The Shared Resource Pattern

An Activity Parallelism Architectural Pattern for Parallel Programming

Jorge L. Ortega-Arjona
Departamento de Matemáticas, Facultad de Ciencias, UNAM
México, D.F. 01000, México
jloa@fciencias.unam.mx

Abstract

The *Shared Resource* pattern is an architectural pattern for parallel programming used when a design problem can be understood in terms of activity parallelism. This pattern proposes a solution in which different operations are performed simultaneously by *sharers* on different pieces of data contained in a *shared resource*. Operations carried out by each sharer are independent of operations by other sharers.

1. Introduction

Parallel processing is the division of a problem, presented as a data structure 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. The main advantage of parallel processing is its ability to handle tasks of a scale that would be unrealistic or not cost-effective for other systems [CG88, Fos94, ST96, Pan96]. The power of parallelism centres on partitioning a big problem in order to deal with complexity. Partitioning is necessary to divide such a big problem into smaller sub-problems that are more easily understood, and may be worked on separately, on a more "comfortable" level. Partitioning is especially important for parallel processing, because it enables software components to be not only created separately but also executed simultaneously.

Requirements of order of data and operations dictate the way in which a parallel computation has to be performed, and therefore, impact on the software design [OR98]. Depending on how the order of data and operations are present in the problem description, it is possible to consider that most parallel applications fall into one of three forms of parallelism: *functional parallelism*, *domain parallelism*, and *activity parallelism* [OR98].

2. The Shared Resource Pattern

The Shared Resource pattern is a specialization of the Blackboard pattern [POSA96], *lacking a control component and introducing aspects of activity parallelism. In the Shared Resource pattern, computations could be performed without a prescribed order on ordered data. Commonly, components perform different computations on different data pieces simultaneously* [OR98].

Activity parallelism is the form of parallelism that involves problems that apply independent computations (as sets of non-deterministic transformations and perhaps repeatedly) on values of a data structure. Activity parallelism can be considered between the extremes of allowing all data to be absorbed by the components (as in domain parallelism) or all processes to be divided into components

(as in functional parallelism) [CG88, OR98, Pan96]. Many components share access to pieces of a data structure. As each component performs independent computations, communication between processing components is often not required. However, the amount of communication is not zero: communication is still required between each processing component and a component that controls the access to the data structure [OR98].

Example: A Token Space

Consider the case of a *token space* [Gray99]. In its simplest form, a token space is merely a passive storage structure for *tokens*, placed there by active processes named *clients*. A token may be a specialised data structure, a list, a data tuple, or any data type defined via inheritance from some base token class. Particularly, in this example a token is considered as a data tuple whose first element is a typed field and whose other elements are name-value pairs, each one referred as a *token item*. Moreover, a token may have one or more token items that contain identification information. The objective is that one or more token items will contain data that are being transferred between parallel clients.

The token space supports two operations: "put" and "request" [Gray99]. A "put" operation places a token in the token space, and it is capable of blocking for flow-control. If a "put" operation cannot be blocked, every data source has the potential to saturate the token space. A "request" operation can only succeed if its tokens are matched. The matching of a token from a request requires matching of each of the token items that it includes. If a request does not match, it is blocked. Requests from different processes are handled by separate threads, operating on the token space. The blocking of any one request does not affect request or put operations from other processes. A simple token space with such characteristics is illustrated in Figure 1.

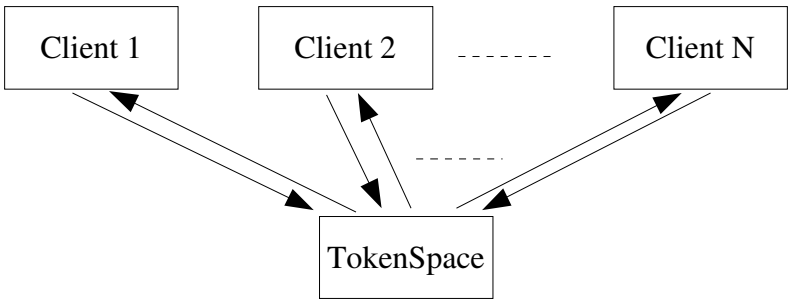

Figure 1. Overview of a simple token space.

Notice that the token space problem is more likely to be considered as an example for concurrent programming (where processes execute simulating concurrency on a single processor) rather than for parallel programming (in which processes execute simultaneously on a group of processors). However, it is simple to explain, and it could be an example of activity parallelism, if the clients would execute in parallel.

Considering the token space as a parallel computation, it should be divided and distributed among a set of processors. Clients send messages to a server running the token space. The server receives messages from the clients, organises and maintains the token space keeping its order and integrity, and sends its contents back to the clients.

Context

Start the design of a software program for performing a parallel computation, using a particular programming language for a certain parallel hardware. Such a computation involves tasks of a scale that would be unrealistic or not cost-effective for single processor systems to handle. The hardware platform or machine to be used is given, offering a reasonably good fit to the parallelism found in the computation. The main objective is to execute the tasks in the most time-efficient way.

Problem

It is necessary to apply a computation on elements of a common centralised data structure. Such a computation is carried out by several sequential processes executing simultaneously. The data structure is concurrently shared among the processes. The details of how the data structure is constructed and maintained are irrelevant to the processes. All the processes know is that they can *send* and *receive* data through the data structure. The integrity of the internal representation, considered as the consistency and preservation of the data structure, is important. However, the order of operations on the data is not a central issue. Generally, performance as execution time is the feature of interest.

For instance, consider the Token Space example. The whole process is based on allowing clients to simultaneously operate, putting or requesting tokens to the token space when needed. Parallelism results from the fact that client processes that have satisfied all their needs for data can then continue concurrently. The processes synchronise activities as necessary by waiting for others to place tokens in the token space. The integrity of the internal representation of the tokens and the token items is important for obtaining a final result after the computation is carried out, but the order of operations on the tokens or token items is not pre-determined.

Forces

Considering the problem description and granularity and load balance as other elements of parallel design [Fos94, CT92] the following forces should be considered:

- The integrity of the data structure must be preserved. This integrity provides the base for result interpretation. For example, in the token space example, it is important to control where and when a token is requested or put, by synchronising these operations for such a token. This allows preserving the overall order and integrity of the token space, so the final state of the token space is considered as the result of the whole computation.
- Each process performs simultaneously and independently a computation on different pieces of data. The objective is to obtain the best possible benefit from activity parallelism. In the token space example, clients indicate their interest in a token. This is the only occasion in which they may

interact with other clients, via the token space. During the rest of the execution time, clients are able to operate independently from the others, using the data of the token.

- Every process may perform different operations, in number and complexity. However, no specific order of data access by processing elements is defined. In the token space example, clients are not restricted to perform the same operation (in fact, performing the same operation is considered as a variation of this pattern). Normally, clients operate or use the information contained in the tokens in different ways. Moreover, as clients execute independently from each other, there is no precise or defined order in which they request or put tokens in the token space.
- Improvement in performance is achieved when execution time decreases. Our main objective is to carry out the computation in the most time-efficient way.

Solution

Introduce parallelism as multiple participating sequential components. Each component executes simultaneously, capable of performing different and independent operations. It also accesses the data structure when needed via a shared resource component, which maintains the integrity of the data structure by defining the synchronising operations that the sequential components can do. Parallelism is almost complete among components: any component can be performing different operations on a different piece of data at the same time, without a prescribed order. Communication can be achieved only as function calls to require data from the shared resource. Components communicate exclusively through the shared resource, by each one indicating its interest in a certain data. The shared resource should provide such data immediately if no other component is accessing it. Data consistency and preservation are tasks of the shared resource. The integrity of the internal representation of data is important, but the order of operations on it is not a central issue. The main restriction is that no piece of data is accessed at the same time by different components. The goal is to make sure that an operation carried out by one sharer component goes on without interference from other sharer components. The Shared Resource pattern is an activity parallel variation of the *Blackboard* pattern [POSA96] without a control instance that triggers the execution of *sources* (the concurrent components of the Blackboard pattern). An important feature is that the execution does not follow a precise order of computations [Shaw95, Pan96].

Structure

In this architectural pattern, the different operations are applied in effect simultaneously to different pieces of data by *sharer* components. Operations in each sharer component are independent of operations in other components. The structure of the solution involves a shared resource that controls the access of different sharer components to the central data structure. Usually, the shared resource component and several different sharer components simultaneously exist and operate during execution time. Therefore, the solution is presented as a centralised network, with the shared resource as the central common component. An Object Diagram, representing the network of elements that follows the shared resource structure, is shown in Figure 2.

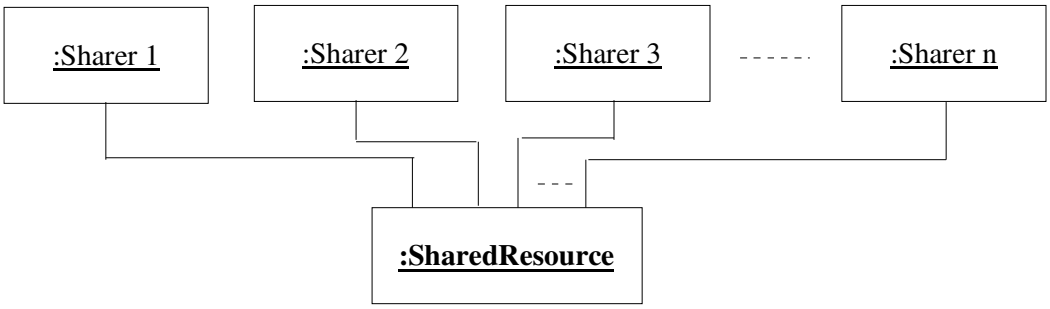

Figure 2. Object Diagram of the Shared Resource pattern.

Participants

- **Shared Resource**. The responsibility of a shared resource is to co-ordinate the access of sharer components, preserving the integrity of data. In the token space example, the token space acts as a shared resource, containing the data structure and defining the operations needed for maintaining and preserving the integrity of the data structure. Such operations are defined to control the request and put operations performed on the token space by the clients.
- **Sharer components**. The responsibilities of a sharer component are to perform its independent computation until requiring data from the shared resource. Then, the sharer component has to cope with any access restriction imposed by the shared resource. Since their computations are independent, all sharer components are able to execute in parallel. In the token space problem, clients act as sharer elements that execute in parallel until they request or put tokens contained in the token space. Once satisfied, clients continue their computations independently.

Dynamics

A typical scenario to describe the basic run-time behaviour of this pattern is described, where all participants (shared resource and sharer components) are active at the same time. This scenario is based on the Token Space example. As it is shown later in the Implementation section, the example program includes data generation, sorting of subsets of the data, merging of sorted subsets of data , and a final reporting element that uses the sorted data. The classes **`Source`**, **`Sorter`**, **`Merger`**, and **`Reporter`** respectively provide each one of these functionalities. However, by now the scenario presented here only considers an instance of each of these classes. A more detailed description of how they really interact to perform a merge-sort computation is presented in the Example Resolved section.

Notice that an instance of the **`Source`**, **`Sorter`**, **`Merger`**, and **`Reporter`** classes behaves as a sharer, performing different operations, and requiring the Token Space (as shared resource) for data tokens. If a data token is not available, the sharer can request another data token. As soon as a data token is made available from the Token Space, the requesting sharer continues its computations. Communications between sharers are normally not allowed. The Token Space is the only common component among the sharers. The processing and communicating scenario is as follows (Figure 3):

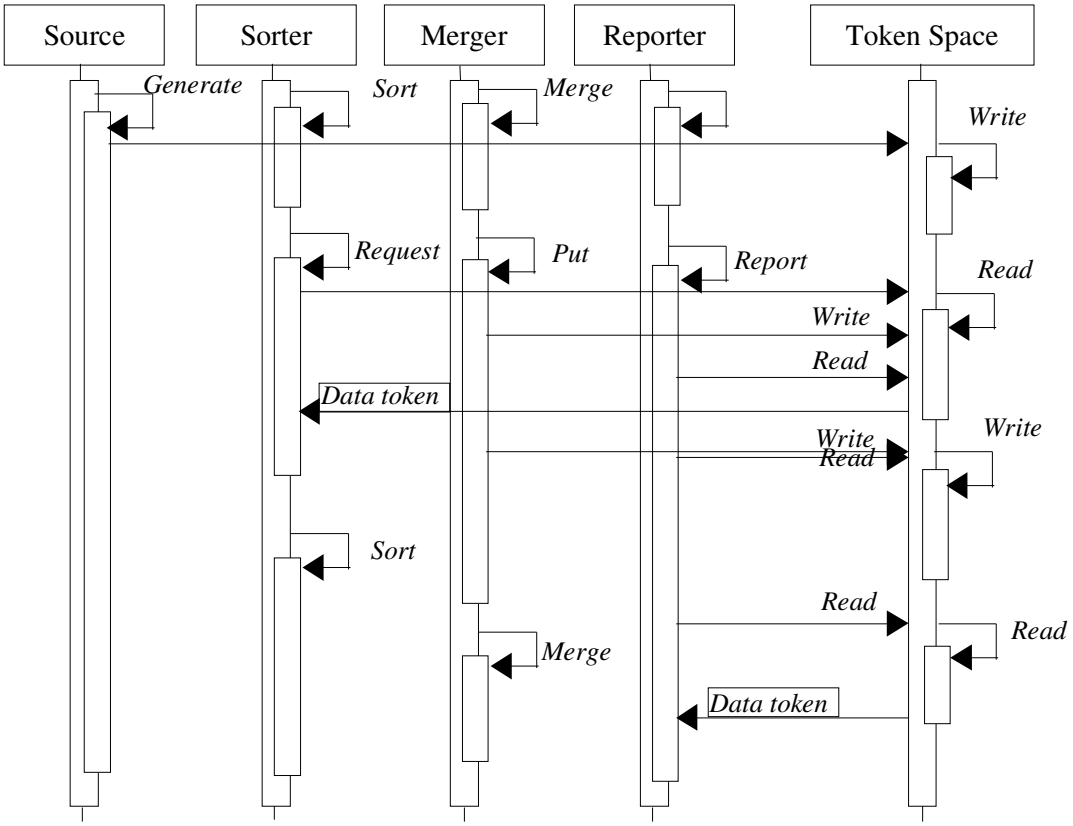

Figure 3. An Interaction Diagram of the Token Space example.

- For this scenario, let us consider a simple **Token Space** which is able to perform a couple of actions, *Read* and *Write*, in order to respectively allow reading or writing data tokens. Each sharer starts processing, performing different, independent operations, and requesting the **Token Space** to execute read or write operations.
- Consider the very basic operation: a **`Source`** object, named **Source**, generates a data token by performing the *Generate* operation, requesting a *Write* operation of the data token to the **Token Space**. If no other sharer component interferes, the **Token Space** is able to immediately serve the request from **Source**, writing the new data token.
- Things become more complex when one sharer component is reading or writing a data token of the **Token Space**, and another sharer component requires to read or write the same data token. Consider, for example, that **Sorter**, (an instance of the class **`Sorter`**) is performing a *Request* operation which requires a *Read* operation of a particular data token to the **Token Space**. If while the **Token Space** is serving this operation, one or more other sharer components (in this scenario, **Merger** or **Reporter**, which are instances of the classes **`Merger`**, and **`Reporter`** respectively) issue calls to the **Token Space** for a *Read* or *Write* operation of the same data token, the **Token Space** should be able to continue until completion of its actual operation, deferring the calls for later execution, or even ignoring them. If this is the case, any sharer component should be able to re-issue its call, requesting for an operation on the same or other data token until it is carried out.

- Another complex situation that may arise is if two or more sharer components issue calls requesting the same data piece to the **Token Space** at precisely the same time. Consider, for example, the previous situation in the scenario: as the **Merger** and **Reporter** calls could not be serviced by the **Token Space**, they have to re-issue their calls, doing it at the very same time. In this particular case, the **Token Space** should be able to resolve the situation by servicing one call (in this scenario, the *Write* request from **Merger**), and deferring or ignoring all other requests for the same data piece for later (as it is the case of the Read operation from **Reporter**). Again, the sharer components whose calls were deferred or ignored, should be able to re-issue them, contesting again for the data piece serviced by the **Token Space**.

Implementation

An architectural exploratory approach to design is described below, in which hardware-independent features are considered early, and hardware-specific issues are delayed in the implementation process [Fos94]. This method structures the implementation process of parallel software based on four stages [OR98]. During the first two stages, attention is focused on concurrency and scalability characteristics. In the last two stages, attention is aimed to shift locality and other performance-related issues. Nevertheless, it is preferred to present each stage as general considerations for design instead of providing details about precise implementation. These implementation details are pointed more precisely in the form of references to design patterns for concurrent, parallel, and distributed systems of several other authors [Sch95, Sch98a, Sch98b, POSA00].

1. *Partitioning.* The computation to be performed can be viewed as the effect of different independent computations on the data structure. Each sharer component is defined to perform an independent computation on data from the shared resource. Sharer components can be executed simultaneously due to their independent processing nature. However, the shared resource implementation should reflect a division and integrity criteria of the data structure, following the basic assumption that no piece of data is operated at the same time by two or more different sharer components. Therefore, sharer components may be implemented by a single entity (for instance, a process, a task, and object, etc.) that performs a defined computation, or a sub-system of entities. Design patterns in general [GHJV95, POSA96, PLoP94, PLoP95] may help with the implementation of the sharer components as sub-system entities. Also, patterns used in concurrent programming like the *Object group* pattern [Maf96], the *Active Object* pattern [LS95, POSA00], and *Categorize Objects for Concurrency* pattern [AEM95] can help to define and implement sharer components.

2. *Communication.* The communication to co-ordinate the interaction of sharer components and shared resource is represented by an appropriate communication interface that allows access to the shared resource. This interface should reflect the form in which requests are issued to the shared resource, and the format and size of the data as argument or return value. In general, an asynchronous coordination schema is used, due to the heterogeneous behaviour of sharer components whose requests can be deferred or ignored by the shared resource. The implementation of a flexible interface between sharer components and shared resource can be done using design patterns for communication, like the *Service Configurator* pattern [JS96], the

Composite Messages pattern [SC95], and the *Compatible Heterogeneous Agents* and *Communication between Agents* patterns [ABM96]. Other design patterns, like the *Double-Checked Locking* pattern [SH96, POSA00], the *Thread-Specific Storage* pattern [HS97, POSA00] and patterns presented dealing with issues about safe use of threads, synchronisation and locks [McKe95, POSA00], can provide help to implement the expected behaviour of the shared resource component.

3. *Agglomeration.* The components and communication structures defined in the first two stages of a design are evaluated and compared with the performance requirements. If necessary, operations can be recombined and reassigned to create different sets of sharer components with different granularity and load-balance. Usually, due to the independent nature of the sharer components, it is difficult to achieve good performance initially, but at the same time, it is easy to make changes on the sharer components without affecting the whole structure. A conjecture-test approach can be used intensively, modifying both granularity and load-balance among sharer components to observe which combination can be used to improve performance. However, special care should be taken with the load-balance between sharer components and a shared resource. The operations of the shared resource should be lighter then any sharer computation, to allow a fast response of the shared resource to requests. Most of the computation activity is meant to be performed by the sharer components.

4. *Mapping*. In the best case, trying to maximize processor utilization and minimize communication costs, each component should be assigned to a different processor. As the number of components is usually expected to be not too large, enough parallel processors can be commonly available. Also, the independent nature of sharers allows for each sharer component to be executed on a different processor. The shared resource also is expected to be executed on a single processor, and all sharers should have communication access to it. However, if the number of processors is limited and less than the number of components, it tends to be difficult and complex to load-balance the whole structure. To solve this, mapping can be determined at run-time by load-balancing algorithms. As a "rule of thumb", systems based on the Shared Resource pattern are very difficult to implement for a SIMD (single-instruction, multiple-data) computer. However, when executed on a MIMD (multiple-instruction, multiple-data) computer, systems based on the Shared Resource pattern tend to have an acceptable performance [Pan96, Pfis95].

Example Resolved

A version of the token space that incorporates mechanisms for process creation has been implemented as a Java class, named class **`TokenSpace`** [Gray99, CN01]. In particular, this version uses threads rather than parallel processes. In the time when this class was developed, in most standard Java runtime systems, the thread packages were unable to use multiple processors, so the token space system of this example is simply a demonstration in which concurrency is simulated. Furthermore, in such a threaded example there is a further simplification: there is no need for a thread in the shared resource itself; the **`put()`** and **`request()`** functions are executed by the threads that simulate the quasi-parallel processes [Gray99].

In this example, an instance of the class **Token** contains a name string and a collection of token items. Client processes use instances of a class **Request** to retrieve required tokens. A **Request** instance contains vectors specifying the required tokens, and their dispositions. Also, a **Request** instance may specify a "termination token". After a failed attempt to match a request for tokens, the matching checks for any specified termination token. Such a token is normally left in the **TokenSpace**. Its presence may affect the operation of many other processes, allowing a process (like, for instance, a data source) to mark the end of data with a token.

A simple parallel sorting program is used to test the **TokenSpace** implementation, which controls the instantiation of processes (more likely, threads) and sequences the phases of a computation. The program includes data generation (a single instance of a class **Source**), sorting of subsets of the data (one or more instances of a class **Sorter**), merging of sorted subsets of data (one or more instances of a class **Merger**), and a final reporting element that uses the sorted data (an instance of a class **Reporter**). Notice that the computation is comparable to a pipeline processing. Nevertheless, it is considered that decomposing a sorting task into several smaller sorting and merging tasks will have a large enhancement for a an $O(N^2)$ sort, and a slight enhancement for a more realistic $O(N\log N)$ sort [Gray99]. Distributing subtasks does add to the computational cost, but if multiprocessors are available, many of the separate sort and merge steps can proceed in parallel, resulting in a shorter elapsed time which is the main interest here, as it is mentioned in the context.

Partitioning

Partitioning refers to defining the computations to be performed on the data contained in the shared resource. In the **TokenSpace** example, a typical client (as a thread) has a **run()** function that may initially submit a number of requests for special initialization tokens. Then, it loops processing further data tokens until some termination condition is met. The **run()** function must end with a call to the **TokenSpace**, notifying the termination of this thread. This allows the record of threads to be maintained correctly. The data identifying a class include information on any token that should be added to the **TokenSpace** when the last instance of a client class is removed. Such tokens can mark the completion of particular phases in a computation and can also trigger the instantiation of objects that will perform a subsequent phase. As it is mentioned above, the parallel sorting example considers four types of clients: a class **Source** for data generation, a class **Sorter** for sorting subsets of the data, a class **Merger** for merging sorted subsets of data, and a class **Reporter** as a reporting element.

Communication

The communication is represented by a communication interface that allows access to the shared resource. In the **TokenSpace** implementation in Java, the access to the token space is based on the modifier **synchronized**. When applied to a method, this modifier causes that such a method only can be invoked when there is no lock held on the **TokenSpace**. If `TokenSpace` is locked, the client is temporary halted till the `TokenSpace` is unlocked. So, **TokenSpace** is locked by the invocation of a **synchronized** method, and unlocked when the method is exited. Additionally, in this implementation, the placement of a token in the **TokenSpace** triggers a check against a table of data that relates token names to the Java classes that may need to be instantiated.

Agglomeration and Mapping

The main process starts and initiates processing. After creating the **`TokenSpace`** object, it declares data structures that must be instantiated to handle them. In the present example, the class **`Source`** handles a **`StartToken`** (only a single instance of this class can be created), the class **`Sorter`** handles **`sort`** tokens (there can be as many instances of this class as seem useful), the class **`Merger`** handles **`merge`** tokens (again, there can be more than one instance of this class), and the class **`Reporter`** responds to the token marking the end of the merging process. Also, an **`endData`** token should be considered, so it marks the end of data processing in the **`TokenSpace`**. Figure 4 shows a test program for the token space example [Gray99].

```
public class Test{
     public static void main (String[] args) {
             TokenSpace tSpace = new TokenSpace();

             TokenHandlerIdentifier thi = new
             TokenHandlerIdentifier(
                     "Source",
                     "StartToken",
                     "sort",
                     "endData",
                     TokenHandlerIdentifier.SINGLETON_HANDLER );
             tSpace.addTokenHandlerInfo(thi);

             thi = new TokenHandlerIdentifier(
                     "Sorter",
                     "sort",
                     "merge",
                     "endSort",
                     TokenHandlerIdentifier.VAR_LOAD_HANDLER);
             tSpace.addTokenHandlerInfo(thi);

             thi = new TokenHandlerIdentifier(
                     "Merger",
                     "merge",
                     "merge",
                     "endMerge",
                     TokenHandlerIdentifier.VAR_LOAD_HANDLER);
             tSpace.addTokenHandlerInfo(thi);

             thi = new TokenHandlerIdentifier(
                     "Reporter",
                     "endMerge",
                     null,
                     "endReport",
                     TokenHandlerIdentifier.SINGLETON_HANDLER);
             tSpace.addTokenHandlerInfo(thi);

             Token t = new Token();
             t.fTokenName = "StartToken";
             t.fItems = null;

             tSpace.put(t,false);
     }
}
```

Figure 4. Class **`Test`** for testing the TokenSpace Example.

A more detailed operation of this program is described as follows:

1. The action of placing a **StartToken** in the **TokenSpace** triggers the creation of a **Source** object with associated thread (or **Source** process). The main thread can now terminate leaving the **TokenSpace** object in existence with running **Source** objects.
2. Each **Source** takes a very large array of randomly ordered doubles, and partitions it into subarrays; each subarray forms the **token_item** of a separate **sort** token placed into the **TokenSpace**. Flow control limits each **Source** from leaving more than ten unprocessed **sort** tokens in the **TokenSpace**. Each **put()** action on the **TokenSpace** results in a re-evaluation of the state of known processes against the data provided in the **TokenHandlerIdentifiers**. The first appearance of a **sort** token in the **TokenSpace** triggers the creation of a **Sorter**; as this class is marked as a **VAR_LOAD_HANDLER** ("variable load handler"), further instances of the class **Sorter** may get created in response to subsequent **put(sort)** actions.
3. The function **Sorter.run()** builds a **Request** object that specifies the need for a **sort** token (this requires no identification or other **token_items**), or the alternative of an **endData** termination token. This request is repeatedly reissued from a loop; if a **sort** token is returned, its subarray is sorted and placed back in the **TokenSpace** as a **merge** token. The loop ends if this termination token is matched.
4. The placement of a **merge** token triggers the creation of a **Merger**. The **Merge.run()** function is similar to that of the **Sorter**, save that its **Request** object involves two **merge** tokens, or an **endSort** termination token. The **Merger** combines the data in the two **merge** tokens that it removes from the **TokenSpace**, and puts back another **merge** token containing an array with their combined data.
5. A **Reporter** object is created when an **endMerge** token appears in the **TokenSpace**. It removes the last remaining **merge** token from the **TokenSpace**. This token contains all elements of the original array (partitioned by the **Source**) and outputs the sorted array or performs any other processing required.

Testing this program on an uniprocesor computer (for a particular size of the data set) employed one or two **Sorters** and a **Merger** as "parallel" (concurrent) objects. In general, measured computation times were just a little longer than using a simple quicksort of the entire data set. These increased times reflect the cost of the more elaborated data ordering (the creation of the various dynamically allocated tokens and subarrays) and the overheads of switching amongst threads.

Known uses

- A Tuple space, used to contain data, presents the parallel programming structure of the Shared Resource pattern. Sharers can generate asynchronous requests to read, remove and add tuples. The tuple space is encapsulated in a single shared resource component that maintains the set of tuples, preventing two parallel sharers from acting simultaneously on the same tuple [Fos94].

- JavaSpaces is a distributed object-sharing structure, constituted as a set of abstractions for distributed programming, which together compose a shared resource structure. In a distributed application, the JavaSpaces structure acts as a virtual space between providers and requesters of network resources or objects, allowing participants in a distributed solution to exchange tasks,

requests and information in the form of Java technology-based objects. Briefly, a JavaSpace is an environment that provides object persistence and facilitates the design of distributed algorithms. Basically, JavaSpaces are client/server systems, with clients calling one set of interfaces - those of the JavaSpace. Clients are encapsulated from details of object-transfer and distributed-function calls. Clients may write and read objects to JavaSpaces and look up the JavaSpace for objects that match some template. JavaSpaces provide developers with the ability to create and store objects with persistence, which allows for process integrity. For a more detailed technical overview of JavaSpaces, refer to [FHA99].

- Mobile robotics control is another concurrent application example of the Shared Resource pattern. The software functions for a mobile robotics system has to deal with external sensors for acquiring input and actuators for controlling its motion and planning its future path in real-time. Unpredictable events may demand a rapid response, for example, imperfect sensor input, power failures, and mechanical limitations in the motion. As an example, the CODGER system uses the Shared Resource pattern to model the cooperation of tasks for coordination and resolution of uncertain situations in a flexible form. CODGER is composed of a "captain", a "map navigator", a "lookout", a "pilot" and a perception system, each one sharing information through a common shared resource [SG96].

- A real-time scheduler is another concurrent application of the Shared Resource pattern. The application is a process control system, in which a number of independent processes are executed, each having its own real-time requirements, and therefore, no process can make assumptions about the relative speed of other processes. Conceptually, they are regarded as different concurrent processes coordinated by a real-time scheduler, accessing, for instance, computer resources (Consoles, printers, I/O devices, etc.) which are shared among them. The real-time scheduler is implemented as a shared resource component to give processes exclusive access to a computer resource, but does not perform any operation on the resource itself. Each different process performs its activities, requiring from time to time the use of computer resources. The shared resource grants the use of resources, maintaining the integrity of the data read from or written to a resource by each different process [Han77].

Consequences

Benefits

- Integrity of data structure within the shared resource is preserved.

- From the perspective of a parallel designer, this pattern is the "simplest" to design and execute, due to the minimal dependence between sharer components. Fundamentally, the operations on each data element are completely independent. That is, each piece of data can be operated in different machines, running independently as long as the appropriate input data are available to each one. It is relatively easy to achieve significant performance in an application that fits the pattern [Pan96].

- As its components (the shared resource and the sharers) are strictly separated, the Shared Resource pattern supports changeability and maintainability [POSA96, Pan96].

- The Shared Resource pattern supports several levels of granularity. If required, the shared resource can provide operations for different data sizes.

- As sharer components perform different and independent operations, they can be reused in different structures. The only requirement for reuse is that the sharer to be reused is able to perform certain operations on the data type in the new shared resource [POSA96, Pan96].

- A shared resource can provide fault tolerance for noise in data [POSA96, SG96].

Liabilities

- Due to the different nature of each component, load-balance is difficult to achieve, even when executing each component on a different processor. The difficulty increases if several components run together on a processor [Pan96].

- The trace of stages for producing a result in a shared resource application is difficult to reproduce. Inherently, computations are not necessarily ordered following a deterministic algorithm [POSA96]. Furthermore, the parallelism of its components introduces a non-deterministic feature to the execution [Pan96].

- Even when parallelism is straightforward, often the shared resource does not consider the use of control strategies to exploit the parallelism of shared and to synchronise their actions. In order to preserve its integrity, the design of the shared resource must consider extra mechanisms or synchronisation constraints to access its data. An alternative is using the Blackboard pattern [POSA96].

Related patterns

The *Shared Resource* pattern is considered a specialization of the *Blackboard* pattern [POSA96] without control component, and introducing aspects of activity parallelism. Also, it is related to the *Repository* architectural style [Shaw95, SG96]. Other patterns that can be considered related to this pattern are the *Compatible Heterogeneous Agents* pattern [ABM96] and the *Object Group* pattern [Maf96].

3. Summary

The goal of the present paper is to provide software designers and engineers with an overview of a common structure used for activity parallel software systems. The architectural pattern 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 [Sch95, Sch98a, Sch98b, POSA00].

4. Acknowledgements

I wish to express my acknowledgements and gratitude to Berna Massingill, who shepherded this paper always providing insights and comments for its improvement. Also, I would like to thank Ralph Jonhnson, for his accurate comments during the shepherding of this paper.

5. References

[ABM96] Amund Aarsten, David Brugali and Giuseppe Menga. *Patterns for Cooperation.* Pattern Languages of Programming Conference (PLoP'96). Allerton Park, Illinois, USA. September 1996.

[AEM95] Aarsten, A., Gabriele Elia, G., and Giuseppe Menga, G. *G++: A Pattern Language for the Object Oriented Design of Concurrent and Distributed Information Systems, with Applications to Computer Integrated Manufacturing.* Department of Automatica e Informatica, Politecnico de Torino. In J. Coplien and D. Schmidt (eds.) Pattern Languages of Program Design. Reading, MA: Addison-Wesley, 1995.

[CG88] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs. A Guide to the Perplexed.* Yale University, Department of Computer Science, New Heaven, Connecticut. May 1988.

[CN01] Peter Carmichael and Joyce Ng. *DSpace Workflow Design Description.* DSpace Durable Digital Documents project, MIT Libraries, 2001. `http://www.dspace.org/`

[CT92] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, Inc., Boston, 1992.

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

[FHA99] Freeman, E., Hupfer, S., and Arnold K. *JavaSpaces. Principles, Patterns and Practice.* Addison Wesley Publishing Co., 1999.

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

[Gray99] Neil Gray. *Architectural Patterns for Parallel Programming.* Personal communication, 1999.

[Han77] Brinch Hansen, P. *The Architecture of Concurrent Programs.* Series in Automatic Computetion. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1977.

[HS97] Harrison, T., and Schmidt, D. C. *Thread-Specific Storage, An Object Behavioral Pattern for Efficiently Accesing per-Thread State.* Department of Computer Science, Washington University. 2nd annual European Pattern Languages of Programming Conference, in Kloster Irsee, Germany, 1997.

[JS96] Prashant Jain and Douglas C. Schmidt. *Service Configurator. A Pattern for DynamicConfiguration and Reconfiguration of Communication Services.* Third Annual Pattern Languages of Programming Conference, Allerton Park, Illinois. September 1996.

[LS95] R. Greg Lavender and Douglas C. Schmidt. *Active Object. An Object Behavioral Pattern for Concurrent Programming* .In *Patterns Languages of Programming 2* (PLOP'95). Addison-Wesley, 1996.

[Maf96] Maffeis, S. *Object Group, An Object Behavioral Pattern for Fault-Tolerance and Group Communication in Distributed Systems.* Department of Computer Science, Cornell University. Proceedings of the USENIX Conference on Object-Oriented Technologies. Toronto, Canada, 1996.

[McKe95] McKenney, P. E. *Selecting Locking Primitives for Parallel Programs.* Sequent Computer Systems, Inc. In J. Vlissides, J. Coplien and N. Kerth (eds.) Pattern Languages of Program Design 2. Reading, MA: 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. Jens Coldewey and Paul Dyson (editors), Universitätsverlag Konstantz GmbH, 1999.

[OR00] Jorge L. Ortega-Arjona. *The Communicating Sequential Elements Pattern.* Proceedings of the 7th Annual Conference on Pattern Languages of Programming, PLoP'98. Eugene Wallingford and Alejandra Garrido (editors), Washigton University Technical Report wucs-00 29, 2000.

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

[Pfis95] Gregory F. Pfister. *In Search of Clusters. The Coming Battle in Lowly Parallel Computing.* Prentice Hall Inc. 1995.

[PLoP94] James O. Coplien and Douglas C. Schmidt (editors). *Patterns Languages of Programming.* Addison-Wesley, 1995.

[PLoP95] James O. Coplien, Norman L. Kerth and John M. Vlissides (editors). *Patterns Languages of Programming 2.* Addison-Wesley, 1996.

[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, Vol. 2 - Patterns for Concurrent and Distributed Objects.* John Wiley and Sons, Ltd., 2000.

[SC95] Aamond Sane and Roy Campbell. *Composite Messages: A Structural Pattern for Communication Between Components.* OOPSLA'95, Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems. October 1995.

[Sch95] Douglas Schmidt. *Accepted Patterns Papers.* OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems. http://www.cs.wustl.edu/~schmidt/OOPSLA-95/html/papers.html. October, 1995.

[Sch98a] Douglas Schmidt. *Design Patterns for Concurrent, Parallel and Distributed Systems.* http://www.cs.wustl.edu/~schmidt/patterns-ace.html. January, 1998.

[Sch98b] Douglas Schmidt. *Other Pattern URL's. Information on Concurrent, Parallel and Distributed Patterns.* http://www.cs.wustl.edu/~schmidt/patterns-info.html. January, 1998.

[SH96] Schmidt, D. C., and Harrison, T. *Double-Checked Locking, An Object Behavioral Pattern for Initializing and Accesing Thread-safe Objects Efficiently.* Department of Computer Science, Washington University. 3rd Pattern Languages of Programming Conference, Allerton Park, Illinois, February 1997.

[Shaw95] Mary Shaw. *Patterns for Software Architectures.* Carnegie Mellon University. In J. Coplien and D. Schmidt (eds.) Pattern Languages of Program Design. Reading, MA: Addison-Wesley, 1995.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall Publishing, 1996.

[ST96] David B. Skillicorn and Domenico Talia. *Models and Languages for Parallel Computation.* Computing and Information Science, Queen's University and Universita della Calabria. October 1996.

[VBT95] Allan Vermeulen, Gabe Beged-Dov and Patrick Thompson. *The Pipeline Design Pattern.* OOPSLA'95, Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems. October 1995.