Concurrency Patterns

David Kocher
00-806-968
dk@sudo.ch

Seminar *Software Architekturen*
Institut für Informatik der Universität Zürich
Prof. H. Gall, Prof. M. Glinz, Ch. Seybold, M. Pinzger

Abstract. Developing muli-threaded applications is harder than developing sequential programs because an object can be manipulated concurrently by multiple threads.
This paper paraphrases the concurrency patterns described in [POSA2], including the *Producer-Consumer*, *Active Object*, *Monitor Object*, *Half-Sync/ Half-Async* and *Leader/ Followers* pattern.

1 Introduction

A concurrent program has multiple threads of control allowing it perform multiple computations in parallel and to control multiple external activities which occur at the same time [MaJk99].

The reasons given why concurrent programming is useful and adds a real benefit to the application's overall performance usually include:

1. Parallel taks can run faster than a sequence of them could, if they are e.g. I/O bound or gain performance from multiprocessing hardware.
2. Enhanced fairness and availability if clients need not to wait for each other's task to complete first [Lea99].
3. Incread application responsiveness; high prority thread for user requests. Putting the user interface into a separate thread from any other work makes the application *feel* more responsive to the user and ensures that an unexpectedly long operation doesn't freeze the application's screen [Shira00].
4. More appropriate structure; to model the environment in a object oriented fashion we have to control multiple activities and handle multiple events.

The hard part of the concurrency model is that the threads are usually not really independent. They must coordinate, synchronize, and share information, and shared resources must be managed carefully to avoid corruption and erroneous computation [Doug02].

The choice of concurrency architecture has a significant impact on the design and performance of multi-threaded networking middleware and applications. No single concurrency architecture is suitable for all workload conditions and hardware and software platforms [POSA2].

Concurrency as a term includes the ability to perform multiple tasks at the same time; we generally refer to that ability as parallelism. But threaded programming is about more than parallelism: it's also about simpler program design. Historically, threading was first exploited to make certain programs easier to write; if a program can be split into separate tasks, it's often easier to program the algorithm as separate tasks or threads [OakWo04].

The benefit of introducing concurrency into an application (by creating threads to perform tasks) must however outweight its costs [Lea99]. This decision is not too different from introducing other kinds of objects in object oriented programming. One has to keep in mind that increasing the number of threads can overwhelm processing times because of scheduling and context switching.

2 Design Patterns for Concurrency

2.1 Producer-Consumer

The Producer/Consumer pattern is used to decouple processes that produce and consume data at different rates. The Producer/Consumer pattern's parallel loops are broken down into two categories; those that produce data, and those that consume the data produced. Data queues are

used to communicate data between loops in the Producer/Consumer design pattern. These queues offer the advantage of data buffering between producer and consumer loops.

One of the more common patterns in threaded programming is the *Producer-Consumer* pattern. The idea is to process data asynchronously by partitioning requests among different groups of threads. The producer is a thread that generates requests to be processed. The consumer is a thread that takes those requests and acts upon them. This pattern provides a clean separation that allows for better thread design and makes development and debugging easier [OakWo04].

The producer and consumer threads are decoupled: the producer never directly calls the consumer (and vice versa). This makes it possible to interchange different producers without affecting the consumer. It also allows us to have multiple producers serviced by a single consumer, or multiple consumers servicing a single producer. More generally, we can vary the number of either based on performance needs or user requirements [OakWo04].

The producer relies on the consumer to make space in the data-area so that it may insert more information whilst at the same time, the consumer relies on the producer to insert information into the data area so that it may remove that information. It therefore follows that a mechanism is required to allow the producer and consumer to communicate so that they know when it is safe to attempt to write or read information from the data-area.

The *Producer/Consumer* pattern is commonly used when acquiring multiple sets of data to be processed in order. Suppose you want to write an application that accepts data while processing them in the order they were received. Because queuing up (producing) this data is much faster than the actual processing (consuming), the *Producer/Consumer* design pattern is best suited for this application. We could conceivably put both the producer and consumer in the same loop for this application, but the processing queue will not be able to add any additional data until the first piece of data is done processing. The *Producer/Consumer* pattern approach to this application would be to queue the data in the producer loop, and have the actual processing done in the consumer loop. This in effect will allow the consumer loop to process the data at its own pace, while allowing the producer loop to queue additional data at the same time.

2.2 Active Object

The Active Object design pattern decouples method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control [POSA2].

This pattern is used in many applications to handle multiple client requests simultaneously and thereby to improve their quality of service. This pattern is called *Active Object* referring to the fact that the concurrent object resides in its own thread of control independantly of the thread of control of the client that

invoked one of its methods. This means that the method invocation and the actual method execution occurs in two different threads. To the client however it appears to invoke an ordinary method.

At runtime the proxy cannot just pass the requests to the servant because of the requirement they both run in different threads. Instead, the proxy constructs a method request object and inserts it into an activation list. This list does not only hold the method request objects inserted by the proxy but also keeps track of which method request can execute. The method request object contains the information to execute the desired method later; i.e. the request parameters and any other context information.

Using the activation list, the threads of the proxy and the servant can run concurrently. Only the activation list must be serialized to protect it from concurrent access using a concurrency control pattern such as the *Monitor Object* [2.3].

A scheduler running in its own thread then invokes the corresponding method on the servant using the information stored in the method request object it takes from the activation list.

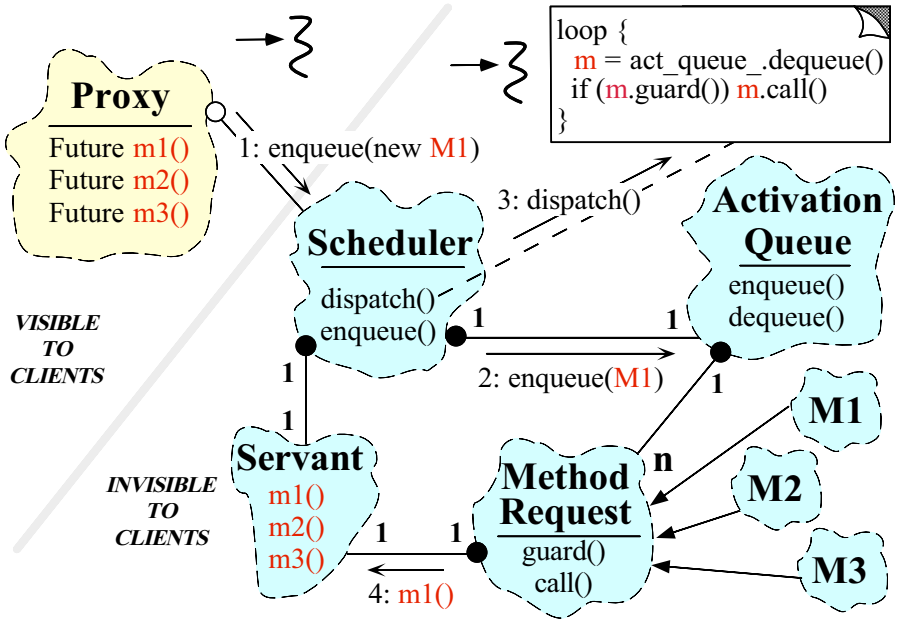

Fig. 1. Structure of the *Active Object* concurrency pattern [Schm00].

Because the proxy can't return the result of the method invocation, a placeholder is returned instead (if the method is a two-way invocation). A future is a mechanism which can be used to provide values that will be resolved to another

value asynchronously. The proxy can return a future to its caller, and the servant will compute the value that the future will resolve to in another thread of control. The future provides a way to ask for the actual value. Two situations are possible:

1. The future has resolved already. That is, the method calculating the value has finished, and stored the result in the future. Asking for the future's value will immediately give the result.
2. The future has not resolved. In this case, the current thread is blocked until the future resolves. The current thread is effectively synchronised with the thread calculating the value. But as long as the program logic does not require immediate usage of the result, the current thread is not blocked.

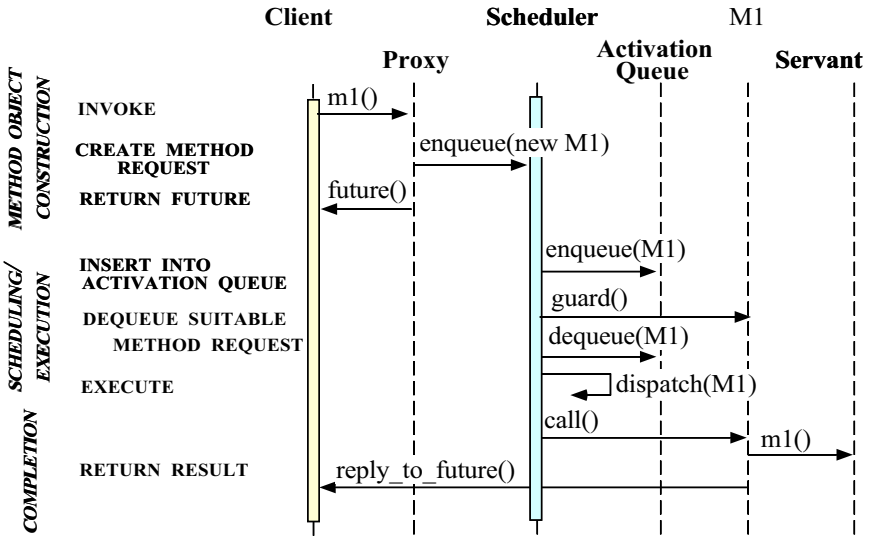

Fig. 2. Sequence diagram of the *Active Object* concurrency pattern [Schm00].

Recapitulating, the *Active Object* concurrency pattern consists of the following participants [POSA2]:

- A *proxy* that advertises the interface of an *Active Object* visible to the client threads.
- The *servant* provides the implementation of the methods defined in the proxy interface.
- A serialized *activation list* which holds the *method request objects* inserted by the proxy and enables the servant to run concurrently.

2.3 Monitor Object

> *The Monitor Object design pattern synchronizes concurrent method execution to ensure that only one method at a time runs within an object. It also allows an object's methods to cooperatively schedule their execution sequences* [POSA2].

In many applications there are multiple threads invoking methods on a object which modify its internal state. In such cases we often have to make sure to synchronize and schedule the access to these methods [POSA2].

The *Monitor Object* belongs to the group of passive objects. In contrary to an *Active Object* [2.2], the monitor does not have its own thread of control. Each method is executed in the thread of the client and access is blocked until the method returns. Also to prevent uncontrolled concurrent changes (race conditions), only one synchronized method can execute within the monitor at any time.

The *Monitor Object* solves this problem implementing the following functionality [POSA2]:

1. The object's interface should define its synchronization boundaries and only one method at a time should be active within the same object.
2. Objects should be responsible for ensuring that any of their methods that requires synchronization are serialized transparently without requriring the client to know anything about. Operations are invoked like ordinary method calls but guaranteed to be mutually exclusive. The condition synchronization is realized using wait and signal primitives (e.g. `wait()` and `notify()` in the Java language).
3. If an object's method must block during execution (e.g. to wait for a condition to become true) it should not block the thread of control. Other clients should be able to access the object in such a case to prevent a deadlock and to take advantage of concurrency mechanisms avilable on hardware.
4. When a method interrupts its thread of control voluntarily, the invariants must always hold; it must leave its object in a stable state.

To fullfill these properties, the *Monitor Object* synchronizes the access to its methods. To serialize concurrent access to an object's state, each *Monitor Object* contains a monitor lock. Synchronized methods can determine the circumstances under which they suspend and resume their execution, based on one or more monitor conditions associated with a *Monitor Object*.

Recapitulating, the *Monitor Object* concurrency pattern consists of the following participants [POSA2]:

1. The *monitor object* itself that exposes the methods to the clients known to be synchronized. Each method executes in the thread of the client, the *Monitor Object* itself does *not* have its own thread of control.
2. *Synchronized methods* of which only one can be execute at any point in time. They implement the thread-safe functions exported by a *Monitor Object's* interface.

3. The *monitor condition* in conjunction with the monitor lock determines if a synchronized method should suspend or resume its processing.

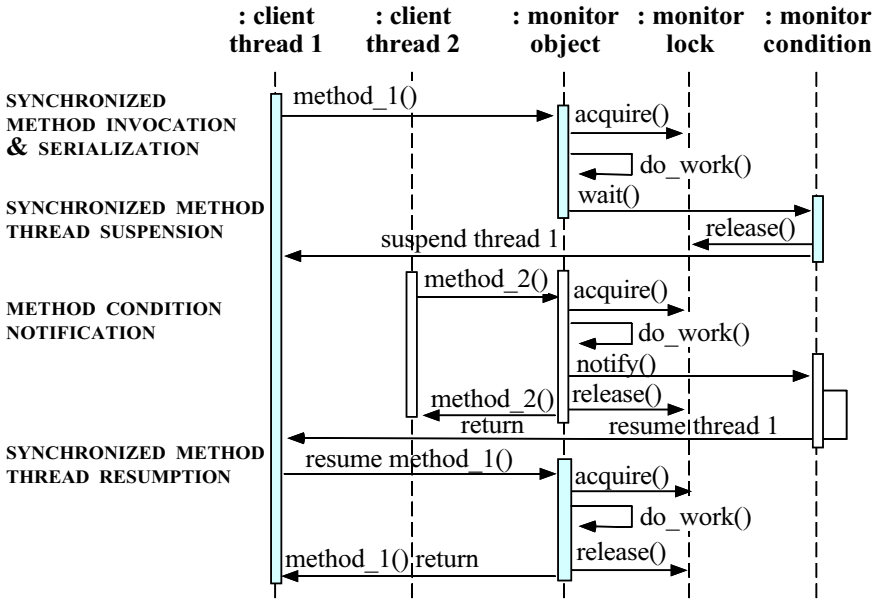

Fig. 3. Sequence diagram of the *Monitor Object* concurrency pattern [Schm00].

The `synchronized` keyword in Java Synchronization ensures that a group of statements (a `synchronized` block) will execute atomically as far as all synchronized threads are concerned. Synchronization does not address the problem of which thread executes the statements first: it is first come, first served.

The main synchronization mechanism in Java is the monitor. The Java language does not provide 'conventional' concurrency control mechanisms, such as mutexes and semaphores, to application developers.

Every object can have a monitor associated with it, so any object can synchronize blocks. Before a synchronized block can be entered, a thread needs to gain ownership of the monitor for that block. Once the thread has gained ownership of the monitor, no other thread synchronized on the same monitor can gain entry to that block (or any other block or method synchronized on the same monitor). The thread owning the monitor gets to execute all the statements in the block, and then automatically releases ownership of the monitor on exiting the block. At that point, another thread waiting to enter the block can acquire ownership of the monitor.

However, threads synchronized on different monitors can gain entry to the same block at any time. For example, a block defined with a `synchronized(this)` expression is synchronized on the monitor of the `this` object. If `this` is an object that is different for two different threads, both threads can gain ownership of their own monitor for that block, and both can execute the block at the same time. This won't matter if the block affects only variables specific to its thread (such as instance variables of this), but can lead to corrupt states if the block alters variables that are shared between the threads. [Shira00]

[Lea99] discusses the implementation of concurrency primitives such as mutexes, semaphores and readers/writers locks in Java in detail.

3 Architectural Patterns for Concurrency

3.1 Half-Sync/Half-Async

> *The Half-Sync/Half-Async architectural pattern decouples asynchronous and synchronous service processing in concurrent systems, to simplify programming without reducing performance. The pattern introduces two intercommunicating layers, one for asynchronous and one for synchronous service processing with a queuing layer between* [POSA2].

Concurrent systems often contain a mixture of both synchronous and asynchronous services. On the system level asynchronous programming is preferred to improve the overall performance. Conversely, an application developer wants to simplify its programming effort and therefore use synchronous processing to reduce complexity. The need for both simplicity and high performance is often contradictory but essential in many concurrent systems. As an example consider an operating system kernel which handles interrupts triggered by network interfaces asynchronously but a higher-level applications services uses synchronous `read()` and `write()` system calls. The challenge is to make these services intercommunicate nevertheless.

To provide such an architecture that enables the synchonous and asynchonous processing services to communicate with each other the *Half-Sync/Half-Async* pattern decomposes the services in the system into layers [POSA1], adding a queuing layer between the synchronous and asynchronous layers. Both the asynchronous and synchronous layers in the *Half-Sync/Half-Async* pattern interact with each other by passsing messages via this queuing layer. A real world example for such a queing layer is the socket layer in many UNIX derivates, serving as a buffering and notification point between the synchonous application process and the asynchonous, interrupt-driven hardware services in the kernel.

The *Half-Sync/Half-Async* pattern has the following participants in the UNIX example mentioned above:

1. Tasks in the *synchronous task layer* (e.g. application processes) perform high-level I/O operations that transfer data synchronously to message queuees in the Queuing layer. Tasks in this layer are *Active Objects* [2.2] that have their own thread of control while performing I/O operations.

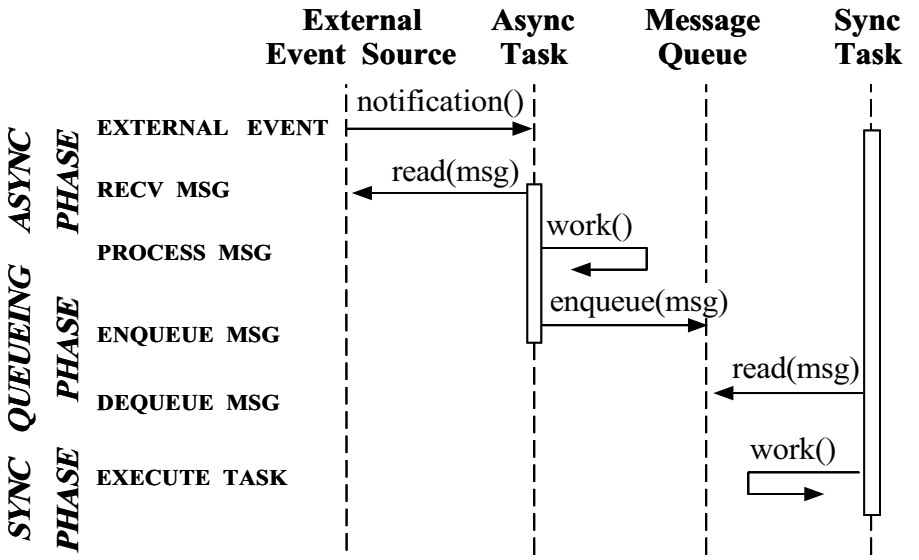

Fig. 4. Sequence diagram of the *Half-Sync/Half-Async* concurrency pattern [POSA2].

2. The *queuing layer* (e.g. Sockets) provides synchronization and buffering between the asynchronous and synchronous layers.
3. Tasks in the *asynchronous task layer* (e.g. Kernel) handle events from the exernal event sources such as network interfaces. These are *Passive Objects* [2.3] that do not have their own thread of control.

The *Half-Sync/Half-Async* has the benefit of simplified higher-level tasks and a single point for inter-layer communication the queueing layer.

3.2 Leader/Followers

> *The Leader/Followers architectural pattern provides an efficient concurrency model where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on the event sources [POSA2].*

The *Leader/Followers* pattern's aim is to provide a solution to process multiple events concurrently such as in multi-threaded server applications. In a typical server scenario a high volume events such as `CONNECT` arrive simultaneously from multiple event sources (such as multiple TCP/IP socket handles). It may not be possible (speaking of performance) to associate a separate thread with each single socket handle because of concurrency-related overhead such as context switching and synchronization issues. The system would not be scalable. A key feature of the *Leader/Followers* pattern thefore is to demultiplex the associations between threads and event sources.

The *Leader/Followers* pattern structures a pool of theads to share a set of event sources efficiently by taking turns demultiplexing events that arrive on these event sources and synchronously dispatching the events to application services that process them [POSA2]. Only one thread in the pool (the leader) is allowed to wait for an event to occur while the other threads (the followers) queue up waiting. When a thread detects an event, it first promotes a follower as the new leader and then takes the role of a processing thread dispatching the event to an application-specific event handler. It is important to note that multiple such processing threads can run concurrently but only one leader thread waits for new events.

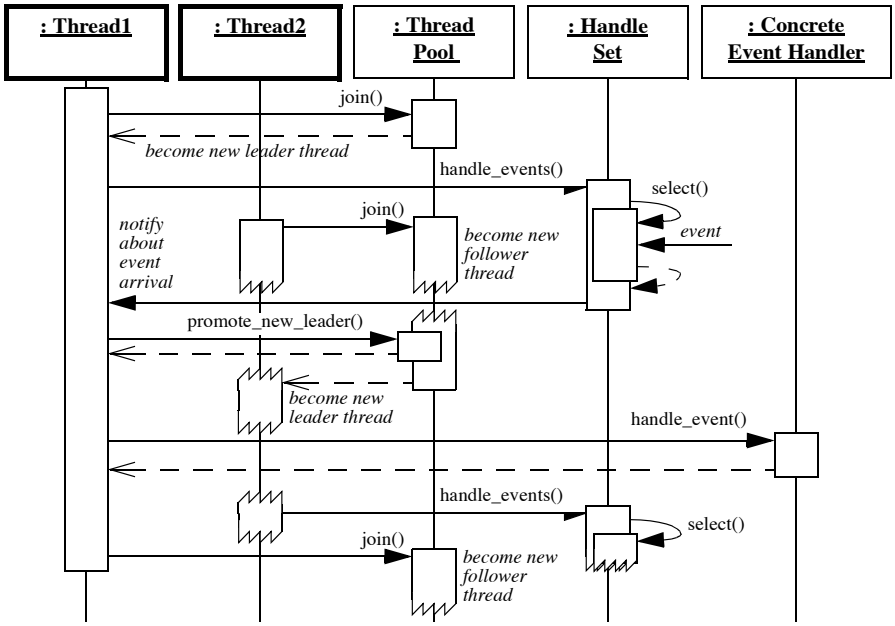

Fig. 5. Sequence diagram of the *Leader/Followers* concurrency pattern [POSA2].

References

[POSA2] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann: Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects, John Wiley & Sons, 2000

[Doug02] Bruce Powel Douglass: Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems, Addison Wesley, 2002

[Malkh04] Dahlia Malkhi: Server Architecture Models, Lecture Slides, Hebrew University, 2004

[Shira00] Jack Shirazi: Java Performance Tuning, O'Reilly, 2000

[Schm00] Douglas C. Schmidt: Monitor Object, An Object Behavioral Pattern for Concurrent Programming, Department of Computer Science, Washington University, St. Louis

[Lea99] Doug Lea: Concurrent Programming in Java: Design Principles and Patterns, Addison-Wesley, 1999.

[OakWo04] Scott Oaks, Henry Wong: Java Threads, 3rd Edition, O'Reilly, 2004

[MaJk99] Jeff Magee and Jeffrey Kramer: Concurrency: State Models & Java Programs, John Wiley, 1999.

[POSA1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, Michael Stal.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns, John Wiley & Sons.