Applying Idioms for Synchronization Mechanisms: Synchronizing communication components for the Fast Fourier Transform

Jorge L. Ortega-Arjona
jloa@ciencias.unam.mx
Departamento de Matemáticas
Facultad de Ciencias, UNAM
MEXICO

Abstract

The Idioms for Synchronization Mechanisms is a collection of patterns related with a method for the implementation of synchronization mechanisms within parallel software systems. The selection of these idioms take as input information *(a)* the design pattern of the communication components to synchronize, *(b)* the memory organization of the parallel hardware platform, and *(c)* the type of communication required.

In this paper, it is presented the application of the Idioms for Synchronization Mechanisms to synchronize the communication components for the Fast Fourier Transform problem. The method takes the information from the Problem Analysis, Coordination Design, and Communication Design, applying an idiom for synchronization mechanisms, and providing elements about its implementation.

CCS Concepts: • **Software and its engineering** → **Designing software**.

Keywords: Idioms, Fast Fourier Transform, Parallel Software Design

ACM Reference Format:
Jorge L. Ortega-Arjona. 2022. Applying Idioms for Synchronization Mechanisms: Synchronizing communication components for the Fast Fourier Transform. In *Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP'22), October 18, 2022*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SugarLoafPLoP'22, October 18, 2022,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

A lot of work and experience has been gathered in concurrent, parallel, and distributed programming around the synchronization mechanisms, as originally proposed during the late 1960s and 1970s by E.W. Dijkstra [4], C.A.R. Hoare [6–8], and P. Brinch-Hansen [1–3]. Further work and experience has been gathered today, such as the formalization of concepts and their representation in different programming languages.

Synchronization is expressed in programming terms as language primitives, known as synchronization mechanisms. Nevertheless, merely including such synchronization mechanisms into a language is not sufficient for creating a complete parallel program. They neither describe a complete coordination system nor represent complete communication subsystems. To be applied effectively, the synchronization mechanisms have to be organized and included within communication structures, which themselves have to be composed and included in an overall coordination structure [11].

Common synchronization mechanisms for concurrent, parallel and distributed programming can be expressed as idioms, that is, as software patterns for programming code in a particular programming language. Several of such synchronization mechanisms have been already expressed as idioms: the Semaphore idiom, the Critical Region idiom, the Monitor idiom, the Message Passing idiom and the Remote Procedure Call idiom [11]. All these idioms describe the use of the synchronization mechanism with a particular parallel programming language, rather than a formal description of their theory of operation.

The objective of this paper is to show how the idioms that provide a pattern description of well-known synchronization mechanisms can be applied for the Fast Fourier Transform, as a particular programming problem under development. The description of synchronization mechanisms as idioms should aid software designers

and engineers with a description of common programming structures used for synchronizing communication activities within a specific programming language, as well as providing guidelines on their use and application during the design and implementation stages of a parallel software system. This development of implementation structures constitutes the main objective of the Detailed Design step within the Pattern-based Parallel Software Design method [11].

When implementing the components that act as synchronization mechanisms within the communication components of a parallel program, it is important to carefully consider how both communication and synchronization are carried out by such synchronization mechanisms. The Idioms for Synchronization Mechanisms (ISM) [11] stand out from many of the sources, references, and descriptions available about how to implement the synchronization between communicating components (or processes) of a parallel program, with the following advantages:

- The ISM represent programming constructs that express synchronization beyond what is properly included within the parallel programming language, but giving the impression that their use is actually part of the parallel language.
- The ISM attempt to reproduce good programming practices, describing some common programmed structures used to detail and implement the synchronization required by a Design Pattern for Communication Components. Thus, their objective is to help the software designer or programmer understand and master features and details of the parallel programming language at hand, by providing low-level, language specific descriptions of code that are used to synchronize between parallel processing components. These Idioms, then, help to solve recurring programming problems in such a parallel programming language. There has been extensive experience and research about such codification in several different parallel programming languages, but unfortunately, they have not been related or linked with general communication structures or overall structures of parallel programs.
- The ISM are descriptions that relate a synchronization function (in run-time terms) with a coded form (in compile-time terms). In many parallel languages, synchronization mechanisms are implemented so their run-time function has little or no resemblance to the code that performs it. Both, function and code, are difficult to relate, so the software designer or programmer cannot notice how communication and synchronization are carried out by coded components. The Idioms here try to relate function and code, providing dynamic and static information about the synchronization mechanisms.
- ISM describe common coded programming structures based on data exchange and function call. As such, they are guidance about how to achieve synchronization between processing components. This is a key for the success or failure of communication. Hence, the Idioms proposed here are classified based on (a) the memory organization and (b) the type of communication between parallel components. These issues deeply affect the selection of synchronization mechanisms and the implementation of communication components.
- The ISM represent programmed forms as regular organizations of code, aiming to allow software designers to understand the synchronization between component, and therefore, reducing their cognitive burden. Moreover, if these idioms are used and learnt, they ease understanding legacy code, since programs tend to be easier to understand.
- The ISM are based on the common concepts and terms originally used for inter-process communication [1–4, 6–8], and as such, they are a vehicle to develop terminology for implementing synchronization components for parallel programs.

Nevertheless, as it is obvious, the ISM present the disadvantage of being non-portable, since they depend on features of the parallel programming language. This does not exclude that several idioms for expressing synchronization mechanisms can be developed for the different parallel programming languages available.

This paper is the third in a series of three papers: the first paper has covered the application of the Parallel Layers as an architectural pattern for solving the Fast Fourier Transform [12] and; the second paper has presented the application of the Multiple Remote Call as a design pattern for dealing with the design of communication components for the Fast Fourier Transform [13]. Here, this paper explains the selection of idioms to be used in solving synchronization issues within these communication components. Together, the three papers attempt to present a complete picture of a parallel implementation for the Fast Fourier Transform.

2 Specification of the System

In the paper, *Applying Architectural Patterns for Parallel Programming. The Fast Fourier Transform* [12], the

Parallel Layers (PL) Architectural Pattern has been selected as a solution for the coordination within the parallel program that solves the Fast Fourier Transform problem. In order to apply the Idioms for Synchronization Mechanisms (ISM), some information is required related to the PL Pattern, such as the parallel platform and programming language.

For this implementation, the parallel platform available for this parallel program is a cluster of computers, specifically, a dual-core server (Intel dual Xeon processors, 1 Gigabyte RAM, 80 Gigabytes HDD) 16 nodes (each with Intel Pentium IV processors, 512 Megabytes RAM, 40 Gigabytes HDD), which communicate through an Ethernet network. The parallel application for this platform is programmed using the Java programming language [12].

3 Specification of the Communication Components

In the paper *Applying Design Patterns for Communication Components. Communicating Parallel Layer components for the Fast Fourier Transform* [13], the Multiple Remote Call Design Pattern has been selected for the communication components of the PL pattern to solve the Fast Fourier Transform problem. In order to apply the ISM, some information related with the Multiple Remote Call Pattern is required as well. This information is summarized as follows.

3.1 The Multiple Remote Call pattern

The communication components are defined so they enable the exchange of values in a bidirectional, one-to-many and many-to-one, remote communication subsystem, having the form of a tree-like communication structure [13]. Hence, the Multiple Remote Call pattern has already been previously chosen as an adequate solution for such communications [9, 12, 13].

- **Description of the communication**. The Multiple Remote Call (MRC) pattern is used to distribute samples to other processing components in lower layers of the PL pattern, executing on other memory systems. Both the higher- and lower-layer components are allowed to execute simultaneously synchronously communicating during each remote call over the network of the distributed memory parallel system [13].
- **Structure and dynamics**
  1. *Structure.* For background information, the structure of the MRC pattern has been taken from [13], as it is shown in Figure 1, using a UML Collaboration Diagram [5]. Notice that the communication component structure allows a synchronous, bidirectional communication between a higher- and two lower-layer components [10, 13].
  2. *Dynamics.* This pattern actually performs a group of remote calls within the available distributed memory parallel platform. Figure 2 is taken from [13], showing the behavior of the participants of this pattern for the actual example [11, 13].

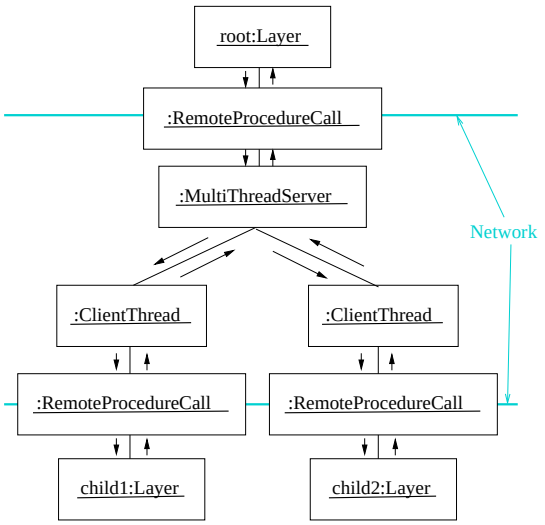

Figure 1. UML Collaboration Diagram of the Multiple Remote Call pattern used for sending samples and combining transforms between layer components of the PL solution to the Fast Fourier Transform problem [13].

4 Detailed Design

This paper introduces the Detailed Design step for solving in parallel the Fast Fourier Transform [11]. In this step, the software designer applies one or more idioms as the basis for synchronization mechanisms. From the decisions taken in the previous steps (Specification of the Problem [12], Specification of the System [12], and Specification of Communication Components [13]), the main objective now is to decide which synchronization mechanisms are to be used as part of the communication substructures.

4.1 Specification of the Synchronization Mechanism

– **The scope.** This section takes into consideration the basic previous information for solving the Fast Fourier Transform problem. The objective is to look for the relevant information

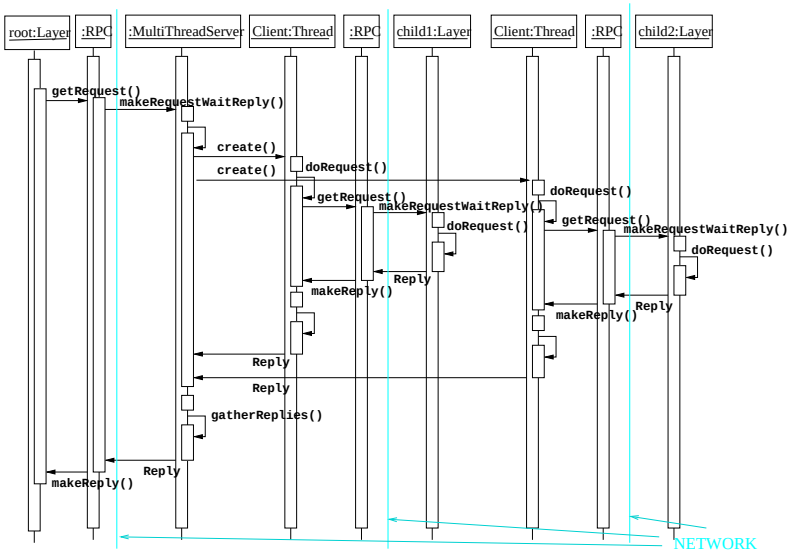

Figure 2. UML Sequence Diagram for the Multiple Remote Call pattern applied for sending samples and combining transforms between a higher- and two lower-layer components of the PL solution for the Fast Fourier Transform problem [13].

for applying a particular idiom as a synchronization mechanism.

For the Fast Fourier Transform problem, the factors that now affect selection of synchronization mechanisms are as follows:

* The available hardware platform is a cluster, this is, a distributed memory parallel platform, programmed using Java as the programming language [12].
* The PL pattern is used as an architectural pattern, requiring to communicate layer software components [12].
* The MRC design pattern is selected for the design and implementation of communication components to support synchronous communication between layers [13].

Based on this information, the procedure for selecting an ISM for the Fast Fourier Transform problem is as follows [11]:

a. *Select the type of synchronization mechanism.* The MRC pattern requires a synchronization mechanism that controls the access and exchange of values between a higher- and two lower-layers as software components that cooperate. These values are communicated using basically remote procedure calls. Hence, the idioms that describe this type of synchronization mechanism are the Message Passing idiom and the Remote Procedure Call idiom [11].

b. *Confirm the type of synchronization mechanism.* The use of a distributed memory platform, given the previous design decisions, confirms that the synchronization mechanisms for communication components in this example may be message passing or remote procedure calls.

c. *Select idioms for synchronization mechanisms.* Communication between layer components needs to be performed synchronously, that is, the high-layer component should wait for a response from its two lower-layer components. This is normally achieved using the MRC pattern. Nevertheless, this design pattern requires synchronization mechanisms. In Java, the Remote Procedure Call idiom allows to develop a synchronization mechanism used here to show how implementation of the MRC pattern can be achieved using this idiom.

d. *Verify the selected idioms.* Checking the Context and Problem sections of the Remote Procedure Call idiom [11]:

   * Context: *'A parallel or distributed application is to be developed in which two or more software components execute simultaneously on a distributed memory platform. Specifically, two software components must communicate, synchronize and exchange data. Each software component must be able to recognize the procedures or functions in the remote address space of the other software component, which is accessed only through I/O operations.'.*
   * Problem: *'To allow communications between two parallel software components executing on different computers on a distributed memory parallel platform, it is necessary to provide synchronous access to calls between their address spaces for an arbitrary number of call and reply operations.'.*

   Comparing these sections with the synchronization requirements of the actual example, the Remoter Procedure Call idiom can be used as the synchronization mechanism for the communication. The use of a distributed memory platform implies the use of message passing or remote procedure calls, whereas the need for synchronous communication between layer components points to the use of remote procedure calls.

The design of the parallel software system can now continue using the Solution section of the Remote Procedure Call idiom, directly implementing it in Java.

– *Structure and Dynamics.*

a. **Structure.** The Remote Procedure Call Idiom is used for implementing the synchronization mechanisms of the communication components for the PL pattern. The Remote Procedure Call idiom in Java is presented as an interface, declaring some basic methods on which synchronization is achieved. Notice that the remote procedure call allows a synchronization over the two basic distributed components: a server and a client [11].

```
interface RemoteProcedureCallInterface{
   public abstract Object makeRequestWaitReply(Object m);
   public abstract Object getRequest();
   public abstract void makeReply();
}
```

b. **Dynamics.** Remote Procedure Calls are used in several ways as synchronization mechanisms. Here, they are used for synchronous communication. The Remote Procedure Call idiom actually synchronizes the operation of the layer components over distributed memory. Figure 3 shows a UML Sequence diagram of the possible execution of the two participants of this idiom as the synchronization mechanism within the MRC pattern. Two parallel software components: a `client c` and a `server s`, which synchronize to exchange values. Since they execute on different nodes of the distributed memory platform, they can only communicate using the remote procedure call methods.

In this scenario, the synchronization over the remote procedure call is performed as follows:

* The communication between software components starts when the `client` invokes `makeRequestWaitReply()`. Assuming that the `remote procedure call` component is free, it receives the call along with its arguments. The `client` waits until the `remote procedure call` component issues a `reply`.
* At the remote end, the `server` invokes `getRequest()` to retrieve any requests issued to the `remote procedure call` component. This triggers the execution of a procedure within the `server`, here `doRequest()`, which serves the call issued by the `client`, operating on the actual parameters of the call.
* Once this procedure finishes, the `server` invokes `makeReply()`, which encapsulates the `reply` and sends it to the `remote procedure` call component.
* Once the `remote procedure call` has the `reply`, it makes it available to the `client`, which unblocks and continues. Note how the `remote procedure call` acts as a synchronization mechanism between `client` and `server`.

– *Synchronization Analysis.* This section describes the advantages and disadvantages of the Remote Procedure Call idiom as a base for the synchronization code proposed [11].

a. **Advantages**

* Multiple parallel layer components can be created in different address spaces of the computers that make up the cluster, as a distributed memory parallel platform. They are able to execute simultaneously, nondeterministically and at different relative speeds. All can execute independently, synchronizing to communicate.
* Synchronization is achieved by blocking every `client` until it receives a reply from the `server`. When implementing `remote procedure calls`, blocking is more manageable than non-blocking: `remote procedure`

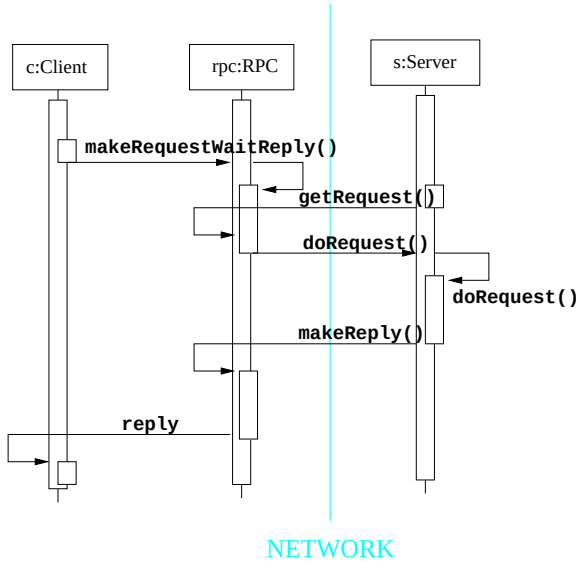

Figure 3. UML Sequence Diagram for the Remote Procedure Call idiom.

`call` implementations map well onto a blocking communication paradigm.
   * Each layer component works its own address space, issuing calls to accessing other layers in a remote address space via network facilities. No other layer component interferes during communication.
   * Data to be sorted is passed as arguments of the remote procedure calls. The integrity of arguments and results is maintained during all communication.

b. **Liabilities**
   * An implementation issue for remote procedure calls in this application example is the number of calls that can be in progress at any time from different threads within a specific layer component. It is important that a number of layer components on a computer within a distributed system should be able to initiate remote procedure calls and, specifically, that several threads of the same layer component should be able to initiate remote procedure calls to the same destination. Consider for example a layer A using several threads to serve remote procedure call requests from different client layers. Layer A may itself need to invoke the service of another layer, say B. It must therefore be possible for a thread on A to initiate a remote procedure call to B and, while it is in progress, another thread on A should be able to initiate other remote procedure calls to layer B.
   * It is commonly argued that the simple and efficient remote procedure call can be used as a basis for all distributed communication requirements of the present Fast Fourier Transform problem. However, there are variations that can be applied here. Such variations include (a) a simple send for event notification, with no requirement for reply, (b) an asynchronous version of a remote procedure call that requests the server to perform the operation and keep the result so the client can picks it up later, (c) a stream protocol for different sources and destinations, such as terminals, I/O and so on.

5 Implementation

The communication components and their respective remote procedure call components are implemented as described in the Detailed Design step, using the Java programming language [12, 13]. So, the implementation is presented here for developing the MRC as communication and synchronization components. Nevertheless, this design and implementation of the whole parallel software system goes beyond the actual purposes of the present paper.

5.1 Communication components – Multiple Remote Calls

The class `RemoteProcedureCall` is used as the synchronization mechanism component of several components of the MRC pattern. For example, let us consider the synchronization within the communication between the high-layer component and the `MultithreadServer`, using remote procedure calls [13].

```
class Layer implements Runnable {
   ...
   private RemoteProcedureCall rpc; // reference to rpc
   private Object data; // Data to be processed
   private Object result; // Result from the call
   ...
   public void run(){
      ...
      rpc = new RemoteProcedureCall(socket s);
      ...
      while(true){
         ...
         result = rpc.getRequest(data);
         ...
      }
   }
}
```

The `MultithreadServer` receives this remote call as follows:

```
class MultithreadServer implements Runnable {
   ...
   private RemoteProcedureCall rpc; // reference to rpc
   private int data[]; // Data to be processed
   private int subData[]; Data to be distributed
   private int reply[]; // Results from client threads
   private int result[]; // Overall result
   private ClientThread clientThread[];
   private int numClients;
   private Boolean request = false; // is there a request?
   ...
   //Function called by the rpc
   private void performRequest(int d[]){
      data = d;
      synchronized(this){
         request = true;
         this.notify();
      }
   }
   ...
   public void run(){
      //Wait until someone make a request
      while(true){
         synchronized(this){
```

```
            while(!request){
                try{wait();}
                catch(InterruptedException e){}
            }
        }
        //Create childthreads
        for(int i=0;i<numClients;i++){
            subdata = getNextSubData(data,i);
            clientThread[i] = new ClientThread(subData);
        }
        //Wait for all child termination
        for(int i=0;i<numClients;i++){
            reply[i] = clientThread[i].returnResult();
            try{
                clientThread[i].join();
            }
            catch(InterruptedException e){}
        }
        result = gatherReplies();
        rpc.makeReply(result);
    }
  }
  ...
}
```

Both components rely on a remote procedure call component to exchange and distribute values as samples and transforms of the FFT computation. Hence, the successful operation of the communication structure relies on how the remote procedure call component implements the methods of the interface `RemoteProcedureCallInterface`: `makeRequestWaitReply()`, `getRequest()`, and `makeReply()`. This is shown in the following section.

5.2 Synchronization Mechanism – Remote Procedure Calls in Java

Based on the Remote Procedure Call idiom and their implementation in the Java programming language, the basic synchronization mechanism that controls the communication between root `Layer` component and the `MultithreadedServer` is presented as follows:

```
import java.net.*;
...

class RemoteProcedureCall extends UnicastRemoteObject
   implements RemoteProcedureCallInterface {

   protected Object data;
   protected Object reply;
   private MultithreadedServer ms;
   ...
   private MessagePassing in = null;
   private MessagePassing out = null;
   ...
   public RemoteProcedureCall(Socket socket) {
      ...
      this.in = new ObjPipedMessagePassing(socket);
      this.out = this.in;
   }
   public Object clientMakeRequestAwaitReply(Object m) {
      send(in, m);
      return receive(out);
   }
   public Object serverGetRequest() {
      return receive(in);
   }
   public void serverMakeReply(Object m) {
      send(out, m);
   }
   ...
}
```

The class `RemoteProcedureCall` implements a two-way flow of information based on sockets, as a one-way flow of information between message passing sender and receiver. The root `Layer` component sends an object to the `MultithreadedServer` that represents a request, and blocks waiting for the reply. The `MultithreadedServer` blocks waiting for a request. When it gets the request, computes the reply, and sends it to the root `Layer` component, unblocking it. As described in the MRC pattern, the `MultithreadedServer` may spawn off a thread to handle the request while it gets additional requests.

Moreover, the `MultithreadedServer` also acts as a call distributor: it waits for requests from the low-layer components that they are able to do some work. The `MultithreadedServer` sends a work command to the layer components, sending the result back later in another call. Notice that this part of the functionality of the MRC pattern is not shown in this code.

The Remote Procedure Calls here are based on synchronous message passing rather than asynchronous because buffering is unnecessary and would waste space; any client blocks on the send in synchronous case and on the receive in asynchronous case. Hence, there is no need of synchronized methods, because synchronization is handled inside the send and receive methods. This method should be synchronized if there are multiple client threads sharing this object.

Finally, it is important to notice that a deadlock possibility exists: if the server makes another call to `serverGetRequest()` before calling `serverMakeReply()` then this `RemoteProcedureCall` object is deadlocked (assuming just one client is using this object, the intended situation) in the sense that the client is blocked on `receive(out)` and the server is blocked on `receive(in)`. This still needs to be fixed for the present implementation.

6 Summary

The ISM are applied here along with a method, in order to show how to apply an idiom that copes with the requirements of the communication components present in the PL solution to the Fast Fourier Transform problem. The main objective of this paper is to demonstrate,

with a particular example, the detailed design and implementation that may be guided by a selected idiom. Moreover, the application of the ISM and the method for selecting them is proposed to be used during the Detailed Design and Implementation for other similar problems that involve synchronous distribution of data, executing on a distributed memory parallel platform.

References

[1] Brinch-Hansen, P., *Structured Multiprogramming.* Communications of the ACM, Vol. 15, No. 17. July, 1972.

[2] Brinch-Hansen, P., *The Programming Language Concurrent Pascal.* IEEE Transactions on Software Engineering, Vol. 1, No. 2. June, 1975.

[3] P. Brinch-Hansen *Distributed Processes: A Concurrent Programming Concept.*, Communications of the ACM, Vol.21, No. 11, 1978.

[4] E.W. Dijkstra *Co-operating Sequential Processes*, In Programming Languages (ed. Genuys), pp.43-112, Academic Press, 1968.

[5] M. Fowler, *UML Distilled.* Addison-Wesley Longman Inc., 1997.

[6] Hoare, C.A.R., *Towards a theory of parallel programming.* Operating System Techniques, Academic Press, 1972.

[7] Hoare, C.A.R., *Monitors: An Operating System Structuring Concept.* Communications of the ACM, Vol. 17, No. 10. October, 1974.

[8] C.A.R. Hoare *Communicating Sequential Processes.* Communications of the ACM, Vol.21, No. 8, August 1978.

[9] J.L. Ortega-Arjona *The Parallel Layers Pattern. A Functional Parallelism Architectural Pattern for Parallel Programming.*, Proceedings of the 6th Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP2007), Porto de Galinhas, Pernambuco, Brazil, 2007.

[10] J.L. Ortega-Arjona *Design Patterns for Communication Components*, Proceedings of the 12th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2007), Kloster Irsee, Germany, 2007.

[11] J.L. Ortega-Arjona *Patterns for Parallel Software Design.* John Wiley & Sons, 2010.

[12] J.L. Ortega-Arjona *Applying Architectural Patterns for Parallel Programming. The Fast Fourier Transform*, Proceedings of the 19th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2014), Kloster Irsee, Germany, 2014.

[13] J.L. Ortega-Arjona *Applying Design Patterns for Communication Components. Communicating Parallel Layer components for the Fast Fourier Transform.*, Proceedings of the 23th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2018), Kloster Irsee, Germany, 2018.