Applying Design Patterns for Communication Components

Communicating CSE components for the One-dimensional Heat Equation

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

Abstract

The design patterns for communication components is a collection of patterns related with a method for developing the communication subsystems of parallel software systems. These design patterns are applied depending on *(a)* the architectural pattern of the overall parallel software system, *(b)* the memory organization of the parallel hardware platform, and *(c)* the type of synchronization required.

In this paper, it is presented the application of the design patterns along with the method for communicating the Communicating Sequential Elements components of the One-dimensional Heat Equation. The method used here takes the information from the problem analysis and coordination design, applies a design pattern for the communication components, and provides elements about its implementation.

This paper is aimed at those interested in the design of parallel software and require a base to understand it, with a background in parallel programming and software patterns, particularly the *Patterns for Parallel Software Design*. The information provided in this paper attempts to guide developers and programmers during the communication design and implementation using design patterns for communication components.

1 Introduction

Parallel programming is characterized by a growing set of parallel hardware architectures, programming paradigms, and parallel languages. This situation makes it 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 presented 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, 17], 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. From the many descriptions about how to organize the communication components of a parallel program [12, 8, 1] the design patterns for communication components [15, 17] have the following advantages:

- The design patterns for communication components describe some common structures used to refine and to detail the communications and synchronization required by an architectural pattern for parallel programming [13, 17]. From this point of view, their objective is to help the software designer or programmer by providing medium-scale descriptions of software compounds that are used to communicate between parallel processing components. There has been an extensive research and development of such software compounds [12, 8, 1], but unfortunately, normally they have not been related or linked with overall structures of parallel programs.

- The design patterns for communication components are descriptions about how to relate a communication function (in run-time terms) with a coded form (in compile-time terms). In many parallel applications, communication components are designed so their run-time communication function is organized having little resemblance to the compile-time organization of code that performs it. In fact, both organizations are by far very independent from each other, making it difficult to notice how communication is performed by coded components. The design patterns for communication components attempt to connect both descriptions into a single description that provides both, dynamic and static information about those communication components.

- The design patterns for communication components describe software subsystems or sub-structures for data exchange and/or function call. As such, they are a guidance about how to find the set of software component that perform communication and synchronization between processing components. The communication between processing components of a parallel software system is a key for the success or failure of such a parallel software system. Hence, the design patterns for communication components are developed and classified based on *(a)* the architectural pattern used for the overall parallel software system, *(b)* the memory organization of the parallel hardware platform, and *(c)* the type of synchronization in the parallel programming language available. These three features strongly affect the design and implementation of communication software components. Thus, the design patterns for communication components describe and document several generic and different software structures for communication components, referring to different kinds of architectural patterns (Parallel Pipes and Filters, Parallel Layers, Communicating Sequential Elements, Manager-Workers, and Shared Resource) [13, 17], the hardware platform memory organization (shared memory or distributed memory), and the different types of synchronization available is common programming languages (synchronous or asynchronous).

- Normally, the design patterns for communication components are described so their components are explained in object-oriented terms. This does not prevent that they can be developed using other different paradigms and programming languages (such as process- or task- oriented). Nevertheless, such a description allows to take into consideration and advantage object-oriented principles, such as the separation between interfaces and implementation, reuse, and modifiability of code [7, 18].
- The design patterns for communication components introduce communication structures as forms in which software components are assembled or arranged together in order to perform communication. The communication structures are represented by forms as regular organizations of software components, aiming to allow software designers to understand complex communication software sub-systems, and therefore, reducing their cognitive burden.
- The design patterns for communication components are based on the common concepts and terms originally used for inter-process communication [5, 9, 2, 10, 3, 11, 4]. As such, these design patterns make use and provide elements to develop a terminology for designing communication components for parallel programs.

The following table presents a brief summary of the design patterns for communication components, and their main classification features [15, 17].

Design Pattern	**Type of Parallelism**	**Memory Organization**	**Type of Synchronization**
Shared Variable Pipe	Functional	Shared Memory	Asynchronous
Multiple Local Call	Functional	Shared Memory	Synchronous
Message Passing Pipe	Functional	Distributed Memory	Asynchronous
Multiple Remote Call	Functional	Distributed Memory	Synchronous
Shared Variable Channel	Domain	Shared Memory	Asynchronous
Message Passing Channel	Domain	Distributed Memory	Asynchronous
Local Rendezvous	Activity	Shared Memory	Synchronous
Remote Rendezvous	Activity	Distributed Memory	Synchronous

2 Specification of the System

In the paper *"Applying Architectural Patterns for Parallel Programming. Solving the One-dimensional Heat Equation"* [16], the Communicating Sequential Elements architectural pattern has been selected as a viable solution for the One-dimensional Wave Equation. Now, in order to apply the design patterns for communication components for developing the communication components for this example, some information related with the Communicating Sequential Elements pattern and the parallel platform and programming language is required. This information is summarized as follows.

2.1 The Communicating Sequential Elements pattern

The algorithmic solution for the One-dimensional Heat Equation is defined in terms of calculating the next temperature of the wire segments as ordered data. Each segment is operated almost autonomously. The exchange of data or communication should be between neighboring segments of the wire. So, the Communicating Sequential Elements (CSE) pattern has been chosen as an adequate solution for the One-dimensional Heat Equation. The design of the parallel software system has been continued based on the Solution section of the CSE pattern, as briefly described as follows [14, 16].

- **Description of the coordination**. The CSE pattern describes a coordination in which multiple **Segment** objects act as concurrent processing software components, each one applying the same temperature operation, whereas **Channel** objects act as communication software component which allow exchanging temperature values between sequential components. No temperature values are directly shared among **Segment** objects, but each one may access only its own private temperature values. Every **Segment** object communicates by sending its temperature value from its local space to its neighboring **Segment** objects, and receiving in exchange their temperature values. This communication is normally asynchronous, considering the exchange of a single temperature value, in a one to one fashion. Therefore, the data representing the whole one-dimensional wire represents the regular logical structure in which data of the problem is arranged. The solution, in terms of a segmented wire, is presented as a network that actually reflects this logical structure in the most transparent and natural form [14, 16].

- **Structure and dynamics**

    1. *Structure*. When applying the CSE pattern for the One-dimensional Heat Equation, the same operation is applied simultaneously to obtain the next temperature values of each segment. However, this operation depends on the partial results in its neighboring segments. Hence, the structure of the actual solution involves a regular, one-dimensional, logical structure, conceived from the wire of the original problem. Thus, the solution is presented as a one-dimensional network of segments that follows the shape of the wire. Identical components simultaneously exist and process during the execution time. An object diagram, representing the network of segments that follows the one-dimensional shape of the wire and its division into segments, is shown in Figure 1 [16].

    2. *Dynamics*. A scenario to describe a basic run-time behavior of the CSE pattern for solving the One-dimensional Heat Equation is shown as follows. Notice that all the segments, as basic processing software components, are active at the same time. Every segment performs

the same temperature operation, as a piece of a processing network. However, for the one-dimensional case here, each segment object communicates with its previous and next neighbors as shown in Figure 2 [16].

The processing and communicating scenario is as follows [16]:

- Initially, consider only a single **Segment** object, **segment(i)**. At first, it exchanges its local temperature value with its neighbors **segment(i-1)** and **segment(i+1)** though the adequate communication **Channel** components. After this, **segment(i)** counts with the different temperatures from its neighbors.
- The temperature operation is simultaneously started by the **segment(i)** component and all the other components of the wire.
- In order to continue, all components iterate as many times as required, exchanging their partial temperature values through the available communication channels.
- The process repeats until each component has finished iterating, and thus, finishing the whole computation.

3. *Functional description of components.* The processing and communicating software components for solving the One-dimensional Heat

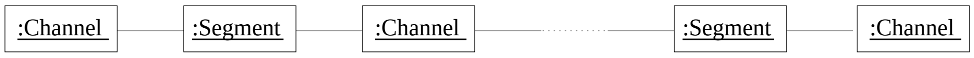

Figure 1: Object diagram of CSE for solving the One-dimensional Heat Equation.

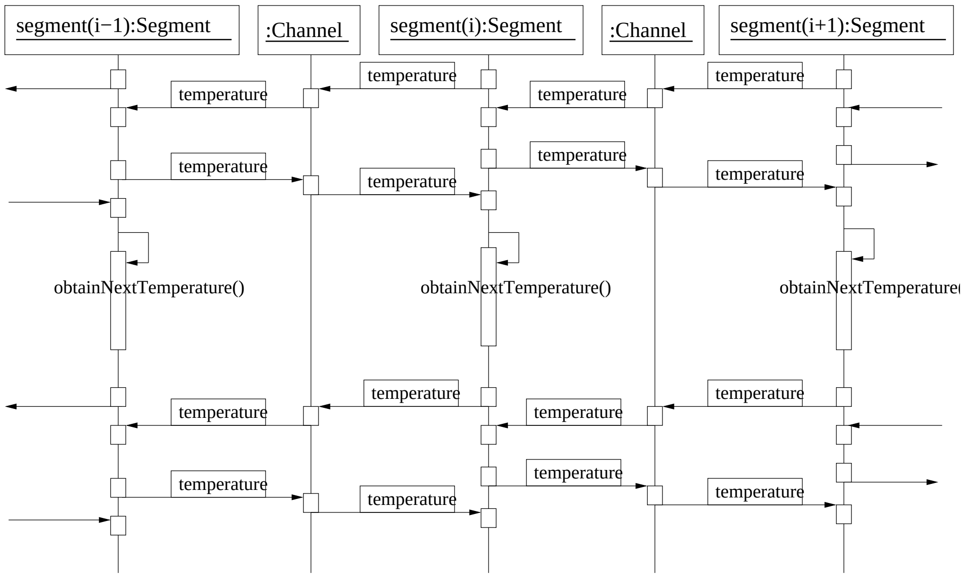

Figure 2: Sequence diagram of CSE for communicating temperatures through channel components for the One-dimensional Heat Equation.

Equation using the CSE pattern are described as follows [16].

– **Segment**. The responsibilities of a segment, as a processing component, are to obtain the next temperature from the temperature values it receives, and make available its own temperature value so its neighboring components are able to proceed.
– **Channel**. The responsibilities of every channel, as a communication component, are to allow sending and receiving temperature values, synchronizing the communication activity between neighboring sequential elements. Channel components are developed as the main design objective of this paper, in section "Communication Design" below.

2.2 Information about parallel platform and programming language

The parallel system available for this example is a SUN SPARC Enterprise T5120 Server. This is a multi-core, shared memory parallel hardware platform, with 18-CoreUltraSPARC T2, 1.2 GHz processors (capable of running 64 threads), 32 Gbytes RAM, and Solaris 10 as operating system. Applications for this parallel platform can be programmed using the Java programming language [13].

3 Communication Design – Specification of Communication Components

3.1 The scope

This section takes into consideration the basic information about the parallel hardware platform and the programming language used, as well as the CSE pattern as the selected coordination for solving the One-dimensional Heat Equation. 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 shared memory multi-core computer), the programming language (Java) and the description of channels as communication software components for the CSE pattern presented in the previous section, the procedure for applying a Design Pattern for the Communication Components for the One-dimensional Heat Equation problem is presented as follows [15, 17]:

1. *Consider the architectural pattern selected in the previous step.* From the CSE pattern description, the design patterns which provide communication components and allow the behavior as described by this architectural pattern for a coordination are the Shared Variable Channel pattern and the Message Passing Channel pattern [15, 17].

2. *Select the nature of the communicating components.* Considering that the parallel hardware platform to be used has a shared memory organization, the nature of the communicating components for such memory organization is considered to be shared variable.

3. *Select the type of synchronization required for the communication.* Normally, the communication between software components that compose an array communicated through point to point communication components makes use of an asynchronous communication. If a synchronous communications would be used, it is very likely that the processing software components would block, waiting for receiving temperature values from their counterpart. As every software component would be waiting, none would be receiving, leading to a deadlock situation. In this case, using a relaxation method such as Gauss-Seidel relaxation or the successive over-relaxation (SOR) at the coordination level would sort this out. Nevertheless, in the problem analysis it is stated that a Jacobi relaxation is to be used [16]. So, this makes it more important to make use of an asynchronous communication scheme which avoids sender waiting for their receivers.

4. *Selection of a design pattern for communication components.* Considering (a) the use of the CSE pattern, (b) the shared memory organization of the parallel platform, and (c) the use of asynchronous communications, therefore the **Shared Variable Channel pattern** is proposed here as the base for designing the communications between Sequential Elements. Let us consider the Context and Problem sections of this pattern [15, 17]:

    - **Context:** 'A parallel program is being developed using the Communicating Sequential Elements architectural pattern as a domain parallelism approach in which the data is partitioned among autonomous processes (elements) as the processing components of the parallel program. The parallel program is developed for a shared memory computer. The programming language to be used counts with synchronization mechanisms for process communication such as semaphores, critical regions, or monitors'
    - **Problem:** 'A sequential element needs to exchange values with its neighboring elements. Every data is locked inside each sequential element, which is responsible for processing that data and only that data'

    From both these descriptions, it is noticeable that for the CSE pattern, on a shared memory parallel platform, and using Java as the programming language, the choice for developing the communication components for this example is the **Shared Variable Channel pattern** [15, 17]. The use of a shared memory parallel platform implies using shared variables, and it is known that the Java programming language counts with the elements for developing semaphores or monitors. Moreover, the channels consider an asynchronous communication scheme between sender and receiver. Therefore, this completes the selection of the Design Pattern for Communication Components of the One-dimensional Heat Equation. The design of the parallel software system continues using the Shared Variable Channel pattern's Solution section as a starting point for communication design and implementation.

3.2 Structure and dynamics

This section takes information of the Shared Variable Channel design pattern, expressing the interaction between its software components that carry out the communication between parallel software components for the actual example [15, 17].

1. *Structure.* The structure of this pattern applied for designing and implementing channel communication components of the CSE pattern is shown in Figure 3 using a UML collaboration diagram [6]. Notice that the channel component structure allows an asynchronous, bidirectional communication between two sequential elements. The asynchronous feature is achieved by allowing an array of temperatures to be stored, so the sender does not wait for the receiver [15, 17].

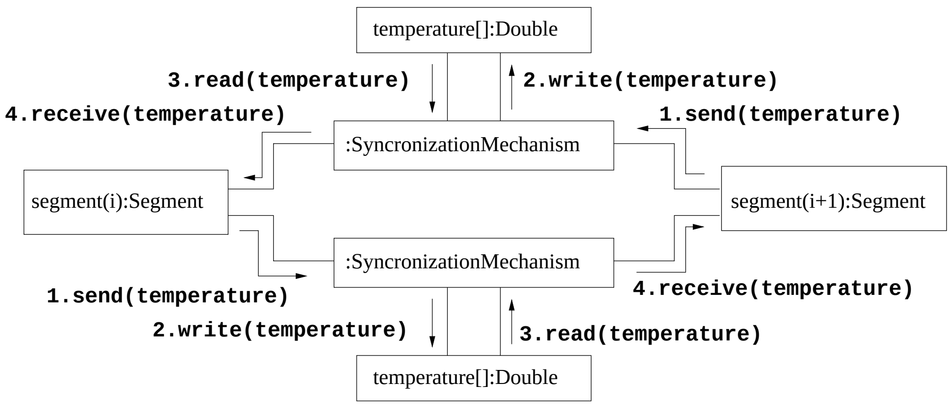

Figure 3: UML collaboration diagram of the Shared Variable Channel pattern used for asynchronously exchange temperature values between sequential components of the CSE solution to the One-dimensional Heat Equation.

2. *Dynamics.* This pattern actually emulates the operation of a channel component within the available shared memory, multi-core parallel platform. Figure 4 shows the behavior of the participants of this pattern for the actual example [15, 17].

   In this scenario, a point to point, bi-directional, asynchronous communication exchange of temperature values of type `Double` is carried out, as follows:

   - The `segment(i)` sequential element sends its local `temperature` value by issuing a `send(temperature)` operation to the sending synchronization mechanism.
   - This synchronization mechanism verifies if the `segment(i+1)` sequential element is not reading the `temperature` shared variable. If this is the case, then it translates the sending operation, allowing a `write(temperature)` operation of the data item on `temperature`. Otherwise, it blocks the operation until the `temperature` can be safely written.

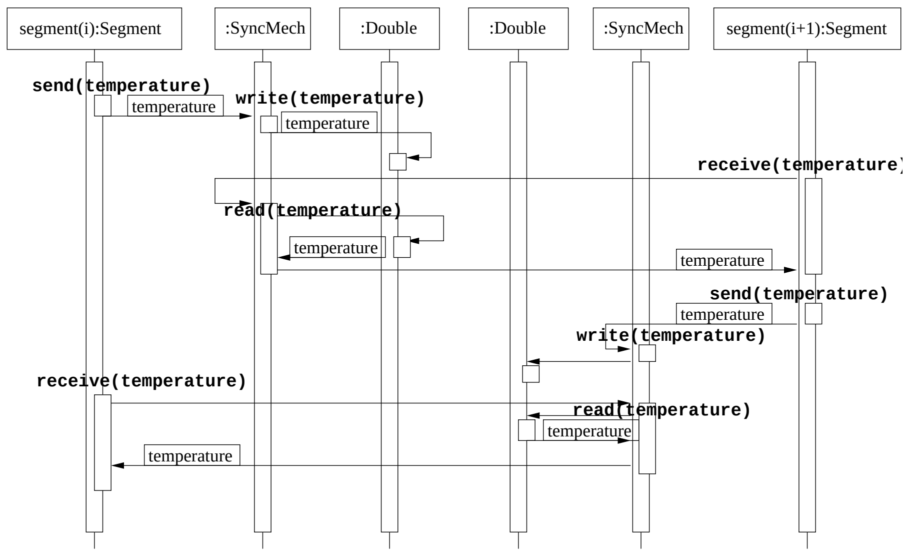

Figure 4: UML sequence diagram for the Shared Variable Channel pattern applied for exchanging temperature values between two neighboring sequential elements of the CSE solution for the One-dimensional Heat Equation.

- When the `segment(i+1)` attempts to receive the temperature value, it does so by issuing a `receive(temperature)` request to the synchronization mechanism. This function returns a `double` type representing the temperature value stored in the shared variable `temperature`. Again, only if its counterpart sequential element (here, `segment(i)`) is not writing on `temperature`, the synchronization mechanism grants a `read(temperature)` operation from it, returning the requested `temperature` value. This achieves the send and receive operations between neighboring segment elements.
- On the other hand, when data flows in the opposite direction, a similar procedure is carried out: the local `temperature` value of `segment(i+1)` is sent by issuing a `send(temperature)` operation to the synchronization mechanism.
- The synchronization mechanism verifies if the counterpart `segment(i)` is not accessing `temperature`. If this is the case, then it translates the sending operation, allowing a `write(temperature)` operation of the temperature value on it. Otherwise, it blocks the operation until the shared variable `temperature` can be modified.
- The `segment(i)` sequential element reads the temperature value by issuing a `receive(temperature)` request to the synchronization mechanism. Again, only if `segment(i+1)` is not writing on `temperature`, the synchronization mechanism grants a `read(temperature)` operation from it, returning the requested temperature value.

3.3 Functional description of software components

This section describes each software component of the Shared Variable Channel pattern as the participant of the communication sub-system, establishing its responsibilities, input, and output [15, 17].

1. **Synchronization mechanisms.** This kind of components is used to synchronize the access to the `Double` shared variables. Notice that they should allow the translation of `send()` and `receive()` operations into adequate operations for writing to and reading from the shared variables. Normally, synchronization mechanisms are used to keep the order and integrity of the shared data.

2. **Shared variables.** The responsibility of the shared variables is to store the `Double` type that holds the temperature values exchanged by sequential elements. These shared variables are designer here as simple variables that buffer during communication, for actually achieving an asynchronous communication.

3.4 Description of the communication

The channel communication component, thus, acts as a single entity, allowing the exchange of information between processing software components. Given that the available parallel platform is a multi-core, shared memory system, the behavior of a channel component is modelled using shared variables. Thus, a couple of shared variables are used to implement the channel component as a bidirectional, shared memory communication means between elements. It is clear that such shared variables require to be safely modified by synchronizing read and write operations from the elements. Hence, the Java programming language provides the basic elements for developing synchronization mechanisms (such as semaphores or monitors). This is required to preserve the order and integrity of the transferred temperature values.

3.5 Communication Analysis

This section describes the advantages and disadvantages of the Shared Variable Channel pattern as a base for the communication structure proposed.

1. **Advantages**

    - A communication sub-structure based on the Shared Variable Channel pattern allows to keep the precise order of the exchanged temperature values by considering a two directional FIFOs for its implementation, as well as synchronizing the access to both `Double` type shared variables.
    - The communication sub-structure based on the Shared Variable Channel pattern allows for a point to point, bidirectional communication component.
    - The use of synchronization mechanisms grants keeping the integrity of transferred temperature values, assuring that at any given moment only one element actual accesses any `Double` type shared variables.

- The use of shared variables implies that the implementation is particularly developed for a shared memory parallel platform.
- The Shared Variable Channel pattern allows the use of asynchronous communications between sequential elements by using the two `Double` type shared variables as two communication buffers.

2. **Liabilities**

- As the available parallel platform is a shared memory one, the communication speed tends to be similar to simple assignation operations over shared variable addresses. Communications are only delayed by the synchronization actions taken by the synchronization mechanisms in order to keep the integrity of the temperature values. Nevertheless, it is very little what can be done to improve communication performance in terms of programming. The only programming action that can be taken is to change the amount of processing of the sequential elements, which modify the granularity, tuning the communication speed.
- The implementation based on shared variables and synchronization mechanisms such as semaphores, conditional regions, or monitors, makes these communication sub-systems only suitable for shared memory platforms. If the parallel software system is considered to be ported to a distributed memory parallel platform, this would require replacing each Shared Variable Channel pattern by a Message Passing Channel pattern [15], and design and implement the communication sub- systems as indicated by this pattern.

4 Implementation

In this section, all the software components described in the communication design section are considered for their implementation using the Java programming language. Here, the implementation of the communication sub-system is developed. It interconnects processing components that implement the actual computation that is to be executed in parallel [16]. So, the implementation is presented here for developing the channel as communication and synchronization components. Nevertheless, describing the entire design and implementation of the whole parallel software system goes beyond the actual purposes of the present paper.

4.1 Synchronization Mechanism

Based on the Java programming language, a basic description of a synchronization mechanism that controls the access to the temperature array is presented as follows:

```
import java.util.Vector;
class SynchronizationMechanism {
    private int numMessages = 0;
    private final Vector temperatures = new Vector();
    public final synchronized void write(double temp){
       ...
       numMessages++;
       temperatures.addElement(temp);
       ...
    }
    public final synchronized double read(){
       double temp = 0.0d;
       numMessages--;
       ...
       temp = temperatures.firstElement();
       temperatures.removeElementAt(0);
       return temp;
    }
}
```

The class `SynchronizationMechanism` presents two synchronized methods, `write()` and `read()`, which allow to safely modify the temperatures buffer and allows an asynchronous communication between `segment` components. This class is used in the following implementation stage as the basic element of the channel components.

4.2 Communication components – Channels

Using the class `SynchronizationMechanism` from the previous section, here it is used as the synchronization mechanism component as described by the Shared Variable Channel pattern, in order to implement the class `Channel`, as follows:

```
public final class Channel {
    private SynchronizationMechanism m0 = null;
    private SynchronizationMechanism m1 = null;
    public Channel(){
       m0 = new SynchronizationMechanism();
       m1 = new SynchronizationMechanism();
    }
    public void send0(Channel c, double temp){
       if(temp == null) throw new NullPointerException();
       m0.write(temp);
    }
    public void send1(Channel c, double temp){
       if(temp == null) throw new NullPointerException();
       m1.write(temp);
    }
    public double receive0(Channel c){
       return m0.read();
    }
    public double receive1(Channel c){
       return m1.read();
    }
}
```

Each channel component is composed of two synchronization mechanisms which allow the bi-directional flow of data through the channel. In order to keep the direction of each message flow, it is necessary to define two methods for sending and other two methods for receiving, and keep attention about the flow of messages. Each method distinguishes on which synchronization mechanism of the channel the message is written. This allows that the channel is capable of allowing a simultaneous bi-directional flow. In the present example, this is used to enforce the use of the Jacobi relaxation [16]. In fact, using (a) a channel communication structure with two-way flow of data, (b) making each one of them asynchronous, and later, (c) taking care on the communication exchanges between segment components, are all design previsions for avoiding any potential deadlock. In parallel programming, it is generally advised that during design, all previsions should be taken against the possibility of a deadlock.

Moreover, in case of modifying the present implementation for executing on a distributed memory parallel system, it would be necessary only to substitute the implementation of the class `Channel`, but using the Message Passing Channel pattern [15] as a base for its definition.

5 Summary

The design patterns for communication components are applied here along with a method for selecting them, in order to show how to select a design pattern that copes with the requirements of communication present in the One-dimensional Heat Equation 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 calculation of differential equations for a one-dimensional problem, executing on a shared memory parallel platform.

6 Acknowledgements

The author wishes to thank Stefan Sobernig, my shepherd for EuroPLoP 2010, for his encouraging comments about the present paper.

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 Communicating Sequential Elements Pattern. An Architectural Pattern for Domain Parallelism*, Proceedings of the 7th Conference on Pattern Languages of Programming (PLoP2000), Allerton Park, Illinois, USA, 2000.

[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 *Applying Architectural Patterns for Parallel Programming. Solving the One-dimensional Heat Equation*, Proceedings of the 14th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2009), Kloster Irsee, Germany, 2009.

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

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