Design Patterns for Communication Components of Parallel Programs

Jorge L. Ortega Arjona
Departmento de Matemáticas
Facultad de Ciencias, UNAM
jloa@fciencias.unam.mx

Abstract
This paper introduces an approach to describing and selecting design patterns for developing communication components of parallel programs. The approach uses the kind of communication requirements of the architectural pattern whose processing components have to be communicated, along with characteristics and features of the parallel hardware platform and programming language synchronisation mechanisms to be used, in order to make selections between different design patterns.

1. Introduction

Parallel processing is the division of a problem, presented as a data structure and/or a set of actions, among multiple processing components that operate simultaneously. The expected result is a more efficient completion of the solution to the problem. Its main advantage is the ability to handle tasks of a scale that would be unrealistic or not cost-effective for other systems [CG90, Fos94, Pan96]. Thus, a parallel program is defined as the specification of a set of software components that simultaneously process and communicate among themselves, in order to achieve a common objective. Hence, a parallel program can be normally described in terms of two types of software components [CG90]:

- *Processing components*. Processing components make up the parallel software system, and their design and implementation focus on actually perform the simultaneous operations on data.
- *Communication components*. Communication components represent the actual cooperation — through exchange of data or the request for operations— between processing components. Communication components are the linking software that allow the information exchange between the processing components of the parallel software system.

The present paper attempts to describe communication components as design patterns, in order to aid in their selection during the design and implementation of a parallel software system.

2. Design Patterns

Design patterns are defined as follows:

"The design patterns ... are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [GHJV95].

"A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context" [POSA96].

The design patterns in this paper focus on describing and refining the communication components of a parallel program, by describing common structures used for communicating, by exchanging data or requesting operations, between processing components.

The design patterns here share a formal structure (using the POSA form, as described in [POSA96]), containing a name, a summary, a context (presenting the design situation in which apply the pattern), a problem statement (including a description of its forces), a solution statement (covering descriptions of

its structure, participants, basic dynamics and implementation), consequences (describing benefits and liabilities), known uses and related patterns. These elements provide a uniform template for browsing pattern descriptions contained in several pattern systems, making it easy to look for and find information about when and how to use each pattern.

3. Classification of Design Patterns for Communication Components

The design patterns for communication components of parallel programs can be classified taking into consideration several characteristics of the communication they perform. Hence, design patterns for parallel programming are defined and classified according to:

- *The parallelism of the overall parallel software system*. The communication components have to be designed to allow communications in parallel systems with *(a)* functional parallelism [OR98, Ort05, Ort07], *(b)* domain parallelism [OR98, Ort00], or *(c)* activity parallelism [OR98, Ort03, Ort04]. These types of parallelism arise from partitioning the algorithm and/or data among the processing components of a parallel program. Hence, functional parallelism focuses on decomposing the algorithm, domain parallelism on dividing data, and activity parallelism on partitioning both, algorithm and data [OR98]. The type of parallelism used in the overall parallel software system is an important contextual indicator of the type of communication component to be designed.
- *The memory organisation of the parallel hardware platform.* The communication components are designed and implemented through programming mechanisms that cope with a parallel hardware platform with *(a)* shared memory, or *(b)* distributed memory [Har98]. In a shared memory parallel system, all memory can be accessed by every processor; in a distributed memory parallel system, each processor counts with a local memory, and it is able to access another processor's memory only through I/O requests. The type of memory organisation is an indicator of the kind of programming mechanisms to be used when designing and implementing communication components.
- *The type of synchronisation.* Depending on the memory organisation, communication components are implemented through programming mechanisms that involve *(a)* synchronous communications, or *(b)* asynchronous communications. Synchronous communications imply blocking the sender or the receiver until its counterpart in the communication is available; asynchronous communications imply that neither the sender nor the receiver waits for its communication counterpart — it continues without blocking [Har98].
- *The type of programming mechanisms.* The practice of parallel programming has provided several programming mechanisms used to design and implement communications between parallel processing components. Common programming mechanisms are *(a)* semaphores [Dij68, And91, Har98, And00], *(b)* monitors, [Hoa74, And91, Har98, And00], *(c)* message passing [Hoa78, And91, Har98, And00], and *(d)* remote procedure call [Bri78, And91, Har98, And00].

Based on this classification criteria, this paper presents eight design patterns commonly used for designing and implementing the communication components of parallel software systems. Table 1 presents these design patterns, classified only regarding to the parallelism of the overall parallel software system, the memory organisation of the parallel hardware platform, the type of synchronisation, and the type of programming mechanisms used for their implementation.

	Parallelism	Memory Organisation	Synchronisation	Programming Mechanism
Shared Variable Pipe	Functional	Shared Memory	Asynchronous	Semaphores or Monitors
Multiple Local Call	Functional	Shared Memory	Synchronous	Semaphores or Monitors
Message Passing Pipe	Functional	Distributed Memory	Asynchronous	Message Passing
Multiple Remote Call	Functional	Distributed Memory	Synchronous	Remote Procedure Call
Shared Variable Channel	Domain	Shared Memory	Asynchronous	Semaphores or Monitors
Message Passing Channel	Domain	Distributed Memory	Asynchronous	Message Passing
Local Rendezvous	Activity	Shared Memory	Synchronous	Semaphores or Monitors
Remote Rendezvous	Activity	Distributed Memory	Synchronous	Remote Procedure Call

Table 1: Design patterns classification**.**

4. Design Patterns for Communication Components of Parallel Programs

Parallel programming is characterised by a growing set of parallel architectures, paradigms and programming languages. This situation makes difficult to propose just one approach containing all the details to design and implement communication components for all parallel software systems. The design patterns proposed here are 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 Shared Variable Pipe pattern

The *Shared Variable Pipe* pattern describes the design of a pipe component based on shared variables and synchronisation mechanisms, which serve for implementing send and receive operations that emulate the behaviour of a pipe component for a shared memory parallel system.

Context

A parallel program is being developed using the Parallel Pipes and Filters architectural pattern [OR98, Ort05] as a functional parallelism approach in which an algorithm is partitioned among autonomous filters 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 synchronisation mechanisms for process communication, such as semaphores [Dij68, Har98] or monitors [Hoa74, Har98].

Problem

A collection of parallel filters require to communicate by exchanging messages, following a single direction data flow; every data and operation over it is carried out inside some filter.

Forces

The following forces should be considered for the *Shared Variable Pipe* pattern:

- Maintain the precise order of the transferred data through the pipe, using a FIFO policy.
- Communication should be point to point and unidirectional.
- Keep the integrity of transferred data.
- The implementation has to consider the shared memory as programming environment.
- The communication should be asynchronous.

Solution

The idea is to emulate the behaviour of a pipe component using a shared variable. Hence, use the shared variable to implement the pipe component, considering it as a one-directional, shared memory communication means between filters. Such a shared variable requires to be safely modified by read and write operations from the filters. Hence, a programming language synchronisation mechanism (such as semaphores or monitors) has to be considered to preserve the order and integrity of the transferred data, along with sending (writing) and receiving (reading) operations.

Structure

The participants and relations that compose the structure of this pattern are shown using a UML Collaboration Diagram [Fow97] for the description (Figure 1).

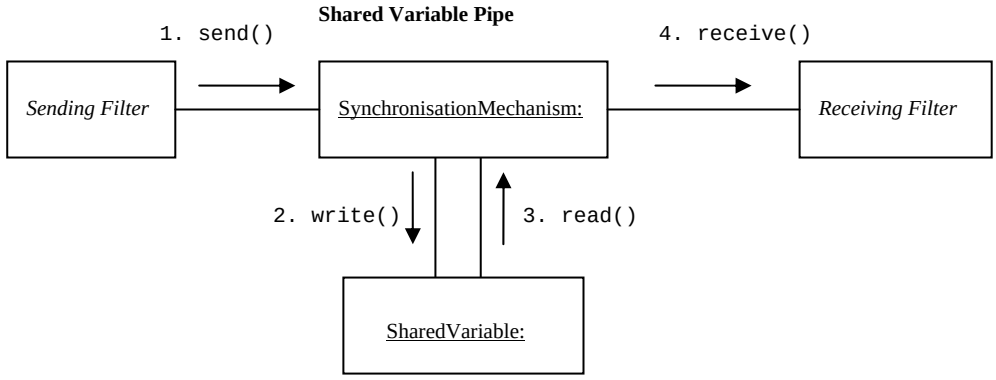

Figure 1. Collaboration Diagram for the Shared Variable Pipe pattern.

Participants

- **Synchronisation Mechanism**. The responsibility of the synchronisation mechanism is to synchronise the access to the shared variable, translating the send and receive operations into adequate operations for writing to and reading from the shared variable. The synchronisation mechanism is, then, in charge of keeping the order and integrity of the shared data.
- **Shared Variable**. The responsibility of the shared variable is to serve as a repository for the data to be transferred. It can be designed as a buffer (an array of a particular type) with an specific size, for accomplishing with the use of asynchronous communication between the sending filter and the receiving filter.

Dynamics

The behaviour of this pattern is expected to emulate the operation of a pipe component within a shared memory parallel system. Hence, Figure 2 shows the behaviour of the participants of this pattern, aiming to carry out such an emulation.

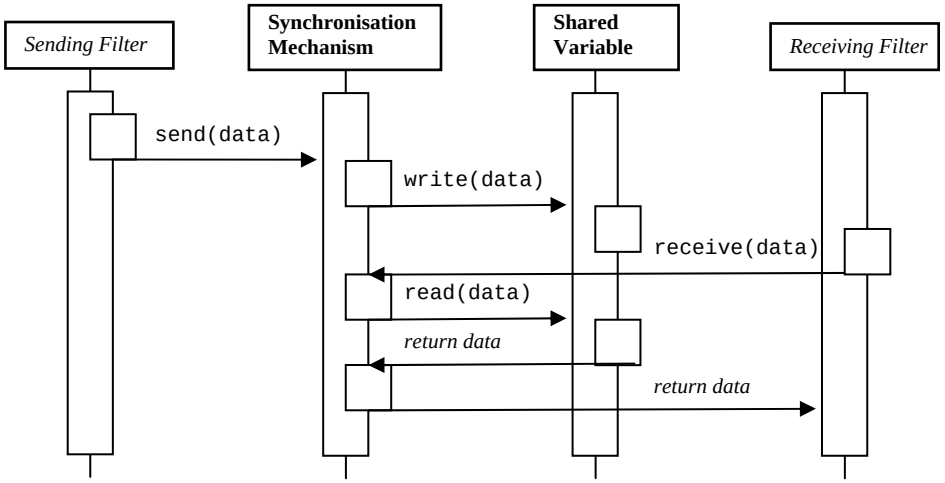

Figure 2. Scenario for the Shared Variable Pipe pattern.

In the scenario shown in Figure 2, the following steps are followed for communicating data from *Sending Filter* to *Receiving Filter*:

- The *Sending Filter* sends the data item to be piped to the Shared Variable Pipe, issuing a `send(data)`operation to the **Synchronisation Mechanism**.
- The **Synchronisation Mechanism** verifies if the *Receiving Filter* is not accessing the **Shared Variable**. If this is the case, then it translates the sending operation, allowing a `write(data)`operation of the data item on the **Shared Variable**. Otherwise, it blocks the operation until the **Shared Variable** can be modified.
- On the other hand, when the *Receiving Filter* attempts to receive data from the Shared Variable Pipe, it does it so by issuing a `receive(data)` request to the **Synchronisation Mechanism**. Again, only if the *Sending Filter* is not modifying the **Shared Variable**, the **Synchronisation Mechanism** grants a `read()` operation from the **Shared Variable**, returning the requested data item.
- The communication flow is kept unidirectional by allowing only send operations to the *Sending Filter*, and receive operations to the *Receiving Filter*.

Implementation

The implementation of the Shared Variable Pipe requires the construction of the Synchronisation Mechanism and the Shared Variable. Both software components exist and execute on a shared memory environment.

The Synchronisation Mechanism can be implemented using semaphores [Dij68, Har98] to synchronise the access to the Shared Variable, considering the P and V operations, respectively just before and after invoking the `write()`or `read()`operations that modify the state of the Shared Variable. Another possibility is the use of monitors [Hoa74, Har98], which consider the synchronisation over the very `write()`or `read()` operations.

The Shared Variable can be implemented as a bounded buffer of a particular type, which can be modified by reading or writing operations from the Synchronisation Mechanism, considering an asynchronous communication approach. The Shared Variable, hence, is capable of keeping several data values in order to cope both the *Sending Filter* and the *Receiving Filter*, if both perform processing activities at different speeds.

Consequences

Benefits

- The Shared Variable Pipe keeps a FIFO policy, by synchronising the access to the Shared Variable. If the *Sending Filter* is faster than the *Receiving Filter*, then the Synchronisation Component would block the *Sending Filter* if the Shared Variable, as a buffer, is full. Otherwise, if the *Receiving Filter* is faster than the *Sending Filter*, the Synchronisation Component would block the *Receiving Filter* if the Shared Variable is empty.
- The Shared Variable Pipe is designed to deal with point to point and unidirectional communication. However, it can be extended to a one-to-many, many-to-one, and many-to-many communications, by using several Synchronisation Mechanisms over several Shared Variables. Also, it keeps a unidirectional flow of data by allowing only sending operations to the *Sending Filter*, and receiving operations to the *Receiving Filter*.
- The Synchronisation Mechanism is in charge of keeping the integrity of transferred data, by assuring that, at any given moment, only one filter has actual access to Shared Variable.
- The implementation is particularly developed for a shared memory programming environment.
- The Shared Variable Pipe uses asynchronous communications, by implementing the Shared Variable as a bounded buffer.

Liabilities

- The communication speed of the Shared Variable Pipe is as slow as the slowest filter it connects. Therefore, to improve communication performance, changes to the amount of processing of the filters have to be considered.
- The Shared Variable Pipe can be used for one-to-many, many-to-one, and many-to-many communications, although the implementation could require the use of several semaphores or monitors. This fact could make it difficult to implement the whole communication component.
- If the *Sending Filter* or the *Receiving Filter* is a lot faster that its communication counterpart, this could produce a great unbalance on the whole computation. This is a signal that the division of the algorithm into steps could be wrong. If it is the case, perhaps removing the pipe and considering both processing components into one could solve the unbalance situation.
- The implementation based on semaphores and monitors makes this pattern only to be used into a shared memory environment. Porting it to a distributed memory parallel platform would require to replace each Shared Variable Pipe by a Message Passing Pipe.
- There could be potential problems if the sending operations are not restricted to the *Sending Pipe*, and /or the receiving operations to the *Receiving Filter*. The structure would not act as a pipe.

Known uses

The Shared Variable Pipe is normally used when the parallel solution of a problem is developed using the Parallel Pipes and Filters architectural pattern [OR98, Ort05] within a shared memory parallel platform. Hence, it has as many known uses as the Parallel Pies and Filters pattern. Particularly, the following known uses are relevant:

- The Shared Variable Pipe pattern is used when implementing the Pipes and Filters version of the Sieve of Eratosthenes for a shared memory computer, in order to allow the flow of integers between the filters in which the test, whether an integer is a prime number or not, is carried out [Har98].
- The Shared Variable Pipe pattern is commonly used when describing a solution based on semaphores or monitors as a bounded buffer communication, in which a producer produces data items and a consumers consumes them [Dij68, Hoa74, And91, Har98, And00].
- The Shared Variable Pipe pattern can be considered a variation of the pipe operation common in several Unix and Unix-based operating systems for communicating processes [And91, And00].

Related patterns

The *Shared Variable Pipe* pattern is directly related with any parallel software system developed on a shared memory environment from the Parallel Pipes and Filters pattern [OR98, Ort05]. It is also related with the pattern for selecting locking primitives, originally proposed by McKenney [McK95], and lately included as part of the POSA 2 book, Patterns for Concurrent and Networked Objects [POSA00].

The Message Passing Pipe pattern

The *Message Passing Pipe* pattern describes the design of a pipe component based on message passing, by implementing send and receive operations that perform the communications of the pipe component for a distributed memory parallel system (although it can be used for a shared memory parallel system as well).

Context

A parallel program is being developed using the Parallel Pipes and Filters architectural pattern [OR98, Ort05] as a functional parallelism approach in which an algorithm is partitioned among autonomous processes (filters) as the processing components of the parallel program. The parallel program is developed for a distributed memory computer, even though it also can be used for a shared memory computer. The programming language to be used counts with synchronisation mechanisms for process communication through message passing [Hoa78, Har98].

Problem

A collection of parallel filters require to communicate by exchanging messages, following a single direction data flow; every data is operated inside some filter.

Forces

The following forces should be considered for the *Message Passing Pipe* pattern:

- Maintain the order of the transferred data through the pipe, using a FIFO policy.
- Communication should be point to point and unidirectional.
- The implementation has to consider a distributed memory as programming environment.
- The data transference should be performed asynchronously.

Solution

Design a pipe component as a distributed software structure connecting the filters that execute on two different processors or computers. The software structure is composed of communication end points (often, sockets), some synchronisation mechanisms, and a couple of data streams. These components are put together in order to achieve a one-directional, distributed memory communication component between filters executing on different processors or computers.

Structure

Figure 3 shows the participants and relations that compose the structure of this pattern, using a UML Collaboration Diagram [Fow97].

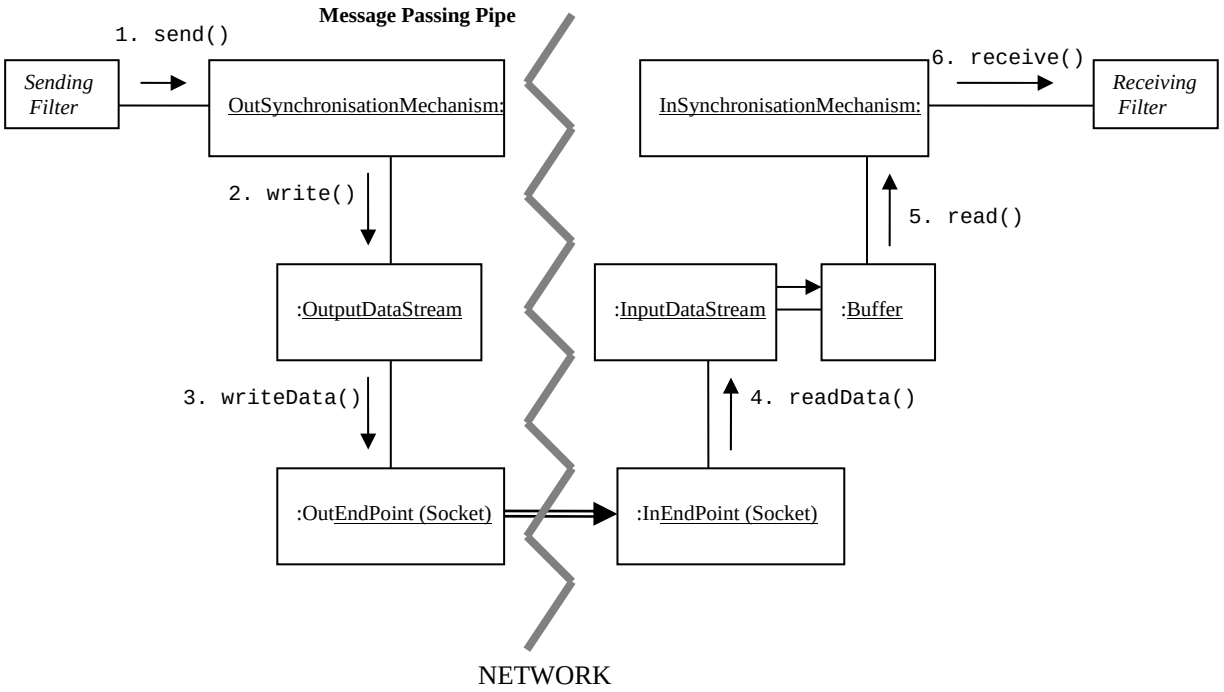

Figure 3. The Message Passing Pipe pattern.

Participants

- **OutSynchronisationMechanism** and **InSynchronisationMechanism**. The responsibility of the synchronisation mechanisms is to synchronise the access to the data streams (**OutDataStream** and **InDataStream**), so only one processing component has access to any of them at a given moment. The synchronisation mechanism is, then, in charge of keeping the order and integrity of the data written to or read from the data streams.
- **OutDataStream** and **InDataStream**. The responsibility of the data streams is to transiently store the serialised data to be passed through the pipe. Every data item must be serialized, that is, converted into a stream of bytes, which is the way in which data is transferred through a network connecting processors or computers. Data streams can be written to and read from communication end points (**OutEndPoint** and **InEndPoint**).
- **OutEndPoint** and **InEndPoint**. The responsibility of the communication end points is to send data back and forth between the processors or computers. These sort of communication end points are commonly implemented as sockets.
- **Buffer**. The responsibility of the buffer is to serve as a repository for the data to be received. It is normally designed as an array of a particular type with an specific size. The buffer allows the use of asynchronous communication between the sending filter and the receiving filter.

Dynamics

This pattern is expected to operate as a pipe component for a distributed memory parallel system. Hence, Figure 4 shows the behaviour of the participants of this pattern, aiming to carry out such an operation.

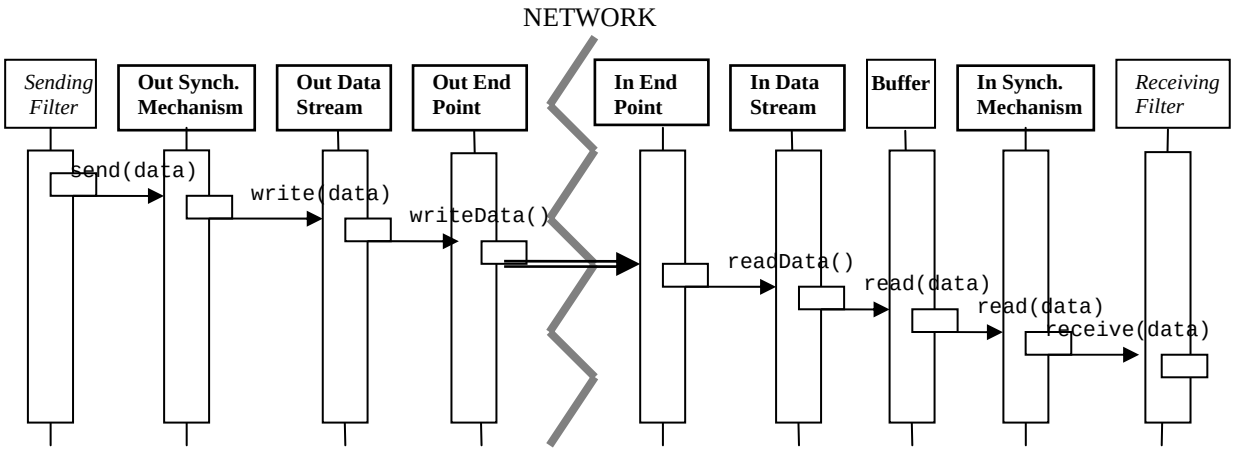

Figure 4. Scenario of the Message Passing Pipe pattern.

In this scenario, the following steps are followed:

- The *Sending Filter* sends the data item to be piped to the Message Passing Pipe, issuing a `send(data)`operation to the **Out Synchronisation Mechanism**.
- The **Out Synchronisation Mechanism** verifies if no other process is accessing the **Out Data Stream**. If this is the case, then it translates the sending operation, allowing a `write(data)`operation of the data item on the **Out Data Stream**. Otherwise, it blocks the operation until the **Out Data Stream** can be written.
- The **Out Data Stream** generates a `writeData()` operation over the **Out End Point** (normally, a socket), so the data item is sent through the network to the appropriate processor or computer.
- The data item is received by the **In End Point** (a socket) which is read by the **In Data Stream** by issuing a `readData()` operation. The data item is allocated into the **Buffer**, so the communication is kept asynchronous. From the **Buffer**, the *Receiving Filter* is able to receive data by issuing a `receive(data)` request to the **In Synchronisation Mechanism**. Again, only if no other process is reading from the **Buffer**, the **Synchronisation Mechanism** grants a `read()` operation from the **Buffer**, allowing to the *Receiving Filter* to read the requested data item.

Implementation

The implementation of the Message Passing Pipe requires the use or construction of the Synchronisation Mechanisms, the Data Streams, the Buffer, and the End Points. All these software components exist and execute on a distributed memory environment, between two communicating processor or computers.

The Synchronisation Mechanism can be implemented using semaphores [Dij68, Har98] to synchronise the access to the Data Streams and the Buffer, considering the P and V operations, respectively just before and after invoking the `write()`or `read()`operations that modify the state of the Data Streams and the Buffer. Another possibility is the use of monitors [Hoa74, Har98], which consider the synchronisation over the very `write()`or `read()` operations.

Data Streams are a common communication form in many programming languages, used to serialise data, that is, converted into a stream of bytes, which is the way in which data is transferred through a network connecting processors or computers.

The End Points are commonly sockets, which is a network communication mechanism common in several programming languages. Sockets are able to send data back and forth between the processors or computers of a network system.

The Buffer can be implemented as an array of a particular type, which can be modified by reading or writing operations from the Synchronisation Mechanism, considering an asynchronous communication approach. The Buffer should be able of keeping several data values in order to cope both the *Sending Filter* and the *Receiving Filter*, since both perform processing activities at different speeds.

Consequences

Benefits

- The Message Passing Pipe keeps a FIFO policy, by synchronising the access to the Buffer on the receiving side.
- The Message Passing Pipe is designed to deal with point to point and unidirectional communication. However, it can be extended to a one-to-many, many-to-one, and many-to-many communications, by using several Synchronisation Mechanisms over several Data Stream, Buffer, and End Point. Also, it keeps a unidirectional flow of data through the Data Streams.
- The implementation based on Data Streams and End Points is explicitly developed for a distributed memory programming environment. However, it can be used within a shared memory programming environment as well.
- The Message Passing Pipe uses asynchronous communications, by implementing a bounded buffer on the receiving side.

Liabilities

- The communication speed of the Message Passing Pipe depends not only on the slowest filter it connects, but also on features and characteristics of the communication network on which it executes. Therefore, communication performance is commonly affected by non-determinism issues which can be determinant on the communication speed.
- The Message Passing Pipe can be used for one-to-many, many-to-one, and many-to-many communications, although the implementation could require the use of several semaphores or monitors. This fact could make it difficult to implement the whole distributed communication components.
- If the *Sending Filter* or the *Receiving Filter* is a lot faster that its communication counterpart, this could produce a great unbalance on the whole computation. This is a signal that the division of the algorithm into steps could be wrong. If it is the case, perhaps removing the pipe and considering both processing components into one could solve the unbalance situation.
- The implementation based on data streams and end points makes this pattern to be suitable for a distributed memory environment. Nevertheless, it can also be used within a shared memory environment, in which, however, it may not have a good performance due to the many communications involved. Hence, for a shared memory parallel platform and to keep a better performance, it would be advisable to replace each Message Passing Pipe by a Shared Variable Pipe.

Known uses

The Message Passing Pipe is commonly used when the parallel solution of a problem is developed using the Parallel Pipes and Filters architectural pattern [OR98, Ort05] within a distributed memory parallel platform. Hence, it has as many known uses as the Parallel Pies and Filters pattern. Particularly, the following known uses are relevant:

- The Message Passing Pipe pattern has been used when implementing the Pipes and Filters version of the graphics rendering for a distributed memory system, in order to allow the flow data between the filters in which the rendering of a scene is carried out [Ort05].
- The Message Passing Pipe pattern is used when describing a distributed memory solution for the Matrix Multiplication problem [Har98].

- The Message Passing Pipe pattern can be used for a point-to-point, unidirectional communication between processes executing on different computers of a network system [And91, Har98, And00].

Related patterns

The *Message Passing Pipe* pattern is related with any parallel software system developed for a point-to-point, unidirectional communication on a distributed memory environment from the Parallel Pipes and Filters pattern [OR98, Ort05]. It is also related with the pattern for networking included as part of the POSA 2 book, Patterns for Concurrent and Networked Objects [POSA00].

Multiple Local Call

The *Multiple Local Call* pattern describes the design of multiple programming modules that encapsulate services or access procedures, which are called or invoked by another component, in order to operate over global and/or local variables. Such an operation is related with delegating a part of a whole processing activity to such a programming module. Both components are allowed to execute simultaneously, and thus, they require a synchronous communication during each call. The call is considered local since all components are designed to exist and execute on a shared memory parallel system.

Context

A parallel program is being developed using the Parallel Layers architectural pattern [OR98, Ort07] as a functional parallelism approach in which an algorithm is partitioned among autonomous processes (layer components) 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 synchronisation mechanisms for process communication like semaphores [Dij68, Har98] or monitors [Hoa74, Har98].

Problem

A collection of parallel layers require to communicate by issuing operation calls, and waiting to receive results; every data is locked inside some layer component.

Forces

The following forces should be considered for the *Multiple Local Call* pattern:

- Maintain the precise order operations.
- Communication commonly should be one to many.
- Keep the integrity and order of the results.
- The implementation has to consider shared memory as programming environment.
- The communication should be synchronous.

Solution

Design the programming modules as a set or array of monitors, encapsulating service procedures able to create more modules or to carry out some processing. Each module receives synchronised calls from a caller component, which delegates it a part of a whole processing activity. By allowing a one to many communication, the whole processing activity tends to be partitioned among several programming modules, which at the same time are able to create further modules, in order to continue partitioning the processing activity until it can be serviced by a single programming module. All these components are designed to exist and execute simultaneously on a shared memory parallel system, synchronising their action during the cascade of calls, so the precise order of operations and the integrity and order of results are kept.

Structure

Figure 5 shows the participants and relations that compose the structure of this pattern at only a single stage of communication, using a UML Collaboration Diagram [Fow97].

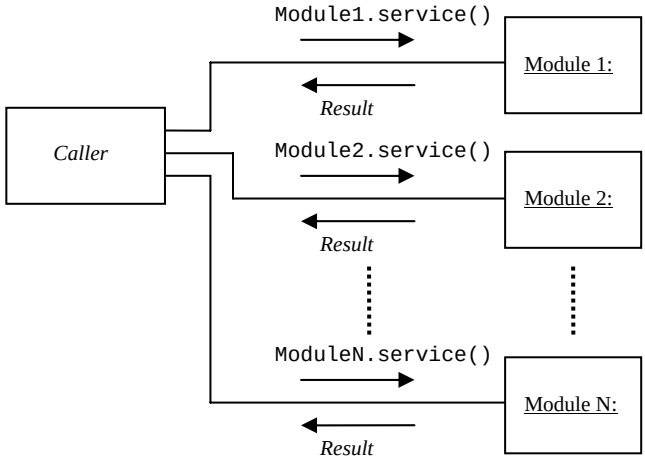

Figure 5. UML Collaboration Diagram for the Local Call pattern.

Participants

- **Caller**. The responsibilities of the caller component is to generate a set of calls to the module components, providing local arguments for each module (and thus, partitioning the processing among the modules) and waiting to receive results from the modules.
- **Module**. The responsibilities of each module component are to re-distribute the call (and the arguments) to another set or array of components, or to carry out the part of the whole processing activity over the local arguments, and hence, produce results which are collected by its caller.

Dynamics

This pattern is expected to operate between components at different layers within a shared memory parallel system. Figure 6 shows the behaviour of the participants of this pattern at a single stage, considering a 1 to N communication.

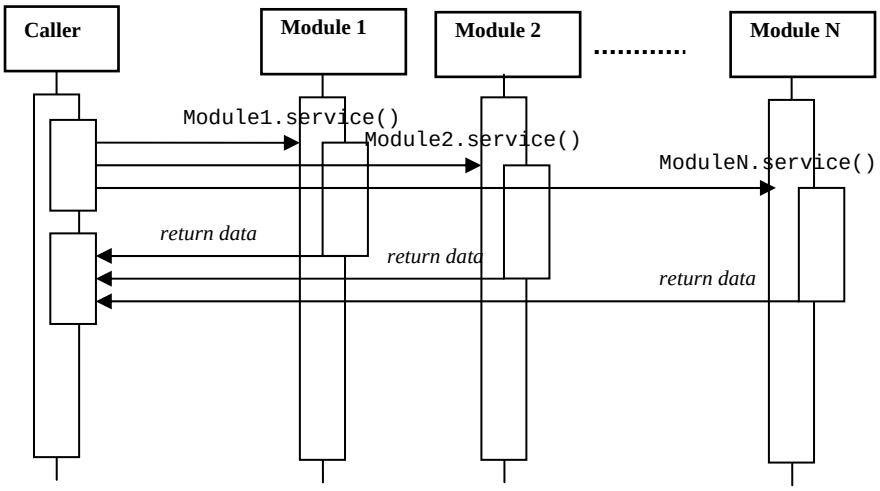

Figure 6. Scenario of the Multiple Local Call pattern.

In this scenario, the following steps are followed:

- The **Caller** produces a group of calls for service to each one of the **Module** components, and waits until they all provide results.

- Each **Module** receives the call, and proceeds to perform the service. As potentially another call can arrive to the **Module** while it is operating, the **Module** is developed using semaphores or a monitor to avoid racing conditions. Once the local operation is finished, every **Module** sends back a result.
- The **Caller** receives the results, and continues processing.

Implementation

The implementation of the Multiple Local Call pattern requires the construction of the Module components to take into consideration its synchronous behaviour. Module software components are allowed to exist and execute on a shared memory environment, as part of a cascade of calls. Commonly, Module components of the same type are used within the same layer.

The Modules can be implemented using semaphores [Dij68, Har98] to synchronise the access to the service, considering the P and V operations, respectively just before and after invoking the service. A better possibility is to consider each Module as a monitor [Hoa74, Har98], which synchronises the access to the service.

Consequences

Benefits

- The Multiple Local Call pattern maintains the precise order operations since it represents a single stage within a cascade of synchronous calls. Hence, only once all the Modules at a layer have completed their operation, the caller is able to continue, perhaps as a single module within another layer.
- As only one caller is used to call and synchronise several modules, communication is kept one to many.
- Due to only synchronous calls are allowed between the caller and the modules, the integrity and order of the results are locally kept.
- The implementation considers the use of shared memory synchronisation mechanisms, such as semaphores and monitors. This simplifies the implementation.
- As modules are implemented as monitors, the caller has to wait until it receives all results from the modules, and thus, the communication is kept synchronous.

Liabilities

- The use of the Multiple Local Call pattern can produce long delays in the communication between components at different layers. Since the caller has to wait until the whole operations are carried out, the communication through the whole hierarchical structure could be slowed down due to the number of modules per caller and the amount of communication between caller and modules.

Known uses

The Multiple Local Call is normally used when the parallel solution of a problem is developed using the Parallel Hierarchies architectural pattern [OR98, Ort07] within a shared memory parallel platform. Hence, it has as many known uses as the Parallel Hierarchies pattern. Particularly, the following known uses are relevant:

- The Multiple Local Call pattern is used in tree-like algorithms, such as searches, in which the data of the problem is provided as arguments to each branch in the tree, and multiple calls are required [Ort07].
- The Multiple Local Call pattern is commonly used when describing a solution based on a divide-and-conquer approach, in which the computation is divided into sub-computations over and over, until a simple operation is required to obtain a result. Assembling all the results provide the global result [And91, And00].

- The Multiple Local Call pattern can be considered a variation of a client-server operation with a simple synchronised call operation, in which a single component acts as a server for a single client [And91, And00].

Related patterns

The *Multiple Local Call* pattern is directly related with any parallel software system developed on a shared memory environment from the Parallel Hierarchies pattern [OR98, Ort07]. It is also related with the pattern for selecting locking primitives, originally proposed by McKenney [McK95], and lately included as part of the POSA 2 book, Patterns for Concurrent and Networked Objects [POSA00].

Multiple Remote Call

The *Multiple Remote Call* pattern describes the design of multiple remote programming modules that encapsulate services or access procedures, which are called or invoked by another component, in order to operate over global and/or local variables. Such an operation is related with delegating a part of a whole processing activity to such a programming module. Components are allowed to execute simultaneously, and thus, they require a synchronous communication during each call. The call is considered remote since components are designed to exist and execute on a distributed memory parallel system.

Context

A parallel program is being developed using the Parallel Layers architectural pattern [OR98, Ort07] as a functional parallelism approach in which an algorithm is partitioned among autonomous processes (layer components) as the processing components of the parallel program. The parallel program is developed for a distributed memory computer, even though it also can be used for a shared memory computer. The programming language to be used counts with synchronisation mechanisms for process communication through remote calls [Bri78, Har98].

Problem

A collection of parallel layers require to communicate by issuing operation calls, and waiting to receive results; every data is locked inside some layer component.

Forces

The following forces should be considered for the *Multiple Remote Call* pattern:

- Maintain the precise order operations.
- Communication commonly should be one to many.
- Keep the integrity and order of the results.
- The implementation has to consider distributed memory as programming environment.
- The communication should be synchronous.

Solution

Design the caller component and the group of programming modules to communicate while they reside in different, remote machines, using remote calls. Each module receives synchronised calls from the caller component, delegating a part of a whole processing activity. In essence, instead of creating a local programming module, a local stub is created and bint to a remote programming module. The local stub is sent messages as if it were the programming module. It receives the messages sent to it and sends the messages onto the remote module, which invokes the adequate service. The result is sent back to the stub, which returns it to the caller. By allowing a one to many communication, the whole processing activity tends to be partitioned among several programming modules, which at the same time are able to create further modules, in order to continue partitioning the processing activity until it can be serviced by a single programming module. All these components are designed to exist and execute simultaneously on a distributed memory

parallel system, synchronising their action during the cascade of calls, so the precise order of operations and the integrity and order of results are kept.

Structure

Figure 7 shows the participants and relations that compose the structure of this pattern at only a single stage of communication, using a UML Collaboration Diagram [Fow97].

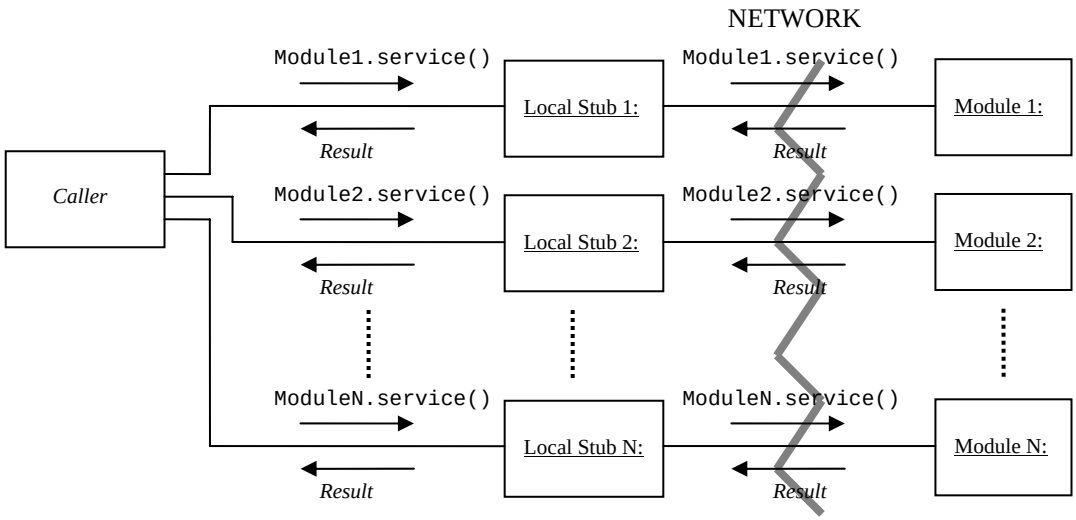

Figure 7. UML Collaboration Diagram for the Multiple Remote Call pattern.

Participants

- **Caller**. The responsibilities of the caller component is to generate a set of calls to the module components, providing local arguments for each module (and thus, partitioning the processing among the modules) and waiting to receive results from the modules.
- **Remote Module**. The responsibilities of each remote module component are to re-distribute the call (and the arguments) to another set or array of components, or to carry out the part of the whole processing activity over the local arguments, and hence, produce results which are collected by its caller.
- **Local Stub**. The responsibilities of is to serve as a local representative of a remote module in a distant computer, receiving calls and re-directing them to its correspondent remote module, as well as receiving the result from the remote module and provide it to the local caller.

Dynamics

This pattern is expected to operate between components at different layers within a distributed memory parallel system. Figure 8 shows the behaviour of the participants of this pattern at a single stage, considering a 1 to N communication.

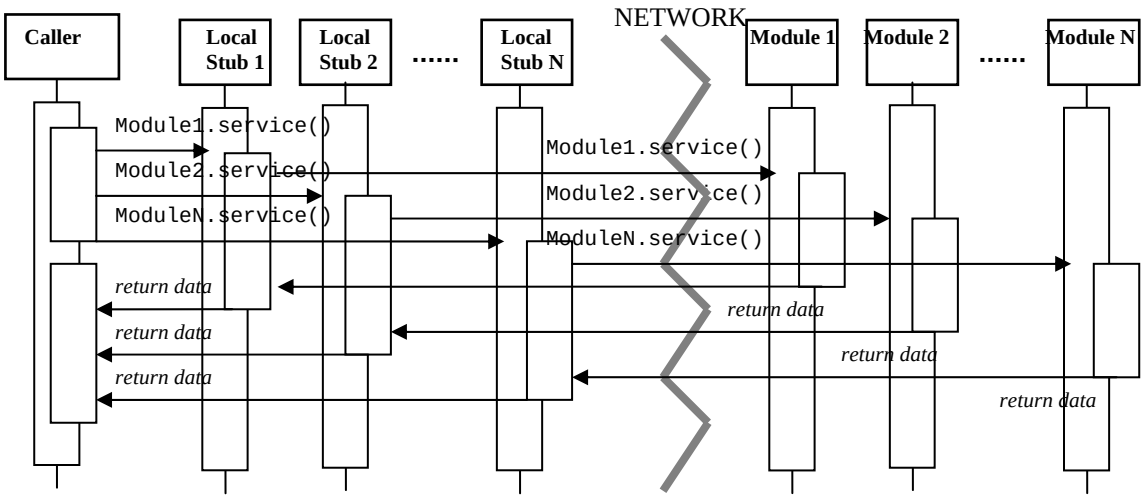

Figure 8. Scenario of Pipes and filters pattern.

In this scenario, the following steps are followed:

- The **Caller** produces a group of calls for service to each one of the **Module** components through the **Local Stubs** locally available. The **Caller** waits until all the **Local Stubs** provide results.
- The **Local Stubs** receive the call and re-direct it through the network to the adequate **Module**, and waits for a result.
- Each **Module** receives the call, and proceeds to perform the service. Once its local operation is finished, every **Module** sends back a result.
- The **Local Stubs** receive the result from its correspondent **Module** through the network, and makes it available to the **Caller**.
- The **Caller** receives the results, and continues processing.

Implementation

The implementation of the Multiple Remote Call pattern requires the construction of the Local Stubs and the Module components, taking into consideration their synchronous behaviour. Both, Local Stubs and Module software components are allowed to exist and execute on a distributed memory environment, as part of a cascade of calls. Commonly, Local Stubs and Module components of the same type are used within the same layer.

Local Stubs and Modules can be implemented as counterparts of remote calls [Bri78, Har98] to carry out the communication and synchronise the access to the service. Particularly, Local Stubs can be implemented to act as monitors [Hoa74, Har98], synchronising the access to the remote service.

Consequences

Benefits

- The Multiple Remote Call pattern maintains the precise order operations since it represents a single stage within a cascade of synchronous calls. Hence, only once all the Modules at a layer have completed their operation, the caller is able to continue, perhaps as a single module within another layer.
- As only one caller is used to call and synchronise several local stubs correspondent to several modules, communication is kept one to many.
- Due to only synchronous calls are allowed between the caller and the local stubs, the integrity and order of the results are locally kept.

- The implementation considers the use of shared memory synchronisation mechanisms, such as monitors, for the local stubs. This simplifies their implementation.
- As local stubs are implemented as monitors, the caller has to wait until it receives all results from them, and thus, the communication is kept locally synchronous.

Liabilities

- The use of the Multiple Remote Call pattern can produce long delays in the communication between components at different layers, due to the use of components related with the remote calls through the network. Also, since the caller has to wait until the whole operations are carried out, the communication through the whole distributed hierarchical structure could be slowed down due to the number of modules per caller and the amount of communication between local stubs and modules.
- The Multiple Remote Call pattern can be used in a shared memory environment. However, due to the calls are considered remote, synchronisation problems could arise which could slow down the operation of the whole structure. In such a case, the use of the Multiple Local Call pattern could simplify the operation, providing a more efficient response.

Known uses

The Multiple Remote Call is normally used when the parallel solution of a problem is developed using the Parallel Hierarchies architectural pattern [OR98, Ort07] within a distributed memory parallel platform (although it can be also used for a shared memory system). Hence, it has as many known uses as the Parallel Hierarchies pattern. Particularly, the following known uses are relevant:

- The Multiple Remote Call pattern is used in hypercube-like platforms to carry out search computations, in which the data of the problem is provided as arguments to processors within a dimension in the hypercube, and multiple remote calls are required to distribute computations to and retrieve data from processors [Ort07].
- The Multiple Remote Call pattern is commonly used when describing a distributed Java program based on a Remote Method Invocation approach, in which the computation is divided into sub-computations among distributed computing resources [Har98].
- The Multiple Remote Call pattern can be considered a variation of a remote procedure call operation, in which a single component acts as a server for a single client [Bri78, And91, And00].

Related patterns

The *Multiple Remote Call* pattern is directly related with any parallel software system developed on a shared memory environment from the Parallel Hierarchies pattern [OR98, Ort07]. It is also related with the Multiple Local Call pattern, as a version for distributed memory systems.

Shared Variable Channel

The *Shared Variable Channel* pattern describes the design of a channel component based on shared variables and synchronisation mechanisms, which serve for implementing send and receive operations that emulate the behaviour of a channel component for a shared memory parallel system.

Context

A parallel program is being developed using the Communicating Sequential Elements architectural pattern [OR98, Ort00] 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 synchronisation mechanisms for process communication such as semaphores [Dij68, Har98] and monitors [Hoa74, Har98].

Problem

An element needs to exchange values with its neighbouring elements. Every data is locked inside an element, which is responsible for processing that data and only that data.

Forces

The following forces should be considered for the *Shared Variable Channel* pattern:

- Maintain the precise order of the transferred data through the channel.
- Communication should be point to point and bidirectional.
- Keep the integrity of transferred data.
- The implementation has to consider the shared memory as programming environment.
- The communication should be asynchronous.

Solution

The idea is to emulate the behaviour of a channel component using shared variables. Thus, use a couple of shared variables to implement the channel component, considering it as a bi-directional, shared memory communication means between elements. Such shared variables require to be safely modified by read and write operations from the elements. Hence, programming language synchronisation mechanisms (such as semaphores or monitors) have to be considered to preserve the order and integrity of the transferred data, along with sending (writing) and receiving (reading) operations.

Structure

The participants and relations that compose the structure of this pattern are shown using a UML Collaboration Diagram [Fow97] for the description (Figure 9).

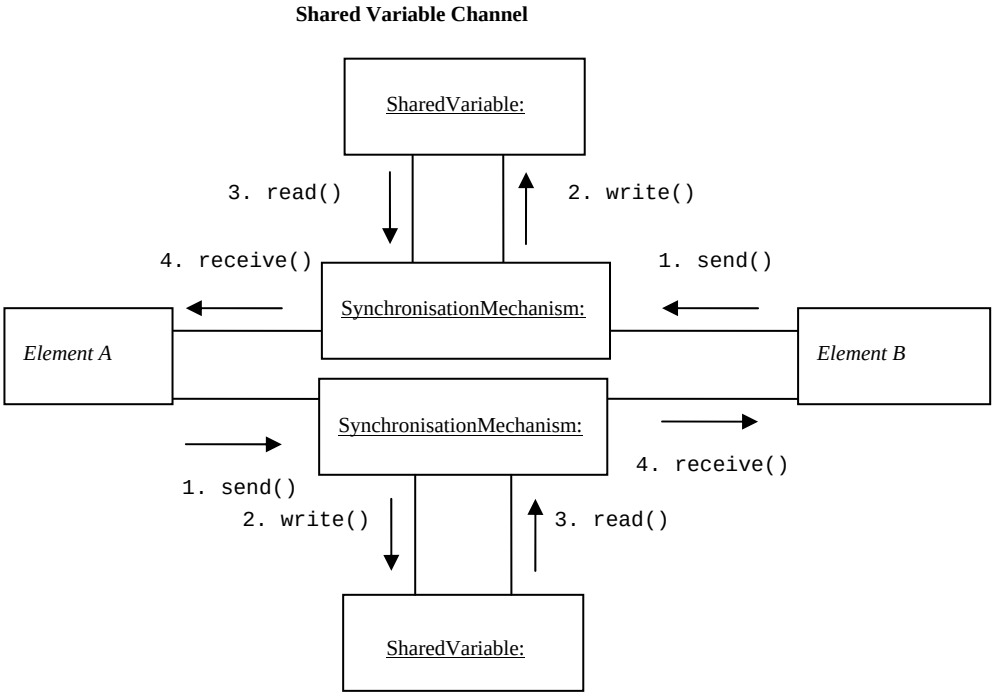

Figure 9. Collaboration Diagram for the Shared Variable Channel pattern.

Participants

- **Synchronisation Mechanisms**. The responsibility of the synchronisation mechanisms is to synchronise the access to the shared variables, translating the send and receive operations into adequate operations for writing to and reading from the shared variables. The synchronisation mechanism is, then, in charge of keeping the order and integrity of the shared data.
- **Shared Variables**. The responsibility of the shared variables is to serve as a repositories for the data to be transferred. Both can be designed as buffers (arrays of a particular type) with an specific size, for accomplishing with the use of asynchronous communication between the communicating elements.

Dynamics

The behaviour of this pattern is expected to emulate the operation of a channel component within a shared memory parallel system. Hence, Figure 10 shows the behaviour of the participants of this pattern, aiming to carry out such an emulation.

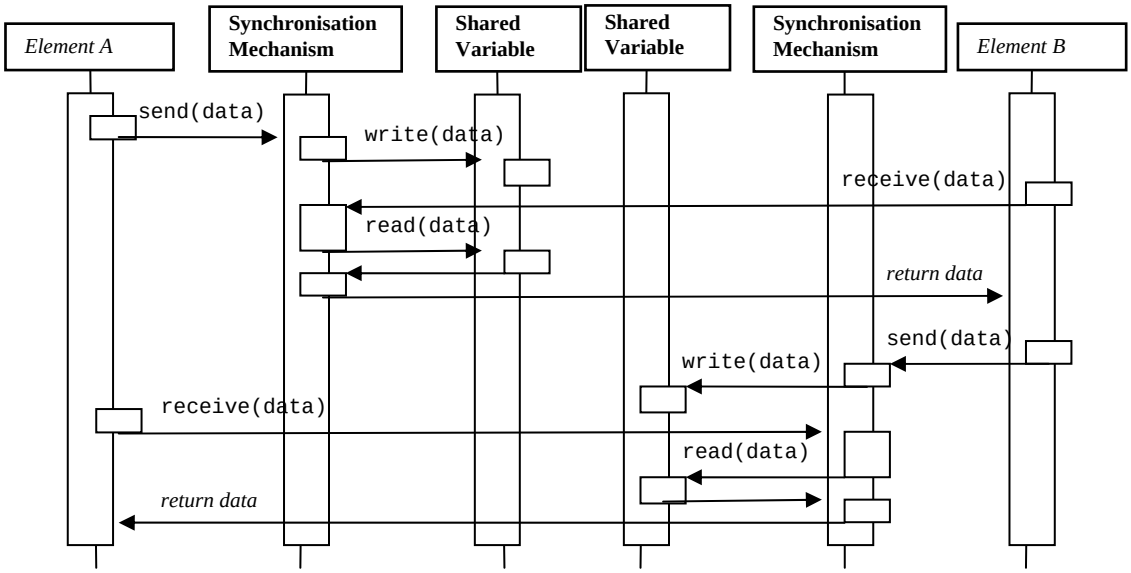

Figure 10. Scenario for the Shared Variable Channel pattern.

In the scenario shown in Figure 10, the following steps are followed for communicating data from *Element A* to *Element B*, and vice versa:

- *Element A* sends a data item by issuing a `send(data)`operation to the **Synchronisation Mechanism**.
- The **Synchronisation Mechanism** verifies if the *Element B* is not reading the **Shared Variable**. If this is the case, then it translates the sending operation, allowing a `write(data)`operation of the data item on the **Shared Variable**. Otherwise, it blocks the operation until the **Shared Variable** can be modified.
- On the other hand, when the *Element B* attempts to receive the data item, it does so by issuing a `receive(data)` request to the **Synchronisation Mechanism**. Again, only if the *Element A* is not writing the **Shared Variable**, the **Synchronisation Mechanism** grants a `read()` operation from the **Shared Variable**, returning the requested data item.
- On the other hand, when *Element B* sends a data item to *Element A*, a similar procedure is carried out: data is sent by issuing a `send(data)`operation to the **Synchronisation Mechanism**.
- The **Synchronisation Mechanism** verifies if the *Element A* is not accessing the **Shared Variable**. If this is the case, then it translates the sending operation, allowing a `write(data)`operation of the data item on the **Shared Variable**. Otherwise, it blocks the operation until the **Shared Variable** can be modified.

- *Element A* reads the data item by issuing a `receive(data)` request to the **Synchronisation Mechanism**. Again, only if the *Element B* is not writing the **Shared Variable**, the **Synchronisation Mechanism** grants a `read()` operation from the **Shared Variable**, returning the requested data item.
- The communication flow is bidirectional, allowing the exchange of data to and from neighbouring elements.

Implementation

The implementation of the Shared Variable Channel requires the implementation of the Synchronisation Mechanisms and the Shared Variables. Both types of software components exist and execute on a shared memory environment.

The Synchronisation Mechanisms can be implemented using semaphores [Dij68, Har98] to synchronise the access to the Shared Variables, considering the P and V operations, respectively just before and after invoking the `write()`or `read()`operations that modify the state of each Shared Variable. Another possibility is the use of monitors [Hoa74, Har98], which consider the synchronisation over the very `write()`or `read()` operations.

The Shared Variables can be implemented as bounded buffers of a particular type, which can be modified by reading or writing operations from the Synchronisation Mechanisms, considering an asynchronous communication approach. The Shared Variables, hence, are capable of keeping several data values in order to allow a bidirectional communication between *Element A* and *Element B*.

Consequences

Benefits

- The Shared Variable Channel keeps the precise order of the transferred data by implementing a two directional FIFO policy, synchronising the access to both Shared Variables.
- The Shared Variable Channel is designed to deal with point to point and bidirectional communication.
- Both Synchronisation Mechanisms are in charge of keeping the integrity of transferred data, by assuring that, at any given moment, only one element has actual access to any of the Shared Variables.
- The implementation is particularly developed for a shared memory programming environment.
- The Shared Variable Channel uses asynchronous communications, by implementing the Shared Variables as two bounded buffers.

Liabilities

- The communication speed of the Shared Variable Channel is as slow as the elements it connects. Therefore, to improve communication performance, changes to the amount of processing of the elements have to be considered.
- The Shared Variable Channel is difficult to extent to one-to-many, many-to-one, and many-to-many communications.
- The implementation based on semaphores and monitors makes this pattern only to be used into a shared memory environment. Porting it to a distributed memory parallel platform would require to replace each Shared Variable Channel by a Message Passing Channel.

Known uses

The Shared Variable Channel is normally used when the parallel solution of a problem is developed using the Communicating Sequential Elements architectural pattern [OR98, Ort00] within a shared memory parallel platform. Hence, it has as many known uses as the Communicating Sequential Elements pattern. Particularly, the following known uses are relevant:

- The Shared Variable Channel pattern is used when implementing a domain parallelism program that solves the Laplace Equation for a shared memory computer. Each element is expected to solve a Laplace Equation locally, exchanging results with its neighbours in a one-, two-, or n-dimensional mesh [KSS96, Har98].
- The Shared Variable Channel pattern is used when using the Communicating Sequential Elements pattern to solve a systolic matrix multiplication. Channels are used to allow the flow of matrix data through components that go on multiplying them, and locally adding the products [Har98].
- The Shared Variable Channel pattern is used in a shared memory computer model of climate, in which each element (or set of elements) compute the variation through time of one or several variables (temperature, humidity, pressure, etc.), and exchange data in order to model the effect of, say, the atmospheric model over the ocean model, and vice versa [Fos94].

Related patterns

The *Shared Variable Channel* pattern is directly related with any parallel software system developed on a shared memory environment from the Communicating Sequential Elements pattern [OR98, Ort00]. It can be considered as a two-directional version of the Shared Variable Pipe pattern. As so, it is related with the pattern for selecting locking primitives, originally proposed by McKenney [McK95], and lately included as part of the POSA 2 book, Patterns for Concurrent and Networked Objects [POSA00].

Message Passing Channel

The *Message Passing Channel* pattern describes the design of a channel component based on message passing, by implementing send and receive operations that perform the communications of the channel component for a distributed memory parallel system (although it can be used for a shared memory parallel system as well).

Context

A parallel program is being developed using the Communicating Sequential Elements architectural pattern [OR98, Ort00] 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 within a distributed memory computer, but it also can be used within a shared memory computer. The programming language to be used counts with synchronisation mechanisms for process communication through message passing [Hoa78, Har98] or rendezvous [Bri78, Har98].

Problem

An element needs to exchange values with its neighbouring elements. Every data is locked inside an element, which is responsible for processing that data and only that data.

Forces

The following forces should be considered for the *Message Passing Channel* pattern:

- Maintain the order of the transferred data through the channel.
- Communication should be point to point and bidirectional.
- The implementation has to consider a distributed memory as programming environment.
- The data transference should be performed asynchronously.

Solution

Design a channel component as a distributed software structure connecting the elements executing on two different processors or computers. The software structure is composed of communication end points (commonly, sockets), some synchronisation mechanisms, and data streams. These components are put together in order to achieve a two-directional, distributed memory communication component between elements executing on different processors or computers.

Structure

Figure 11 shows the participants and relations that compose the structure of this pattern, using a UML Collaboration Diagram [Fow97].

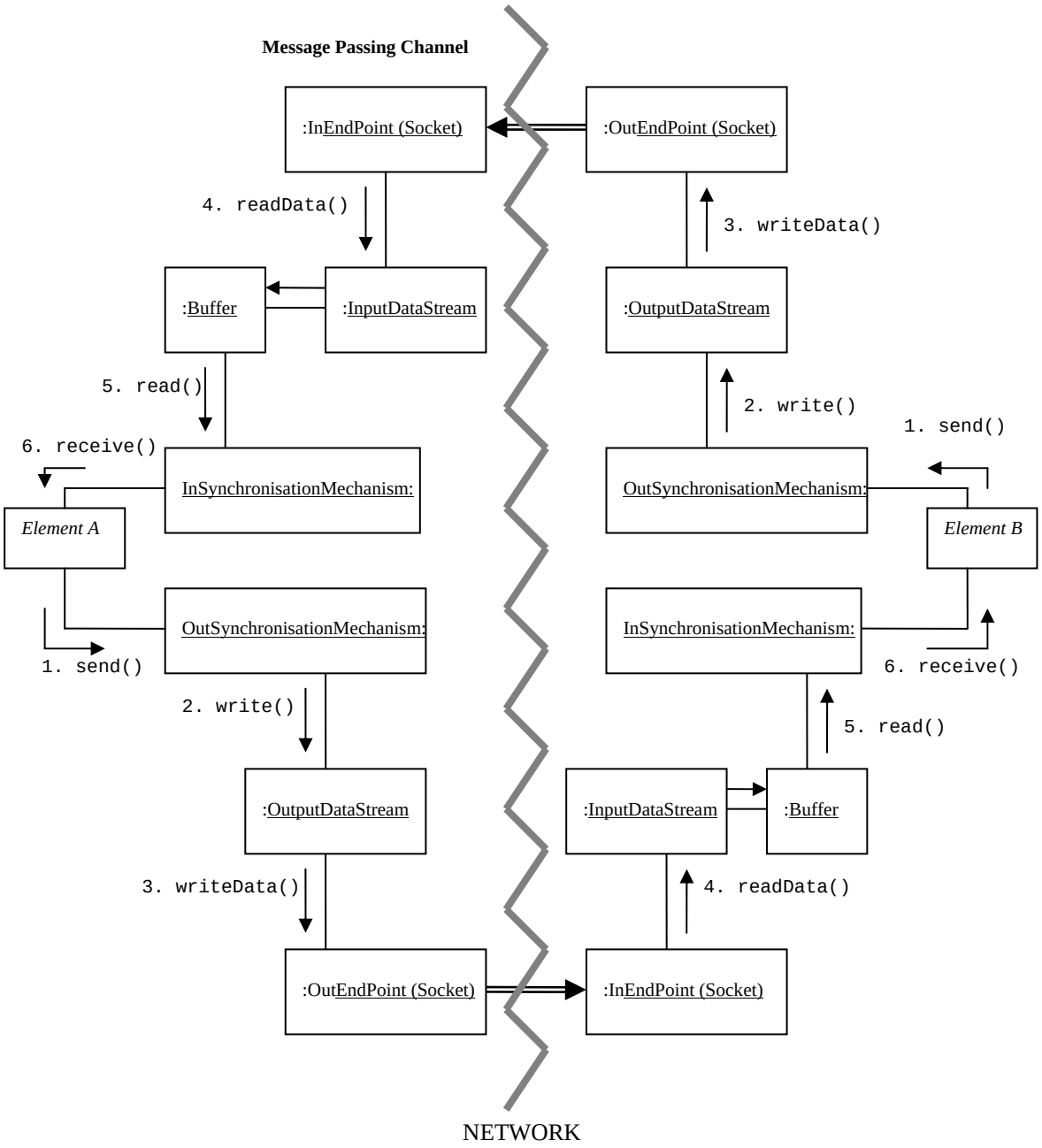

Figure 11. The Message Passing Channel pattern.

Participants

- **OutSynchronisationMechanisms** and **InSynchronisationMechanisms**. The responsibility of the synchronisation mechanisms is to synchronise the access to the data streams (**OutDataStreams** and **InDataStreams**), so only one processing component has access to any of them at a given moment. The synchronisation mechanisms are, then, in charge of keeping the order and integrity of the data written to or read from the data streams.
- **OutDataStreams** and **InDataStreams**. The responsibility of the data streams is to transiently store the serialised data to be passed through the channel. Every data item must be serialized, that is, converted into a stream of bytes, which is the way in which data is transferred through a

network connecting processors or computers. Data streams can be written to and read from communication end points (**OutEndPoints** and **InEndPoints**).

- **OutEndPoints** and **InEndPoints**. The responsibility of the communication end points is to send data back and forth between the processors or computers. These sort of communication end points are commonly implemented as sockets.
- **Buffers**. The responsibility of the buffers is to serve as repositories for the data to be received. They are normally designed as arrays of a particular type with an specific size. The buffers allow the use of asynchronous communication between elements.

Dynamics

This pattern is expected to operate as a channel component for a distributed memory parallel system. Hence, Figure 12 shows the behaviour of the participants of this pattern, aiming to carry out such an operation.

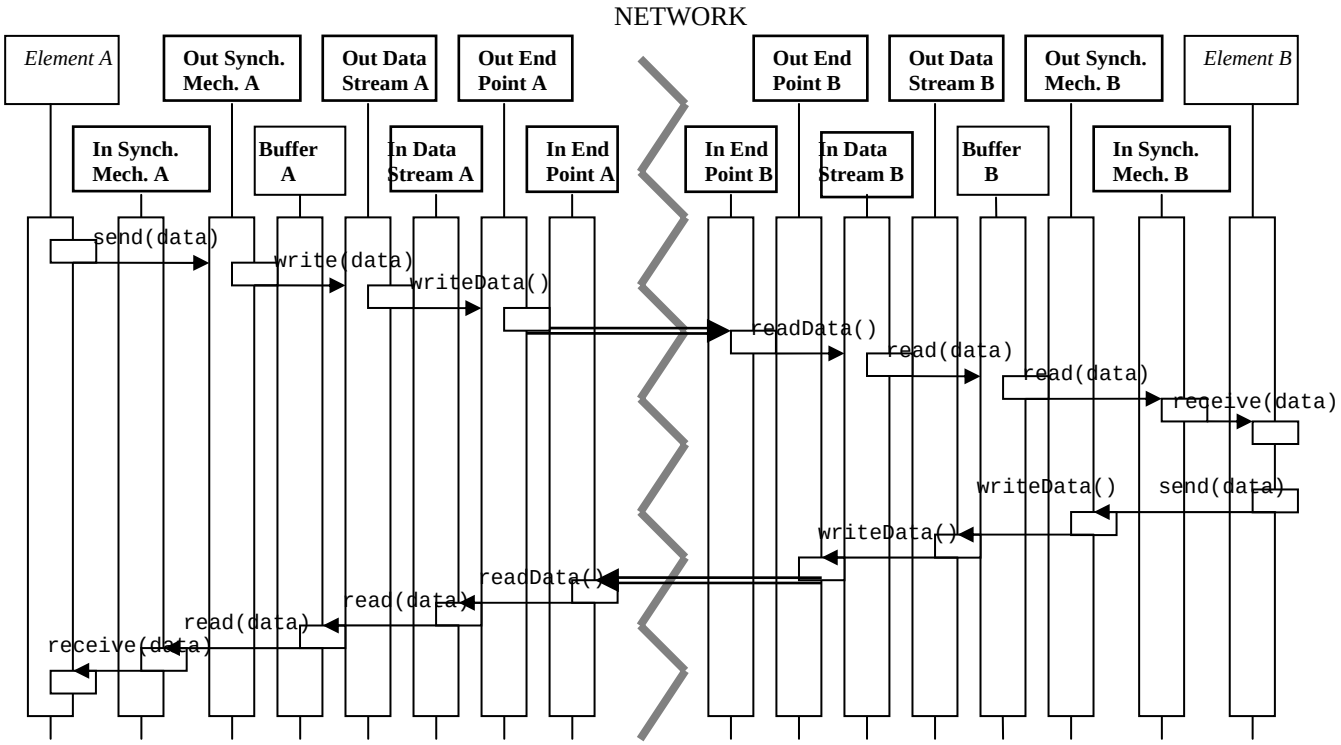

Figure 12. Scenario of the Message Passing Channel pattern.

In this scenario, the following steps are followed:

- *Element A* sends a data item to the Message Passing Pipe, issuing a `send(data)`operation to the **Out Synchronisation Mechanism A**.
- The **Out Synchronisation Mechanism A** verifies if no other process is accessing the **Out Data Stream A**. If this is the case, then it translates the sending operation, allowing a `write(data)`operation of the data item on the **Out Data Stream A**. Otherwise, it blocks the operation until the **Out Data Stream A** can be written.
- The **Out Data Stream A** generates a `writeData()` operation over the **Out End Point A** (normally, a socket), so the data item is sent through the network to the appropriate processor or computer.
- The data item is received by the **In End Point B** (a socket) which is read by the **In Data Stream B** by issuing a `readData()` operation. The data item is allocated into the **Buffer B**, so the communication is kept asynchronous. From the **Buffer B**, *Element B* is able to receive data by issuing a `receive(data)` request to the **In Synchronisation Mechanism B**. Again, only if

no other process is reading from the **Buffer B**, the **Synchronisation Mechanism B** grants a `read()` operation from the **Buffer B**, allowing to *Element B* to read the requested data item.

- On the other hand, *Element B* is able to send a data item to the Message Passing Pipe, by issuing a `send(data)`operation to the **Out Synchronisation Mechanism B**.
- The **Out Synchronisation Mechanism B** verifies if no other process is accessing the **Out Data Stream B**. If this is the case, then it translates the sending operation, allowing a `write(data)`operation of the data item on the **Out Data Stream B**. Otherwise, it blocks the operation until the **Out Data Stream B** can be written.
- The **Out Data Stream B** generates a `writeData()` operation over the **Out End Point B** (normally, a socket), so the data item is sent through the network to the appropriate processor or computer.
- The data item is received by the **In End Point A** (a socket) which is read by the **In Data Stream A** by issuing a `readData()` operation. The data item is allocated into the **Buffer A**, so the communication is kept asynchronous. From the **Buffer A**, *Element A* receives the data item by issuing a `receive(data)` request to the **In Synchronisation Mechanism A**. Again, only if no other process is reading from the **Buffer A**, the **Synchronisation Mechanism A** grants a `read()` operation from the **Buffer A**, allowing *Element A* to read the requested data item.

Implementation

The implementation of the Message Passing Channel requires the use or construction of the Synchronisation Mechanisms, the Data Streams, the Buffers, and the End Points. All these software components should exist and execute on a distributed memory environment, between two communicating processor or computers.

The Synchronisation Mechanisms can be implemented using semaphores [Dij68, Har98] to synchronise the access to the Data Streams and the Buffers, considering the P and V operations, respectively just before and after invoking the `write()`or `read()`operations that modify the state of the Data Streams and the Buffers. Another possibility is the use of monitors [Hoa74, Har98], which consider the synchronisation over the very `write()`or `read()` operations.

Data Streams are a common communication form in many programming languages, used to serialise data, that is, converted into a stream of bytes, which is the way in which data is transferred through a network connecting processors or computers.

The End Points are commonly sockets, which is a network communication mechanism common in several programming languages. Sockets are able to send data back and forth between the processors or computers of a network system.

The Buffers can be implemented as arrays of a particular type, which can be modified by reading or writing operations from the Synchronisation Mechanisms, considering an asynchronous communication approach. The Buffers should be able of keeping several data values in order to cope *Elements* executing processing activities on different processors, and hence, very likely at different speeds.

Consequences

Benefits

- The Message Passing Channel keeps a FIFO policy in each one of both directions, by synchronising the access to the Buffers on both communicating sides.
- The Message Passing Channel is designed to deal with point to point and bidirectional communication. However, it can be extended to a one-to-many, many-to-one, and many-to-many communications, by using several Synchronisation Mechanisms over several Data

Streams, Buffers, and End Points. Also, it able to keep a simultaneous bidirectional flow of data by using two different Data Streams between processing components.
- The implementation based on Data Streams and End Points is explicitly developed for a distributed memory programming environment. However, it can be used within a shared memory programming environment as well.
- The Message Passing Channel uses asynchronous communications, by implementing a bounded buffer on the receiving side.

Liabilities
- The communication speed of the Message Passing Channel depends not only on the slowest element it connects, but also on features and characteristics of the communication network on which it executes. Therefore, communication performance is commonly affected by non-determinism issues which can be determinant on the communication speed.
- The Message Passing Channel can be used for one-to-many, many-to-one, and many-to-many communications, although the implementation could require the use of several semaphores or monitors. This fact could make it difficult to implement the whole distributed communication components.
- If any element is a lot faster than any of its communicating counterparts, this could produce a great unbalance on the whole computation. This is a signal that the division of the data among elements could be wrong. If it is the case, perhaps removing the channel and a change on the granularity of the processing components, by considering a different distribution of data among the processing components, could solve the unbalance situation.
- The implementation based on data streams and end points makes this pattern to be suitable for a distributed memory environment. Nevertheless, it can also be used within a shared memory environment, in which, however, it may not have a good performance due to the many communications involved. Hence, for a shared memory parallel platform and to keep a better performance, it would be advisable to replace each Message Passing Channel by a Shared Variable Channel.

Known uses

The Message Passing Channel is commonly used when the parallel solution of a problem is developed using the Communicating Sequential Elements architectural pattern [OR98, Ort00] within a distributed memory parallel platform. Hence, it has as many known uses as the Communicating Sequential Elements pattern. Particularly, the following known uses are relevant:
- The Message Passing Channel pattern has been used when implementing the distributed solutions for the One-dimensional Heat Equation, allowing the data exchange among processing components that compute the Heat Equation for an interval of a one-dimensional substrate, such as a wire [Ort05].
- The Message Passing Channel pattern is used when describing a domain parallelism, distributed memory solution for the Two-dimensional Wave Equation [DW96].
- The Message Passing Channel pattern can be used for a point-to-point, bidirectional data exchange between processes executing on different computers of a network system [And91, Har98, And00].

Related patterns

The *Message Passing Channel* pattern is directly related with any parallel software system developed on a distributed memory environment from the Communicating Sequential Elements pattern [OR98, Ort00]. It can be considered as a two-directional version of the Message Passing Pipe pattern.

Local Rendezvous

The *Local Rendezvous* pattern describes the design of a local, point-to-point, bi-direction, and synchronous communication component that allows the exchange of information between a manager and a worker or between a shared resource and a sharer. The manager and/or shared resource encapsulate a data structure, whose parts or pieces can be synchronously read or written at a precise moment by a single worker and/or a single sharer component. Data is allowed to flow from the manager/shared resource to the worker/sharer, and vice versa. Components are allowed to execute simultaneously. The rendezvous is considered local since components are designed to exist and execute on a shared memory parallel system.

Context

A parallel program is being developed using the Manager-Workers architectural pattern [OR98, Ort04] or the Shared Resource architectural pattern [OR98, Ort03] as activity parallelism approaches in which algorithm and data are partitioned among the autonomous processes (workers or sharers) as the processing components of the parallel program. The parallel program is developed within a shared memory computer. The programming language to be used counts with synchronisation mechanisms for process communication, such as semaphores [Dij68, Har98] or monitors [Hoa74, Har98].

Problem

Communication is required for workers or sharers by reading and writing data objects from the manager or the shared resource, within a shared memory system.

Forces

The following forces should be considered for the *Local Rendezvous* pattern:

- Keep the integrity and order of the encapsulated data structure.
- Communication commonly should be point-to-point, bi-directional, and synchronous.
- The implementation has to consider shared memory as programming environment.

Solution

Use a common component called *rendezvous* to carry out a point-to-point, bi-direction, and synchronous exchange of information between a manager and a worker or between a shared resource and a sharer. The communication considers the isolated case of communication between a single worker and the manager, or a single sharer and the shared resource. A worker or a sharer read or write a piece of data from the manager or shared resource synchronously. This keeps the integrity and order of the encapsulated data structure. Components (manager and workers, or shared resource and sharers) are allowed to simultaneously exist and execute on a shared memory parallel system.

Structure

Figure 13 shows the participants and relations that compose the structure of this pattern, using a UML Collaboration Diagram [Fow97].

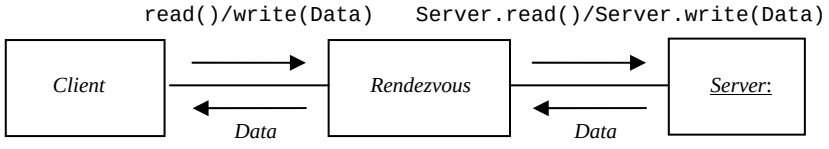

Figure 13. Local Rendezvous pattern.

Participants

- **Client**. The Client component can be a worker or a sharer, whose responsibilities are to requests read operations to get pieces of the data structure from the Server, process them, and request write operations of the resulting data to the data structure within the Server.
- **Server**. The Server component can be a manager or a shared resource, whose responsibilities are to keep the integrity and order of its local data structure and to server the read and write requests from the Clients.
- **Rendezvous.** The Rendezvous component is in charge of allowing the point-to-point, bi-directional, and synchronous communication between a client and a server. It does so by encapsulating the read and write calls, so the client can only direct requests to the Rendezvous. There should be as many Rendezvous components as Clients within the Manager-Workers structure or the Shared Resource-Sharers structure.

Dynamics

Figure 14 shows the behaviour of the participants of this pattern, considering a single client for the server.

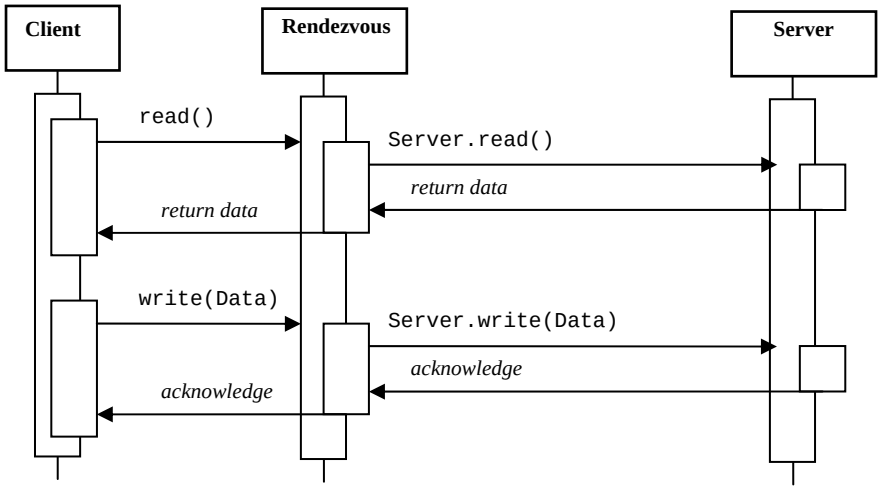

Figure 14. Scenario of the Local Rendezvous pattern.

In this scenario, the following steps are followed:

- A **Client** (whether a worker or a sharer) requests a read operation of data from a **Server** (whether a manager or a shared resource). So, it directs a read operation to the **Rendezvous** component, so it re-directs the read call to the proper **Server**. Once the **Server** makes the data available to the **Rendezvous** component, this provides the data to the **Client**, finishing a single read operation.
- On the other hand, when the **Client** (whether a worker or a sharer) requests a write operation of data to the **Server** (whether a manager or a shared resource), it directs a write operation to the **Rendezvous** component. This re-directs the write call to the proper **Server**. Once the **Server** receives the data from the **Rendezvous** component, it provides an acknowledgement message to the **Client**, finishing a single write operation.

Implementation

The implementation is mainly based on the creation of the Rendezvous component. Since it is considered to be developed for a shared memory parallel system, the Rendezvous component is created as a common component accessible to Client and Server as well, using a synchronisation mechanism such as semaphores [Dij68, Har98] to synchronise the access to the Rendezvous component, considering the P and V operations, respectively just before and after invoking the `write()`or `read()`operations. Another possibility is the use of monitors [Hoa74, Har98], which consider the synchronisation over the very `write()`or `read()` operations.

Consequences

Benefits

- The integrity and order of the encapsulated data structure is kept by allowing only point-to-point, synchronous read/write operations between clients and server.
- The Rendezvous component is developed to keep a point-to-point, bi-directional, and synchronous communication.
- The implementation is carried out using semaphores or monitors, considering a shared memory programming environment.

Liabilities

- The use of synchronous communications between server and clients may slow down the performance of the whole structure, particularly if the number of clients tends to be large, or the communications are very frequent. This can be mitigated by changing the granularity of the pieces of data which are made available in read operations and/or included to the data structure due to a write operation.

Known uses

The Local Rendezvous is normally used when the parallel solution of a problem is developed using the Manager-Workers architectural pattern [OR98, Ort04] or the Shared Resource pattern [OR98, Ort03] within a shared memory parallel platform. Hence, it has as many known uses as these architectural patterns. Particularly, the following known uses are relevant:

- The Local Rendezvous pattern is used when implementing a Manager-Workers activity parallelism program that solves a matrix multiplication for a shared memory computer. Each element is expected to solve a local scalar product of a row from the first matrix and a column from the second. Both sub-arrays have to be read by the workers from the manager. The result of such a product is a number which is returned to the manager, who writes it on the proper position within the resulting matrix [KSS96, Har98].
- The Local Rendezvous pattern is used when using the Shared Resource pattern for creating a Token Space. It is used to allow reading and writing over the token space by the source, sorter, merger, and reporter components [Ort03].
- The Local Rendezvous pattern is used in a shared memory computer to model the Dinning Philosophers problem, originally proposed by E.W. Dijkstra, and developed as a Shared Resource. Every time a philosopher takes the forks, it reads data from a dinning server. Synchronisation and communication is carried out by Rendezvous component, which allow the execution of take and deposit procedures [Har98].

Related patterns

The *Local Rendezvous* pattern is directly related with any parallel software system developed on a shared memory environment from the Manager-Workers pattern [OR98, Ort04] or the Shared Resource pattern [OR98, Ort03]. It is related with the pattern for selecting locking primitives, originally proposed by McKenney [McK95], and lately included as part of the POSA 2 book, Patterns for Concurrent and Networked Objects [POSA00].

Remote Rendezvous

The *Remote Rendezvous* pattern describes the design of a remote, point-to-point, bi-direction, and synchronous communication component that allows the exchange of information between a manager and a worker, or between a shared resource and a sharer. The manager and/or shared resource encapsulate a data structure, whose parts or pieces can be synchronously read or written at a precise moment by a single remote worker and/or a single remote sharer component. Data is allowed to flow from the manager/shared resource to the worker/sharer, and vice versa. Components execute simultaneously. The rendezvous is considered remote since components are designed to exist and

execute on a distributed memory parallel system (although they can be used within a shared memory parallel platform).

Context

A parallel program is being developed using the Manager-Workers architectural pattern [OR98, Ort04] or the Shared Resource architectural pattern [OR98, Ort03] as activity parallelism approaches in which data is partitioned among the autonomous processes (workers or sharers) as the processing components of the parallel program. The parallel program is developed within a distributed memory computer, but it also can be used within a shared memory computer. The programming language to be used counts with synchronisation mechanisms for process communication through remote calls [Bri78, Har98].

Problem

Communication is required so workers or sharers are able to read and write data by sending and receiving data objects from the manager or the shared resource, within a distributed memory system.

Forces

The following forces should be considered for the *Remote Rendezvous* pattern:

- Keep the integrity and order of the encapsulated data structure.
- Communication commonly should be point-to-point, bi-directional, and synchronous.
- The implementation has to consider distributed memory as programming environment, although it could be used on a shared memory system.

Solution

Design a remote, point-to-point, bi-direction, and synchronous *rendezvous* component to allow the exchange of information between a manager and a worker, or between a shared resource and a sharer. Such a component allows data to flow from the manager/shared resource to the worker/sharer, and vice versa. The rendezvous is considered remote since components are designed to exist and execute on a distributed memory parallel system (although they can be used within a shared memory parallel platform).

Structure

Figure 15 shows the participants and relations that compose the structure of this pattern, using a UML Collaboration Diagram [Fow97].

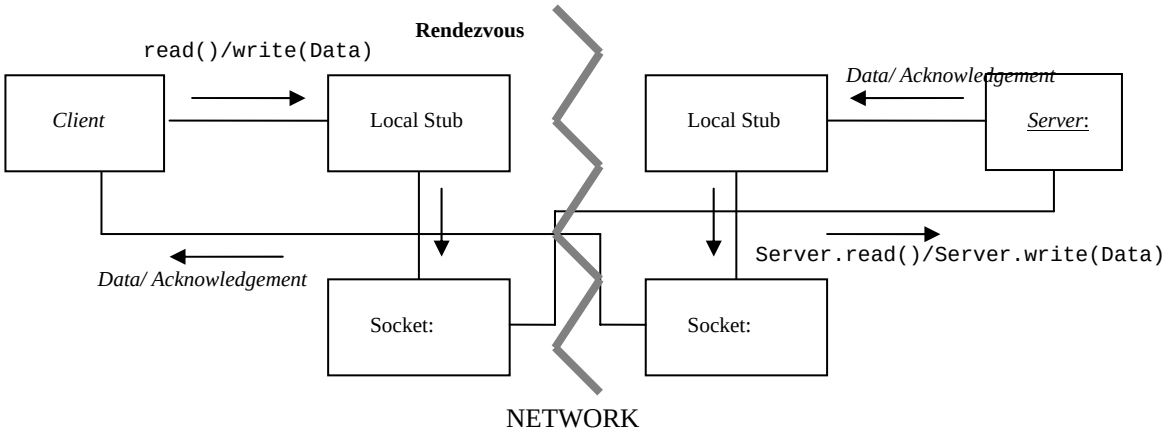

Figure 15. Remote Rendezvous pattern.

Participants

- **Client**. The Client component can be a worker or a sharer, whose responsibilities are to requests read operations to get pieces of the data structure from the Server, process them, and request write operations of the resulting data to the data structure within the Server.
- **Server**. The Server component can be a manager or a shared resource, whose responsibilities are to keep the integrity and order of its local data structure and to server the read and write requests from the Clients.
- **Local Stubs.** The Local Stubs are in charge of controlling the communication between a client and a server. It does so by issuing read and write calls through the Sockets. There should be a Local Stub and a Socket for the server and for each client.
- **Sockets**. The responsibility of the Sockets is to send data back and forth between the processors or computers.

Dynamics

Figure 16 shows the behaviour of the participants of this pattern, considering a single client for the server.

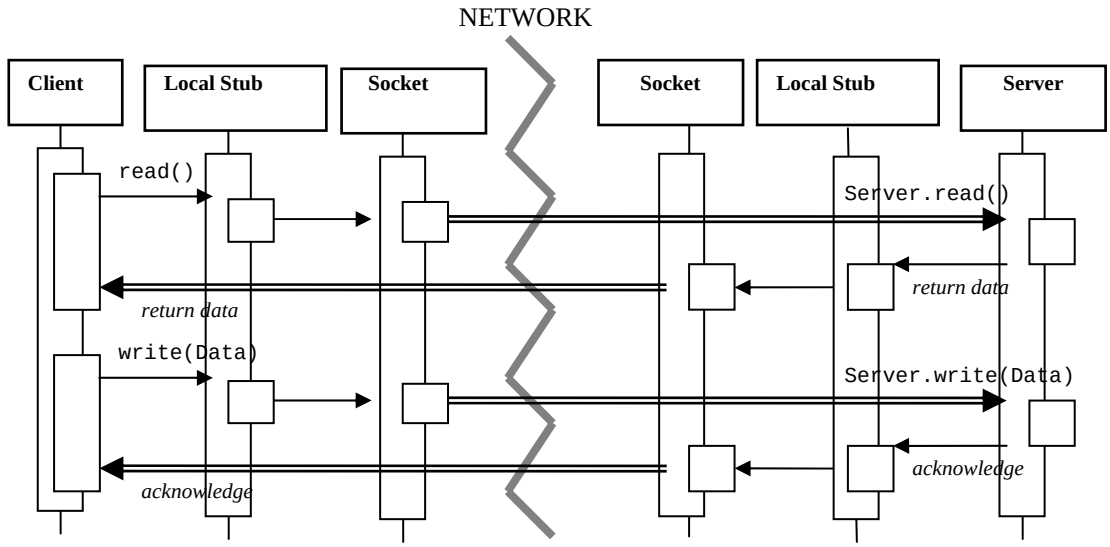

Figure 16. Scenario of the Remote Rendezvous pattern.

In this scenario, the following steps are followed:

- The **Client** requests reading data from the **Server**, so it issues a read operation to its **Local Stub**. This re-directs the call to the **Server** through its correspondent **Socket**.
- The **Server** receives the request, and makes available the data requested by the **Client**, issuing a call to its **Local Stub**. This re-directs the call to the **Client** through its correspondent **Socket**. The read data is now available to the **Client**.
- On the other hand, the **Client** requests writing data to the **Server**, so it issues a write operation to its **Local Stub**. This re-directs the call to the **Server** through its correspondent **Socket**.
- The **Server** receives the request, and takes the data sent by the **Client**, issuing a call to its **Local Stub** in order to acknowledge the operation. This re-directs the call to the **Client** through its correspondent **Socket**. The written data is now on the **Server**.

Implementation

The implementation is mainly based on the creation of the Local Stub components. Each Local Stub has to be created to be locally accessible, using a synchronisation mechanism such as semaphores [Dij68, Har98] to synchronise the access, considering the P and V operations, respectively just before and after invoking the `write()`or `read()`operations. Another possibility is

the use of monitors [Hoa74, Har98], which consider the synchronisation over the very `write()`or `read()` operations.

The Sockets are network communication mechanisms common in several programming languages. Sockets are able to send data back and forth between the processors or computers of a network system.

Consequences

Benefits

- The integrity and order of the encapsulated data structure is kept by allowing only point-to-point, synchronous read/write operations between clients and server.
- The Rendezvous component is developed to keep a point-to-point, bi-directional, and synchronous communication.
- The implementation is carried out considering a distributed memory programming environment, although it can be used on a shared memory platform.

Liabilities

- The use of synchronous communications between remote server and clients slows down the performance of the whole structure, particularly if the number of clients tends to be large and located far from the server, or when the communications are very frequent. This problem can be mitigated by changing the granularity of the pieces of data which are made available in read operations and/or included to the data structure due to a write operation.
- Even though this pattern can be used on a shared memory platform, due to the number of components, it tends to make communications between server and clients complex and slow. An alternative would be to use the *Local Rendezvous* pattern instead.

Known uses

The Remote Rendezvous is normally used when the parallel solution of a problem is developed using the Manager-Workers architectural pattern [OR98, Ort04] or the Shared Resource pattern [OR98, Ort03] within a distributed memory parallel platform. Hence, it has as many known uses as these architectural patterns. Particularly, the following known uses are relevant:

- The Remote Rendezvous pattern is used when implementing a Manager-Workers activity parallelism program that solves the N-Queens problem for a distributed memory system [Har98].
- The Remote Rendezvous pattern is used when using the Manager-Workers pattern for solving the Polygon Overlay problem [Ort04].
- The Remote Rendezvous pattern is used in the JavaSpaces system that acts like a Shared Resource on a distributed environment, allowing reading and writing operations over the virtual space [Ort03].

Related patterns

The *Remote Rendezvous* pattern is directly related with any parallel software system developed on a distributed memory environment from the Manager-Workers pattern [OR98, Ort04] or the Shared Resource pattern [OR98, Ort03].

5. Summary

The goal of the present work is to provide software designers and engineers with an overview of the common structures used for parallel software systems, and provide a guidelines on the selection of architectural patterns during the initial design stages of parallel software applications. However, as a first attempt at the creation of a more organised pattern system for parallel programming it is not complete or detailed enough to consider every issue of parallel programming. The patterns described

here can be linked with other current pattern developments for concurrent, parallel and distributed systems. Work on patterns that support the design and implementation of such systems has been addressed previously by several authors [POSA00].

References

[And91] Gregory R. Andrews. *Concurrent Programming. Principles and Practice.* The Benjamin/Cummings Publishing Company, 1991.

[And00] Gregory R. Andrews. *Multithread, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[Bri78] P. Brinch-Hansen. *Distributed Processes: A Concurrent Programming Concept.* Communications of the ACM, Vol. 21, No. 11, November 1978.

[CG90] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs. A First Course.* The MIT Press, 1990.

[CSG97] David Culler, Jaswinder Pal Singh and Anoop Gupta. *Parallel Computer Architecture. A Hardware/Software Approach (Preliminary draft).* Morgan Kaufmann Publishers, 1997

[Dij68] Edsger W. Dijkstra. *Co-operating Sequential Processes.* Programming Languages, Academic Press, 1968.

[DW96] S. Dobson and C.P. Wadsworth. *Towards a Theory of Shared Data in Distributed Systems.* In Software Engineering for Parallel and Distributed Systems, Chapman & Hall, 1996.

[Fos94] Ian Foster. *Designing and Building Parallel Programs, Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Publishing Company, 1994.

[Fow97] Martin Fowler. *UML Distilled.* Addison-Wesley Longman Inc., 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Systems.* Addison-Wesley, Reading, MA, 1994.

[Har98] Stephen J. Hartley. *Concurrent Programming. The Java Programming Language.* Oxford University Press, 1998.

[Hoa74] C.A.R. Hoare. *Monitors: An Operating System Strucutring Concept.* Communications of the ACM, Vol. 17, No. 10, October 1974.

[Hoa78] C.A.R. Hoare. *Communicating Sequential Processes.* Communications of the ACM, Vol. 21, No. 8, August 1978.

[KSS96] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads.* SunSoft Press, 1996.

[McK95] Paul E. McKenney. *Selecting Locking Primitives for Parallel Programs.* In *Patterns Languages of Programming 2* (PLoP'95). Addison-Wesley, 1996.

[OR98] Jorge L. Ortega-Arjona and Graham Roberts. *Architectural Patterns for Parallel Programming.* Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing, EuroPLoP'98. 9-12 july, 1998. Irsee, Germany.

[Ort00] Jorge L. Ortega-Arjona. *The Communicating Sequential Elements Pattern. An Architectural Pattern for Domain Parallelism.* Proceedings of the 7th Conference on Pattern Languages of Programming, PLoP 2000. 13-15 august, 2000. Allerton Park, Illinois, USA..

[Ort03] Jorge L. Ortega-Arjona. *The Shared Resource Pattern. An Activity Parallelism Architectural Pattern for Parallel Programming.* Proceedings of the 10th Conference on Pattern Languages of Programming, PLoP 2003. 8-12 september, 2003. Allerton Park, Illinois, USA.

[Ort04] Jorge L. Ortega-Arjona. *The Manager-Workers Pattern. An Activity Parallelism Architectural Pattern for Parallel Programming.* Proceedings of the 9th European Conference on Pattern Languages of Programming and Computing, EuroPLoP 2004. 7-11 july, 2004. Irsee, Germany.

[Ort05] Jorge L. Ortega-Arjona. *The Parallel Pipes and Filters Pattern. A Functional Parallelism Architectural Pattern for Parallel Programming.* Proceedings of the 10th European Conference On Pattern Languages of Programming and Computing, EuroPLoP 2005. 6-10 july, 2005. Irsee, Germany.

[Ort07] Jorge L. Ortega-Arjona. *The Parallel Layers Pattern. A Functional Parallelism Architectural Pattern for Parallel Programming.* Accepted to the 6th Latinamerican Conference On Pattern Languages of Programming and Computing, SugarLoafPLoP 2007. 27-30 may, 2007. Porto de Galinhas, Brazil.

[Pan96] Cherri M. Pancake. *Is Parallelism for You?* Oregon State University. Originally published in Computational Science and Engineering, Vol. 3, No. 2. Summer, 1996.

[PB90] Cherri M. Pancake and Donna Bergmark. *Do Parallel Languages Respond to the Needs of Scientific Programmers?* Computer magazine, IEEE Computer Society. December 1990.

[POSA96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal. *Pattern-Oriented Software Architecture.* John Wiley & Sons, Ltd., 1996.

[POSA00] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. *Pattern-Oriented Software Architecture. Patterns for Concurrent and Netwroked Objects.* John Wiley & Sons, Ltd., 2000.