with the multithreading abilities;
2. Single program, separated communication object:
   Description: The communication abilities are placed all together in a separated RPC_Socket object.
   The RPC_Socket object embeds all the abilities which were introduced in the previous step: initialize the lower-level winsock socket, send and receive a message, keep waiting for a reply, ...
   Objectives:
   (a) Enforcing modularity, reusability and functional separation;
   (b) Avoiding code redundancy;
3. Separate client and server application:
   Description: The client application and the server application are now two different sets of source code files and two different executable files.
   Objectives:

(a) Prepare code for functional differentiation between the two programs;

4. Communication object differentiation:
Description: Now that the client and the server are separated, the two sockets can develop different abilities. The client RPC socket becomes able to find the server (the servers) with a broadcast message. Both the RPC sockets specialize in handling the required type of messages.
Objectives:

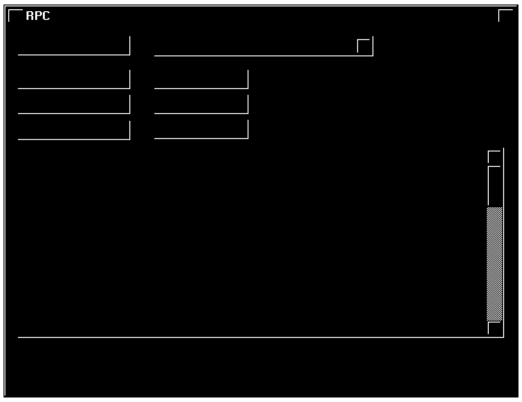
   (a) Make the RPC client socket able to pack request messages and unpack response message;

   (b) Make the RPC server socket able to unpack request messages and pack response message;

   (c) Create the structures and the procedures that allow at runtime registration and invocation of the exported procedures;

   (d) Create the code sections required to enforce the exactly once call semantics;

5. Account manager service development:
Description: The client becomes a "real" application, providing the user with a set of graphic controls useful to run the application domain functions. The server "puts the meat around the skeleton", thus implementing a very simple bank account management.
The results of this phase in the GUI can be seen in the top part of the following picture: The user can select from a combo box the desired operation, the composition of the dialog box changes dynamically depending on that choice. For example if the user selects the withdraw operation, the program prepares two labeled entry field, one for the account number and one for the amount and a read-only test field for the final balance. The number, label and purpose of the fields change in accordance with the current selected operation.



Objectives:

(a) creating on the client side the user interface and features to collect arguments from and show results to the user;

(b) implementing on server side the bank account manager with a simple data structure which keeps at runtime only (no persistence is required) the account information;

6. Inspection interface development:
Description: Both on client and on server side, a set of graphic controls is implemented to let the user control the programs' evolution step by step. Both the RPC client and server sockets check now a set of debug flags before performing any action.

This is the implementation phase at which I arrived, Nov. 7th, 2000

The user has now three VCR-like buttons as depicted in the following figure: the first button enables the step-by step mode, the second performs one step and the third disables this mode. In step-by-step mode, before performing any action the program wait the user to press the middle button (the "play" button). This way the user can take all his time to understand what is the state of both client and server before the action, what happens during the action and what are its effects on both sides. If the user is not anymore interested in what happens on the client or on the server side, he can press the "fast forward" button: now the program will just perform the actions it want to do without any delay.

Objectives:

(a) provide an easy to use mechanism for the learner to control the execution pace;

7. RPC code brought out from client and server:
Description: The RPC code which implements the calls required by the client and its symmetrical counterpart which implements the RPC

calls required by the server are isolated from the client's and server's code respectively. The isolated code sections become separate entities: the *RPC client dynamic link library* and the *RPC server dynamic link library*. The client and the server will import from them the RPC calls they formerly used to run in the same executable.

Objectives:

(a) Enforcing modularity and reusability, and ease of redistributability;

(b) Making the RPC libraries available to other applications;

7 Communication protocol

7.1 Request messages

1. REQUEST_IS_THERE_ANYONE
   This message is broadcasted as first message by the client. Any active server should answer with a RESPONSE_HERE_I_AM message. The client waits for this response, extracts the server address from the response message and starts communicating with that server. If more than one server answers and multiple responses are received, the first response is chosen and the others are ignored, the first answering server is the one which has the minimum response delay and it is assumed that it is the best choice, for network delay and workload.

Content:	"S_RPC_R"	0
Size (bytes):	←7→	←1→

2. REQUEST_ARE_YOU_ALIVE
   This message could be sent anytime by the client to its chosen server, for a number of reasons: to verify if the server is up and running at a particular time, to calculate the network latency, ... The recipient server should answer, with a RESPONSE_I_AM_ALIVE message.

   In order to avoid confusion due to orphan responses and to avoid a delayed response for a previous request to be taken as a response to the current one, the client encloses a random number (R#) in the request and waits for an answer that quotes that particular number.

Content:	"S_RPC_R"	1	R#
Size (bytes):	←7→	←1→	←1→

3. REQUEST_EXECUTE_PROCEDURE

   This is the message used by the client in order to start a remote procedure execution. It can be repeated as many times as wanted, if the client (in association with a timeout mechanism) has a reasonable suspect that its request message was lost.

Content:	"S_RPC_R"	2	Seq#	Srvc_id	Proc_id	
Size (bytes):	←7→	←1→	←2→	←var→	←var→	

	In#	Out#	$Param_1$	...	$Param_n$
	←1→	←1→	←4→	...	←4→

where:

- Seq# is the sequential number which identifies the request. It is a 16-bit little-endian word. It is required in order to enforce the exactly one semantics. It should be incremented at each new request.
- Srvc_id is the service identifier, a non-zero-length null-terminated ASCII string. It is used to identify the interface which is to be called. Its name should match the service name specified in the interface definition language file, in the `interface` statement. In our example application, this string has the `"account"` value.
- Proc_id is the procedure identifier, a non-zero-length null-terminated ASCII string. It is used to identify the procedure that the client wants to invoke. Its name must match the name of the procedure specified in the interface definition language file.
- In# is the number of input parameters taken by the procedure. It is a byte. It is used to perform a runtime signature check, in order to avoid version mismatch between the client and the server.
- Out# is the number of output parameters yielded by the procedure. It is a byte. It is used to perform a runtime signature check, in order to avoid version mismatch between the client and the server. In our implementation, this number can only assume the 0 or the 1 value.
- $Param_1$ ... $Param_n$ are the input parameters for the procedure. They are 32-bit signed integers sent as 4 bytes from the most significant to the least significant one.

7.2 Response messages

1. `RESPONSE_HERE_I_AM`
   This message is returned by any server upon the receipt of a `REQUEST_IS_THERE_ANYONE` message from a client. The recipient of this response message is the same client which originated the request message. The client is supposed to extract the server address from the UDP message header, therefore there is no need to add supplemental information in the body of the message.

Content:	`"S_RPC_A"`	0
Size (bytes):	←7→	←1→

2. `RESPONSE_I_AM_ALIVE`
   For every request message of the `REQUEST_ARE_YOU_ALIVE` coming from a client, the server should reply to that client with a `RESPONSE_I_AM_ALIVE`-type message containing the same R# number that was present in the request message.

Content:	"S_RPC_A"	1	R#
Size (bytes):	←7→	←1→	←1→

3. RESPONSE_EXECUTION_STARTED
   Upon receiving a execution request message (type REQUEST_EXECUTE_PROCEDURE), the server should check if the requested service and procedure are registered and if the procedure signature appearing in the request message match the signature the procedure was registered with. If all these checks are successful, the server launches a new thread and starts the execution of the desired procedure in that thread; at the same time, a RESPONSE_EXECUTION_STARTED response message is sent to the client announcing the starting of the requested procedure.

   If the server receives multiple requests from the client (the client has no other way to deal with the possibility of lost messages), and the execution of the required task is not yet completed, the server replies with multiple RESPONSE_EXECUTION_STARTED response messages without taking any other actions (of course the server does not want to launch any other thread) as far as the sequence numbers are not increased.

Content:	"S_RPC_A"	2	Seq#
Size (bytes):	←7→	←1→	←1→

4. RESPONSE_EXECUTION_COMPLETE
   This message can be originated in two ways: it can be solicited (the response to a REQUEST_EXECUTE_PROCEDURE message coming from the client) or unsolicited (as soon as a procedure execution is complete, the server sends this message to the client).

   The client can keep asking multiple times about the result of a procedure execution and the server keeps responding as far as no other execution requests with a following number arrived. When an execution request with the n sequence number reaches the server, the server is authorized to forget about the result of the $n-1^{th}$ execution and can ignore any request message coming from the client with a number less than n.

   This policy is safe since the client does not start requesting an execution if the previous was not complete, so we can assume that after the server receives the n^{th} request, if an older request message arrives, it is certainly a delayed copy of a request which was already carried out.

Content:	"S_RPC_A"	3	Seq#	$RetVal_1$	...	$RetVal_n$
Size (bytes):	←7→	←1→	←2→	←4→	...	←4→

5. `RESPONSE_UNKNOWN_PROCEDURE`
   This message is sent by the server as a response to a `REQUEST_EXECUTE_PROCEDURE` message coming from the client in which an unknown service or an unknown procedure in a correct service was requested.

   This case should never happen if the client stub and the server servant were compiled with the ScarpazIDL compiler from the same IDL specification, unless the programmer was so unwary to declare two different interfaces (or two different version of the same interface) with the same name and tried to use the client of the one with the server of the other.

   After receiving this message, the client should warn the user of the fatal error and die.

Content:	`"S_RPC_A"`	128	Seq#
Size (bytes):	←7→	←1→	←2→

6. `RESPONSE_WRONG_SIGNATURE`
   This message is sent by the server as a response to a `REQUEST_EXECUTE_PROCEDURE` message coming from the client which tries to execute a procedure with numbers of input or output parameters different from numbers declared in the signature with which the procedure was originally registered in the server.

   As for the previous message, this case should never happen if the client stub and the server servant were compiled with the ScarpazIDL compiler from the same IDL specification, and the client and server are correctly matched.

   After receiving this message, the client should warn the user of the fatal error and die.

Content:	`"S_RPC_A"`	129	Seq#
Size (bytes):	←7→	←1→	←2→