Managing Shared Resources [1]

K. Hendrikx, E. Duval, H Olivié
Department of Computer Science
K.U.Leuven
koenh@cs.kuleuven.ac.be

1 Introduction

Management of software and hardware resources is important, especially in distributed applications. Resources such as file handles and database cursors are in limited supply. The available number may depend on the machine's load and configuration, and may not be known in advance.

In this context we will define a resource to be anything that is shared – by users, processes or threads. The most important resources in computing are processor cycles, memory and bandwidth, but in principle, anything that can be looked up and referenced can be shared. Management of such software resources becomes an issue when either:

- The resource requires exclusive access to another (physical) resource such as memory, processor cycles, communication hardware etc.
- The number of clients that concurrently use it in a distributed system can increase without limit.

2 Context

Resource management typically involves a tradeoff between time and space usage. Time is traded for space by sharing. Space is traded for time by replication. Both sharing and replication may introduce an extra cost to maintain consistency. Consistency of shared resources is usually maintained by carefully synchronising access to it, for instance by a semaphore or monitor. Consistency of replicated resources is maintained by synchronising the state of each copy of the resource.

2.1 Coping with growth

When the number of concurrent users of a resource grows, there are basically three strategies to cope with such growth:

Reconfigure Adding more memory, or use a faster CPU, or add more bandwidth can solve the problem. This is usually not done automatically, but the software should be configurable to allow it to make use of more or less hardware resources. Reconfiguration can be done statically at compile

time by changing parameters in the code. But it is often preferable to allow dynamic reconfiguration, by setting parameters at run time. For in-depth discussion of reconfiguration the reader is referred to [Sommerlad, 1999].

Multiplex Requests from different clients are sequenced: while one client uses the resource, other clients wait. An obvious disadvantage is that response times increase, which may be unacceptable for interactive applications. Another disadvantage is that processes may get blocked indefinitely because of deadlock or live-lock, causing failures that must be dealt with.

Replicate Replication increases concurrency as operations proceed independently in each replica. Replicated services usually share some underlying resources, so that speedup is never proportional to the number of replicas. A second benefit of replication is that the replica is often located closer to the client, so that access is faster and less bandwidth is consumed. This is usually called *caching*.

There exist many policies for replication and sharing. Replication is either synchronous or asynchronous. In synchronous replication updates to a resource are applied to all replicas atomically. Asynchronous replication has no such atomicity guarantee but is more scalable [Gray *et al.*, 1996]. There exist many techniques and algorithms for synchronising replicas asynchronously.

Sharing policies are either optimistic or pessimistic. Optimistic sharing does not block processes when accessing a shared resource, but will abort any operation that could cause an inconsistency. Exclusive locking is the most basic pessimistic policy. More advanced sharing methods allow multiple concurrent readers. In general more complex pessimistic policies allow more concurrency by taking into account more knowledge about the semantics of the operations.

2.2 Architecture

Resource management has an important impact on the (distributed) architecture of applications. Layered architectures [Mowbray and Malveau, 1997; Buschmann *et al.*, 1996], in which each layer provides services to a layer at a higher level of abstraction, is useful to manage the complexity of (distributed) systems. In such systems the lower layers deal with resource management. This goes for the classic three-tier client-server system, but also for protocol stacks where the bottom layers manage physical connections.

For instance, a two-tier client-server application may present problems of scalability if clients interact immediately with a database using SQL. As the number of concurrent clients grows, the server may be overwhelmed. One solution is to switch to a three-tier or multitier application architecture where the interaction with the database is encapsulated in a server-side *middleware* layer.

2.3 Forces

There are three conflicting approaches to problems in hardware and software: sharing, isolation and replication. Sharing is done by replacing the shared object with a name or an address – a token that defines a method for looking up the actual resource. Replication is providing each client with a copying of the resource. The difference between sharing and replication is illustrated by parameter binding mechanisms in languages like Pascal. Within the confines of structured programming, the semantics of copying and referencing are much more clear-cut than they are in an object-oriented setting. Both sharing and replication can lead to inconsistency. When inconsistency gets in the way, we must move away from these by isolation in the temporal domain: synchronisation.

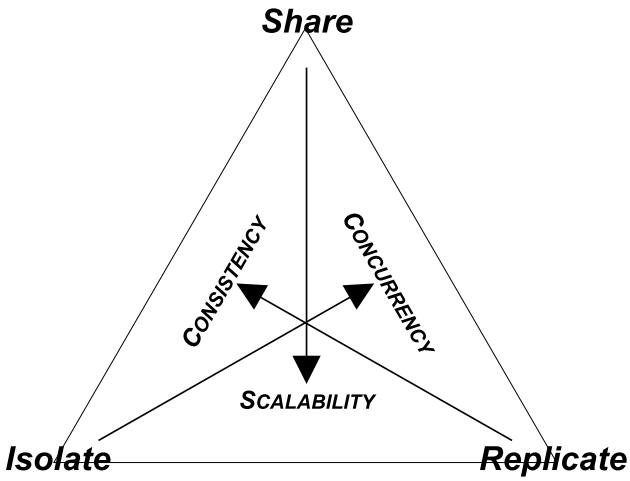

Figure 1: Overview of Forces

Figure 1 relates these approaches to the conflicting forces they resolve: scalability, concurrency and consistency. We want to maximise all three, but must strike a compromise. More isolation means less concurrency, because locking will exclude concurrent access to resources. More replication means less consistency, because as each copy may be modified independently. More sharing means less scalability, because shared resources get overwhelmed as the number of clients grows.

2.4 Roadmap

Figure 2 shows the patterns presented in this mini pattern language in relationship with the main forces. Each pattern represents a hot-spot allowing an implementation to vary along an axis in this three-dimensional solution space.

The Handle/Body patterns discussed in [Coplien, 2000] (especially Counted

Body) allow one to control how a resource is shared or replicated, for instance by allowing multiple clients to share one resource until one updates it. Synchroniser allows one to vary the level of sharing versus isolation of the resource by temporal synchronisation. A Resource Pool allows one to control sequential isolation versus replication by having a pool of equivalent resources, forcing clients only to wait when all resource are checked out. A pool of size one corresponds to sequential isolation, while a very large pool is equivalent of replication.

Transactions impose constraints on the order in which operations are applied to shared resources. There are basically three flavours of transactions: Optimistic Transaction relies on sharing the resource and detecting nonserialisability; Two-Phase Locking relies on locking to guarantee serialisability; Multiversion Two-Phase Locking uses replication to increase concurrency in locking protocols.

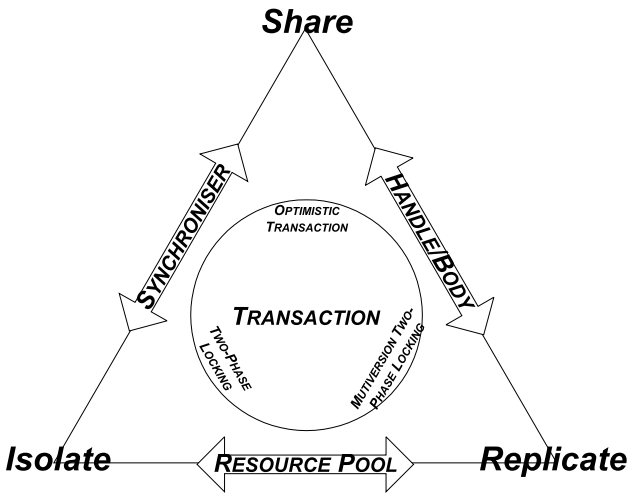

Figure 2: Overview of Forces and Patterns

3 The Wrapper Ideom

Many problems in software are solved by adding an extra indirection. This is especially true for resource management. A well-known ideom is the Wrapper ideom (also known as *Handle/Body* [Coplien, 2000]). A wrapper is an object that holds a reference to the actual resource. It may implement the same interface as the underlying resource by delegation, or it may use another mechanism to make it appear as the underlying resource (e.g. by overloading the dereference operator in C++).

Wrappers are used for a number of different resource-management applications:

Lazy Initialisation A wrapper is created without creating or looking up the actual resource. Only when the resource is first called upon does the wrapper allocate or look up the resource. A variation of this the is a *future* object, which represents the result of a computation before it is available.

Garbage Collection Garbage collection or centralised deallocation is achieved through wrappers, e.g. using reference counting. Combined with lazy initialisation, wrappers can be used to implement fairly complex caching mechanisms.

Synchronisation One application of a wrapper is to provide synchronisation for an underlying data structure. The wrapper is a Decorator [Gamma *et al.*, 1995] for the resource, only adding synchronisation constructs such as a semaphore or monitor (e.g. Java collection framework).

Copying/Referencing Combined with lazy initialisation and synchronisation, a wrapper can be used to hide whether the underlying resource is shared or not. This would allow an application to decide at run time whether to use an extra copy of the resource or to share an existing one. An application of this idea is the *copy-on-update* or *Counted Body* ideom, in which the wrapper creates a new copy of the shared resource when the client updates it.

Hiding Multiplicity A wrapper may hide the fact that the underlying resource is replicated and broadcast requests to each replica to implement fault tolerant systems.

Access Control A wrapper may be used to separate the access control policy from the actual resource.

In general, wrappers allow certain system or protocol issues to be factored out as first class entities, increasing flexibility. A disadvantage is that the wrapper usually depends on the resource class, creating an inappropriate dependency of the system perspective on the application perspective. For this reason, wrappers should be modelled at a higher level of abstraction, either as a language feature or based on reflection.

Pattern Name SYNCHRONISER

Context Multiple clients concurrently access shared resources. Some operations on these resources require clients to have exclusive access.

There are basically three ways in which the semantics of a resource dictate synchronisation:

Intra-Object: *Critical sections* of code (usually methods) that share variables may not be executed concurrently to avoid inconsistent behaviour. Before such code fragment is executed it must acquire a semaphore that is associated with the shared variable.

Client Transaction: Synchronisation depends on the semantics of the client's transaction, for instance when a client does a sequence of interdependent reads and updates to a data structure. The client must explicitly acquire and release exclusive access.

Inter-Object: Each method is guarded by a boolean condition that determines if it is available to clients. Usually, only one thread is active in the shared object, and messages to it are queued in a buffer. This paradigm allows dynamic scheduling of messages, and it is transparent to distribution. It is however not supported by many programming languages and usually relies on middleware tools and code generation.

Problem How can clients flexibly synchronise access to resources?

Forces

1. The mechanisms used to provide synchronisation generally depends on the support provided by the programming language, operating system and/or available libraries. If portability is an issue, synchronisation code should be factored out into separate components.

2. Embedding synchronisation within reusable classes, removes the responsibility from the programmer, protecting against programmer error.

3. Protocols for dealing with deadlock, fairness or non-blocking resource acquisition are difficult to implement using only the synchronisation constructs available in general-purpose programming languages.

4. Because of the inheritance anomaly [Crnogorac *et al.*, 1997], subclassing objects with built-in synchronisation may break the synchronisation of inherited behaviour, which may limit the reusability of these classes.

Solution Avoid mixing synchronisation code with business logic in reusable classes.

Use the Decorator pattern [Gamma *et al.*, 1995] to add synchronisation to a shared resource, so client code can use it transparently. Only for client transactions, the client must explicitly demarcate transaction boundaries by calling a transaction manager object.

Using separate Semaphore abstraction can be used to guarantee language and system independence. It also guarantees more flexibility to allow for deadlock detection and fair scheduling. Figure 3 shows a diagram of a possible class structure.

Collaborations

1. In case of a client-transaction, the client first creates a transaction, and locks the resource through that transaction.

2. A client makes a request to a synchronised resource.

3. Depending on the type of synchronisation:

   Critical Section: The method first calls the semaphore to acquire it. If it is unavailable, the method blocks. Then, the corresponding method of the actual resource is called, and the semaphore is released.

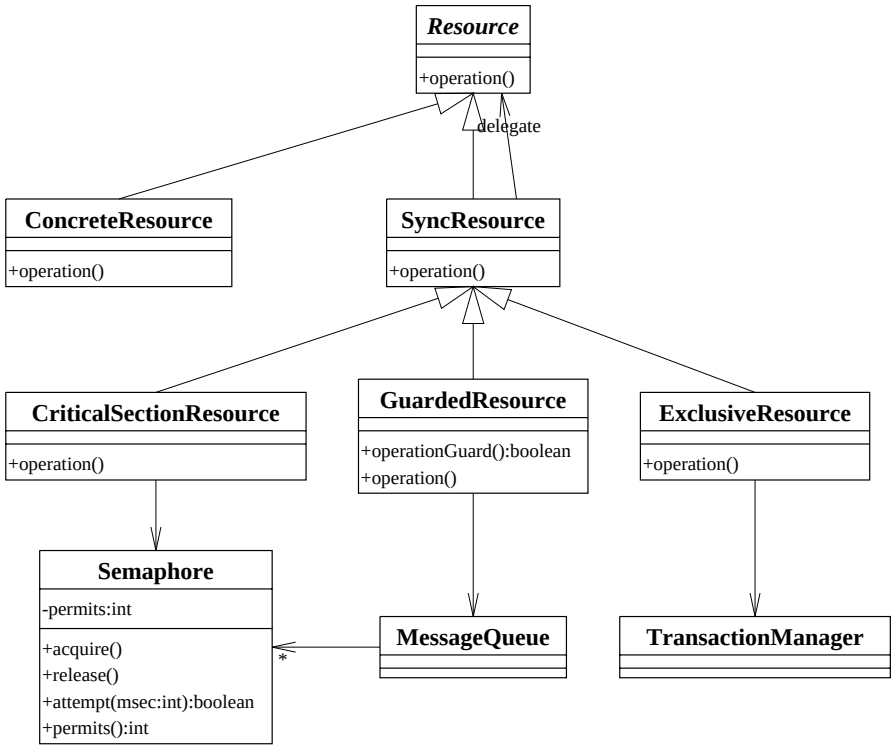

Figure 3: Synchroniser Structure

Client Transaction: The decorator does nothing more than assert that the shared resource is enlisted by the current transaction. If not, there must be a programmer error, and a run-time exception is raised. Otherwise, it calls the corresponding method of the actual resource and returns the result.

Guarded Method: The called method is *reified*[2] and enqueued on a message queue with its parameters. The queue returns a placeholder object that will contain the result when it becomes available, and a semaphore. The thread then blocks on the semaphore. When the queue has processed the message the semaphore is released, and the decorator can return the result. The message queue has its own thread of control, calling available methods and rescheduling unavailable methods.

4. The semaphore may cooperate with a deadlock detector to determine whether or not granting the lock would create a deadlock.

[2]Reification is conversion to an object. The method-object may be represented simply by its name, by an object of the language's reflection system, or by a Command object [Gamma *et al.*, 1995].

Consequences The use of this synchroniser pattern has the following consequences:

1. Synchronisation and business logic are separated and stand a better chance of being portable and reusable.
2. The synchronisation strategy can be easily changed, even at run-time, allowing more flexibility.
3. For transactional resources, the decorator can assert that the resource was properly locked before it is used.
4. By explicitly modelling semaphores instead of using the programming language's synchronisation primitives, it is easier to experiment with different mechanisms to avoid live-lock and to detect deadlock.

However

1. Programming low-level synchronisation primitives is difficult and error-prone.
2. There is considerable extra work in writing the extra classes needed.
3. Changes to the resource class also impact the synchronisation decorator classes. A practical solution to this problem is to automatically generate these classes based on e.g. IDL interface descriptions.

Known Uses

- The ACE [Schmidt, 1995] toolkit provides C++ implementations of semaphores, guards and events.
- Smalltalk has explicit classes, such as `ProcessScheduler`, `Semaphore` and `SharedQueue` to flexibly manage communication between processes.

Pattern Name RESOURCE POOL

Context A server process handles requests for an unlimited number of clients, using limited resources, such as database connections or server CPU time.

Problem How can we flexibly control concurrent access to stateless server resources?

Forces

- Simply allocating new resources by request of the clients may overwhelm the server, slowing down overall progress.
- The number of concurrently handled tasks must be large enough to fully utilise the server.

Solution Resources are stored in a resource pool. A client can check a resource out, use it, and check it back in. When resources are all checked out the client is blocked until one becomes available.

Because a client may break or invalidate a resource the resource pool must validate the resource before handing it to a client. If found invalid, it is replaced by a new resource.

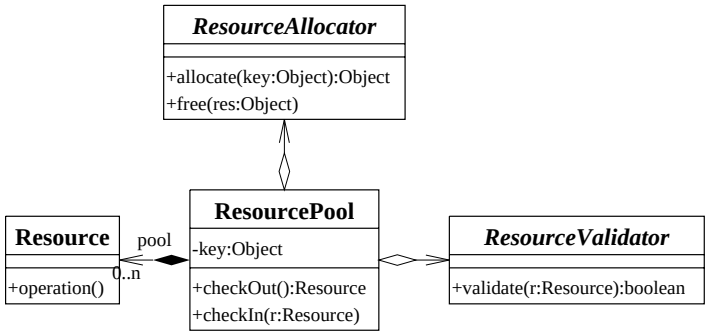

Figure 4: Resource Pool Class Structure

Structure Figure 4 shows the structure of the solution, separating the roles for allocation and validation into separate classes.

Applicability The resources cannot maintain state in between acquisitions: they are returned in their original state, or are reset by the pool on their return. The only state change supported is invalidation: a thread may be invalidated if it is killed by its task; a database connection may be invalidated if the connection to the server was temporarily lost.

Consequences

- The number of concurrently available resources can be adapted dynamically for optimal performance.
- Resources are preallocated, and acquiring one from the pool takes very little time.
- If no conversational state is maintained in between calls of a single client, a wrapper can be used that acquires and releases the resource for each method call, hiding the resource pool from the client.

Sample Code A typical example of a resource pool is a thread pool in a Web server. Each client request is handled by a thread from the thread pool. By limiting the number of concurrently handled requests the amount of memory, CPU-time and file handles used by the server can be kept below acceptable bounds.

A task thread object is an active consumer of tasks. To get a task to be executed, an idle task thread must be checked out of the resource pool. When the task is finished, the thread is checked back in. This provides a simple way of configuring how many threads can be active at the same time. An example sequence diagram is shown in Figure 5.

Known Uses

- Thread pooling is often used in Web servers such as the Apache Web server.
- Enterprise Java Beans specifies a type of components called Session Beans that maintain no conversational state between clients so that can be pooled dynamically in the server.

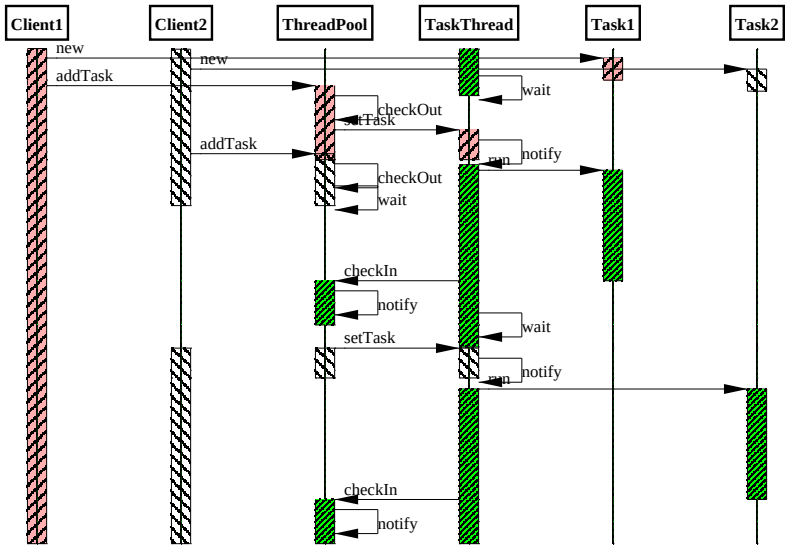

Figure 5: Task Thread Pool Sequence Diagram

- The ACE framework supports pooling for both threads and heap memory management.

Pattern Name TRANSACTION

Context A number of concurrent processes perform operations on shared resources.

Problem How can we ensure that subsequent operations are performed atomically so that other processes cannot see intermediate (inconsistent) states of the resources?

Forces

- By exclusively locking the shared resources a transaction can force other transactions to wait. However, this could constrain concurrency too much and increase the risk of deadlock and live-lock.
- Sharing constraints may vary:
  - The most strict is *serialisability*, which means that the accumulated result of multiple concurrent transactions is guaranteed to be identical to that of a serial execution of those transactions.
  - A less strict constraint is *repeatable read* or *session semantics*: each client only sees consistent data but clients may concurrently perform updates. Only updates from the last client to commit persist.

Solution Depending on the nature of the resource, the transaction and the expected amount of contention, choose one of the following patterns:

Optimistic Transaction aborts operations that jeopardise serialisability without blocking. Use only if contention is low or combined with other techniques that avoid conflicts.

Two-Phase Locking exclusively locks resources when read or written and releases locks on commit. Locking creates a risk of deadlock.

Multiversion Two-Phase Locking copies resources before they are updated, and at commit time consolidates updates in a way that guarantees serialisability. Multiple reading transactions can coexist with a single writer.

Each of these solutions achieves a different balance of concurrency, consistency and scalability by using different amounts of sharing, isolation and replication, as shown in Figure 2.

Consequences Flexible transaction abstractions provide fine-grained control over how the mechanisms of replication and locking are used while preserving transaction correctness.

However Transactions usually imply the possibility of failure, e.g. because of deadlock, meaning that participating operations must be aborted and undone.

Known Uses

- Transactions are supported by most relational and object-oriented database management systems.
- Sun's Java Transaction API specifies interfaces between a transaction manager and the parties involved in a distributed transaction system: the resource manager, the application server, and the transactional applications.
- The Microsoft Transaction Manager is a well-known middleware product that supports (distributed) transactions.

Pattern Name OPTIMISTIC TRANSACTION

Context A client is performing a sequence of operations on a shared resource, but concurrent clients may be updating the same resource, causing inconsistency.

Problem How can we detect that a shared resource was updated by another client?

Forces

- If contention for the resource is expected to be low, it may not be worthwhile to invest a lot of programming effort to enforce serialisability by locking.
- Locking has a tendency to create deadlocks, which have to be dealt with.
- Copying the resource may be a solution, but that could be an unreasonable expense if contention is low.

Solution During the transaction both the resource and the client keep track of a number that changes in a predictable manner with each update, for instance a simple count of the number of updates. At the start of a transaction, the client copies it and before each update, the client checks that the resource's number is what it expects it to be. If another client updated the resource concurrently, the numbers don't match and an exception is raised.

Consequences

- Consistency and serialisability are guaranteed.
- No locking is required at all, if the updates and access to the update count are implemented with atomic, non-interruptible instructions.
- No copying is required.
- There is no risk of deadlock or live-lock.

However Concurrent access will often result in exceptions. Partially completed transactions cannot be undone, unless operations are logged. Therefore concurrent access will often be regarded as a programming error, and force the programmer to come up with a solution that avoids it. Alternatively, if transactions are *idempotent* they can safely be retried.

Known Uses Java 2 collection framework uses this mechanism to ensure consistency between collections and their iterators.

Pattern Name TWO-PHASE LOCKING

Context Clients are concurrently performing operations on a shared resource. The resources represent important data that must remain consistent and updates may not be lost.

Problem How can we ensure serialisability of transactions?

Forces

- If contention for the resource is expected to be high, it may be worthwhile to invest programming effort to enforce serialisability by locking.
- Locking has a tendency to create deadlocks, which have to be dealt with.
- Copying the resource may be difficult because of its size or nature.

Solution Set up a two-phase locking scheme. Clients can only access the resource in the context of a transaction, which is associated with the current thread. Clients explicitly call the transaction to lock the resource, and the transaction keeps track of the locks. When the client commits the transaction, all locks are released.

Serialisability is guaranteed if no client releases any locks before acquiring a lock. The transaction object verifies this constraint.

Differentiating between read-locks and write-locks or otherwise introducing knowledge of the semantics of the operations may increase concurrency. For instance, read-locks can coexist with other read-locks.

Consequences

- Serialisability is guaranteed.
- No copying of resources is required.

However

- Locking reduces concurrency.
- With high contention, the risk of deadlock increases. The number of deadlocks (per second) in a distributed system is very sensitive to the duration of transactions and the number of resources locked by each transaction[3].
- When deadlock occurs, it must be detected and the transaction must be undone. To enable undoing of operations, the Command pattern [Gamma *et al.*, 1995] can be used to log and undo operations of aborted transactions.

Known Uses Most database management systems use two-phase locking to implement transactions.

Pattern Name MULTIVERSION TWO-PHASE LOCKING

Context Client are concurrently performing operations on a shared resource. The resources represent important data that must remain consistent and updates may not be lost.

Problem How can we ensure serialisability of transactions?

Forces

- Two-Phase Locking causes queries to lock large regions of data for long periods of time, causing update transactions to suffer long delays.
- If copying the resource is easy, clients can be handed a copy of the resource that each can use concurrently.

Solution Resources are accessed through a separate handle or wrapper, using the Decorator pattern [Gamma *et al.*, 1995]. A client must explicitly call the transaction to acquire a handle for the resource. The transaction holds an exclusive lock on each handle. The handle is either a read-only handle or a read-write handle, depending on the type specified by the client. Many read-only handles may exist for a given resource, but only one read-write handle. Clients requesting a read-write handle that is in use are blocked until it becomes available.

When a method is called on the handle, the handle checks that it exists in the current transaction, raising an exception if not. Read-only handles also raise an exception when a mutator method is called on them.

When a mutator method is called on a read-write handle, the handle creates a new copy of the resource and delegates all subsequent messages to that copy.

[3] In [Gray *et al.*, 1996] the deadlock rate is estimated to be $\frac{TPS \times ActionTime \times Actions^5}{4 \times DbSize^2}$ for a system performing TPS transactions per second, each performing $Actions$ operations, each taking $ActionTime$ seconds.

When the client commits the transaction, the transaction acquires an exclusive lock on all the read-write handles it handed out, and on all read-only handles for the updated resource. These handles are locked in a fixed order determined by the resource identities, in order to avoid deadlock. The transaction then unifies all modified copies of the resources with the original version, either by updating them with the modifications, or by simply replacing the original. Finally, all locks are released.

Structure Figure 6 shows a possible class structure for this pattern.

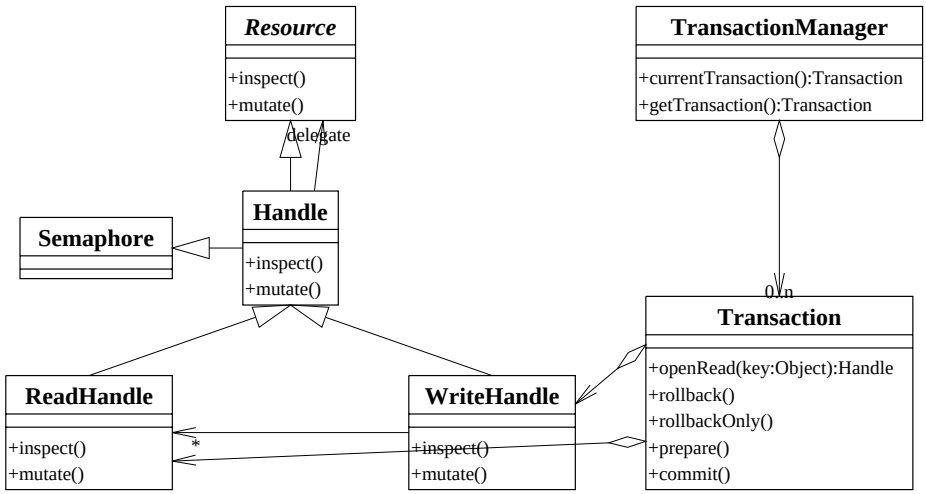

Figure 6: Multiversion Two-Phase Locking Class Structure

Consequences

- Multiple clients can concurrently read a resource.
- One client can update a resource concurrently with other readers.
- There is more concurrency and less risk of deadlock than with simple two-phase locking.
- No operations must be undone when a transaction is aborted. The updated copy can simply be discarded.
- Copying of resources can be combined with client-side caching in distributed systems.

However

- Copying the resources and their data requires extra memory.
- Committing requires a transaction to wait for all other transactions that have read the resources that it has updated to complete first.
- Failure because of deadlock can occur during commit.

Known Uses Multiversion two-phase locking has been incorporated in various relational database management systems, among which Prime DBMS, DEC Rdb/VMS and Interbase.

Acknowledgments

I thank John Letourneau, my EuroPloP shepherd, for his many helpful comments.

References

[Buschmann *et al.*, 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stad. *Pattern-Oriented Software Architecture – A System of Patterns.* John Wiley, 1996.

[Coplien, 2000] James O. Coplien. C++ idioms patterns. In Brian Foote, Neil Harrison, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, chapter 10, pages 167–197. Addison Wesley, Reading, MA, 2000.

[Crnogorac *et al.*, 1997] Lobel Crnogorac, Anand S. Rao, and Kotagiri Ramamohanarao. Inheritance anomaly — A formal treatment. Technical Report 96/42, Department of Computer Science, University of Melbourne, April 1997.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995.

[Gray *et al.*, 1996] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of theACM SIGMOD International Conference on Management of Data*, volume 25, 2 of *ACM SIGMOD Record*, pages 173–182, New York, June 4–6 1996. ACM Press.

[Mowbray and Malveau, 1997] Thomas J. Mowbray and Raphael Malveau. *CORBA Design Patterns.* John Wiley & Sons, New York, 1997.

[Schmidt, 1995] Douglas C. Schmidt. An OO encapsulation of lightweight OS concurrency mechanism in the ACE toolkit. Technical Report WUCS-95-31, Washington University, St. Louis., 1995.

[Sommerlad, 1999] Peter Sommerlad. Configurability. In *Proceedings of the Fourth European Conference on Pattern Languages of Programming and Computing (Europlop)*, 1999.