Applying Design Patterns for Communication Components

Communication between Manager and Worker components for an N-body Simulation

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

Abstract

This paper presents the application of the communication components for a parallel version of the N-body simulation. 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 [4, 6] 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 [2, 5, 6] 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 application 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 [6]. The present paper preciselly focuses on the second level of design: communication design.

In this paper, it is presented the application of the Remote Rendezvous pattern for designing the communication components of a parallel program that performs an N-body simulation. For this problem, the paper *"Applying Architectural Patterns for Parallel Programming. An N-body Simulation"* [7] has already presented the Manager-Workers 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 Remote Rendezvous design pattern for continuing the design of the whole parallel program that performs the N-body simulation. The design development here is part of the method for designing parallel programs as presented in the book *"Pattern for Parallel Software Design* [6]. However, in this paper, only the Communication Design is specifically performed to solve the communication requirements of the Manager-Workers-based N-body simulation, making use of Design Patterns for Communication Components [4, 6], taking information from the architectural decisions in [7], and providing elements about the design and implementation of communication components for the N-body simulation.

2 The Manager-Workers pattern: the N-body simulation case

In the paper *"Applying Architectural Patterns for Parallel Programming. An N-body Simulation"* [7], the Manager-Workers Architectural Pattern has been selected as a viable solution for an N-body simulation. Now, in order to apply the Design Patterns for Communication Components for developing the communication components for this example, some information related with the Manager-Workers pattern, the parallel platform, and programming language is required. This information is summarized as follows.

2.1 The Manager-Workers pattern

- **Description of the coordination**. The Manager-Workers (MW) architectural pattern uses activity parallelism to execute the N-body simulation, allowing the simultaneous existence and execution of more than one worker components through time. Each one of these instances at the same time obtain the force summation and time integration for a single body, and during a time step. In a parallel system like this, the N-body simulation involves the distribution and execution of data for several time steps. Each time step starts by distributing the data among all workers, and finish only when all workers provide the manager with the new state of its correspondent body [3, 7, 6].

- **Structure and dynamics**

  1. *Structure*. Using the Manager-Worker architectural pattern for an N-body simulation, all bodies information is distributed by a manager component, and operated by workers as conceptually-independent components. Each worker performs the same operations to obtain an update of the state of a body, at a time step. So, the operations defined for a body are simultaneously performed. An Object Diagram, representing this structure is shown in Figure 1 [7, 6]. Notice that this organization effectively allows to distribute the data of all bodies among worker components, as previously described in the Problem Analysis [7], so each body's new state can be computed independently from the others.

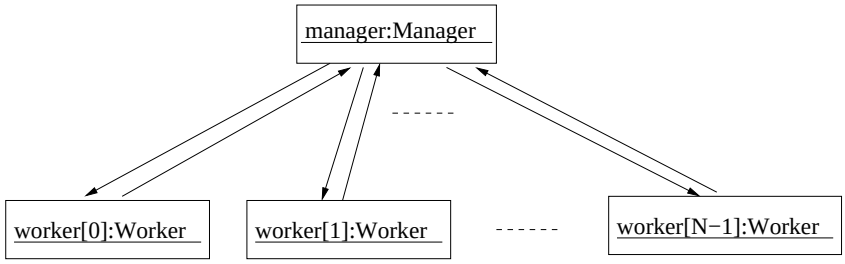

Figure 1: Object Diagram of MW for solving the N-body simulation.

  2. *Dynamics*. A typical scenario is used here to describe the basic runtime behavior of this pattern when applied to the *N-body* simulation. All components, whether manager or workers, are active at the same time, distributing and processing the information of different bodies, and assemlbling an overall state of the system at each time step. A single time step is described in Figure 2 [7, 6].

     The processing and communicating scenario is as follows [7, 6]:

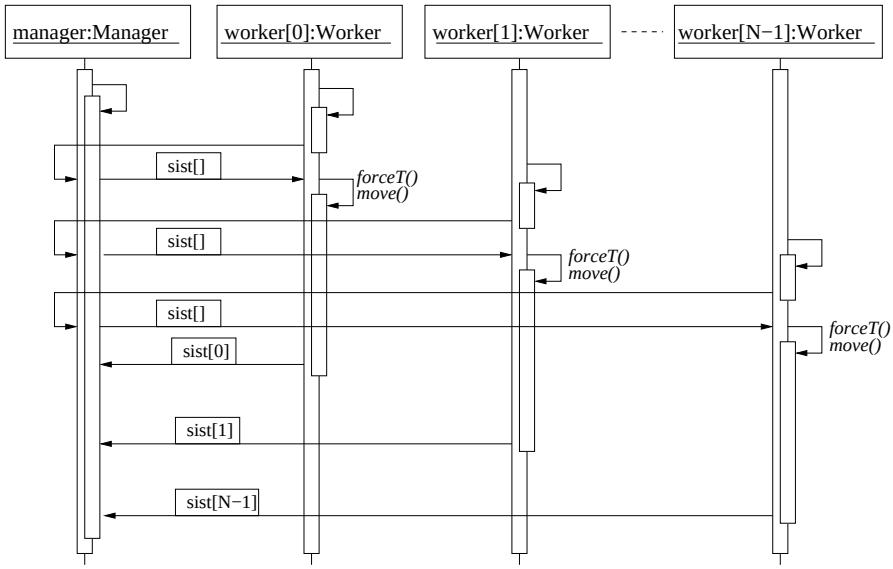

Figure 2: Sequence Diagram of MW for the N-body simulation.

- All participants are created, and wait until a data array of all the bodies `sist[]` is provided to the manager. When such data is available to the manager, it sends all the array by request to each waiting worker.
- Each worker receives a copy of the array. Notice that the i-th worker is associated with obtaining the force summation and the time integration for the i-th body. So, every worker starts processing an operation `forceT()` and `move()`. These operations are independent of the operations on other workers. When the i-th worker finishes processing, it returns a result of only the state of the i-th body to the manager. Once all results are received by the manager, and if the simulation is to be continued for further time steps, each worker requests again for the array representing all the state of all the bodies, and the process repeats.
- For each time step, the manager is usually replying to requests of the array from the workers or receiving their partial results. Once all time steps have been processed, the manager assembles a total result from the partial results and the program finishes. Any non-serviced requests of data from the workers are ignored.

3. *Functional description of components.* The processing and communicating software components for simulating the N-body problem using the MW pattern are described as follows [7].
   - **Manager**. The responsibilities of a manager are to create a

number of workers, to distribute work among them, to start up their execution, and to assemble the overall simulation result from the sub-results from the workers.

– **Worker**. The responsibility of a worker is to seek for the array of bodies, to implement the operations of force summation and time integration on a single body, and to perform such operations.

2.2 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 [7].

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 MW pattern as the selected coordination for solving the N-body simulation. The objective is to look for the relevant information for applying 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 MW pattern presented in the previous section, the procedure for selecting a Design Pattern for the Communication Components for the N-body simulation is presented as follows [4, 6]:

  1. *Consider the architectural pattern selected in the previous step.* From the MW pattern description, the design patterns which provide communication components and allow the behavior as described by this architectural pattern for a coordination are the Local Rendezvous pattern and the Remote Rendezvous pattern [4, 6].
  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 manager and two or more workers makes use of a synchronous communication. In each synchronous communication, a worker component calls to the manager component and blocks, waiting for receiving a response from it. Once a result is received from all workers, the manager component either assembles the global result, or continues distributing work among the workers by receiving further calls.

4. *Selection of a design pattern for communication components.* Considering (a) the use of the MW pattern, (b) the distributed memory organization of the parallel platform, and (c) the use of synchronous communications, therefore the **Remote Rendezvous pattern** is proposed here as the base for designing the communications between manager and each worker. Let us consider the Context and Problem sections of this pattern [4, 6]:

   – **Context:** 'A parallel program is to be developed using the ManagerWorkers architectural pattern (...) as an activity parallelism approach in which data is partitioned among autonomous processes that make up the processing components of the parallel program. The parallel program is to be developed for a distributed memory computer (although it also can be used on a shared memory computer). The programming language to be used includes synchronization mechanisms for interprocess communication through remote procedure calls'.

   – **Problem:** 'A means of communication is required that allows processes to read and write data by sending and receiving data objects from the manager (...), within a distributed memory system'.

   From both these descriptions, it is noticeable that for the MW 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 **Remote Rendezvous 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 worker and its manager. Therefore, this completes the selection of the Design Pattern for Communication Components for the N-body simulation. The design of the parallel software system continues using the Remote Rendezvous pattern's Solution section as a starting point for communication design and implementation.

- **Structure and dynamics.** This section takes information of the Remote Rendezvous 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 MW pattern is shown in Figure 3 using a UML Collaboration Diagram [1]. Notice that this component structure allows a synchronous, bidirectional communication between a manager component and a worker component. The synchronous feature is achieved by using issuing a call from the manager component, which does wait for the related worker response [4, 6].

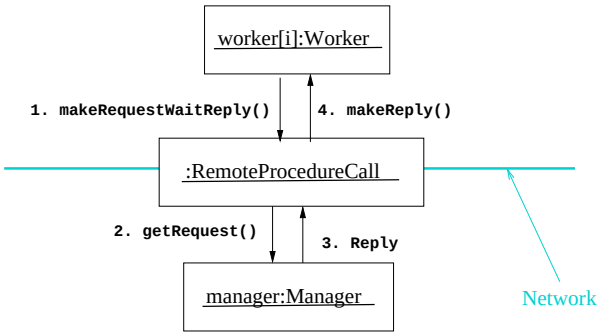

Figure 3: UML Collaboration Diagram of the Remote Rendezvous pattern used for synchronous remote calls between the manager and a worker of the MW solution to the N-body simulation.

2. *Dynamics.* This pattern actually performs a remote call through 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 worker requests data from the manager, so it issues a request operation to its remote procedure call component. This redirects the call to the manager through a socket, synchronizing the call so the worker remains blocked until it receives a response. If it made a read request for data, it waits until the data is made available: if it made a write request, the worker blocks until it receives an acknowledgement from the manager.

   – The manager receives the request. If it is a request for data, it makes the data available by issuing a reply to the remote procedure call component (normally via a socket). On the other hand, if the request was for a write operation, the manager writes the partial result at the relevant place within the data structure and issues an acknowledgement message to the worker, enabling the it to request more work, if needed.

3. *Functional description of software components.* This section describes

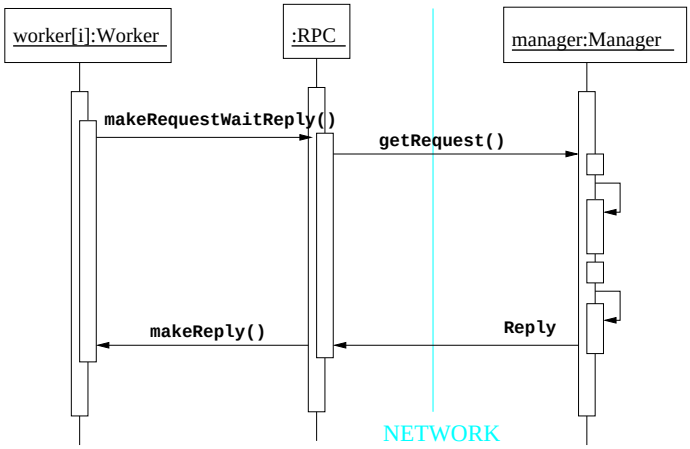

Figure 4: UML Sequence Diagram for the Remote Rendezvous pattern applied for synchronous remote calls between manager and a worker of the MW solution for the N-body simulation.

each software component of the Remote Rendezvous pattern as the participant of the communication sub-system, establishing its responsibilities, input, and output.

(a) **Worker.** The worker component has responsibility for requesting read operations of the set of bodies that act on a particular body, processing these information, and requesting write operations of the new position of such a body.

(b) **Manager.** The manager component has responsibility for maintaining the integrity and order of the local data structure of bodies, and serving read and write requests from the workers.

(c) **Remote procedure call.** The remote procedure call components in this pattern have two main responsibilities: (a) to serve as a remote communication and synchronization mechanism, allowing bidirectional synchronous communication between any two components on different computers that it connects, and (b) to serve as a remote communication stage for the distributed memory organization between the components, decoupling them so that communication between them is synchronous. Remote procedure calls are normally used for distributed memory environments.

4. *Description of the communication.* The Remote Rendezvous pattern provides a bidirectional, one-to-one, remote communication subsystem for the N-body simulation, based on the MW pattern. This subsystem is based on a remote procedure call communication structure. It describes a communication component that performs a remote calls

to components executing on different processors or computer systems. Hence, this pattern is used to allow the communication of a body's information from the manager to a worker, and viceversa. The manager and all workers 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 Remote Rendezvous pattern as a base for the communication structure proposed.

   (a) **Advantages**
   - The integrity and order of the bodies data structure is maintained by allowing only point-to-point, bidirectional synchronous read/write operations between workers and manager.
   - The implementation is carried out for a distributed memory programming environment, although it can also be used on a shared memory platform.

   (b) **Liabilities**
   - The use of synchronous communications between manager and workers slows the performance of the whole program, particularly if the number of workers is large and/or they are located far from the manager, or when communications are very frequent. This problem can be mitigated by changing the granularity of the data available in read operations and/or inserted into the bodies data structure in a write operation.
   - Even though this pattern can be used on a shared memory platform, it tends to make communications between manager and workers complex and slow due to the number of components involved. An alternative would be to use the Local Rendezvous pattern [6].

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 [7]. So, the implementation is presented here for developing the remote rendezvous 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 Remote Rendezvous 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 `manager` and `worker` 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 Remote Rendezvous pattern, in order to be implemented and used within the class `Worker`. In the current example, the worker component, acting as a client, performs the `method getRequest()`, directed to the manager through the respective remote procedure call component, as follows:

```
class Worker implements Runnable {
    ...
    private RemoteProcedureCall rpc; // reference to rpc
    private Body [] sist; // Array of bodies to be processed
    private Body [] reply; // Result from the movement of this body
    private Boolean ack; // Acknowledge for write
    ...
    public void run(){
         ...
         rpc = new RemoteProcedureCall(socket s);
         ...
         while(true){
              ...
              result = rpc.makeRequestWaitReply(sist); // read sist[]
```

```
            ...
            // Perform operations forceT() and move() for the
            // actual body correspondent to this worker
            ...
            ack = rpc.makeRequestWaitReply(reply); // write result
        }
    }
}
```

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 `Manager` through the method `makeRequestWaitReply()`. The `Manager` that receives this remote call is made as a multithreaded server, as shown as follows:

```
class Manager implements Runnable {
   ...
   private RemoteProcedureCall rpc; // reference to rpc
   private Body sist[]; // Data to be processed
   private Body reply[]; // Results from client threads
   private Body result[]; // Overall result
   private WorkerThread workerThread[];
   private int numWorkers;
   private Boolean request = false; // is there a request?
   ...
   //Function called by the rpc
   private void getRequest(Body d[]){
      data = d;
      synchronized(this){
         request = true;
         this.notify();
      }
   }
   ...
   public void run(){
      //Wait until a worker makes a request
      while(true){
         synchronized(this){
            while(!request){
                try{wait();}
                catch(InterruptedException e){}
            }
         }
         //Create workerThreads
         for(int i=0;i<numWorkers;i++){
            workerThread[i] = new WorkerThread(sist);
         }
         //Wait for all workers termination
         for(int i=0;i<numWorkers;i++){
            reply[i] = workerThread[i].returnResult();
```

```
            try{
               workerThread[i].join();
            }
            catch(InterruptedException e){}
         }
         result = gatherReplies();
         rpc.makeReply(result);
      }
   }
   ...
}
```

The `Manager` is in charge of creating several new `WorkerThreads`. These handle that part of the `sist` to be processed by each call: after creating all of them, the `Manager` waits until all the results are received. The `Manager` then gathers all results.

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

```
class WorkerThread extends Thread{
   ...
   private RemoteProcedureCall rpc; //reference to rpc
   private Body data[]; //Data to be processed
   private Body result[]; //Result from the call
   private Boolean isResult = false; //Is there result
   ...
   public WorkerThread(Body 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 Body returnResult(){
       synchronized(result){
           while(!isResult){
```

```
            try{wait();} //Wait for result become available
            catch(InterruptedException e){}
         }
      }
      return result[];
   }
}
```

Each `WorkerThread` acts as a single thread for managing each worker components. 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 worker components.

Each `WorkerThread` starts working when created, performing the `doRequest()` method and receiving the `sist` it should send to its respective worker component. The `WorkerThread` does this through a `RemoteProcedureCall` component. Once it receives a `result`, the `WorkerThread` sends it back to the `Manager`, which assembles the overall `result`.

5 Summary

The Design Patterns for Communication Components are used here along with a method for applying them, in order to show how to cope with the requirements of communication present in the N-body simulation. The main objective of this paper is to demonstrate, with a particular example, the detailed design and implementation that may be guided by applying a design pattern. Moreover, the application of the Design Patterns for Communication Components and the method for applying 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.

References

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

[2] 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.

[3] J.L. Ortega-Arjona *The Manager-Workers Pattern. An Activity Parallelism Architectural Pattern for Parallel Programming.*, 9th European Conference on Pattern Languages of Programming and Computing 2004 (EuroPLoP2004) Kloster Irsee, Germany. 7-11 july, 2004

[4] 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.

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

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

[7] J.L. Ortega-Arjona *Applying Architectural Patterns for Parallel Programming. An N-body Simulation*, Accepted to the 2nd Asian Conference on Pattern Languages of Programs (AsianPLoP2011), Tokyo, Japan, 2011.