Synchronizer

an Object Behavioral Pattern
for Concurrent Programming

Ennio Grasso
CSELT
Via Reiss Romoli 274 - 10148 Torino (Italy)
e-mail: ennio.grasso@cselt.it
fax: +39-11-2286862

Abstract

This paper describes the *Synchronizer* pattern, a specialization of the *Active Object* pattern described in [SCHM96a]. The intent of the Active Object pattern is the decoupling of method execution form method invocation to simplify synchronized access to shared objects by different threads. By assigning one thread to a shared object, the Active Object does not provide the maximum amount of concurrency that could be achieved. The Synchronizer specializes the Active Object by allowing many concurrent threads to access a shared object.

Intent

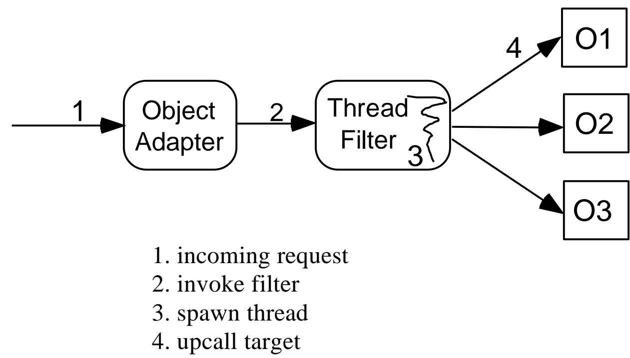
The Synchronizer decouples method execution form method and externalizes the synchronization policy of the operations on a shared object so as to allow concurrent non-conflicting threads to access the object.

Introduction

Concurrency is a paramount concern in a distributed system, where application servers may receive multiple requests in parallel with the problem of synchronizing those requests. A number of concurrency models have been proposed to tackle the synchronization problem, all based on the *Command* pattern [GAMM94] that encapsulates a request as a first class object. In a single-threaded environment, the *Reactor* pattern [SCHM95b] provides a cooperative event loop model to manage and serve queues of requests. Although suited for single-threaded systems, the Reactor is fairly complex and requires subtle programming to avoid deadlocks. In a multi-threaded environment other options are available that exploit the parallelism provided by the underlying hardware and software platform.
In particular, two concurrency patterns, called respectively *thread-per-request* and *thread-per-object*, are possible. In the remainder we will assume that thread management be performed by a separate filter object that use the *Chain of Responsibility* [GAMM94] pattern to decouple request demultiplexing from request dispatching.

Thread-per-request

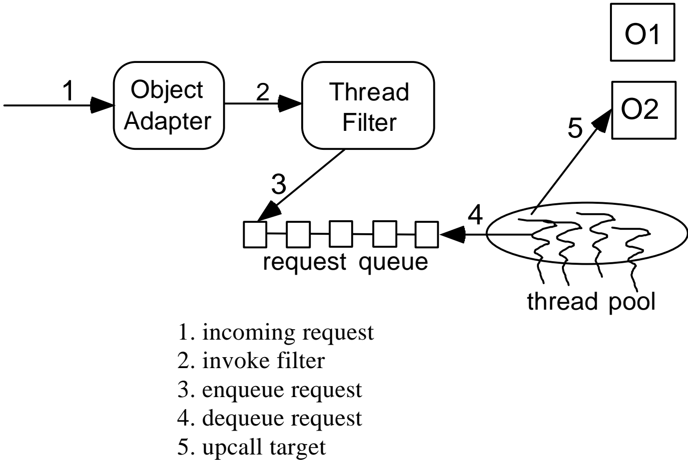
In the thread-per-request model every incoming request to a server causes a new thread to be spawned to process the request.



This model is fairly simple to implement and provides the greatest amount of concurrency that can possibly be achieved becuse every request is assigned to a different thread. However, the thread-per-request model reveals two drawbacks:

1. spawning a new thread for each incoming request may cause an overhead on the system;
2. since many requests may be dispatched to the same object, the application is responsible for synchronizing concurrent threads on shared objects.

A variation of the thread-per-request pattern that avoid the cost of dynamic thread creation employs the strategy of pre-allocating a pool of threads in the server that will be dynamically assigned to incoming requests.



If the number of requests exceeds the number of threads in the pool, the exceeding requests will be waiting for a thread to become available. Compared with the first approach, the pool model is more complex because of the management of the request-queue and the assignment of requests to threads. Requests are enqueued by a master thread that executes the filter code while each working thread in the pool runs in a loop waiting for a new request, processing it, and sending back the result:

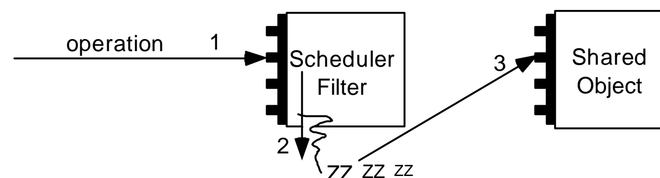
```
for (;;) {
        Request r =  RequestQueue->dequeue();
        // suppose this will also send back the result
        ObjectAdapter->dispatch(r);
}
```

The drawback to the pool approach is that the number of working threads must be decided beforehand, which is a compromise between the potential concurrency level on the one hand and thread consumption on the other. The pool approach appears best suited in cases where bursts of requests and idle periods are infrequent, which is statistically the case in many servers.

Synchronization issues

From the application point of view there is not much difference between the two variants of the thread-per-request pattern, because in both cases the application is in charge of handling concurrency control on shared objects. A number of concurrency control techniques may be used, each enforcing a particular synchronization policy: *mutexes*, *semaphores*, *condition variables*, *guards*, *locks*, to name a few. For example, the synchronization policy of the locking model is specified in terms of the conflict relationship between lock modes [GRAY93]. If a thread requests a lock mode that conflicts with any of the lock modes already granted to other concurrent threads, the first thread is suspended until its lock request can be fulfilled. Other techniques, such as *condition variables*, use synchronization policies based on constraints on the state of the shared object. If the constraints are not satisfied the thread will be suspended until a different thread changes the state of the shared object and notify the condition variable about this change.

If the synchronization code is embedded into the operations of the shared object the result will be an obtrusive implementation that will inhibit the by derived classes that need different synchronization policies. A more flexible implementation is to use the *Chain of Responsibility* pattern by separating the synchronization code into a *Scheduler* filter object in charge of enforcing the synchronization policy for the shared object.



1. operation dispatched to the Scheduler filter
2. if synchronization constraints are not satisfied suspend thread
3. when synchronization constraints are satisfied invoke the operation

With a locking technique the code of the Scheduler filter would be something like:

```
SchedulerFilter::opA(...) {
        // acquire lock in (e.g.) exclusive mode
        lock->acquire( exclusive_mode);

        // dispatch to the shared object
        obj->opA(...);

        // release lock
        lock->release( exclusive_mode);
}
```

With condition variables:

```
SchedulerFilter::opA(...) {
        // wait till predicate is true
        while (!predicate(...))   cond->wait() 1;

        // dispatch to the shared object
        obj->opA(...);

        // possibly signal an awaiting thread
        if (predicate(...))   cond->signal();
}
```

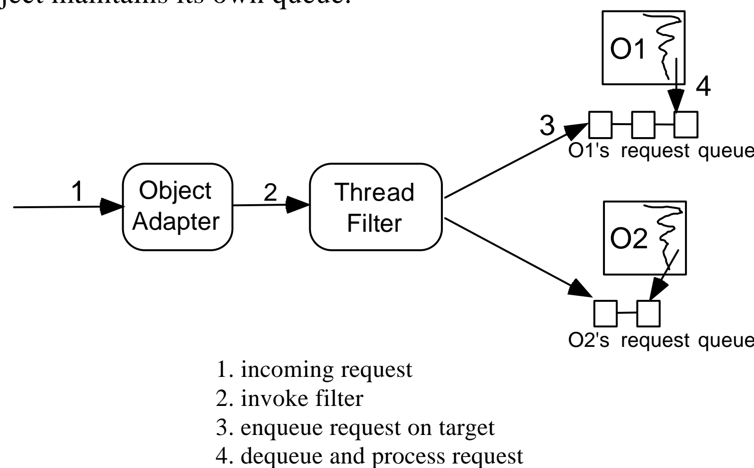
---

[1] *A condition variable typically relies on amutex to evaluate the predicate inmutal exclusion, but this detail is omitted here for the sake of simplicity.*

Whatever the synchronization policy, if the synchronization constrains enforced by the Scheduler are not satisfied the thread will be suspended until a second thread alters the state of affair, possibly allowing the first thread to proceed. Such waiting state is usually limited to the time needed by the second thread to complete its processing on the shared object and corresponds to the execution of a single object operation. However, it is not uncommon that an operation calls a chain of operations thereby prolonging the processing on the object. In such cases this forced waiting state may definitely be a wastage of thread resources since the suspended thread cannot be used to process other incoming requests.

Thread-per-object

Instead of dynamically assigning a thread to every incoming request, whether by spawning a new thread or allocating the thread from the pool, the thread-per-object pattern, called Active Object in [SCHM96a], makes the correspondence thread/object by assigning one thread per object. From the implementation point of view the thread-per-object pattern bears a close resemblance with the pool variant of the thread-per-request pattern. In this case, however, requests are not inserted in a global queue but each object maintains its own queue.



1. incoming request
2. invoke filter
3. enqueue request on target
4. dequeue and process request

Having one single thread per object, synchronization policies based on the conflict relationship between object operations are not necessary since all invocations are forcibly serialized. The only synchronization policies that make sense are constraints on the state of the object. For example, consider a *queue* object with enqueue and dequeue operations. The enqueue operation can only be executed if the queue is not full and conversely the dequeue operation can only be executed if the queue is not empty.

Contrary to the thread-per-request pattern, in the thread-per-object pattern the Scheduler does not cause the thread to wait. Instead, the thread selects a request from the queue only if it satisfies the constraints enforced by the Scheduler [SCHM96a]:

```
for (;;) {
        // select a request that satisfies the predicate
        for (Request r =  RequestQueue->head(); r; r = r->next)
                if (predicate(r,...)) {
                                RequestQueue->dequeue(r);
                                break;
                }
        // suppose this will also send back the result
        ObjectAdapter->dispatch(r);
}
```

The thread-per-object pattern avoids the undesirable waiting state that affects the thread-per-request pattern for threads that do not satisfy the synchronization constraints of the Scheduler. However the

thread-per-object pattern may reduce the overall concurrency level of the server when compared with the thread-per-request pattern. This is because threads are statically associated with objects and not dynamically assigned to requests. To understand the difference, consider the scenario in which many concurrent requests are dispatched to the same object and suppose many of them could be executed concurrently according to the synchronization constraints of the Scheduler. In the thread-per-request pattern there is one thread per request and many are allowed to proceed in parallel. On the other hand, in the thread-per-object pattern there is only one thread in charge of processing all the requests dispatched to the same object and even though many requests are potentially concurrent they are in fact forcibly serialized. Note that there may be many threads in the server associated with different objects but these threads will remain idle because no requests are sent to them.

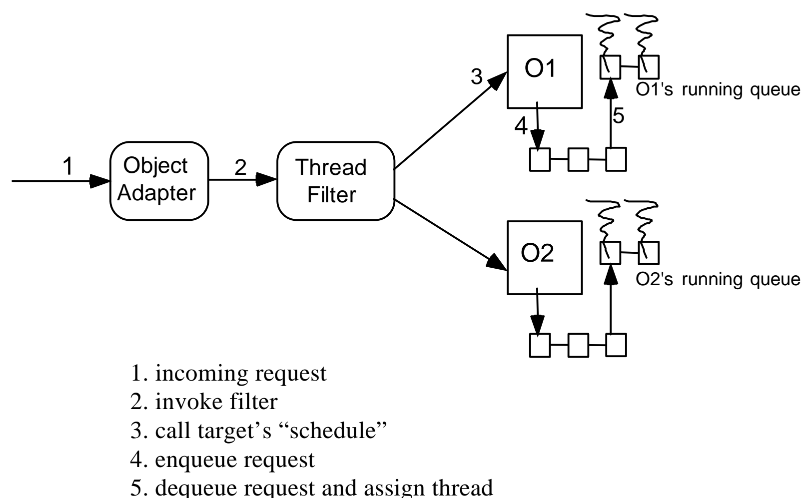
Synchronizer

As described above, the thread-per-request pattern permits greater concurrency than the thread-per-object pattern, but the inconvenient is that threads may be forcibly suspended until some constraints are satisfied. The *Synchronizer* pattern complements the thread-per-request pattern with the synchronization model of the thread-per-object pattern so as to avoid threads waiting on synchronization constraints.

A typical implementation of the thread-per-request pattern employs a two-step approach:

1. a thread, either spawned or allocated from a pool, is assigned to an incoming request;
2. the thread is dispatched to the Scheduler that may force the thread into a waiting state.

The idea of the Synchronizer is to merge these two steps by assigning a thread to a request only if the request satisfy the synchronization constraints enforced by the Scheduler.

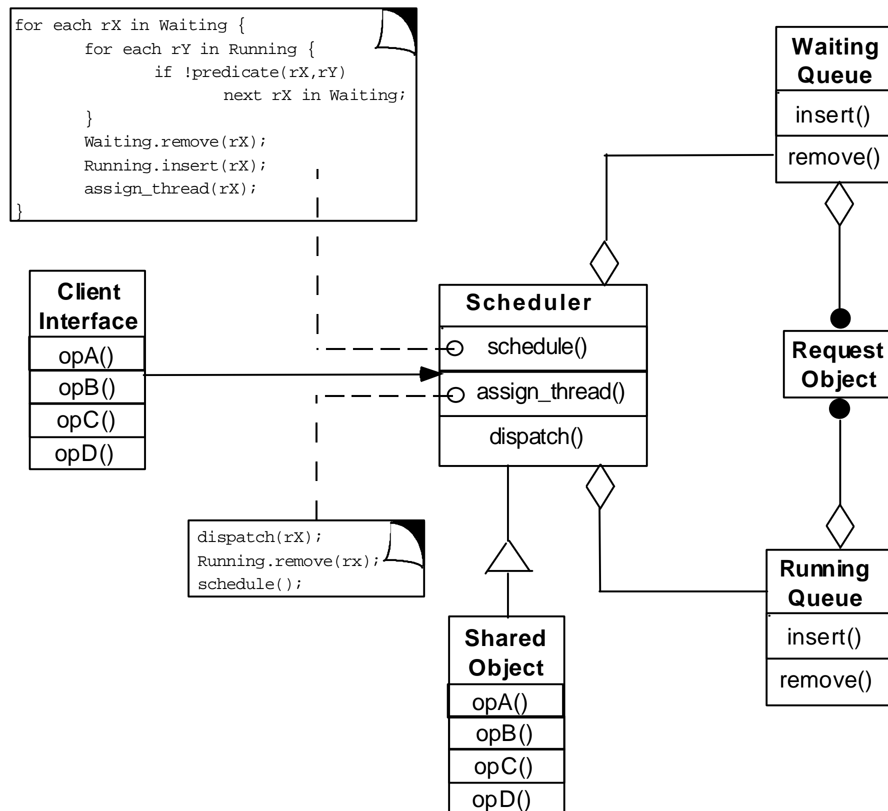


To summarize the difference between the three approaches:

- in the standard thread-per-request pattern the Scheduler suspends the threads assigned to requests that do not satisfy the synchronization constraints;
- in the thread-per-object pattern the Scheduler selects a request that satisfies the synchronization constraints and assigns it to the thread associated with the object;
- in the Synchronizer pattern, the Scheduler selects all requests that satisfy the synchronization constraints and assign each of them to a different thread. Note that the Synchronizer is silent on whether the threads are spawned or allocated from a server pool. Either approach can be applied according to application requirements.

Structure and Participants

The structure of the Synchronizer bears many similarities with the structure of the Active Object pattern [SCHM96a] and is illustrated in the following OMT class diagram:



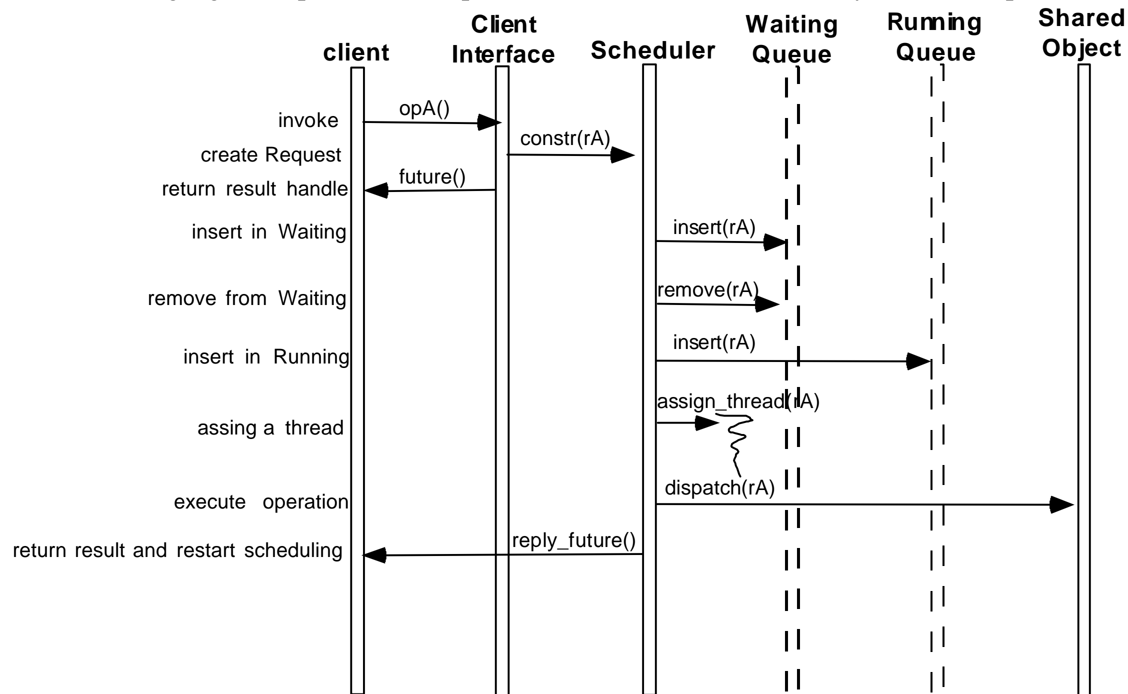
The participants in the Synchronizer pattern are:

- **Client Interface** is a *Proxy* [GAMM94] that presents an operation interface to client applications. The invocation of an operation in the Client Interface triggers the construction of a Request Object.
- **Request Object** is constructed for any invocation that needs synchronized access to a Shared Object managed by the Scheduler. Each Request Object maintains the argument bindings to the operation as well as any other information to execute the operation and return the result.
- **Running Queue** maintains a queue of requests which are being processed by concurrent threads on the shared object.
- **Waiting Queue** maintains a priority queue of pending requests that cannot be executed because they do not satisfy the synchronization constraints enforced by the scheduler.
- **Scheduler** manages the Waiting and Running Queues and enforces the synchronization policy for the shared object. The policy determines whether a request can be executed and inserted in the Running Queue or has to be inserted in the Waiting Queue. The decision to process a request may be based on predicates on the state of the shared object or on the conflict relationship with any requests that are being processed by a concurrent threads.
- **Shared Object** is the shared resource protected by the Synchronizer and defines the same operations of the Client Interface. Although in the OMT diagram the Shared Object class inherits from the Scheduler, it may well be designed as a separate object and use delegation instead of inheritance.

The Scheduler of the Synchronizer pattern augments the Scheduler of the Active Object pattern by allowing the designer to specify synchronization policies based on the conflict relationship between concurrent operations. This is certainly not necessary in the Active Object that applies the thread-per-object approach so that there will never be concurrent threads in the same object. On the other hand, the Synchronizer uses the thread-per-request approach and there may be synchronization policies to enforce based on conflicting operations.

Collaborations

The following figure depicts the five phases of collaborations in the Synchronizer pattern:



1. *Request Object construction* - the client application invokes an operation in the Client Interface proxy object. This triggers the creation of a Request Object, which maintains the argument bindings to the operation as well as any other information to execute the operation and return the result.
2. *Queuing* - the Scheduler inserts the Request Object in the Waiting Queue.
3. *Scheduling* - the Scheduler obtains a mutex and for each pending request in the Waiting Queue determines if the request can be executed by checking its synchronization constraints, possibly in relationship with all the requests in the Running Queue. Every request that satisfy the constraints is removed from the Waiting Queue and inserted in the Running Queue.
4. *Execution* - each request inserted in the Running Queue is assigned a thread (which may be spawned or allocated from a pool) that dispatches the request to the Shared Object.
5. *Return result* - after completing the execution of the request, the thread returns the result (if any) to the client, then obtains a mutex and removes the Request Object from the Running Queue. This may possibly allow some pending requests to execute: the thread notifies the Scheduler which restart from the Scheduling phase.

Transactional Synchronizer

Transactions are a useful instrument to guarantee consistency of applications, though they have traditionally been investigated in the context of conventional database applications. More recently the

transaction concepts have been extended to the broader context of object distributed architectures, such as the Object Transaction Service [OMG95] for CORBA which provides support for controlling the scope and completion of distributed nested transactions.

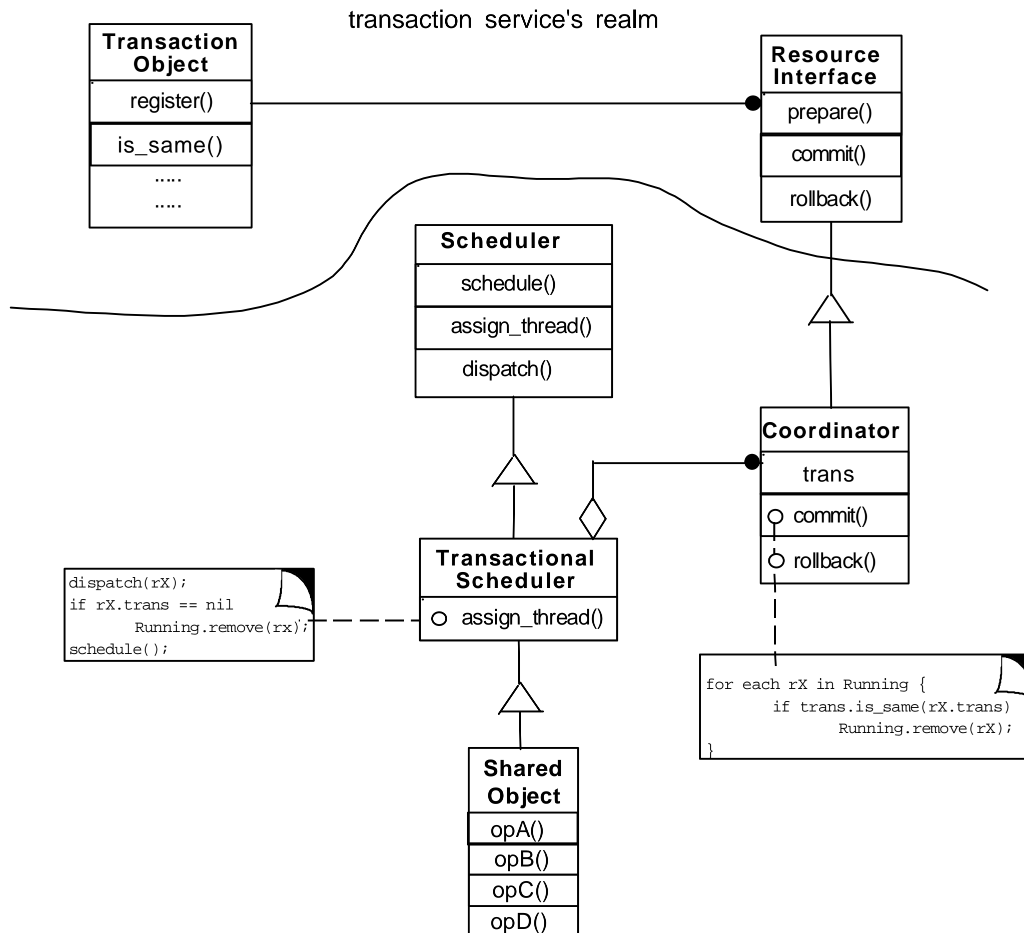
A transaction service needs to coordinate access to shared objects and ensure isolation of concurrent transactional activities. In a locking based concurrency control technique, when a thread obtains a lock on behalf of a transaction the lock is not explicitly released by the application but automatically by the transaction service at the end of the transaction. This so called *two-phase-locking* policy[2] is enough to guarantee *serializability* of concurrent transactions. To become transaction-aware and coordinate requests on behalf of transactions the Synchronizer must follow the same principle of the two-phase-locking policy. The *Transactional Synchronizer* augments the Synchronizer with:

- a Transaction Identifier associated with Object Requests, and
- a *Coordinator* object used to enforce the two-phase-locking policy for requests processed on behalf of transactions.

Recall that in the Synchronizer when a thread completes the execution of an operation, the corresponding Request Object is removed from the Running Queue. This removal may possibly allow some pending requests in the Waiting Queue to be executed according to the synchronization policy enforced by the Scheduler. Maintaining the Request Object in the Running Queue even after the execution of an operation is tantamount to retaining a lock obtained on behalf of a transaction: the conflicting pending requests are not allowed to proceed. On the other hand, when a transaction completes, a means is needed to remove the Request Objects in the Running Queue pertaining to that transaction, and the Coordinator is used to carry out that task.

How is the Coordinator coupled with the transaction service? Every transaction service must offer a means to let resources register their involvement in transactions. If the Coordinator is a resource that a participates in a transaction, it will be notified at the completion of the transaction. For example, in the Object Transaction Service, an object that wants to be involved in a transaction must support the *Resource* interface and register itself with the *Transaction Object*. When the transaction completes the Resource is notified through the commit or rollback operations. The structure of the Transactional Synchronizer leverage on the vanilla Synchronizer and is illustrated in the following OMT class diagram (classes of the Synchronizer are not depicted):

[2] *While the transaction is active, it progressively acquires locks (growing phase) which are maintained until the transaction completes (shrinking phase).*



The new participants in the Transactional Synchronizer pattern are:

- **Coordinator** inherits from the Resource Interface so as to be capable of registering as a participant in the Transaction Object. The Coordinator is notified by the Transaction Object when the transaction completes so that it can remove all Resource Objects pertaining to that transaction from the Running Queue. Note that the Transaction Object and the Resource Interface are not part of the Transactional Synchronizer.
- **Transactional Scheduler** refines the assign_thread operation. Request Objects are now removed from the Running Queue only if they were not carried out on behalf of transactions.

Sample usage in the PCCS

We applied the Synchronizer and Transactional Synchronizer patterns in the implementation of the Programmable Concurrency Control Service (PCCS) [GRAS96a], [GRAS96b] and the eXtended Transaction Service (XTS) [GRAS96c] for Orbix-MT [IONA95].

The PCCS defines a *SynchronizedObject* IDL interface with no operations but simply used as a marker to indicate that an object wants to be synchronized through the Synchronizer pattern. As an example, consider the following IDL interface specification:

```
#include " pccs.idl"

interface Account :   ProgConcurrencyControl::SynchronizedObject {
       void deposit(in lon  g x);
       void withdraw(in long x);
       long balance();
};
```

The interface *Account* inherits from *SynchronizedObject* meaning that each invocation to the Account object will be synchronized transparently by the PCCS. Let us consider the implementation to the *Account* interface:

```
#include < pccs.h>
#include " account.hh"

class account : public   AccountBOAImpl, public   Scheduler_i {
       CORBA::Long x;
public:
       account( ProgConcurrencyControl::ConflictTable &t) :     Scheduler_i(t) {}
       virtual void* _ deref() {return (  Scheduler_i*)this;}

       // IDL operations
       virtual void deposit(  CORBA::Long l,  CORBA::Environment &  env) {
             x += l;}

       virtual void withdraw(  CORBA::Long l,  CORBA::Environment &  env) {
              x -=l;}

       CORBA::Long balance(  CORBA::Environment &  env) {
              return x;}
};
```

Two things are worth noting in the *account* class: it inherits from the *Scheduler_i* class that implements the Scheduler component of the Synchronizer pattern, and it defines the _ deref method so as to return the reference to the Scheduler_i object. Indeed, when an invocation arrives at the server, the thread filter of the PCCS checks whether the target interface inherits form *SynchronizedObject*. If so, the filter must have a hook to pass the request to the Scheduler. The only general way that Orbix offers to proceed from the IDL interface to the implementation class is through the _ deref method and this is used by the thread filter to get access to the Scheduler. The filter expects a pointer to the Scheduler_i class and that is why the _deref method must cast up to the Scheduler_i class.

If inheriting from the Scheduler_i class is considered too restraining, e.g. because the inheritance hierarchy has already been defined, it is possible to resort to delegation instead of inheritance. Plainly, it will suffice to create a separate Scheduler_i object and provide a reference to this object in the _ deref operation as in this example:

```
#include < pccs.h>
#include " account.hh"

class account : public   AccountBOAImpl {
       CORBA::Long x;
       Scheduler_i *s;
public:
       account( ProgConcurrencyControl::ConflictTable &t) {
           s = new  Scheduler_i(t);
       }
       virtual void* _ deref() {return s;}

       // IDL operations
       //... same as before
```

As said before, the Synchronizer pattern allows the specification of different synchronization policies. In the implementation of the PCCS we have applied a synchronization policy based on the conflict relationships between the operations on the IDL interface. Note that the constructor of the Scheduler_i class requires a *ConflictTable* parameter that allows the designer to declare the conflict relationships of the operations according to application semantics. Since all object instances that support the same IDL interface bear the same conflict specification, only one *ConflictTable* instance is created using the

Singleton pattern [GAMM94]. In the example, suppose the table for the *Account* interface be as follows:

	•	•	•
	•	•	•
	•	•	

The specification of the table can be done in either two ways:

1. create an instance of the *ConflictTable* class and then initialize it;
2. subclass the *ConflictTable* class with a specialized class whose constructor initializes the table according to the desired conflict policy.

We will consider the second option as the first is almost identical:

```
#include " account.h"

class AccountTable : public  ProgConcurrencyControl::ConflictTable {
public:
       AccountTable() :  ProgConcurrencyControl::ConflictTable(3,16) {

       // set the name according to the operations in the Account interface
       ProgConcurrencyControl::lock_mode    dep =
                           account_table.string_to_mode("deposit");

       ProgConcurrencyControl::lock_mode wit =
                           account_table.string_to_mode("withdraw");

       ProgConcurrencyControl::lock_mode    bal =
                           account_table.string_to_mode("balance");

       // initialize the conflict relationship
       account_table.set_exclusive(  dep);
       account_table.set_exclusive(wit);
       account_table.set_conflict(  bal, dep);
       account_table.set_conflict(  bal, wit);
       }
};

// create a singleton account table
AccountTable  account_table;

main () {
       // create  two account objects
       Account_var a1 = new account(   account_table);
       Account_var a2 = new account(   account_table);

       CORBA::Orbix.impl_is_ready();
}
```

Before specifying the conflict relationship the designer must specify the operation names in string format using the `string_to_mode` method of the table. This is important because Orbix dispatches requests according to operation names in string format. The Scheduler_i class uses this information to match a request with the name specified in the `string_to_mode` method. Note that the constructor of the ConflictTable takes two integers. One is the size of the table, i.e. the number of the entries corresponding to the number of operations in the IDL interface (3 in the example). The second specifies the maximum length of the operation names (16 characters in the example).

References

[BUSC96], F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, “A System of Patterns”, John Wiley & Sons, 1996.

[GAMM94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns”, Addison Wesley, 1994.

[GRAS96a] E. Grasso; "A Programmable Concurrency Control Service for Real-Time Object Request Broker"; in Proc. 8th Euromicro Workshop on Real Time Systems, L'Aquila, 1996.

[GRAS96b] E. Grasso; "A Programmable Concurrency Control Service CORBA"; in Proc. 3rd International Workshop on Services in Distributed Networked Environments, Macau, 1996.

[GRAS96c] E. Grasso; "An Extended Transaction Service for Telecom Object Request Brokers "; in Proc. 2nd International Workshop on Distributed Object Oriented Computing, Frankfurt, Germany, 1996.

[GRAY93] J. Gray, A. Reuter; "Transaction Processing: Concepts and Techniques"; Morgan Kaufmann Publishers, 1993.

[IONA95] IONA Technologies Inc., “Orbix-MT 2 programming guide”, 1995.

[OMG91] "The Common Object Request Broker Architecture and Specification", OMG doc. n. 91-12-1, December 1991.

[OMG95] "CORBAservices: Common Object Services Specification", OMG doc. n. 95-3-31, March 1995.

[SCHM95a] R. G. Lavender, D. C. Schmidt, “Active Object”, conf. Pattern Lnaguages of Programming,Monticello, Illinois, September 1995.

[SCHM95b] D. C. Schmidt, “Reactor”, in Pattern Languages of Programming Design, Addison Wesley, 1995.

[SCHM95c] D. C. Schmidt, “Evaluating Concurrency Models for CORBA Servers”, online tutorial: http://www.cs.wustl.edu/~schmidt/tutorials-corba.html