RPC Arranger Design Pattern

Yoad Gidron
NewACT
Apollo House, Shaar Yokneam,
Yokneam Illit, Israel
+972-52-689-9929
yoad.gidron@newact.com

Lev Kozakov
IBM T.J.Watson Recearch Center
19 Skyline Drive,
Hawthorne, NY, U.S.A.
+1-914-784-7002
kozakov@us.ibm.com

Uri Shani
IBM Haifa Labs
University of Haifa Campus,
Mt. Carmel, Haifa, Israel
+972-4-8296282
shani@il.ibm.com

ABSTRACT
Remote Procedure Call (RPC) is a commonly used mechanism for synchronous communicating client/server applications. RPC implements a tightly synchronized client/server interaction that is analogous to the regular procedure call in non-distributed applications. The general underlying principle common to all RPC-based tools and standards is that both client and server share a common interface definition file (e.g., IDL). Automatic tools process the interface definition file and generate source files that, when compiled with the application, ensure client/server run-time compliancy. With DCE, CORBA and Web Services (SOAP), client and server do not have to be implemented in the same language, although the most common language used in these application had initially been C, then C++, and later Java and C#. With the modern languages, such as Java, the built-in Remote Method Invocation (RMI) method provides a very easy solution for developing distributed applications.

We introduce a new design pattern for single-language RPC-based object-oriented client/server applications that offers a solution based on sharing of base classes, in place of an interface definition file. This pattern provides the developer with full control on how client and server interact, while releasing her from the dependence on, and need to learn complex RPC infrastructures, tools and standards. We implemented this pattern in C++ to provide a convenient and safe solution to a real problem. An implementation in Java was done too.

The pattern is presented using the GOF pattern template [1].

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Network**]: distributed Systems – *client/server, Distributed applications;* D.1.3 [**Programming Techniques**]: Concurrent Programming – *Distributed Programming;*

General Terms
Design Pattern, Remote Procedure Call, Synchronous Communication

Keywords
RPC

1. RPC ARRANGER SCOPE
Distributed client/server applications. Remote Procedure Call (RPC) mechanism implementation.

2. INTENT
Encapsulate synchronous RPC mechanism for both client and server parts in a distributed application providing direct control of the process to the developer rather than hiding it within third-party services and component. Ensure seamless interaction between client and server by sharing of the same class declarations. Simplify writing reliable RPC-based applications without requiring the use of external tools.

3. FORCES
- Synchronized communication between client and server must also be maintained in full synchrony between client and server code
- Developer of RPC needs to have full control over progress of RPC at execution time.
- Need full control on how RPC passes parameters between client and serve, how they are coded, and how much of the data is passed.
- External RPC tool imposes dependency on the tool, require to learn the technology, and dependency on its revisions.
- External RPC tools may require the use of an additional mediator broker server to facilitate the client/server interaction.

4. MOTIVATION

Remote Procedure Call (RPC) is a widely used paradigm and a classic method for developing client/server applications [2][3]. The essence of RPC is that clients and servers are tightly synchronized and exchange information and control analogously to procedure calls in a regular non-distributed application [1]. RPC is implemented usually via tools provided in products that are generally also based on international standards for development of such applications [4][5][6][7]. The underlying principle of operation of these tools is that the compliance between the client and server comes from a common interface definition file, which defines the functions, their parameters, and their types. Interface files are written in an Interface Definition Language (IDL). A stub compiler generates stub files from IDL file. The stubs are compiled and linked with the client and server application code. Thus the same source interface definition is shared between applications to ensure workable protocol and seamless interaction.

The stages in an RPC are depicted in Figure 1. The remote procedure call mechanism is encapsulated in stubs, which consist of source and header files. The applications need to compile their code using stub header files and link with the compiled stubs. In addition, the applications will link with the runtime support of the RPC package. Input and output parameters are encoded and streamed via the communication protocol in a process called marshaling. Marshaling ensures that parameters are correctly transferred between different machines and processes. For that reason, the IDL file defines precisely the semantics of built-in and compound types so that the exact byte ordering (little/big endian) and number of bytes per data element is clearly and unambiguously specified.

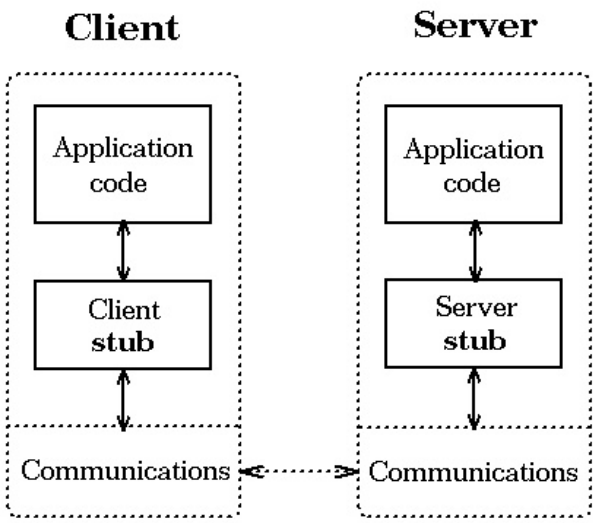

Figure 1. RPC Mechanism. Client application code invokes a regular procedure call in the client stub, which marshals input and output parameters via a communication layer. Server stub plays the partner role, which invokes the actual service routine in the server application.

This pattern comes to alleviate several issues; Firstly, some of the tools providing access to classical RPC mechanism are quite complex, and depend on special tools and run-time libraries. Yet, that is not always true, specifically with modern RPC such as the Java RMI [14]. The later is pretty easy to use. Secondly, all RPC tools, including Java RMI, leave very little control to the programmer as to how the interaction between client and server are done. In fact, the "protocol" between the parties is totally sealed behind the limited freedom of defining the parameters of the RPC and their direction – whether input to the server, or a return from it. More control of the application over the RPC execution may serve additional purposes. For instance, RPC operates on independent calls with no sense of a state in the server that is preserved across calls. This limitation is not necessarily true in all RPC systems, e.g. program partitioning [13] and in some sense also Java RMI [14] can preserve state.

The RPC Arranger design pattern comes to fill this niche. The main idea of the pattern is to share the same stub code between client and server applications by the use of a common base class, instead of generating the stubs from the same interface definition file. Within that process, programmer can also explicitly control how the RPC call protocol is implemented down to the most primitive data element exchanged between client and server.

The RPC Arranger is described and demonstrated using the C++ language. A Java version of this design has been

implemented in order to evaluate efficiency comparing this design and the use of Java RMI [16].

The RPC Arranger pattern borrows its structure from the known Bridge and Template design patterns [8], and the Dispatcher design pattern [15]. To clarify the mechanism of the RPC Arranger, consider a fragment of a simple client/server application, depicted in Figure 2A (object diagram) and Figure 2B (class diagram). This fragment presents a remote login operation.

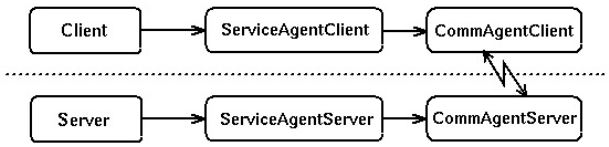

Figure 2A. Fragment of RPC based client/server application (object diagram).

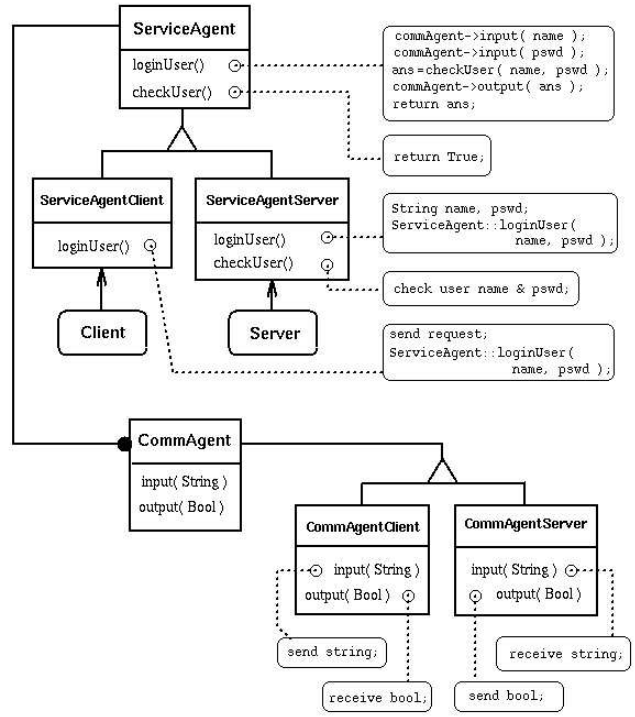

Figure 2B. Fragment of RPC based client/server application (class diagram).

The client application issues the RPC to the server application, providing user's name and password as string parameters. The server application checks the user's attributes and returns boolean parameter (answer) to the client application.

The objects in this fragment interact in the following way:

- The client calls the *loginUser()* method of its *ServiceAgentClient* object.
- This method sends appropriate request to the server to identify the required service and then calls the loginUser() method of its base ServiceAgent class, which performs the client stub of the RPC.
- The server identifies the request and calls the loginUser() method of its ServiceAgentServer object.
- This method allocates space for the input parameters and then calls the loginUser() method of its base ServiceAgent class, which performs the server stub of the RPC.
- The loginUser() method of the ServiceAgent class (wich comprises the common stub code) calls the input() methods of the CommAgent class to transfer input parameters, then it gets the output parameter value, calling the checkUser() method (the actual service function), and calls the output() method of the CommAgent class to transfer the output parameter.

The *input()* method in the client object sends parameters, while in the server object it receives parameters. The *output()* method in the client object receives parameters, while in the server object it sends parameters. The *checkUser()* method is implemented in the server object only to perform actual user authentication.

The key objects in this fragment are the ***service*** and ***communication agents***.

Base communication agent class defines marshaling methods for built-in data types. These methods are implemented differently in the client and server communication agent objects (in the client *input()* means sending and *output()* means receiving, while in the server *input()* means receiving and *output()* means sending).

One service agent object is required for each of the client and server to implement the RPC (*loginUser()*). The common base class for these objects implements the stub for this procedure. It uses marshaling (*input()/output()*) methods, implemented in an appropriate communication agent object, and calls the actual service function (*checkUser()*), which completes the task (user authentication). The actual service function is implemented in the server object only.

Note that the *loginUser()* function is an instance of the Template Method [8] design pattern - it is composed of functions, that are defined in abstact classes and implemented differently in concrete classes.

By sharing of the same base class between client and server service objects, the same common code (stub) will run on both sides, ensuring synchronous interaction between client and server parts of the application.

In real distributed applications each semantically related group of services requires one set of service agent objects as follows:

- One common base class, which implements common stubs
- One object for each client and server, whose class is derived form this common base class.

Each RPC will be identified in two levels:

- ***High level dispatcher*** - to resolve the group (class) of services (service agent object)
- ***Low level dispatcher*** - to resolve the specific procedure in the service group

High level dispatcher is implemented in the server application itself. The service-specific low level dispatchers are implemented in each service agent object of the server.

5. APPLICABILITY

The pattern could be useful for developing RPC-based object-oriented distributed applications, when both client and server parts are written in the same programming language and use the same underlying communication interface. Depending on the low-level marshaling implementation, it may be possible to walk also between heterogeneous machines.

6. STRUCTURE

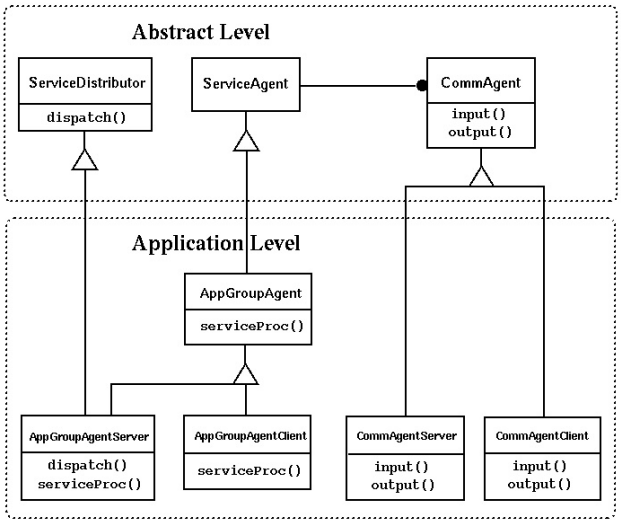

Figure 3. Class diagram of the RPC Arranger design pattern.

7. PARTICIPANTS

7.1 Abstract level

7.1.1 CommAgent

Abstract class. Defines the marshaling interface (input/output methods) for built-in data types; it also implements all methods needed for the RPC initiation;

7.1.2 ServiceDistributor

Abstract class. Defines an interface for the low level dispatcher; a reference to an instance of this class could be used for registration of service groups in the high level dispatcher (not shown here);

7.1.3 ServiceAgent

Base class for all application-specific service agent classes; it provides an access to the communication agent object.

7.2 Application level

7.2.1 CommAgentServer

Derived from the *CommAgent* class; it implements the server side of the marshaling interface (input/output methods) for built-in data types (in the server input means receiving and output means sending);

7.2.2 CommAgentClient

Derived from the *CommAgent* class; it implements the client side of the marshaling interface (input/output methods) for built-in data types (in the client input means sending and output means receiving);

7.2.3 AppGroupAgent

Base service agent class for a particular group of services, derived from the *ServiceAgent* class; it implements the application-specific group of common procedures (stubs) for both client and server; it may also implement the common marshaling interface (input/output methods) for the application-specific group of data types; it defines the application-specific group of actual service functions, as well;

7.2.4 AppGroupAgentServer

Service agent class for a particular group of services, provided in the server, derived from both the *ServiceDistributor* and the *AppGroupAgent* classes; it implements the low-level dispatcher as well as the application-specific group of actual service functions for the server application;

7.2.5 AppGroupAgentClient

Service agent class for a particular group of services, used in the client, derived from the *AppGroupAgent* class; it overrides the base procedures (stubs), defined in its base

class, so that it sends the service specific request before the base procedure is called.

8. COLLABORATIONS

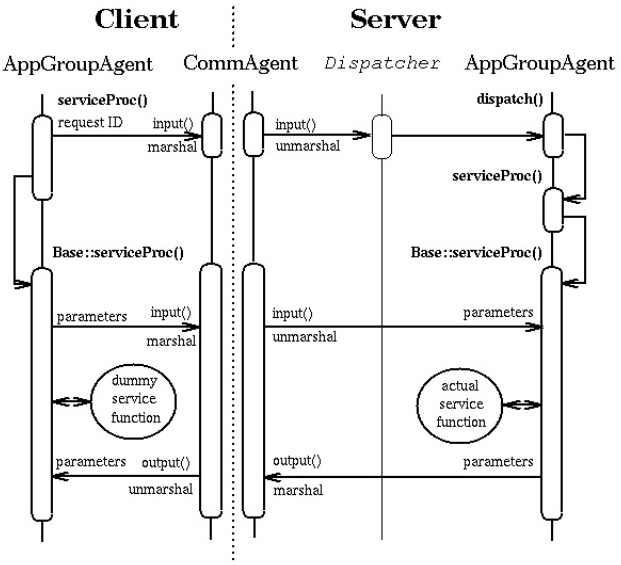

Figure 4. Collaborations between the RPC Arranger objects.

- The client invokes a specific remote procedure by using a method of the appropriate service group (*AppGroupAgentClient* class).
- The ID of this call is sent to the server as a request message. In the server, a top-level dispatcher is awaiting incoming messages, and will receive the request ID, identify the proper service group (*AppGroupAgentServer* class) and invoke the low level dispatcher of the group.
- The group dispatcher, defined in the service distributor (*ServiceDistributor* class) and implemented in the service agent object of the server (*AppGroupAgentServer* class) will invoke the correct procedure.
- From this point, both client and server run exactly the same common service method (stub), which is implemented in their mutual base class (*AppGroupAgent* class). This method transfers parameters by using marshaling functions of two kinds. For the group-specific data types, marshaling is implemented in the base service agent class (*AppGroupAgent* class). For built-in data types, marshaling is defined in the proper communication agent object (*CommAgent* class), which is accessed through the common base class (*ServiceAgent* class) . Note, that the latter functions are implemented differently in the client and server communication agent objects (*CommAgentClient/CommAgentServer* classes). To accomplish the task, the common service method calls appropriate service functions, defined in the base service agent class (*AppGroupAgent* class) and implemented in the server object (*AppGroupAgentServer* class).

9. CONSEQUENCES

The benefits and liabilities of the RPC Arranger pattern include the following:

- It eliminates the need for interface definition (IDL) files and tools, used to generate client/server stubs. Instead both client and server parts of the application use the same code (stub) to implement the actual RPC.
- It gives a developer more control over the marshaling process. The marshaling methods for all built-in data types as well as other necessary methods of the communication agent could be shared between different applications. The marshaling methods for application-specific data types should be based on methods provided by the communication agent. The application-specific marshaling methods are written by a developer of the server application.
- It adds certain overhead when implementing new services, as described in the Implementation section.
- It makes using new services easy. To make use of a new available service procedure a client application should be recompiled and linked with the new/updated service agent class code.

10. IMPLEMENTATION

To make new service procedure available a developer should go through the following steps:

1. Implement common part
   - Add two methods (new common procedure stub and new service function definition) to existing base service agent class (or add new base service agent class);

- Add marshaling methods for new service-specific data types (if necessary);

2. Implement server part
   - Add one trivial method (new service procedure) to existing service agent class (or add new service agent class, derived from the appropriate base service agent class);
   - Implement the actual service function in the service agent class;
   - Add new entries to the high level and low level dispatchers.
3. Implement client part
   - Add one trivial method (new service procedure) to existing service agent class (or add new service agent class, derived from the appropriate base service agent class).

The following are further implementation details:

- Each application initializes a communication channel via TCP/IP. The most useful method is via sockets, but the pattern does not prefer any mechanism as long as it can be considered a reliable stream of bytes in both directions.
- In our sample we use the Socket++ library [9], which encapsulates the standard socket library API. The *Socket* class in our sample inherits from the standard *iostream* class and uses the buffer of *streambuf*. Thus, we can write to the socket by using the '<<' operator, and read from the socket by using the '>>' operator. Data is sent over the socket implicitly (when the buffer is full) or explicitly by calling *sync()*. End of input data stream is sensed when the input buffer is empty. Our *Socket* class definition is very similar to the *iosockstream* in the Socket++ library.
- The registration of available service groups could be done after the communication channel has been established. Each service group is associated with one service agent object. To register particular service group its service agent object (presented as a pointer to the *ServiceDistributor* object) should be added to the registrar object, accompanied by all service specific request IDs. In our sample the *registrar* object is implemented as the STL - like [10] *Map* class instance.
- If certain application-specific data types are used by more than one service group, the marshaling functions for these data types could be defined separately.
- To avoid multiple inheritance in the service agent object (server) one could use the alternative structure of this object, depicted in Figure 5.

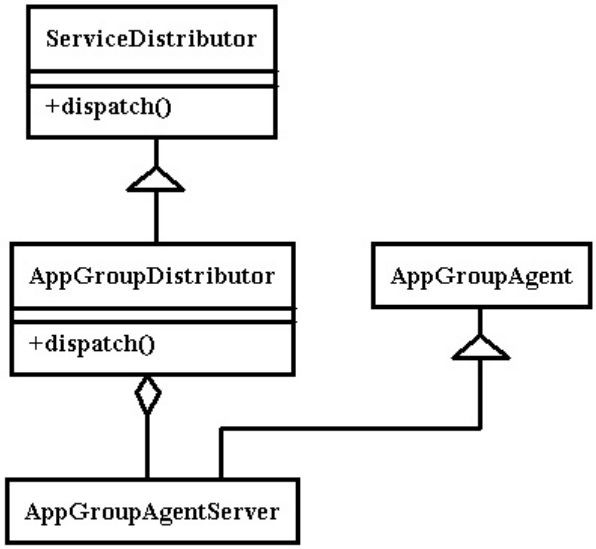

Figure 5. Alternative structure of the service agent object in server application.

11. SAMPLE CODE AND USAGE

11.1 Sample Application

Our example is of a simple calculator. The calculator can perform the four basic arithmetic operations: '+', '-', '*', and '/' on integer operands. We define the arithmetic operations as a type:

```
typedef char Operator;
#define PLUS  '+'
#define MINUS '-'
#define MULT  '*'
#define DIV   '/'
```

The calculate() function returns the result of a given expression:

```
int calculate(int a, Operator op, int b)
{
  switch (op) {
    case PLUS:  return a + b;
    case MINUS: return a - b;
    case MULT:  return a * b;
```

```
    case DIV:   return a / b;
    default:          //  should  throw  an
exception
  }
}
```

The application is as simple as:

```
main()
{
  int a, b;
  Operator op;

  cin >> a >> op >> b;

  cout << calculate(a, op, b) << endl;
}
```

To adapt this application to a client/server model, the client will invoke the calculate() function via an RPC to the server, after passing to it the operands and operator as input parameters. The result will be returned as an output parameter.

11.2 The RPC Staff

Two enumerated types are defined: **Request ID**s that are sent from the client to the server and **Return Code**s that are sent from the server to the client.

```
enum RequestId {
  NO_REQUEST = 0,
  // Here the application adds more RPC
  // ids (e.g. CALCULATE for our example)
};

enum ReturnCode {
  SUCCESS = 0,
  INTERNAL_FAILURE,
  COMMUNICATION_FAILURE,
  CONNECTION_REFUSED,
  INVALID_REQUEST,
  FUNCTION_FAILED
};
```

The **Communication Agent** *CommAgent* is an abstract base class that defines the input and output methods for the built-in data types (which have different implementations in the client and server) as pure virtual functions. Here, we are only using characters, integers and strings, but more input/output functions can be defined for other built-in types or user-defined types. The Communication Agent also defines the first two steps in the client/server protocol: connect and request. The *isValid()* method is used to identify the client application. It validates that the client and server were indeed compiled with the same shared stub code. This method is implemented in the server only.

```
class CommAgent {
private: // private methods
  virtual boolean isValid() const {
    return True; }
protected: // protected attributes
  Socket &socket;
public: // authentication string
  String clientAuthentication;
public: // base methods
  CommAgent(Socket &sock) : socket(sock) {}
  virtual ~CommAgent() {}
public: // abstract methods
  virtual void input(char &character) = 0;
  virtual void input(int &number) = 0;
  virtual void input(String &string) = 0;
  virtual boolean endOfInput() = 0;
  virtual void output(char &character) = 0;
  virtual void output(int &number) = 0;
  virtual void output(String &string) = 0;
  virtual boolean endOfOutput() = 0;
public: // public methods
  ReturnCode connect() {
    // client: send; server: receive
    input(clientAuthentication);
    // check if the input was successful
    if (!endOfInput())
      return COMMUNICATION_FAILURE;
    ReturnCode response;
    // server: check client authentication
    if (isValid())
      response = SUCCESS;
    else
      response = CONNECTION_REFUSED;
    // server: send; client: receive
    output(response);
    return endOfOutput() ? serverResponse :
COMMUNICATION_FAILURE;
  }
  ReturnCode  request(RequestId  &requestId)
{
    // client: send; server: receive
    input(requestId);
    return   endOfInput()   ?   SUCCESS   :
COMMUNICATION_FAILURE;
  }
};
```

The **Communication Agent Client** *CommAgentClient* implements the actual input and output methods for the built-in data types. In the client, input means sending, and output means receiving.

```
class CommAgentClient : public CommAgent {
public: // base methods
  CommAgentClient(Socket     &socket)     :
CommAgent(socket) {}
  ~CommAgentClient() {}
public: // marshaling methods
  void input(char &character) {    // send
    socket << character; }
  void input(int &number) {        // send
    socket << number; }
  void input(String &string) {     // send
    socket << string; }
  boolean endOfInput() {           // flush
    return socket.sync(); }
  void  output(char  &character)  {      //
receive
    socket >> character; }
  void  output(int  &number)  {          //
receive
    socket >> number; }
```

```
  void output(String &string) {       //
receive
    socket >> string; }
  boolean endOfOutput() {          // check
    return socket.good(); }
};
```

The **Communication Agent Server** *CommAgentServer* implements the actual input and output methods for the built-in types. In the server, input means receiving, and output means sending. It also implements the *isValid()* method that checks the authentication of the client.

```
class CommAgentServer : public CommAgent {
private: // private methods
  boolean isValid() const {
   // check the client authentication
 }
public: // base methods
  CommAgentServer(Socket     &socket)     :
CommAgent(socket) {}
  ~CommAgentServer() {}
public: // marshaling methods
  void input(char &character) {       //
receive
    socket >> character; }
  void input(int &number) {           //
receive
    socket >> number; }
  void input(String &string) {        //
receive
    socket >> string; }
  boolean endOfInput() {           // check
    return socket.good(); }
  void output(char &character) {   // send
    socket << character; }
  void output(int &number) {       // send
    socket << number; }
  void output(String &string) {    // send
    socket << string; }
  boolean endOfOutput() {          // flush
    return socket.sync(); }
};
```

The **Service Distributor** *ServiceDistributor* is an abstract class that defines the pure virtual function *dispatch()* which serves as the low level dispatcher. A pointer to an instance of this class will be used for the registration of services.

```
class ServiceDistributor {
public: // base methods
  ServiceDistributor() {}
  virtual ~ServiceDistributor() {}
public: // abstract low level dispatcher
  virtual    boolean    dispatch(RequestId
requestId) = 0;
};
```

The **Service Agent** *ServiceAgent* is the base class for all service agent classes. It holds a reference to a **Communication Agent**.

```
class ServiceAgent {
protected: // attributes
  CommAgent &commAgent;
public: // base methods
  ServiceAgent(CommAgent   &aCommAgent)   :
commAgent(aCommAgent) {}
  virtual ~ServiceAgent() {}
};
```

11.3 RPC Version of the Application

The **Calculator Agent** is the base class (*AppGroupAgent* class) for the client and server service-specific classes. It implements the *calc_RPC()* method - the common procedure for both client and server. The *calc_RPC()* method takes three input arguments (two operands and an operator) and returns the result as an output argument. This method defines the protocol of passing arguments between the client and server. The *doCalculate()* method performs the actual calculation and is implemented only in the server by calling our original *calculate()* function.

```
class CalcAgent : public ServiceAgent {
private: // dummy service functions
  virtual  ReturnCode  doCalculate(int  &a,
Operator &op, int &b,
                                   int
&result) {
    return SUCCESS; }
protected: // common RPC methods
  virtual   ReturnCode   calc_RPC(int   &a,
Operator &op, int &b,
                                int   &result)
{
    // This is the common RPC stub
    commAgent.input(a);
    commAgent.input(op);
    commAgent.input(b);
    if (!commAgent.endOfInput())
      return COMMUNICATION_FAILURE;
    // The following call does nothing in
    // the client, but in the server it
    // performs the actual calculation.
    ReturnCode rc = doCalculate(a, op, b,
result);
    commAgent.output(rc);
    // needed to verify success of the
    // function itself.
    if (rc == SUCCESS)
      commAgent.output(result);
    return    commAgent.endOfOutput()    ?
SUCCESS : COMMUNICATION_FAILURE;
  }
public: // base methods
  CalcAgent(CommAgent     &commAgent)     :
ServiceAgent(commAgent) {}
};
```

The **Calculator Agent Client** overrides the *calc_RPC()* function of the base class. First, a request is sent to the server and then the the base *calc_RPC()* function is called.

```
class CalcAgentClient : public CalcAgent {
```

```
public: // base methods
  CalcAgentClient(CommAgentClient
&commAgent) : CalcAgent(commAgent) {}
public: // service RPC methods
  ReturnCode calc_RPC(int &a, Operator &op,
int &b, int &result) {
    RequestId     reqId     =     CALCULATE;
// send request
    ReturnCode              rc             =
commAgent.request(reqId);
    if (rc == SUCCESS)
      return  CalcAgent::calc_RPC(a,  op,  b,
result);
    else
      return rc;
  }
};
```

The **Calculator Agent Server** implements the *doCalculate()* function. It also implements the low level dispatcher that selects the method to invoke. The *calculate()* function with no arguments is defined because the server should allocate parameters before calling the *calc_RPC()* function.

```
class     CalcAgentServer     :     public
ServiceDistributor,
                      public CalcAgent {
private: // actual service functions
  ReturnCode  doCalculate(a,  op,  b,  result)
{
    result = calculate(a, op, b);
    return SUCCESS;
  }
private:  //  methods  to  be  invoked  by
dispatcher
  ReturnCode calculate() {
    int a, b, result;
    Operator op;
    return  CalcAgent::calc_RPC(a,  op,  b,
result);
  }
public: // base methods
  CalcAgentServer(CommAgentServer
&commAgent) : CalcAgent(commAgent) {}
public: // public interface
  boolean dispatch(RequestId requestId) {
    // low level dispatcher
    ReturnCode rc = INVALID_REQUEST;
    switch(requestId) {
    case CALCULATE:
      rc = calculate();
      break;
    default:
      break;
    }
    return (rc == SUCCESS) ? True : False;
  }
};
```

The *Client* class defines the methods of the client application. The *connect()* method is used for establishing a connection to the server. It also provides an interface to the *calc_RPC()* function of the **Calculator Agent Client**.

```
class Client {
private: // attributes
  Socket socket;
  CommAgentClient *pCommAgent;
public: // base methods
  Client() {}
  ~Client() {
    socket.close(); }
public: // public interface
  ReturnCode  connect(const  char  *hostName,
int portNumber) {
    if           (!socket.connect(hostName,
portNumber))
      return COMMUNICATION_FAILURE;
    pCommAgent             =             new
CommAgentClient(socket);
    // The client should initialize the
    //  clientAuthentication  string  in  the
pCommAgent
    return pCommAgent->connect();
  }
  void disconnect() {
    socket.shutdown();
    delete pCommAgent;
  }
public: // service methods
  ReturnCode   calculate(int   &a,   Operator
&op, int &b,
                       int &result) {
    CalcAgentClient calcAgent(*pCommAgent);
    return   calcAgent.calc_RPC(a,   op,   b,
result);
  }
};
```

The *Server* class defines the methods of the server application. It contains the top-level dispatcher that selects the agent class to service the request and calls the low-level dispatcher. The *handleRequest()* method receives a *requestID* from the client and calls the top level dispatcher.

```
class Server {
private: // attributes
  Socket socket;
  CommAgentServer *pCommAgent;
  Map<  RequestId,  ServiceDistributor  *  >
registrar;
private: // private interface
  // service registration
  void registerServices () {
    registrar.insert     (CALCULATE,     new
CalculatorAgentServer(*pCommAgent));
  }
  // top level dispatcher
  boolean dispatch(RequestId reqId) {
    ServiceDistributor       *pAgent       =
registrar.find (reqId);
    if (pAgent != NULL)
      return pAgent->dispatch(reqId);
    else
      return False;
```

```
 }
public: // base methods
  Server(int portNumber) {
    socket.bind(portNumber); }
  ~Server() {
    socket.close(); }
public: // public interface
  void listen() {
    socket.listen(); }
  boolean connect() {
    Socket sock(socket.accept());
    pCommAgent = new CommAgentServer(sock);
    registerServices();
    return    (pCommAgent->connect()    ==
SUCCESS) ? True : False;
  }
  void disconnect() {
    registrar.erase(registrar.begin(),
registrar.end());
    delete pCommAgent;
  }
  boolean handleRequest() {
    RequestId requestId = NO_REQUEST;
    if  (pCommAgent->request(requestId)  !=
SUCCESS) // receive a request
      return False;
    return dispatch(requestId);
  }
};
```

Client code example:

```
Client client;
client.connect("hadar.haifa.ibm.com",
1997);
int a, b;
Operator op;
cin >> a >> op >> b;
int result;
client.calculate(a, op, b, result);
cout << result << endl;
```

Server code example (this server handles only one request per connection):

```
Server server(1997);
while(1) {
  server.listen();
  server.connect();
  server.handleRequest();
  server.disconnect();
}
```

12. KNOWN USES

This methodology was first presented at the Eighth Israeli Conference on Computer Systems and Software Engineering [11].

The *RPC Arranger* design pattern was assumed as a basis for the implementation of the client/server solution in the large-scale hospital Picture Archive and Communication System (PACS) [12]. The primary purpose of the client/server solution was to provide remote Windows-based image viewers with an access to the central UNIX-based system archive. The solution alleviated the need to implement security and data access on the Windows platform based on the implementation already available on the UNIX platform. The server part was implemented as an UNIX-based application, while the client part - as a Windows-based API. Low level marshaling interface was implemented by using the Socket++ library [9].

13. RELATED PATTERNS

The *RPC Arranger* has something in common with a few classical design patterns. The structure of the (ServiceAgent, CommAgent, AppGroupAgent) triad reminds the structural Bridge pattern [8]. The construction of the common ‘stub’ procedure in the ServiceAgent and its different implementations in the ServiceAgentServer and ServiceAgentClient remind the behavioral Template Method pattern [8]. The Dispatcher pattern [15] may be used to implement the dispatcher component of the RPC Arranger pattern.

The major distinction of the RPC Arranger pattern is that it combines several classical structural and behavioral ideas to form a specialized design pattern for a wide class of distributed systems.

14. REFERENCES

[1] A.D. Birrell, B.J. Nelson, "Implementing Remote Procedure Calls", ACM Trans. on Computer Systems, vol.2(1), pp.39-59, February 1984.

[2] W. Richard Stevens, "UNIX Network Programming", Prentice Hall, 1990, ISBN 0-13-949876-1.

[3] Open Software Foundation, "OSF DCE Application Development Guide", Prentice Hall, 1993, ISBN 0-13-643826-1.

[4] Object Management Group, "The Common Object Request Broker: Architecture and Specifications", Object Management Group, Inc., edition 2.0, July 1995.

[5] Douglas C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications", Proc. of the 6-th USENIX C++ Technical Conference, Cambridge, Massachusets, USENIX Association, April 1994.

[6] J. Dilley, "OODCE: A C++ Framework for the OSF Distributed Computing Environment", Proc. of the USENIX Technical Conference on UNIX and Advanced Computing Systems, New Orleans, Louisiana, January 16-20, 1995, USENIX Association, Berkeley, CA, USA, ISBN 1-880446-67-7.

[7] I. Gold, U. Shani, "Wrapping DCE/OSF Client/Server Applications", Proc. of the USENIX UNIX Applications Development Symposium, Toronto, Canada, April 25-28, 1994.

[8] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, "Design Patterns: Elements of Reusable Object Oriented Software", Addison-Wesley Professional Computing Series, 1995, ISBN: 0-201-63361-2.

[9] Gnanasekaran Swaminathan, C++ Socket Library (Socket++), copyright by SAIC/ASSET, 1996.

[10] David R. Musser, Atul Saini, "STL Tutorial and Reference Guide: C++ Programming with the Sandard Template Library", Addison-Wesley Professional Computing Series, 1996, ISBN: 0-201-63398-1.

[11] Y. Gidron, L. Kozakov, U. Shani, "An RPC-Based Methodology for Client/Server Application Development in C++", Proc. of the 8-th Israeli Conference on Computer Systems and Software Engineering, pp.39-46, Herzliya, Israel, June 18-19, 1997.

[12] Y. Gidron, L. Kozakov, E. Salant, U. Shani, "Deploying a Large-Scale Hospital-Wide PACS", to be published in the Proc. of the SPIE Conference on Medical Imaging, vol.3339, paper 58, San Diego, California, USA, February 21-27, 1998.

[13] U. Shani, et. al., "C Programs Partitioning for Heterogeneous Machines", Proc of the 6-th Israeli Conference on Computer Systems and Software Engineering, pp. 136-145, June 2-3, Israel, 1992.

[14] P. Heller and S. Roberts, "Java 1.1 Developer's Handbook", Sybex, 1997, ISBN 0-7821-1919-0.

[15] Christian Thilmany and Todd McKinney, “BizTalk Implement Design Patterns for Business Rules with Orchestrated Designer,” From the October 2001 issue of MSDN Magazine. http://msdn.microsoft.com/msdnmag/issues/01/10/BizTalk/default.aspx

[16] Anat Hashavit, Sigal Ishay and Vladimir Lazebny, ”An RPC-Based Methodology for Client/Server Application Development in Java,” DSL Lab, Computer Science department, Technion – Israel Institute of Technology, Haifa, Israel. Final project report in: http://dsl.cs.technion.ac.il/completed_projects/rpc-pattern/index.htm, 2004.