Applying Design Patterns for Communication Components Communicating Parallel Layer components for an Hypercube Sorting

Jorge L. Ortega Arjona

[1]Departamento de Matemáticas Facultad de Ciencias, UNAM
jloa@fciencias.unam.mx

Abstract. *This paper presents the design and implementation of the communication components for a parallel version of the Hypercube Sorting. The method used here makes use of Design Patterns for Communication Components, which take information from the Problem Analysis and Coordination Design, and provide elements about its implementation.*

1. Introduction

Parallel programming is characterized by a growing set of parallel hardware architectures, programming paradigms, and parallel languages. This situation makes difficult to propose just a single approach containing all the details to design and implement communication components for all parallel software systems. Hence, the Design Patterns for Communication Components [15, 17] are proposed as an effort to help a programmer to design the communication components depending on particular characteristics and features of the communication to be carried out between the processing components, when designing a parallel program.

The Design Patterns for Communication Components focus on describing and refining the communication components of a parallel program, by describing common programming structures used for communicating, exchanging data or requesting operations, between processing components. Their application directly depends on the Architectural Pattern for Parallel Programming [13, 16, 17] which they are part of, detailing a communication and synchronization function as a local problem, and providing a form as a local solution of software components for such a communication problem.

When designing the communication components of a parallel program, it is important to think carefully how communication and synchronization are to be actually carried out by those communication components.

However, design patterns for communication are not applied in isolation. A parallel program is the result of applying several patterns at different levels of design and implementation. The design and implementation of a whole parallel program requires applying more than a single pattern. Different patterns are applied at different levels of design. Designing and programming a parallel software system requires, then, several patterns at least at three levels of design: coordination, communication, and synchronization. Several different patterns have been proposed for each one of these levels: architectural patterns for coordination, design patterns for communication, and idioms for

synchronization [17]. The present paper preciselly focuses on the second level of design: communication design.

In this paper, it is presented the application of the multiple remote call pattern for designing the communication components of a parallel program that solves the Hypercube Sorting. For this problem, the paper *"Applying Architectural Patterns for Parallel Programming. An Hypercube Sorting"* [18] has already presented the Communicating Sequential Elements pattern for designing the coordination level of the whole parallel program. Here, this paper continues and complements the design of the whole parallel program, by applying the multiple remote call design pattern for continuing the design of the whole parallel program that solves the Hypercube Sorting. The design development here is part of the method for designing parallel programs as presented in the book *"Pattern for Parallel Software Design* [17]. However, in this paper, only the Communication Design is specifically performed to solve the communication requirements of the Hypercube Sorting, making use of Design Patterns for Communication Components [15, 17], taking information from the architectural decisions in [18], and providing elements about the design and implementation of communication components for the Hypercube Sorting.

2. The Parallel Layers pattern: the Hypercube Sorting case

In the paper *"Applying Architectural Patterns for Parallel Programming. An Hypercube Sorting"* [18], the Parallel Layers Architectural Pattern has been selected as a viable solution for an Hypercube Sorting. Now, in order to apply the Design Patterns for Communication Components for developing the communication components for this example, some information related with the Parallel Layers pattern and the parallel platform and programming language is required. This information is summarized as follows.

2.1. The Parallel Layers pattern

- **Description of the coordination**. The Parallel Layers pattern makes use of functional parallelism to execute sub-algorithms, allowing the simultaneous existence and execution of more than one instance of a layer component through time. Each of these instances are composed of the simplest sub-algorithms. In a layered system, an operation involves the execution of operations in several layers. These operations are triggered by a call, and data is vertically shared among layers in the form of arguments for these function calls. During the execution of operations in each layer, usually the higher layers have to wait for a result from lower layers. However, if each layer is represented by more than one component, they can be executed in parallel and service new requests. Therefore, at the same time, several ordered sets of operations are carried out by the same system. Several computations can be overlapped in time [14, 18, 17].
- **Structure and dynamics**
    1. *Structure*. When applying the PL pattern for the Hypercube Sorting, the set to be sorted is divided over and over until a simple operation can be carried out simultaneously to obtain a partial sorting. Once this is achieved, the sorted result is sent back to the component above, and the sorting is performed again, over and over, until reaching the root component of the whole structure. Hence, the structure of the actual solution involves a tree-like logical structure. Thus, the solution is presented as a tree of processing

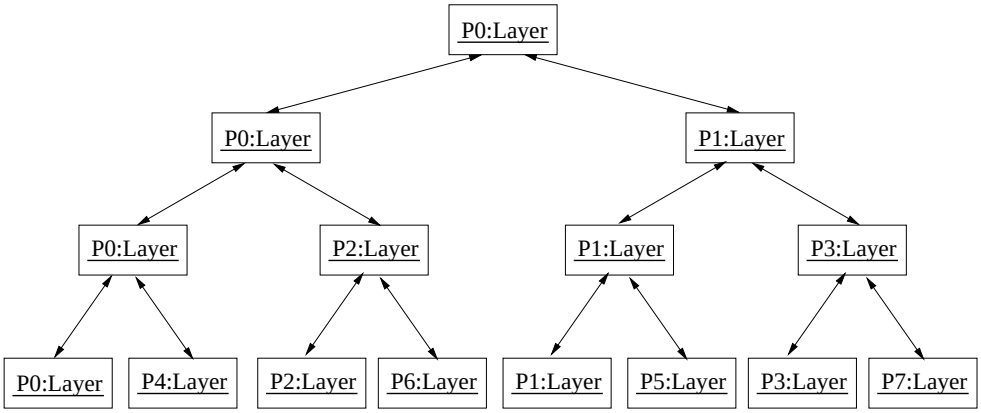

Figure 1. Object Diagram of PL for solving the Hypercube Sorting.

layer components. These are identical components that simultaneously exist and process during the execution time. An Object Diagram, representing this structure is shown in Figure 3 [18, 17].

2. *Dynamics*. A scenario to describe a basic run-time behavior of the PL pattern for solving the Hypercube Sorting is described as follows. Notice that all layer components, as basic processing software components, are active at the same time. Every layer component performs the same division, and once the set is completely divided, the layer component sorts its subset, providing its result to the layer component above. This operation is repeated until the whole set is sorted and made available to the root component of the tree structure as shown in Figure 4 [18, 17].

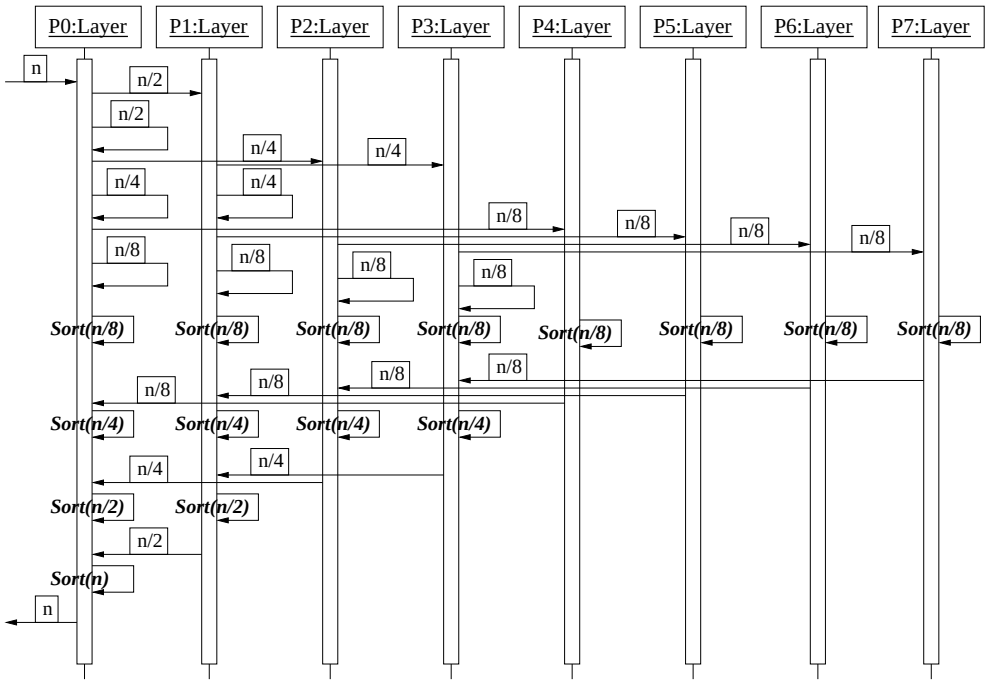

Figure 2. Sequence Diagram of PL for Hypercube Sorting.

2.2. Information about parallel platform and programming language

Information about parallel platform and programming language. 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 [18].

3. Communication Design

3.1. Specification of Communication Components

- **The scope.** This section takes into consideration the basic information about the parallel hardware platform and the programming language used, as well as the PL pattern as the selected coordination for solving the Hypercube Sorting. The objective is to look for the relevant information for choosing a particular design pattern as a communication structure.

  Based on the information about the parallel platform (a distributed memory cluster), the programming language (Java) and the description of software components for the PL pattern presented in the previous section, the procedure for selecting a Design Pattern for the Communication Components for the Hypercube Sorting is presented as follows [15, 17]:

  1. *Consider the architectural pattern selected in the previous step.* From the PL pattern description, the design patterns which provide communication components and allow the behavior as described by this architectural pattern for a coordination are the Multiple Local Call pattern and the Multiple Remote Call pattern [15, 17].
  2. *Select the nature of the communicating components.* Considering that the parallel hardware platform to be used has a distributed memory organization, the nature of the communicating components for such memory organization is considered to be message passing or remote call.
  3. *Select the type of synchronization required for the communication.* Normally, the communication between software components that act as a root and two or more children makes use of a synchronous communication. In a synchronous communications, the root software component calls to its children and blocks, waiting for receiving a response from them. Once the response in received, this software component operates on the results from its children, and acting as a child, provides a results to its own root software component.
  4. *Selection of a design pattern for communication components.* Considering (a) the use of the PL pattern, (b) the distributed memory organization of the parallel platform, and (c) the use of synchronous communications, therefore the **Multiple Remote Call pattern** is proposed here as the base for designing the communications between root and children. Let us consider the Context and Problem sections of this pattern [15, 17]:
     - **Context:** ‘A parallel program is to be developed using the Parallel Layers architectural pattern as a functional parallelism approach

in which an algorithm is partitioned among autonomous processes (layer components) that make up the processing components of the parallel program. The parallel program is to be developed for a distributed memory computer, but also can be used with a shared memory computer. The programming language to be used has rendezvous or remote procedure calls as synchronization mechanisms for remote process communication'.

– **Problem:** 'A collection of distributed, parallel layer components need to communicate by issuing multiple remote procedure calls, synchronously waiting to receive the multiple results of those calls. All data is contained in a distributed layer component and only disseminated to layer components below, or gathered and passed to layer components above'.

From both these descriptions, it is noticeable that for the PL pattern, on a distributed memory parallel platform, and using Java as the programming language, the choice for developing the communication components for this example is the **Multiple Remote Call pattern**. The use of a distributed memory parallel platform implies using remote calls, and it is known that the Java programming language counts with the elements for developing such calls. Moreover, this calls consider a synchronous communication scheme between a 'client' and its 'server'. Therefore, this completes the selection of the Design Pattern for Communication Components of the Hypercube Sorting. The design of the parallel software system continues using the Multiple Remote Call pattern's Solution section as a starting point for communication design and implementation.

- **Structure and dynamics.** This section takes information of the Multiple Remote Call design pattern, expressing the interaction between its software components that carry out the communication between parallel software components for the actual example.

  1. *Structure.* The structure of this pattern applied for designing and implementing remote call communication components for the PL pattern is shown in Figure 3 using a UML Collaboration Diagram [6]. Notice that this component structure allows a synchronous, bidirectional communication between a root component and a group of children. The synchronous feature is achieved by using a barrier synchronization on the root side, so the root component does wait for all its children [15, 17].
  2. *Dynamics.* This pattern actually performs a groups of remote calls within the available distributed memory parallel platform. Figure 4 shows the behavior of the participants of this pattern for the actual example.
     In this scenario, a group of bi-directional, synchronous remote calls is carried out, as follows:
     – The root component issues a remote procedure call through a remote procedure call component to the multithread server, which executes on a different processor within the distributed memory computer. Once this remote procedure call has been issued, the root component blocks, waiting for a result.

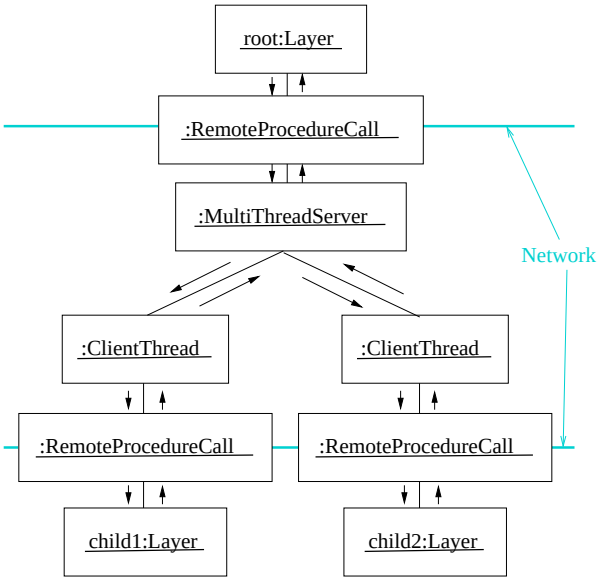

Figure 3. UML Collaboration Diagram of the Multiple Remote Call pattern used for synchronous remote calls between root and two children of the PL solution to the Hypercube Sorting.

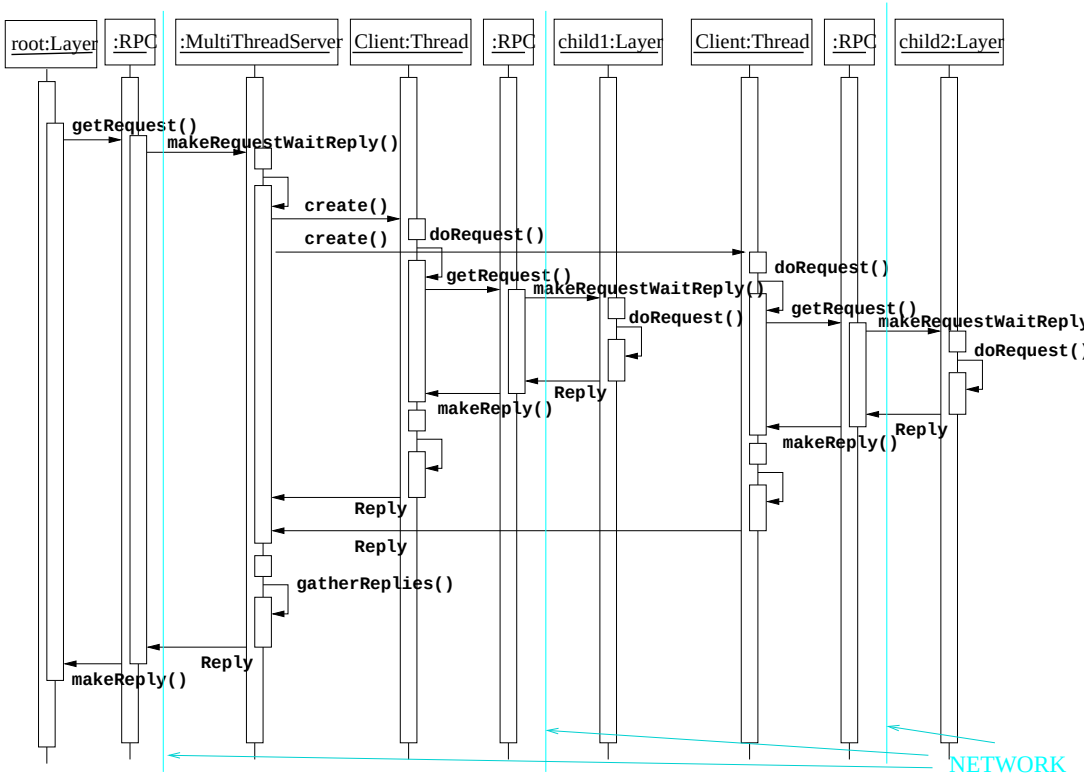

Figure 4. UML Sequence Diagram for the Multiple Remote Call pattern applied for synchronous remote calls between root and two children of the PL solution to the Hypercube Sorting.

- The multithread server receives the remote call from the remote procedure call component through the network and creates a group of client threads to distribute the call to child components executing on other computers.
- Once created, each client thread is passed part of the data and transmits it by issuing a remote procedure call through a new remote procedure call component, one for each client thread. Remote procedure call components have been proposed and used as communication and synchronization mechanisms for distributed memory environments: here they are used to maintain the synchronous feature of communications within the whole Parallel Layers structure, distributed among several processors. Once every call is issued to remote processes, all the client threads wait until they receive the results from the remote procedure call components.
- Once each child component produces a result, it returns it through the network to the remote procedure call component that originally called it, and thus to its respective client thread.
- Each client thread passes its result to the multithread server. Once results have been received from all client threads, the multithread server assembles them into a single result, which is passed through the network via the remote procedure call component to the remote root component that originally issued the call.

3. *Functional description of software components.* This section describes each software component of the Multiple Remote Call pattern as the participant of the communication sub-system, establishing its responsibilities, input, and output.
   (a) **Multithread server.** The responsibilities of the multithread server component are to receive remote procedure calls and their respective data, as arguments, from a higher-layer component, divide the data and create a client thread for each data subset. The server then waits for all client threads to produce their results: once received, the multithread server assembles an overall result and returns it to the higher-layer component that originally called it.
   (b) **Client thread.** The responsibilities of each client thread, once created, are to receive a local call from the multithread server with a subset of data to be operated on, and to generate a remote procedure call to a single layer component on the layer below. Once the called procedure produces a result, the client thread retrieves it, returning it to its multithread server.
   (c) **Remote procedure call.** The remote procedure call components in this pattern have two main responsibilities: *(a)* to serve as a communication and synchronization mechanism, allowing bidirectional synchronous communication between any two components it connects (which execute on different computers), and *(b)* to serve as a remote communication stage within the distributed memory organization between the components of adjacent layers, decoupling them so that communications between them are performed

synchronously. Remote procedure calls are normally used for distributed memory environments.

4. *Description of the communication.* The Multiple Remote Call pattern provides a bidirectional, one-to-many and many-to-one, remote communication subsystem for Hypercube Sorting solution, based on the PL pattern. This subsystem has the form of a tree-like communication structure. It describes a set of communication components that disseminate remote calls to multiple communication components executing on different processors or computer systems. These communication components act as surrogates or proxies of the processing components, sorting local subsets of `int` variables, and then, returning a sorted array. Hence, this pattern is used to distribute a part of the whole set to be sorted to other processing components in lower layers, executing on other memory systems. Both the higher- and lower-layer components are allowed to execute simultaneously. However, they must communicate synchronously during each remote call over the network of the distributed memory parallel system.
5. *Communication Analysis.* This section describes the advantages and disadvantages of the Multiple Remote Call pattern as a base for the communication structure proposed.
   (a) **Advantages**
      – The Multiple Remote Call pattern preserves the precise order of sorting operations, since it represents a single stage within a cascade of synchronous remote procedure calls. Hence the multithread server is able to continue only when all the child components of a layer have completed their operations.
      – As only one multithread server is used to call and synchronize several local client threads, corresponding to several child components, one-to-many communication is maintained during the distribution of data and many-to-one when retrieving results. This is useful from a reusability standpoint.
      – As only synchronous calls are allowed, the integrity and order of the sorted results are preserved.
      – The implementation includes the use of remote procedure calls as synchronization mechanisms. This simplifies their implementation and use for the distributed memory parallel platform available.
      – All communications are synchronous.
   (b) **Liabilities**
      – The use of the Multiple Remote Call pattern may produce long delays in communication between remote components on different layers due to the use of remote calls through the network between components. As every layer component has to wait until all operations on the next lower layer are carried out, communication through the entire distributed hierarchical structure could be slowed due to the number

of component per layer and the volume of communication between root and child components.

4. Implementation

In this section, all the software components described in the Communication Design step are considered for their implementation using the Java programming language. Here, it is only presented the implementation of the communication sub-system, which interconnects processing components that implement the actual computation that is to be executed in parallel [18]. So, the implementation is presented here for developing the multiple remote calls 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.

4.1. Synchronization Mechanism – Remote Procedure Calls

Based on the Java programming language, an interface for the remote procedure call that provides the basic functionalities of a synchronization mechanism for the Multiple Remote Call pattern is presented as follows:

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

The interface `RemoteProcedureCall` presents three abstract methods which allow to produce the calls between distributed objects and allow a synchronous communication between `root` and `child` components. This interface is used in the following implementation stage as the basic synchronization element of the remote call components.

The methods of the interface `RemoteProcedureCall` are normally used in a common ‘client-server’ way: the method `makeRequestWaitReply()` is used by any ‘client’ component to generate a remote procedure call. It then blocks until it receives a result. The method `getRequest()` is used by any ‘server’ to receive the remote procedure call. Finally, the method `makeReply()` is used by the ‘server’ to communicate a result to the client remotely, unblocking it.

4.2. Communication components

Using the interface `RemoteProcedureCall` from the previous section, here it is used as the synchronization mechanism component as described by the Multiple Remote Call pattern, in order to be implemented within the class `Layer`. In the current example, the any layer component, acting as a root (or client), performs the `method getRequest()`, directed to the remote multithread server through the respective remote procedure call component, as follows:

```
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);
              ...
         }
    }
}
```

Notice that the `RemoteProcedureCall` component has a `socket` as argument. This means that this component makes use of the network to carry out its operation, translating the call into a synchronous remote call to the `MultithreadServer` through the method `makeRequestWaitReply()`. The `MultithreadServer` that receives this remote call is shown 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);
       }
   }
   ...
}
```

The `MultithreadServer` is in charge of creating several new `ClientThreads`. These handle that part of the `int` data to be processed by each call: after creating all of them, the `MultithreadServer` waits until all the results are received. The `MultithreadServer` then gathers all results and sends them back to the root component through the remote procedure call component, which keeps the root component waiting until it receives the results.

Now, the code for the `ClientThread` is shown as follows.

```
class ClientThread extends Thread{
   ...
   private RemoteProcedureCall rpc; //reference to rpc
   private int data[]; //Data to be processed
   private int result[]; //Result from the call
   private Boolean isResult = false; //Is there result
   ...
   public ClientThread(int data){
       this.data = data;
       this.start();
   }
   ...
   public void run(){
       synchronized(result){
           result = doRequest();
           isResult = true;
           result.notify();
       }
       ...
   }
   ...
   private []int doRequest(){
       ...
       rpc = new RemoteProcedureCall(socket);
       ...
       return rpc.getRequest(data);
   }
   ...
   public Object returnResult(){
       synchronized(result){
           while(!isResult){
              try{wait();} //Wait for result become available
              catch(InterruptedException e){}
           }
       }
       return result[];
   }
}
```

Each `ClientThread` acts as a single server for the child components in the layer below, this is, the components in one layer are the ‘clients’ for the lower layers, and

at the same time the 'servers' for the higher layers. Notice that the code of the respective `RemoteProcedureCall` component again makes use a `socket`, allowing it to make use of the network to communicate with the child layer components.

Each `ClientThread` starts working when created, performing the `doRequest()` method and receiving the `int` data it should send to its respective child layer components. The `ClientThread` does this through a `RemoteProcedureCall` component. Once it receives a `result`, the `ClientThread` sends it back to the `MultithreadServer`, which assembles the overall `result` and replies to the root layer component through the `RemoteProcedureCall` that originally issued the whole call.

5. Summary

The Design Patterns for Communication Components are applied here along with a method for selecting them, in order to show how to cope with the requirements of communication present in the Hypercube Sorting 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 design pattern. Moreover, the application of the Design Patterns for Communication Components and the method for selecting them is proposed to be used during the Communication Design and Implementation for other similar problems that involve the distribution of data between identical processing components executing on a distributed memory parallel platform.

6. Ackowledgements

This work is part of an ongoing research in the Departmento de Matemáticas. Facultad de Ciencias, UNAM, funded by project IN109010-2, PAPIIT-DGAPA-UNAM, 2010.

References

[1] G.R. Andrews *Foundation of Multithreaded, Parallel and Distributed Programming.*, Addison-Wesley Longman, Inc., 2000.

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

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

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

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

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

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Systems. Addison-Wesley, Reading, MA, 1994.

[8] S. Hartley *Concurrent Programming. The Java Programming Language.*, Oxford University Press Inc., 1998.

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

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

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

[12] S. Kleiman, D. Shah, and B. Smaalders *Programming with Threads,* 3rd ed. SunSoft Press, 1996.

[13] J.L. Ortega-Arjona and G.R. Roberts *Architectural Patterns for Parallel Programming*, Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing (EuroPLoP98), Kloster Irsee, Germany, 1998.

[14] J.L. Ortega-Arjona *The Parallel Layers Pattern. A Functional Parallelism Architectural Pattern for Parallel Programming.*, 6th Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP 2007), Porto de Galinhas, Pernambuco, Brasil. 25-31 May, 2007.

[15] 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.

[16] J.L. Ortega-Arjona *Architectural Patterns for Parallel Programming. Models for Performance Estimation*, VDM Verlag, 2009.

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

[18] J.L. Ortega-Arjona *Applying Architectural Patterns for Parallel Programming. An Hypercube Sorting*, Submitted to the 15th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2010), Kloster Irsee, Germany, 2010.

[19] Shalloway, A., and Trott, J.R., *Design Patterns Explained: A New Perspective on Object-Oriented Design.* Software Pattern Series. Addison-Wesley, 2002.