Active Object

An Object Behavioral Pattern for
Concurrent Programming

R. Greg Lavender
G.Lavender@isode.com
ISODE Consortium Inc.
Austin, TX

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis

An earlier version of this paper appeared in a chapter in the book "Pattern Languages of Program Design 2" ISBN 0-201-89527-7, edited by John Vlissides, Jim Coplien, and Norm Kerth published by Addison-Wesley, 1996.

Abstract

This paper describes the Active Object pattern, which decouples method execution from method invocation in order to simplify synchronized access to an object that resides in its own thread of control. The Active Object pattern allows one or more independent threads of execution to interleave their access to data modeled as a single object. A broad class of producer/consumer and reader/writer applications are well-suited to this model of concurrency. This pattern is commonly used in distributed systems requiring multi-threaded servers. In addition, client applications, such as windowing systems and network browsers, employ active objects to simplify concurrent, asynchronous network operations.

1 Intent

The Active Object design pattern decouples method execution from method invocation to enhance concurrency and simplify synchronized access to an object that resides in its own thread of control.

2 Also Known As

Concurrent Object and Actor

3 Example

To illustrate the Active Object pattern, consider the design of a communication Gateway [1]. A Gateway decouples cooperating components and allows them to interact without having direct dependencies among each other [2]. The Gateway shown in Figure 1 routes messages from one or more supplier processes to one or more consumer processes in a distributed system [3].

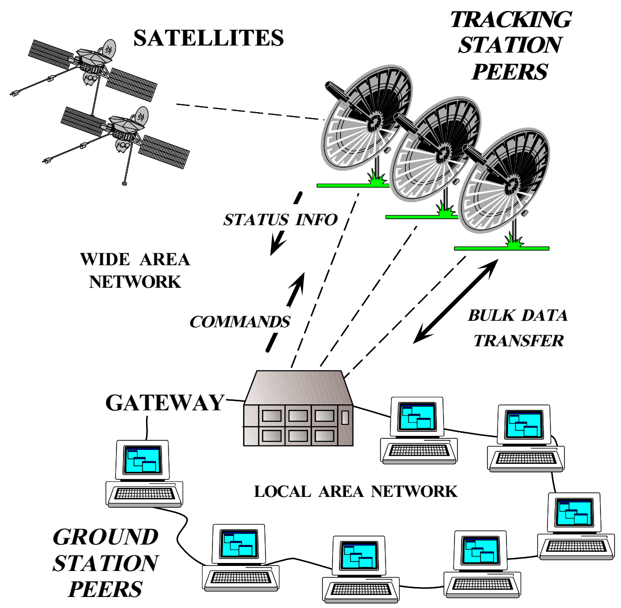


Figure 1: Communication Gateway

In our example, the Gateway, suppliers, and consumers communicate over TCP, which is a connection-oriented protocol [4]. Therefore, the Gateway software may encounter flow control from the TCP transport layer when it tries to send data to a remote consumer. TCP uses flow control to ensure that fast suppliers or Gateways do not produce data more rapidly than slow consumers or congested networks can buffer and process the data.

To improve end-to-end quality of service (QoS) for all suppliers and consumers, the entire Gateway process must not block waiting for flow control to abate over any one connection to a consumer. In addition, the Gateway must be able to scale up efficiently as the number of suppliers and consumers increase.

An effective way to prevent blocking and to improve performance is to introduce concurrency into the Gateway design. Concurrent applications allow the thread of control of an object O that executes a method to be decoupled from the threads of control that invoke methods on O. Moreover, using concurrency in the gateway enables threads whose TCP connections are flow controlled to block without impeding the progress of threads whose TCP connections are not flow controlled.

4 Context

Clients that access objects running in separate threads of control.

5 Problem

Many applications benefit from using concurrent objects to improve their QoS, *e.g.*, by allowing an application to handle multiple client requests in parallel. Instead of using single-threaded *passive objects*, which execute their methods in the thread of control of the client that invoked the methods, concurrent objects reside in their own thread of control. However, if objects run concurrently we must synchronize access to their methods and data if these objects are shared by multiple client threads. In the presence of this problem, three forces arise:

1. Methods invoked on an object concurrently should not block the entire process in order to prevent degrading the QoS of other methods: For instance, if one outgoing TCP connection to a consumer in our Gateway example becomes blocked due to flow control, the Gateway process should still be able to queue up new messages while waiting for flow control to abate. Likewise, if other outgoing TCP connections are *not* flow controlled, they should be able to send messages to their consumers independently of any blocked connections.

2. Synchronized access to shared objects should be simple: Applications like the Gateway example are often hard to program if developers must explicitly use low-level synchronization mechanisms, such as acquiring and releasing mutual exclusion (mutex) locks. In general, methods that are subject to synchronization constraints should be serialized transparently when an object is accessed by multiple client threads.

3. Applications should be designed to transparently leverage the parallelism available on a hardware/software platform: In our Gateway example, messages destined for different consumers should be sent in parallel by a Gateway over different TCP connections. If the entire Gateway is programmed to only run in a single thread of control, however, performance bottlenecks cannot be alleviated transparently by running the Gateway on a multi-processor.

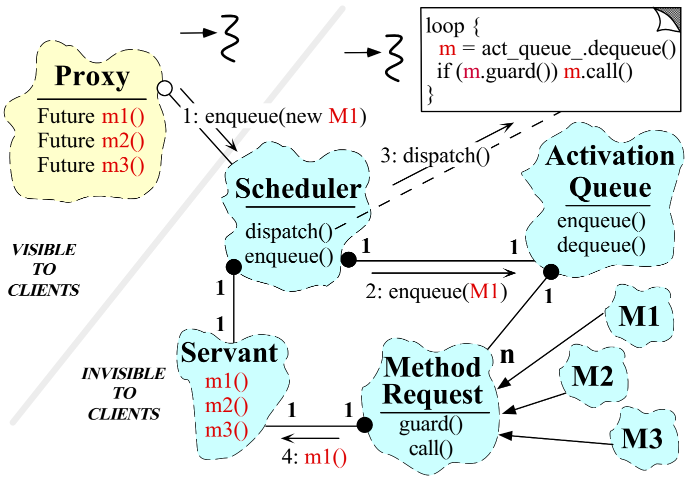
6 Solution

For each object that requires concurrent execution, decouple method invocation on the object from method execution. This decoupling is designed so the client thread appears to invoke an ordinary method. This method is automatically converted into a method request object and passed to another thread of control, where it is converted back into a method and executed on the object implementation.

An active object consists of the following components. A *Proxy* [5, 2] represents the interface of the object and a *Servant* [6] provides the object's implementation. Both the Proxy and the Servant run in separate threads so that method invocation and method execution can run concurrently: the proxy runs in the client thread, while the servant runs in a different thread. At run-time, the Proxy transforms the client's method invocation into a *Method Request*, which is stored in an *Activation Queue* by a *Scheduler*. The Scheduler runs continuously in the same thread as the servant, dequeueing Method Requests from the Activation Queue when they become runnable and dispatching them on the Servant that implements the Active Object. Clients can obtain the results of a method's execution via the *Future* returned by the Proxy.

7 Structure

The structure of the Active Object pattern is illustrated in the following Booch class diagram:



There are six key participants in the Active Object pattern:

Proxy

- A Proxy [2, 5] provides an interface that allows clients to invoke publically accessible methods on an Active Object using standard, strongly-typed programming language features, rather than passing loosely-typed messages between threads. When a client invokes a method defined by the Proxy, this triggers the construction and queueing of a Method Request object onto the Scheduler's Activation Queue, all of which occurs in the client's thread of control.

Method Request

- A Method Request is used to pass *context information* about a specific method invocation on a Proxy, such as method parameters and code, from the Proxy to a Scheduler running in a separate thread. An abstract Method Request class defines an interface for executing methods of an Active Object. The interace also contains *guard* methods that can be used to determine when a Method Request's synchronization constraints are met. For every Active Object method offered by the Proxy that requires synchronized access in its Servant, the abstract Method Request class is subclassed to create a concrete Method Request class. Instances of these classes are created by the proxy when its methods are invoked and contain the specific context information necessary to execute these method invocations and return any results back to clients.

Activation Queue

- An Activation Queue maintains a bounded buffer of pending Method Requests created by the Proxy. This queue keeps track of which Method Requests to execute. It also decouples the client thread from the servant thread so the two threads can run concurrently.

Scheduler

- A Scheduler runs in a different thread than its clients, managing an Activation Queue of Method Requests that are pending execution. A Scheduler decides which Method Request to dequeue next and execute on the Servant that implements this method. This scheduling decision is based on various criteria, such as *ordering*, *e.g.*, the order in which methods are inserted into the Activation Queue, and *synchronization constraints*, *e.g.*, the fulfillment of certain properties or the occurrence of specific events, such as space becoming available for new elements in a bounded data structure. A Scheduler typically evaluates synchronization constraints by using method request guards.

Servant

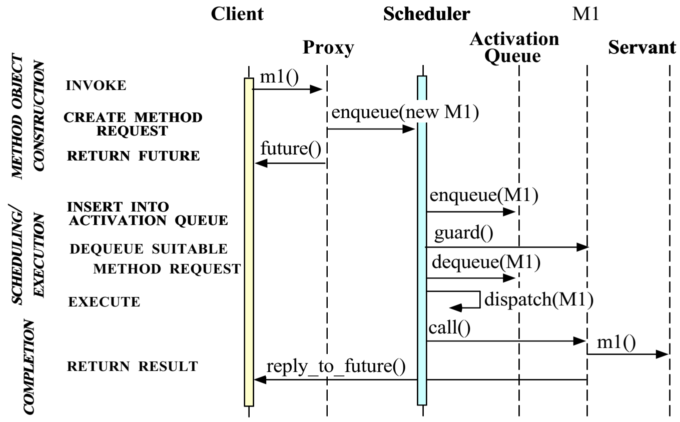
- A Servant defines the behavior and state that is being modeled as an Active Object. Servants implement the methods defined in the Proxy and the corresponding Method Requests. A Servant method is invoked when its corresponding Method Request is executed by a Scheduler; thus, Servants execute in the Scheduler's thread of control. Servants may provide other methods used by Method Requests to implement their guards.

Future

- A Future [7, 8] allows a client to obtain the results of method invocations after the Servant finishes executing the method. When a client invokes methods through a Proxy, a Future is returned immediately to the client. The Future reserves space for the invoked method to store its results. When a client wants to obtain these results, it can "rendezvous" with the Future, either blocking or polling until the results are computed and stored into the Future.

8 Dynamics

The following figure illustrates the three phases of collaborations in the Active Object pattern:



1. Method Request construction and scheduling: In this phase, the client invokes a method on the Proxy. This triggers the creation of a Method Request, which maintains the argument bindings to the method, as well as any other bindings required to execute the method and return its results. The Proxy then passes the Method Request to the Scheduler, which enqueues it on the Activation Queue. If the method is defined as a *two-way* [6], a binding to a Future is returned to the client that invoked the method. No Future is returned if a method is defined as a *oneway*, *i.e.*, it has no return values.

2. Method execution: In this phase, the Scheduler runs continuously in a different thread than its clients. Within this thread, the Scheduler monitors the Activation Queue and determines which Method Request(s) have become runnable, *e.g.*, when their synchronization constraints are met. When a Method Request becomes runnable, the Scheduler dequeues it, binds it to the Servant, and dispatches the appropriate method on the Servant. When this method is called, it can access/update the state of its Servant and create its result(s).

3. Completion: In the final phase, the results, if any, are stored in the Future and the Scheduler continues to monitor the Activation Queue for runnable Method Requests. After a two-way method completes, clients can retrieve its results by rendezvousing with the Future. In general, any clients that rendezvous with the Future can obtain its results. The Method Request and Future are deleted or garbage collected when they are no longer referenced.

9 Implementation

This section explains the steps involved in building a concurrent application using the Active Object pattern. The application implemented using the Active Object pattern is a portion of the Gateway from Section 3. Figure 2 illustrates the structure and participants in this example. The example in this section uses reusable components from the ACE framework [9]. ACE provides a rich set of reusable C++ wrappers and framework components that perform common communication software tasks across a wide range of OS platforms.

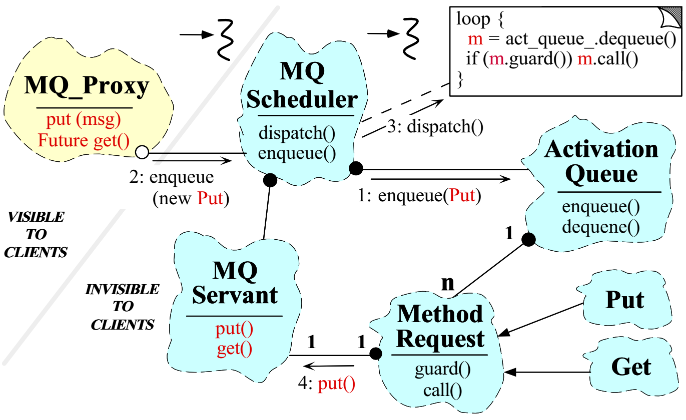


Figure 2: Implementing a Message Queue as an Active Object for `Consumer Handlers`

1. Implement the Servant: A Servant defines the behavior and state that is being modeled as an Active Object. The methods a Servant implements are accessible by clients via a Proxy. In addition, a Servant may contain other methods that Method Requests can use to implement guards that allow a Scheduler to evaluate run-time synchronization constraints. These constraints determine the order in which a Scheduler dispatches Method Requests.

In our Gateway example, the Servant is a message queue that buffers messages that are pending delivery to consumers. For each remote consumer, there is a `Consumer Handler` that contains a TCP connection to the consumer process. In addition, a `Consumer Handler` contains a message queue model as an Active Object and implemented with an `MQ_Servant`. Each `Consumer Handler`'s Active Object message queue stores messages passed from supplier to the Gateway while they are waiting to be sent to their remote consumer. The following class provides an interface for this Servant:

```
class MQ_Servant
{
public:
  MQ_Servant (size_t mq_size);

  // Message queue implementation operations.
  void put_i (const Message &msg);
  Message get_i (void);

  // Predicates.
  bool empty_i (void) const;
  bool full_i (void) const;

private:
  // Internal Queue representation, e.g., a
  // circular array or a linked list, etc.
};
```

The `put_i` and `get_i` methods implement the insertion and removal operations on the queue, respectively. In addition, the servant defines two *predicates*, `empty_i` and `full_i`, that distinguish three internal states: (1) empty, (2) full, and (3) neither empty nor full. These predicates are used in the implementation of the Method Request `guard` methods, which allow the Scheduler to enforce run-time synchronization constraints that dictate the order in which `put_i` and `get_i` methods are called on a Servant.

Note how the `MQ_Servant` class is designed so that synchronization mechanisms remain external to the Servant. For instance, in our Gateway example, the methods in the `MQ_Servant` class do not include any code that implements synchronization. This class only provides methods that implement the Servant's functionality and check its internal state. This design avoids the *inheritance anomaly* [10, 11, 12, 13] problem, which inhibits the reuse of Servant implementations if subclasses require different synchronization policies. Thus, a change to the synchronization constraints of the Active Object need not affect its servant implementation.

2. Implement the Proxy and Method Requests: The Proxy provides clients with an interface to the Servant's methods. For each method invocation by a client, the Proxy creates a Method Request. A Method Request is an abstraction for the context[1] of a method. This context typically includes the method parameters, a binding to the Servant the method will be applied to, a Future for the result, and the code for the Method Request's `call` method.

In our Gateway example, the `MQ_Proxy` provides an abstract interface to the `MQ_Servant` defined in Step 1. This message queue is used by a `Consumer Handler` to queue messages for delivery to consumers, as shown in Figure 2. In addition, the `MQ_Proxy` is a factory that constructs instances of Method Requests and passes them to a Scheduler, which queues them for subsequent execution in a separate thread. The C++ implementation of `MQ_Proxy` is shown below:

```
class MQ_Proxy
{
public:
  // Bound the message queue size.
  enum { MAX_SIZE = 100 };

  MQ_Proxy (size_t size = MAX_SIZE)
    : scheduler_ (new MQ_Scheduler (size)),
      servant_ (new MQ_Servant (size)) {}

  // Schedule <put> to execute on the active object.
  void put (const Message &m) {
    Method_Request *method_request =
```

[1] This context is often called a *closure*.

```
      new Put (servant_, m);
    scheduler_->enqueue (method_request);
  }

  // Return a Message_Future as the ``future''
  // result of an asynchronous <get>
  // method on the active object.
  Message_Future get (void) {
    Message_Future result;

    Method_Request *method_request =
      new Get (servant_, result);
    scheduler_->enqueue (method_request);
    return result;
  }

  // ... empty() and full() predicate implementations ...

protected:
  // The Servant that implements the
  // Active Object methods.
  MQ_Servant *servant_;

  // A scheduler for the Message Queue.
  MQ_Scheduler *scheduler_;
};
```

Each method of an MQ_Proxy transforms its invocation into a Method Request and passes the request to its MQ_Scheduler, which enqueues it for subsequent activation. A Method_Request base class defines virtual guard and call methods that are used by its Scheduler to determine if a Method Request can be executed and to execute the Method Request on its Servant, respectively, as follows:

```
class Method_Request
{
public:
  // Evaluate the synchronization constraint.
  virtual bool guard (void) const = 0;

  // Implement the method.
  virtual void call (void) = 0;
};
```

The methods in this class must be defined by subclasses, one subclass for each method defined in the Proxy. The rationale for defining these two methods is to provide Schedulers with a uniform interface to evaluate and execute concrete Method_Requests. Thus, Schedulers can be decoupled from specific knowledge of how to evaluate the synchronization constraints or trigger the execution of concrete Method_Request.

For instance, when a client invokes the put method on the Proxy in our Gatway example, this method is transformed into an instance of the Put subclass, which inherits from Method_Request and contains a pointer to the MQ_Servant, as follows:

```
class Put : public Method_Request
{
public:
  Put (MQ_Servant *rep,
       Message arg)
    : servant_ (rep), arg_ (arg) {}

  virtual bool guard (void) const {
    // Synchronization constraint: only allow
    // <put_i> calls when the queue is not full.
    return !servant_->full_i ();
  }

  virtual void call (void) {
    // Insert message into the servant.
    servant_->put_i (arg_);
  }

private:
  MQ_Servant *servant_;
  Message arg_;
};
```

Note how the guard method uses the MQ_Servant's full_i predicate to implement a synchronization constraint that allows the Scheduler to determine when the Put method request can execute. When a Put method request can be executed, the Scheduler invokes its call hook method. This call hook uses its run-time binding to the MQ_Servant to invoke the Servant's put_i method. This method is executed in the context of that Servant and does not require any explicit serialization mechanisms since the Scheduler enforces all the necessary synchronization constraints via the Method Request guards.

The Proxy also transforms the get method into an instance of the Get class, which is defined as follows:

```
class Get : public Method_Request
{
public:
  Get (MQ_Servant *rep,
       const Message_Future &f)
    : servant_ (rep), result_ (f) {}

  bool guard (void) const {
    // Synchronization constraint:
    // cannot call a <get_i> method until
    // the queue is not empty.
    return !servant_->empty_i ();
  }

  virtual void call (void) {
    // Bind the dequeued message to the
    // future result object.
    result_ = servant_->get_i ();
  }

private:
  MQ_Servant *servant_;

  // Message_Future result value.
  Message_Future result_;
};
```

For every two-way method in the Proxy that returns a value, such as the get_i method in our Gateway example, a Message_Future is returned to the client thread that calls it, as shown in implementation Step 4 below. The client may choose to evaluate the Message_Future's value immediately, in which case the client blocks until the method request is executed by the scheduler. Conversely, the evaluation of a return result from a method invocation on an Active Object can be deferred, in which case the client thread and the thread executing the method can proceed asynchronously.

3. Implement the Activation Queue: Each Method Request is enqueued on an Activation Queue. This is typically implemented as a thread-safe bounded-buffer that is shared between the client threads and the thread where the Scheduler and Servant run. An Activation Queue also provides an iterator that allows the Scheduler to traverse its elements in accordance with the Iterator pattern [5].

The following C++ code illustrates how the `Activation_Queue` is used in the Gateway:

```
class Activation_Queue
{
public:
  // Block for an "infinite" amount of time
  // waiting for <enqueue> and <dequeue> methods
  // to complete.
  const int INFINITE = -1;

  // Define a "trait".
  typedef Activation_Queue_Iterator
          iterator;

  // Constructor creates the queue with the
  // specified high water mark that determines
  // its capacity.
  Activation_Queue (size_t high_water_mark);

  // Insert <method_request> into the queue, waiting
  // up to <msec_timeout> amount of time for space
  // to become available in the queue.
  void enqueue (Method_Request *method_request,
                long msec_timeout = INFINITE);

  // Remove <method_request> from the queue, waiting
  // up to <msec_timeout> amount of time for a
  // <method_request> to appear in the queue.
  void dequeue (Method_Request *method_request,
                long msec_timeout = INFINITE);

private:
  // Synchronization mechanisms, e.g., condition
  // variables and mutexes, and the queue
  // implementation, e.g., an array or a linked
  // list, go here.
  // ...
};
```

The `enqueue` and `dequeue` methods provide a "bounded-buffer producer/consumer" concurrency model that allows multiple threads to simultaneously insert and remove `Method_Requests` without corrupting the internal state of an `Activation_Queue`. One or more client threads play the role of producers, enqueueing `Method_Requests` via a Proxy. The Scheduler thread plays the role of consumer, dequeueing `Method_Requests` when their `guards` evaluate to "true" and invoking their `call` hooks to execute Servant methods.

The `Activation_Queue` is designed as a bounded-buffer using condition variables and mutexes [14]. Therefore, the Scheduler thread will block for `msec_timeout` amount of time when trying to remove `Method_Requests` from an empty `Activation_Queue`. Likewise, client threads will block for up to `msec_timeout` amount of time when they try to insert onto a full `Activation_Queue`, *i.e.*, a queue whose current `Method_Request` count equals its high water mark. If an `enqueue` method times out, control returns to the client thread and the method is not executed.

4. Implement the Scheduler: A Scheduler maintains the Activation Queue and executes pending Method Requests whose synchronization constraints are met. The public interface of a Scheduler typically provides one method for the Proxy to enqueue Method Requests into the Activation Queue and another method that dispatches method requests on the Servant. These methods run in separate threads, *i.e.*, the Proxy runs in different threads than the Scheduler and Servant, which run in the same thread.

In our Gateway example, we define an `MQ_Scheduler` class, as follows:

```
class MQ_Scheduler
{
public:
  // Initialize the Activation_Queue to have the
  // specified capacity and make the Scheduler
  // run in its own thread of control.
  MQ_Scheduler (size_t high_water_mark);

  // ... Other constructors/destructors, etc.,

  // Insert the Method Request into
  // the Activation_Queue.  This method
  // runs in the thread of its client, i.e.,
  // in the Proxy's thread.
  void enqueue (Method_Request *method_request) {
    act_queue_->enqueue (method_request);
  }

  // Dispatch the Method Requests on their Servant
  // in the Scheduler's thread.
  virtual void dispatch (void);

protected:
  // Queue of pending Method_Requests.
  Activation_Queue *act_queue_;

  // Entry point into the new thread.
  static void *svc_run (void *arg);
};
```

The Scheduler executes its `dispatch` method in a different thread of control than its client threads. These client threads make the Proxy enqueue `Method Requests` in the Scheduler's `Activation_Queue`. The Scheduler monitors its `Activation_Queue` in its own thread, selecting a `Method_Request` whose *guard* evaluates to "true," *i.e.*, whose synchronization constraints are met. This `Method_Request` is then executed by invoking its `call` hook method. Note that multiple client threads can share the same Proxy. The Proxy methods need not be thread-safe since the Scheduler and Activation Queue handle concurrency control.

For instance, in our Gateway example, the constructor of `MQ_Scheduler` initializes the `Activation_Queue` and spawns a new thread of control to run the `MQ_Scheduler`'s `dispatch` method, as follows:

```
MQ_Scheduler (size_t high_water_mark)
  : act_queue_ (new Activation_Queue
                (high_water_mark))
{
  // Spawn a separate thread to dispatch
  // method requests.
  Thread_Manager::instance ()->spawn (svc_run,
                                      this);
}
```

This new thread executes the svc_run static method, which is simply an adapter that calls the dispatch method, as follows:

```
void *
MQ_Scheduler::svc_run (void *args)
{
  MQ_Scheduler *this_obj =
    reinterpret_cast<MQ_Scheduler *> (args);

  this_obj->dispatch ();
}
```

The dispatch method determines the order that Put and Get method requests are processed based on the underlying MQ_Servant predicates empty_i and full_i. These predicates reflect the state of the Servant, such as whether the message queue is empty, full, or neither. By evaluating these predicate constraints via the Method Request guard methods, the Scheduler can ensure fair shared access to the MQ_Servant, as follows:

```
virtual void
MQ_Scheduler::dispatch (void)
{
  // Iterate continuously in a
  // separate thread.
  for (;;) {
    Activation_Queue::iterator i;

    // The iterator's <begin> call blocks
    // when the <Activation_Queue> is empty.
    for (i = act_queue_->begin ();
         i != act_queue_->end ();
         i++) {
      // Select a Method Request 'mr'
      // whose guard evaluates to true.
      Method_Request *mr = *i;

      if (mr->guard ()) {
        // Remove <mr> from the queue first
        // in case <call> throws an exception.
        act_queue_->dequeue (mr);
        mr->call ();
        delete mr;
      }
    }
  }
}
```

In our Gateway example, the dispatch implementation of the MQ_Scheduler class continuously executes the next Method_Request whose guard evaluates to true. Scheduler implementations can be more sophisticated, however, and may contain variables that represent the synchronization state of the Servant. For example, to implement a multiple-readers/single-writer synchronization policy several counter variables can be stored in the Scheduler to keep track of the number of read and write requests. The Scheduler can use these counts to determine when a single writer can proceed, that is, when the current number of readers is 0 and no other writer is currently running. Note that the counter values are independent of the Servant's state since they are only used by the Scheduler to enforce the correct synchronization policy on behalf of the Servant.

5. Determine rendezvous and return value policies: The rendezvous policy determines how clients obtain return values from methods invoked on active objects. A rendezvous policy is required since Active Object servants do not execute in the same thread as clients that invoke their methods. Implementations of the Active Object pattern typically choose from the following rendezvous and return value policies:

1. *Synchronous waiting* – Block the client thread synchronously in the Proxy until the Method Request is dispatched by the Scheduler and the result is computed and stored in the future.
2. *Synchronous timed wait* – Block only for a bounded amount of time and fail the result of a two-way call is not returned within the allocated time period. If the timeout is zero the client thread "polls," *i.e.*, it returns to the caller without queueing the Method Request if the Scheduler cannot dispatch it immediately.
3. *Asynchronous* – Queue the Method Request and return control to the client thread immediately. If the method is a two-way call that produces a result then some form of Future mechanism must be used to provide synchronized access to the value (or to the error status if the method call fails).

The Future construct allows two-way asynchronous invocations that return a value to the client. When a Servant completes the method execution, it acquires a write lock on the Future and updates the Future with a result value. Any client threads that are currently blocked waiting for the result value are awakened and may access the result value concurrently. A Future object can be garbage collected after the writer and all readers no longer reference the Future. In languages like C++, which do not support garbage collection natively, the Future objects can be reclaimed when they are no longer in use via idioms like Counter Pointer [2].

In our Gateway example, the get method invoked on the MQ_Proxy ultimately results in the Get::call method being dispatched by the MQ_Scheduler, as shown in Step 2 above. Since the MQ_Proxy get method returns a value, a Message_Future is returned when the client calls it. The Message_Future is defined as follows:

```
class Message_Future
public:
  // Copy constructor binds <this> and <f> to the
  // same <Message_Future_Rep>, which is created if
  // necessary.
  Message_Future (const Message_Future &f);

  // Constructor that initializes <Message_Future> to
  // point to <Message> <m> immediately.
  Message_Future (const Message &m);

  // Assignment operator that binds <this> and <f>
  // to the same <Message_Future_Rep>, which is
  // created if necessary.
  void operator= (const Message_Future &f);

  // ... other constructors/destructors, etc.,

  // Type conversion, which blocks
```

```
  // waiting to obtain the result of the
  // asynchronous method invocation.
  operator Message ();
};
```

The Message_Future is implemented using the Counted Pointer idiom [2]. This idiom simplifies memory management for dynamically allocated C++ objects by using a reference counted Message_Future_Rep *body* that is accessed solely through the Message_Future *handle*.

In general, a client can obtain the Message result value from a Message_Future object in either of the followings ways:

- **Immediate evaluation:** The client may choose to evaluate the Message_Future's value immediately. For example, a Gateway Consumer Handler running in a separate thread may choose to block until new messages arrive from suppliers, as follows:

```
MQ_Proxy mq;
// ...

// Conversion of Message_Future from the
// get() method into a Message causes the
// thread to block until a message is
// available.
Message msg = mq.get ();

// Transmit message to the consumer.
send (msg);
```

- **Deferred evaluation:** The evaluation of a return result from a method invocation on an Active Object can be deferred. For example, if messages are not available immediately, a Consumer Handler can store the Message_Future return value from mq and perform other "bookkeeping" tasks, such as exchanging *keep-alive messages* to ensure its consumer is still active. When the Consumer Handler is done with these tasks it can block until a message arrives from suppliers, as follows:

```
// Obtain a future (does not block the client).
Message_Future future = mq.get ();

// Do something else here...

// Evaluate future in the conversion operator;
// may block if the result is not available yet.
Message msg = Message (future);
```

10 Example Resolved

Internally, the Gateway software contains Supplier and Consumer Handlers that act as local proxies [2, 5] for remote suppliers and consumers, respectively. As shown in Figure 3, Supplier Handlers receive messages from remote suppliers, inspect address fields in the messages, and use the address as a key into a Routing Table that identifies which remote consumer should receive the message. The Routing Table maintains a map of Consumer

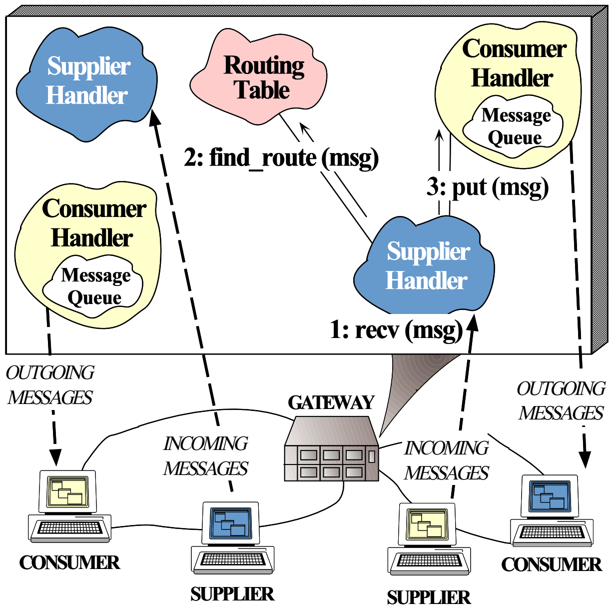


Figure 3: Communication Gateway

Handlers, each of which is responsible for delivering messages to its remote consumer over a separate TCP connection.

To handle flow control over various TCP connections, each Consumer Handler contains a Message Queue implemented using the Active Object described in Section 9. The Consumer_Handler class is defined as follows:

```
class Consumer_Handler
{
public:
  Consumer_Handler (void);

  // Put the message into the queue.
  void put (const Message &msg) {
    message_queue_.put (msg);
  }

private:
  // Proxy to the Active Object.
  MQ_Proxy message_queue_;

  // Connection to the remote consumer.
  SOCK_Stream connection_;

  // Entry point into the new thread.
  static void *svc_run (void *arg);
};
```

Supplier Handlers running in their own threads put messages into the appropriate Consumer Handler's Message Queue, as follows:

```
Supplier_Handler::route_message (const Message &msg)
{
  // Locate the appropriate consumer based on the
  // address information in the Message.
  Consumer_Handler *ch =
    routing_table_.find (msg.address ());

  // Put the Message into the Consumer Handler's queue.
  ch->put (msg);
};
```

To process the messages placed into its message queue, each Consumer_Handler spawns a separate thread of control in its constructor, as follows:

```
Consumer_Handler::Consumer_Handler (void)
{
  // Spawn a separate thread to get messages
  // from the message queue and send them to
  // the consumer.
  Thread_Manager::instance ()->spawn (svc_run,
                                       this);
}
```

This new thread executes the svc_run method, which gets the messages placed into the queue by Supplier Handler threads and sends them to the consumer over the TCP connection, as follows:

```
void *
Consumer_Handler::svc_run (void *args)
{
  Consumer_Handler *this_obj =
    reinterpret_cast<Consumer_Handler *> (args);

  for (;;) {
    // Conversion of Message_Future from the
    // get() method into a Message causes the
    // thread to block until a message is
    // available.
    Message msg = this_obj->message_queue_.get ();

    // Transmit message to the consumer.
    this_obj->connection_.send (msg);
  }
}
```

Since the message queue is implemented as an Active Object the send operation can block in any given Consumer_Handler object without affecting the quality of service of other Consumer_Handlers.

11 Variants

The following are variations of the Active Object pattern.

Integrated Scheduler: To reduce the number of components needed to implement the Active Object pattern, the roles of the Proxy and Servant are often integrated into the Scheduler component, though servants still execute in a different thread than the proxies. Moreover, the transformation of the method call into a Method Request can also be integrated into the Scheduler. For instance, the following is another way to implement the Message Queue example using an integrated Scheduler:

```
class MQ_Scheduler
public:
  MQ_Scheduler (size_t size)
    : act_queue_ (new Activation_Queue (size))
  {}

  // ... other constructors/destructors, etc.,

  void put (const Message &msg) {
    Method_Request *method_request =
      // The <MQ_Scheduler> is the servant.
      new Put (this, msg);
    act_queue_->enqueue (method_request);
  }

  Message_Future get (void) {
    Message_Future result;

    Method_Request *method_request =
      // The <MQ_Scheduler> is the servant.
      new Get (this, result);
    act_queue_->enqueue (method_request);
    return result;
  }

  // ...
private:
  // Message queue servant operations.
  void put_i (const Message &msg);
  Message get_i (void);

  // Predicates.
  bool empty_i (void) const;
  bool full_i (void) const;

  Activation_Queue *act_queue_;
  // ...
};
```

By centralizing where Method Requests are generated, the pattern implementation can be simplified since there are fewer components. The drawback, of course, is that the Scheduler must know the type of the Servant and Proxy, which makes it hard to reuse a Scheduler for different types of Active Objects.

Message passing: A further refinement of the integrated Scheduler variant is to remove the Proxy and Servant altogether and use direct *message passing* between the client thread and the Scheduler thread, as follows:

```
class Scheduler
public:
  Scheduler (size_t size)
    : act_queue_ (new Activation_Queue (size))
  {}

  // ... other constructors/destructors, etc.,

  // Enqueue a Message Request in the thread of
  // the client.
  void enqueue (Message_Request *message_request) {
    act_queue_->enqueue (message_request);
  }

  // Dispatch Message Requests in the thread of
  // the Scheduler.
  virtual void dispatch (void) {
    Message_Request *mr;

    // Block waiting for next request to arrive.
    while (act_queue_->dequeue (mr)) {
      // Process the message request <mr>.
    }
  }
}

protected:
  Activation_Queue *act_queue_;
  // ...
};
```

In this design, there is no Proxy, so clients simply create an appropriate type of Message_Request and call enqueue, which inserts the request into the Activation_Queue. Likewise, there is no Servant, so the dispatch method running in the Scheduler's thread simply dequeues the next Message_Request and processes the request according to its type.

In general, it is easier to develop a message passing mechanism than it is to develop an Active Object since there are fewer components to develop. However, message passing is typically more tedious and error-prone since application developers are responsible for programming the Proxy and Servant logic, rather than letting the Active Object developers write this code.

Polymorphic futures: A Polymorphic Future [15] allows parameterization of the eventual result type represented by the Future and enforces the necessary synchronization. In particular, a Polymorphic Future result value provides write-once, read-many synchronization. Whether a client blocks on a future depends on whether or not a result value has been computed. Hence, a Polymorphic Future is partly a reader-writer condition synchronization pattern and partly a producer-consumer synchronization pattern.

The following class illustrates a polymorphic future template in C++:

```
template <class T>
class Future
{
  // This class implements a 'single write, multiple
  // read' pattern that can be used to return results
  // from asynchronous method invocations.
public:
  // Constructor.
  Future (void);

  // Copy constructor that binds <this> and <r> to
  // the same <Future> representation
  Future (const Future<T> &r);

  // Destructor.
  ~Future (void);

  // Assignment operator that binds <this> and
  // <r> to the same <Future>.
  void operator = (const Future<T> &r);

  // Cancel a <Future>. Put the future into its
  // initial state. Returns 0 on success and -1
  // on failure.
  int cancel (void);

  // Type conversion, which obtains the result
  // of the asynchronous method invocation.
  // Will block forever until the result is
  // obtained.
  operator T ();

  // Check if the result is available.
  int ready (void);

private:
  Future_Rep<T> *future_rep_;
  // Future representation implemented using
  // the Counted Pointer idiom.
};
```

A client can use a polymorphic future as follows:

```
// Obtain a future (does not block the client).
Future<Message> future = mq.get ();

// Do something else here...

// Evaluate future in the conversion operator;
// may block if the result is not available yet.
Message msg = Message (future);
```

Distributed Active Object: In this variant, a distribution boundary exists between the Proxy and the Scheduler, rather than a threading boundary, as with the conventional Active Object pattern. Therefore, the client-side Proxy plays the role of a *stub*, which is responsible for marshaling the method parameters into a Method Request format that can be transmitted across a network and executed by a Servant in a separate address space. In addition, this variant also typically introduces the notion of a server-side *skeleton*, which performs demarshaling on the Method Request parameters before they are passed to a Servant method in the server.

Thread pool: A *thread pool* is a generalization of Active Object that supports multiple Servants per Active Object. These Servants can offer the same services to increase throughput and responsiveness. Every Servant runs in its own thread and actively ask the Scheduler to assign a new request when it is ready with its current job. The Scheduler then assigns a new job as soon as one is available.

12 Known Uses

The following are specific known uses of the Active Object pattern:

CORBA ORBs: The Active Object pattern has been used to implement concurrent ORB middleware frameworks, such as CORBA [6] and DCOM [16]. For instance, the TAO ORB [17] implements the Active Object pattern for its default concurrency model [18]. In this design, CORBA stubs correspond to the Active Object pattern's Proxies, which transform remote operation invocations into CORBA Requests. The TAO ORB Core's Reactor is the Scheduler and the socket queues in the ORB Core correspond to the Activation Queues. Developers create Servants that execute the methods in the context of the server. Clients can either make synchronous two-way invocations, which block the calling thread until the operation returns, or they can make asynchronous method invocations, which return a Poller future object that can be evaluated at a later point [19].

ACE Framework: Reusable implementations of the Method Request, Activation Queue, and Future components in the Active Object pattern are provided in the ACE framework [9]. These components have been used to implement many production distributed systems.

Siemens MedCom: The Active Object pattern is used in the Siemens MedCom framework, which provides a black-box component-oriented framework for electronic medical systems [20]. MedCom employ the Active Object pattern in conjunction with the Command Processor pattern to simplify client windowing applications that access patient information on various medical servers.

Siemens Call Center management system: This system uses the thread pool variant of the Active Object pattern.

Actors: The Active Object pattern has been used to implement Actors [21]. An Actor contains a set of instance variables and behaviors that react to messages sent to an Actor by other Actors. Messages sent to an Actor are queued in the Actor's message queue. In the Actor model, messages are executed in order of arrival by the "current" behavior. Each behavior nominates a replacement behavior to execute the next message, possibly before the nominating behavior has completed execution. Variations on the basic Actor model allow messages in the message queue to be executed based on criteria other than arrival order [22]. When the Active Object pattern is used to implement Actors, the Scheduler corresponds to the Actor scheduling mechanism, Method Request correspond to the behaviors defined for an Actor, and the Servant is the set of instance variables that collectively represent the state of an Actor [23]. The Proxy is simply a strongly-typed mechanism used to pass a message to an Actor.

13 Consequences

The Active Object pattern provides the following benefits:

Enhance application concurrency and simplify synchronization complexity: Concurrency is enhanced by allowing client threads and asynchronous method executions to run simultaneously. Synchronization complexity is simplified by the Scheduler, which evaluates synchronization constraints to guarantee serialized access to Servants, depending on their state.

Transparently leverage available parallelism: If the hardware and software platforms support multiple CPUs efficiently, this pattern can allow multiple active objects to execute in parallel, subject to their synchronization constraints.

Method execution order can differ from method invocation order: Methods invoked asynchronously are executed based on their synchronization constraints, which may differ from their invocation order.

However, the Active Object pattern has the following liabilities:

Performance overhead: Depending on how the Scheduler is implemented, *e.g.*, in user-space vs. kernel-space, context switching, synchronization, and data movement overhead may occur when scheduling and executing active object method invocations. In general, the Active Object pattern is most applicable on relatively coarse-grained objects. In contrast, if the objects are very fine-grained, the performance overhead of active objects can be excessive, compared with other concurrency patterns such as Monitors.

Complicated debugging: It may be difficult to debug programs containing active objects due to the concurrency and non-determinism of the Scheduler. Moreover, many debuggers do not support concurrent applications adequately.

14 See Also

The Monitor pattern ensures that only one method at a time executes within a Passive Object, regardless of the number of threads that invoke this object's methods concurrently. Monitors are generally more efficient than Active Objects since they incur less context switching and data movement overhead. However, it is harder to add a distribution boundary between client and server threads using the Monitor pattern.

The Reactor pattern [24] is responsible for demultiplexing and dispatching of multiple event handlers that are triggered when it is possible to initiate an operation without blocking. This pattern is often used in lieu of the Active Object pattern to schedule callback operations to passive objects. It can also be used in conjunction of the Active Object pattern to form the Half-Sync/Half-Async pattern described in the next paragraph.

The Half-Sync/Half-Async pattern [25] is an architectural pattern that decouples synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency. This pattern typically uses the Active Object pattern to implement the Synchronous task layer, the Reactor pattern [24] to implement the Asynchronous task layer, and a Producer/Consumer pattern to implement the Queueing layer.

The Command Processor pattern [2] is similar to the Active Object pattern. Its intent is to separate the issuing of requests from their execution. A command processor, which corresponds to the scheduler, maintains pending service requests, which are implemented as Commands [5]. These are executed on suppliers, which correspond to servants. The Command Processor pattern does not focus on concurrency, however, and clients, the command processor, and suppliers reside in the same thread of control. Thus, there are no proxies that represent the servants to clients. Clients create commands and pass them directly to the command processor.

The Broker pattern [2] has many of the same components as the Active Object pattern. In particular, clients access Brokers via Proxies and servers implement remote objects via Servants. The primary difference between the Broker pattern and the Active Object pattern is that there's a distribution boundary between proxies and servants in the Broker pattern vs. a threading boundary between proxies and servants in the Active Object pattern.

The Mutual Exclusion (Mutex) pattern [26] is a simple locking pattern that is often used in lieu of Active Objects when implementing concurrent Passive Objects, also known as Monitors. The Mutex pattern can occur in slightly different forms, such as a spin lock or a semaphore. The Mutex pattern can have various semantics, such as recursive mutexes and priority inheritance mutexes.

Acknowledgements

The genesis for the Active Object pattern originated with Greg Lavender. Thanks to Frank Buschmann, Hans Rohnert, Martin Botzler, Michael Stal, Christa Schwanninger, and Greg Gallant for extensive comments that greatly improved the form and content of this version of the pattern description.

References

[1] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[3] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, vol. 2, no. 1, 1996.

[4] W. R. Stevens, *TCP/IP Illustrated, Volume 1*. Reading, Massachusetts: Addison Wesley, 1993.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[6] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[7] R. H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. Programming Languages and Systems*, vol. 7, pp. 501–538, Oct. 1985.

[8] B. Liskov and L. Shrira, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 260–267, June 1988.

[9] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6^{th} USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[10] P. America, "Inheritance and Subtyping in a Parallel Object-Oriented Language," in *ECOOP'87 Conference Proceedings*, pp. 234–242, Springer-Verlag, 1987.

[11] D. G. Kafura and K. H. Lee, "Inheritance in Actor-Based Concurrent Object-Oriented Languages," in *ECOOP'89 Conference Proceedings*, pp. 131–145, Cambridge University Press, 1989.

[12] S. Matsuoka, K. Wakita, and A. Yonezawa, "Analysis of Inheritance Anomaly in Concurrent Object-Oriented Languages," *OOPS Messenger*, 1991.

[13] M. Papathomas, "Concurrency Issues in Object-Oriented Languages," in *Object Oriented Development* (D. Tsichritzis, ed.), pp. 207–245, Centre Universitaire D'Informatique, University of Geneva, 1989.

[14] W. R. Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[15] R. G. Lavender and D. G. Kafura, "A Polymorphic Future and First-Class Function Type for Concurrent Object-Oriented Programming in C++," in *Forthcoming*, 1995. http://www.cs.utexas.edu/users/lavender/papers/futures.ps.

[16] D. Box, *Essential COM*. Addison-Wesley, Reading, MA, 1997.

[17] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[18] D. C. Schmidt, "Evaluating Architectures for Multi-threaded CORBA Object Request Brokers," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

[19] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed., May 1998.

[20] P. Jain, S. Widoff, and D. C. Schmidt, "The Design and Performance of MedJava – Experience Developing Performance-Sensitive Distributed Applications with Java," *IEE/BCS Distributed Systems Engineering Journal*, 1998.

[21] G. Agha, *A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[22] C. Tomlinson and V. Singh, "Inheritance and Synchronization with Enabled-Sets," in *OOPSLA'89 Conference Proceedings*, pp. 103–112, Oct. 1989.

[23] D. Kafura, M. Mukherji, and G. Lavender, "ACT++: A Class Library for Concurrent Programming in C++ using Actors," *Journal of Object-Oriented Programming*, pp. 47–56, October 1992.

[24] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[25] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Proceedings of the 2^{nd} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–10, September 1995.

[26] Paul E. McKinney, "A Pattern Language for Parallelizing Existing Programs on Shared Memory Multiprocessors," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.