MyRPC

Birsan C. Ioana (cas. Amariei), B5

Alexandru Ioan Cuza University of Iasi

Introduction

This technical report is mainly addressed to computer science students that have basic knowledge regarding computer network concepts. The purpose of this report is to:

1. Understand the requirements for the MyRPC project: we are asked to implement a client-server application that allows calls for remote procedures. We know that the server has access to a list of predefined procedures and the client has the ability to get the list and send requests to the server. A protocol for making requests and providing answers will be defined.
2. Propose a solution that meets those requirements: in doing this, we describe the technologies we will use and explain each choice, create an application architecture, and provide use-cases that helps us clarify the client-server interaction.

Technologies

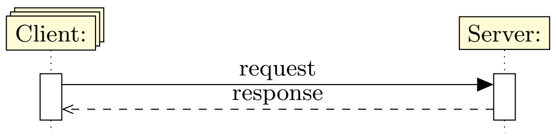
In implementing this project I chose the following technologies:

1. C++ because it is an object-oriented programming language that allows data abstraction and modularity;
2. The TCP client-server model folds better in this particular case because data reaches its destination in time without duplication and it automatically brakes data into packages. Despite being slower than the UDP client-server model, the last one comes with no guarantees regarding delivery, I have to manually break the data into packages and it would be difficult to reconstruct packages.
3. pugixml for parsing an XML document because the library is extremely portable and easy to integrate and use.

Application architecture

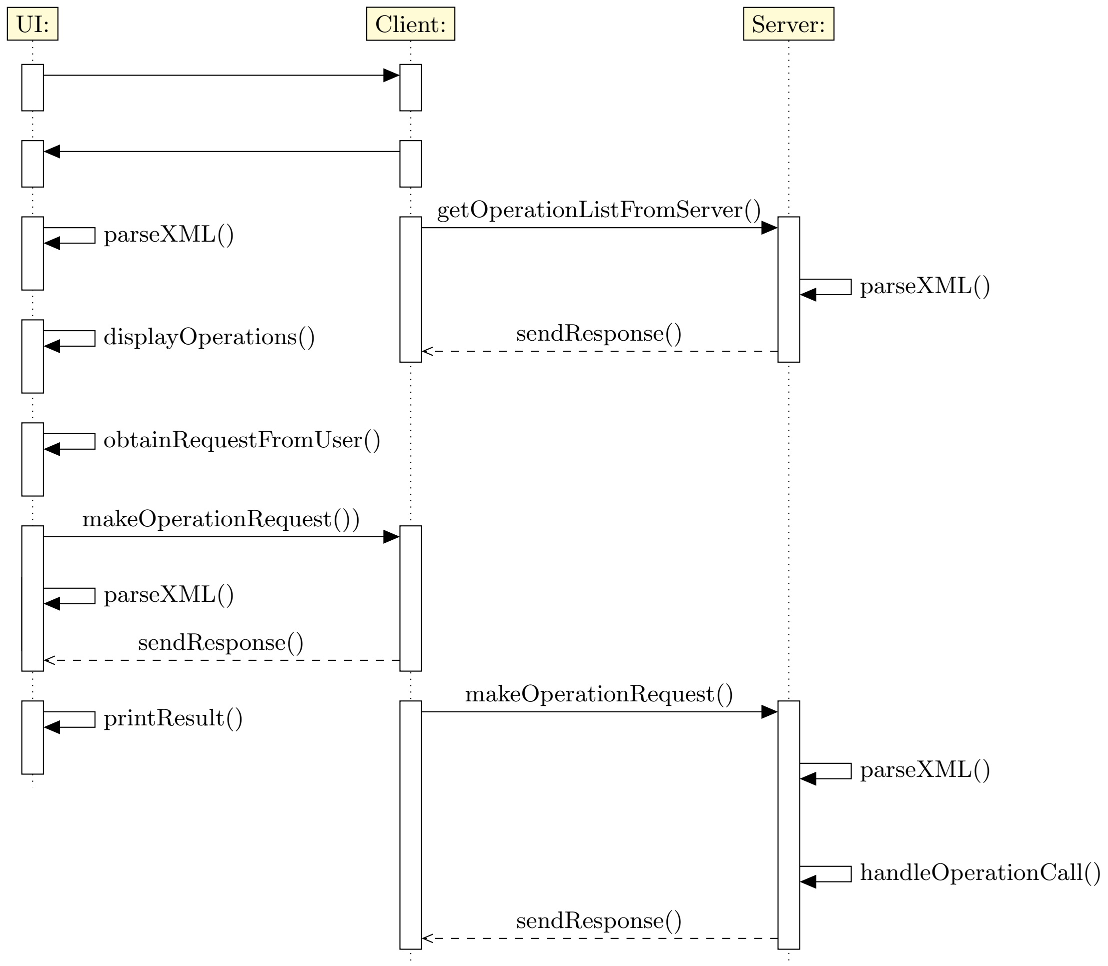
In the figure below is presented the client-server application architecture: the server component provides a function or service to one or many clients, which initiate requests for such services.

Fig. 1. Application architecture



In the sequence diagram below is presented the detailed interaction between the client and the server. The client makes a request for the procedures list that are contained in server and the server answers to this request by sending the available procedures in XML format. The client is responsible with the parsing of the document. When is finished it sends a new request with the name of the procedure and its parameters. The server computes the operation and returns its result to the client which will then display the result.

Fig. 2. UI - Client - Server interaction



Implementation details

The application uses a TCP client-server model, with a concurrent server implemented using the fork() aproach. I have used fork() as a mechanism of creating new processes because it is simple to implement and completely satisfies the requiremets for application. The server is responsible with receiving requests from clients (operation list or operation call) and sending back the appropiate response.

```
void Server::startServer() {
    while (1) {
        socklen_t length = sizeof(this->from);

        printf("[server]Waiting at port %d...\n", port);
        fflush(stdout);

        int clientSocketDescriptor = accept(socketDescriptor,
                                (struct sockaddr *) &this->from, &length);

        if (clientSocketDescriptor < 0) {
            perror("[server] Error at accept().\n");
            continue;
        }

        this->pid = fork();

        if (this->pid < 0) {
            perror("Error at fork().\n");
            continue;
        }

        if (this->pid == 0) {
            int size = readInt(clientSocketDescriptor);
            char *requestBuffer = readBuffer(clientSocketDescriptor, size);

            printf("[server]Message received...%s\n", requestBuffer);

            string request(requestBuffer);

            xml_document doc;
            xml_parse_result result = doc.load_string(request.c_str());

            string requestType = doc.document_element().attribute("type").value();

            if (!result) {
                cout << "Parsed with errors" << endl;
                cout << result.description() << endl;
                const char *buffer = "The XML document could not be parsed.";
                sendResponse(clientSocketDescriptor, buffer);
            } else if (requestType.compare("operationList") == 0) {
                sendFile(clientSocketDescriptor);
            } else if (requestType.compare("operationCall") == 0) {
                handleOperationCall(clientSocketDescriptor, doc);
            } else {
                cout << "Received an invalid request: " << request << endl;
                const char *buffer = "The request is not valid. No such operation.";
                sendResponse(clientSocketDescriptor, buffer);
            }

            close(clientSocketDescriptor);
        } else {
            /**
             * @https://profs.info.uaic.ro/~eonica/rc/lab07e.html
             */
            waitpid(pid, &this->status, WNOHANG);
        }
    }
}
```

In the server, the child processes that ended can be ”cleaned” by calling a non-blocking waitpid() using the WNOHANG parameter. The advantage, besides simultaneously handling multiple clients is also that the client waits a small amount of time for the connection to be accepted. The server continues the loop for accepting client connections almost immediately, the children handling the client requests.

In terms of the list of procedures that the server makes available to clients, I considered a list of mathematical procedures. I used a XML document format as appropiate to represent the information.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<operations>
    <operation>
        <name>add</name>
        <arguments>
            <type>int</type>
            <type>int</type>
        </arguments>
        <resultType>int</resultType>
    </operation>
    <operation>
        <name>sub</name>
        <arguments>
            <type>int</type>
            <type>int</type>
        </arguments>
        <resultType>int</resultType>
    </operation>
    <operation>
        <name>mul</name>
        <arguments>
            <type>int</type>
            <type>int</type>
        </arguments>
        <resultType>int</resultType>
    </operation>
    <operation>
        <name>div</name>
        <arguments>
            <type>int</type>
            <type>int</type>
        </arguments>
        <resultType>int</resultType>
    </operation>
    <operation>
        <name>sum</name>
        <arguments>
            <type>int+</type>
        </arguments>
        <resultType>int</resultType>
    </operation>
    <operation>
        <name>to_uppercase</name>
        <arguments>
            <type>string</type>
        </arguments>
        <resultType>int</resultType>
    </operation>
</operations>
```

The UI class is used to realize the interaction between the user and the client, by following the next steps:

1. obtain the list of procedures
2. display the list of procedures
3. obtain request from user
4. send request to client
5. obtain result and display it

```
 void UI::startUserInteraction() {
    string operations = newClient().getOperationListFromServer();
    displayAvailableOperations(operations);
    string request = obtainRequestFromUser();
    string result = newClient().makeOperationRequest(request);
    printResult(result);
}

void UI::displayAvailableOperations(string ops) {
    xml_document doc;
    xml_parse_result result = doc.load_string(ops.c_str());

    if (!result) {
        cout << "Parsed␣with␣errors." << endl;
        cout << result.description() << endl;
        exit(1);
    }

    cout << "The␣list␣of␣available␣operations:␣" << endl;

    int item = 1;
    xml_node operations = doc.document_element();
    for (xml_node operation : operations.children()) {
        cout << item++ << ".␣" << operation.child("name").child_value() << "␣";

        xml_node arguments = operation.child("arguments");
        for (xml_node type: arguments.children()) {
            cout << encloseTypeIdentifier(type.child_value()) << "␣";
        }
        cout << endl;
    }
}

string UI::obtainRequestFromUser() {
    cout << "Call␣desired␣function:␣";
    char request[256];
    cin.getline(request, 256);

    xml_document doc;
    xml_node requestNode = doc.append_child("request");
    requestNode.append_attribute("type") = "operationCall";
    xml_node operation = requestNode.append_child("operation");

    list<string> tokens = split(request, "␣");
    string& name = tokens.front();
    operation.append_child("name")
                        .append_child(pugi::node_pcdata).set_value(name.c_str());
    tokens.pop_front();

    xml_node arguments = operation.append_child("arguments");
    for(string& argument : tokens) {
        arguments.append_child("argument")
                .append_child(pugi::node_pcdata).set_value(argument.c_str());
    }

    stringstream ss;
    doc.save(ss, "␣␣");

    return string(ss.str());
}

void UI::printResult(string result) {
    xml_document doc;
    xml_parse_result parse_result = doc.load_string(result.c_str());

    if (!parse_result) {
        cout << "Parsed␣with␣errors." << endl;
        cout << parse_result.description() << endl;
        exit(1);
    }

    cout << "The␣result␣is:␣" << doc.document_element().child_value() << endl;
}
```

The client is responsible with: obtaining the list of procedures from server, making the operation request to server and obtaining the result.

```
string Client::getOperationListFromServer() {
    sendOperationListRequestToServer();
    return getOperationListResponseFromServer();

}

void Client::sendOperationListRequestToServer() {
    string request = "<request␣type=\"operationList\"␣/>";
    writeInt(socketDescriptor, request.length());
    writeBuffer(socketDescriptor, request);
}

string Client::getOperationListResponseFromServer() {
    int length = readInt(socketDescriptor);
    char *buffer = readBuffer(socketDescriptor, length);

    return string(buffer);
}

string Client::makeOperationRequest(string request) {
    writeInt(socketDescriptor, request.length());
    writeBuffer(socketDescriptor, request);
    int size = readInt(socketDescriptor);
    char* buffer = readBuffer(socketDescriptor, size);

    return string(buffer);
}
```

Below we present two scenarios of use in which the actors are represented by the client, server and user. The first scenario describes the manner in which initialization of client-server communication takes place. The second one shows how the user interacts with the client (application) to send the input needed to perform some operations, then the client is responsible for passing to the server the information which then provides the answer.

Fig. 3. Initialization

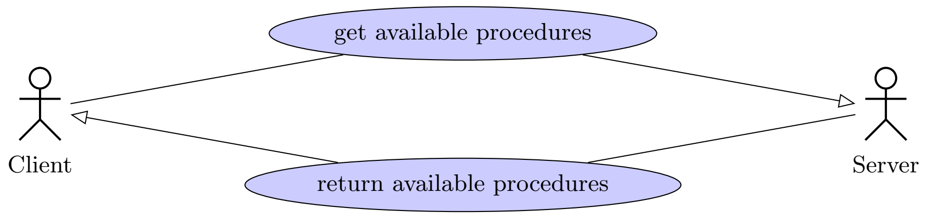
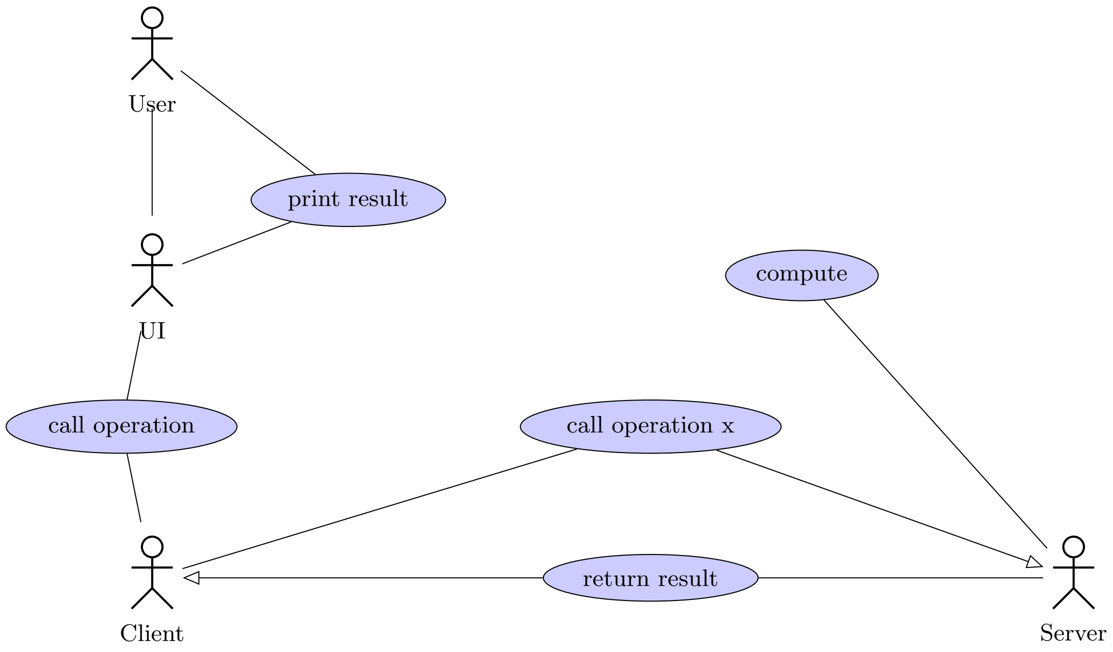


Fig. 4. Procedure call



Conclusions

We presented the requirements for the MyRPC project and described a solution that satisfies them. With regard to a possible improvement of the proposed solution, optimization is desirable for the server side, because in this model there is a problem with the potential overburden in server with children and also the time lost for creating and eliminating from the system new processes for each new connection.

Bibliography

– http://csapp.cs.cmu.edu/2e/ch12-preview.pdf
– http://www.tenouk.com/Module39.html
– https://pugixml.org/docs/quickstart.html#access
– https://profs.info.uaic.ro/~eonica/rc/lab07e.html
– http://perso.ensta-paristech.fr/~kielbasi/tikzuml/var/files/html/web-tikz-uml-userguide.html
– https://en.wikipedia.org/wiki/Client%E2%80%93server_model
– https://docstore.mik.ua/orelly/xml/xmlnut/index.htm
– https://en.wikipedia.org/wiki/Use_case