Some Resource Management Patterns

Submission to RM Patterns Focus Group at EuroPLoP 2002

Markus Völter
voelter@acm.org

Version 0.1, May 14, 2002

This little paper suggests a set of patterns in the area of resouce management in local or distributed systems. Please note that some of the patterns might overlap with material that is already written. This paper is just a suggestion – don't consider the stuff as full-fledged patterns!

Introduction

All the patterns in this collection *save* something. This is because resources are typically considered expensive or somehow limited. Saving resources is thus always desirable. The patterns in this paper save

- programmers from having to code thread-safe resources
- expensive resources while using cheaper but worse, other resources
- memory for large volumes of data
- the resource provider from dangling resources in the face of unreliable clients.
- memory and conversion time.
- the developer from having to care about concurrency.

The Patterns

RESOURCE-PER-THREAD

safes the programmer from having to code thread-safe resources

You have a multithreaded system that needs to access resources.

* * *

Accessing a resource concurrently is not always easy, because as a consequence, the resource must be thread-safe. This means that the resource's program code has to take into account that several threads might be "in the code" at the same time.

Consider a database connection. Such a connection is an object that manages communication with the database. It forwards queries, and returns the results.

Typically, it also needs to store the results for a while, until they are used by the client. If several clients use the connection at the same time, the connection must manage queries and store results for several clients.

Therefore:

Instead of accessing *one* resource from several threads, provide each thread with a *separate* resource. Use pooled SELF-CONTAINED RESOURCES to make sure the allocation and deallocation protocol is safe from failing clients.

* * *

Note that this of course only works well if the resource does not include global, shared state. Stateless resources such as services, connections, etc. are especially well suited. One the downside, using this patterns comes with some management overhead for pooling.

* * *

Typical examples are database connection pools. Another proninent example are stateless session EJBs, they are also pooled by the container.

SWAPPING

Saves expensive resources while using cheaper but worse, other resources

You have an environment with limted resources.

* * *

Usually, when keeping data in memory (or another expensive resource) you don't need all the data all the time. The data is typically not accessed all the time, there are "idle times" when the data is not really accessed, but the state needs to be preserved.

Therefore:

Swap data that occupies scarce resources to cheapter, but less optimal resources while the data is not actually accessed. Use proxies to reload the data when it is needed again. Use algorithms such as LRU to determine which portions of the data to swap out.

* * *

Swapping of course might have performance problems: the data has to be reloaded when it is accessed after it has been passivated. Also, a thread might have to run in the background to run the LRU (or other) algorithm.

* * *

Operating systems swap memory regions to hard disks. EJB application servers passivate stateful session beans to a hard disk.

STREAM + CALLBACK

Safes memory!

You want to process large chunks of data.

* * *

When processing large chunks of data, you usually have the problem that it does not really fit into memory, and that it takes a lot of time to build the data structure in memory.

Consider an image processing application that processes high-resolution radiography data. Such images can easily have 50 to 100 megs of data. If you process one or several of these images, memory can become scarce.

Therefore:

Don't load the the whole data at once. Instead, process the data in small bits, incrementally, and use a callback-based interface to integrate with the processing application

* * *

Memory requirements are significantly lower. However, the programmer API is perhaps not as nice as it could be otherwise, you may have to code your own state machine to keep track of history-dependent data (just compare DOM to SAX…)

* * *

The SAX interface for XML parsing uses this approach. There are several pipes and filters based image processing tools.

SELF-CONTAINED RESOURCE

Safes the resource provider from dangling resources in the face of unreliable clients.

A (resource) server provides rersouces to clients.

* * *

A resource server is responsible for efficient management of the resources it serves, and for the integrity of the managed set of resources. However, often the server runs into problems if the client does not behave well (doesn't stick to a defined API) or crashes.

Consider a database connection pool. A client requests a connection from the pool, uses it, and when finished, returns it to the pool. There might be a *close()* operation that returns the connection to the pool (the resource server). However, if a client forgets to call *close()*, or crashes just before, the resource remains dangling – marked as "in use" but not really used any more.

Therefore:

Make sure that the protocol of using resources is self-contained. This means, it works correctly also in the face of non-reliable, wrong, or crashing clients.

* * *

Using this pattern makes resource server significantly more reliable – but might also incur a performance hit.

* * *

Examples for this pattern are smart pointers in C++ (which release a resource once it runs out of scope) or leases, pings or heartbeats for resource management in distributed systems.

AUTO-CONVERTING REFERENCE

Safes memory and conversion time.

You use a reference to a resource which can have several representations.

* * *

Converting a resource from one representation to another one takes a lot of time and computing power. Keeping a resource in several representations costs money. Both solutions don't work in practice.

Consider a data structure that also has an XML representation. This typically results in three different, but content-wise similar data structures: the native object representation, the DOM representation for the XML, and usually, the XML string. You should make sure that all versions are logically up to date, but that only those versions that are actually needed are really kept in memory at any time.

Therefore:

Provide a specific reference (pointer) class to reference such a data structure. Provide accessor operations for all versions of the data, and convert to this version lazily, i.e. on demand when the resource is accessed.

* * *

As an advantage, there is only always the representation that's really needed in menory. On the downside, invoking a getter operation might take some time, because in the background, a format conversion could be triggered.

* * *

I have used this pattern several times in XML based applications. I'm sure 2 others will have used it, too ✍

DELEGATED CONCURRENY

Safes the developer from having to care about concurrency.

You need to access a resource concurrently.

* * *

Managing concurrent access to a resource is not an easy task. You might have to detect deadlocks, prevent race conditions, or, generally, provide semantically correct behaviour in the face of concurrent access (thread-safety).

Consider an object that serves as a façade (an "objectified interface") for a data stucture in a database. You have to make sure that concurrent client can access the object (I have to write more here….)

Therefore:

Instead of handling concurrency in your resouce objects, keep the object layer stateless and delegate concurrency to the underlying resource (if there is one, that is!). An underlying resource might be better suited for handling the concurrency.

* * *

???

* * *

A typical example are databases. They are traditionally quite good at handling concurrency issues. You should not try to redo this – delegate to the database.