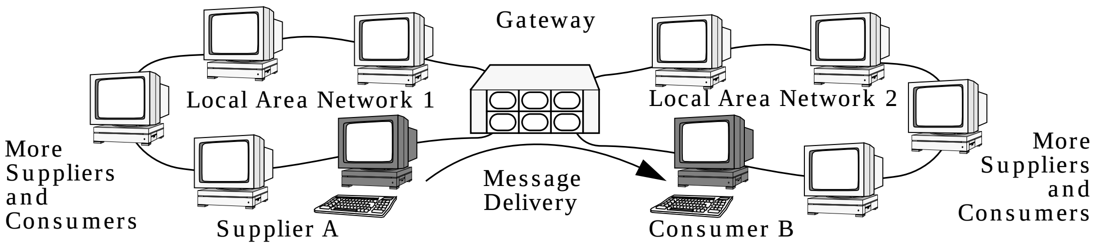
Active Object

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

Also Known As Concurrent Object, Actor

Example 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. For example, a gateway may route messages from one or more supplier processes to one or more consumer processes in a distributed system.



The suppliers, consumers, and gateway communicate using TCP, which is a connection-oriented protocol [Ste93]. 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 while 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 improve performance is to introduce *concurrency* into the gateway design. In a concurrent appli-

1. See the Acceptor-Connector pattern (117) for further details on this example.

cation, the thread of control of an object *O* that executes a method can be decoupled from the threads of control that invoke methods on *O*. 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.

Context Clients that access objects running in separate threads of control.

Problem Many applications benefit from using concurrent objects to improve their QoS, for instance, 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:

- Methods invoked on an object concurrently should not block the entire process to prevent degrading the QoS of other methods.

  ➥ For instance, if one outgoing TCP connection in our gateway example is 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. ❏

- 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.

- 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. ❑

Solution For each object exposed to the above forces, 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 invocation is converted automatically into a method request object and passed to another thread of control, where it is later converted back into a method invocation that is executed on the object implementation.

An Active Object consists of the following components. A *proxy* [POSA1] [GHJV95] represents the interface of the object and a *servant* [OMG98b] provides the object's implementation. Both the proxy and the servant run in separate threads so that method invocations and method executions can run concurrently, that is, 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 invocations into *method requests*, which are stored in an *activation queue* by a *scheduler*. The scheduler's event loop runs continuously in the same thread as the servant, dequeueing method requests from the activation queue and dispatching them on the servant. Clients can obtain the result of a method's execution via a *future* returned by the proxy.

Structure There are six key participants in the Active Object pattern:

A *proxy* [POSA1] [GHJV95] 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.

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 the methods of an Active Object. This interface 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 a 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 result back to clients.

Class	***Collaborator***
Proxy	• Method Request • Scheduler • Future
Responsibility	
• Defines the Active Object's interface to clients • Creates Method Request • Runs in the client's thread	

Class	***Collaborator***
Abstract Method Request	• Servant • Future
Responsibility	
• Represents a method call on the Active Object • Provides guards to check when the method request becomes runnable	

Class	***Collaborator***
Concrete Method Request	• Servant • Future
Responsibility	
• Implements the representation of a specific method call	

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.

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*, for example the order in which methods are inserted into the activation queue, or *synchronization constraints*, for instance the fulfillment of certain properties or the occurrence of specific events, such as space becoming available for new elements in a data structure with a bounded size. A scheduler typically evaluates synchronization constraints by using the method requests' guards.

Class Activation Queue	***Collaborator***
Responsibility • Maintains method requests pending for execution • Runs in the Active Object's thread	

Class Scheduler	***Collaborator*** • Activation Queue • Method Request
Responsibility • Executes method requests • Runs in the Active Object's thread	

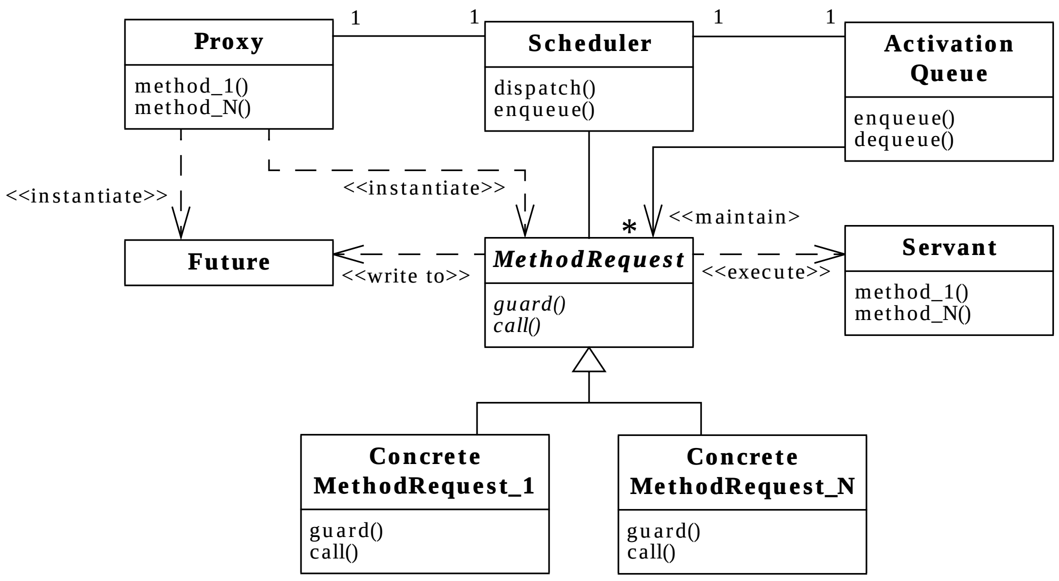
A *servant* defines the behavior and state that is being modeled as an Active Object. A servant implements the methods defined in the proxy and the corresponding method requests. A servant method is invoked when its corresponding method request is executed by a scheduler; thus, a servant executes in the scheduler's thread of control. A servant may provide other predicate methods that can be used by method requests to implement its guards.

When a client invokes a method on a proxy, a *future* is returned immediately to the client. A *future* [Hal85] [LS88] allows a client to obtain the result of method invocations after the servant finishes executing the method. The future reserves space for the invoked method to store its result. When a client wants to obtain this result, it can *rendezvous* with the future, either blocking or polling until the result is computed and stored into the future.

Class Servant	***Collaborator***
Responsibility • Implements the Active Object • Runs in the Active Object thread	

Class Future	***Collaborator***
Responsibility • Contains the result of a method call on an Active Object	

The UML class diagram for the Active Object pattern looks as follows:

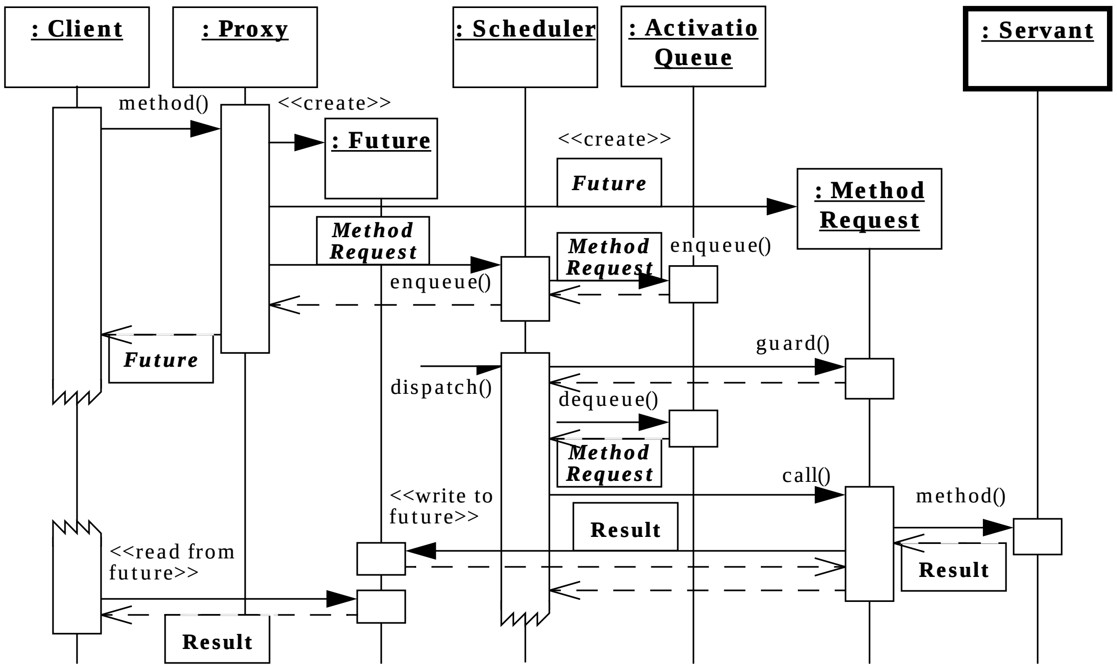


Dynamics The behavior of the Active Object pattern can be divided into three phases:

Method request construction and scheduling. A 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 result. 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 [OMG98b], a binding to a future is returned to the client. No future is returned if a method is defined as a *one-way*, which means it has no return values.

Method request execution. The scheduler runs continuously in a different thread than its clients. In this thread, the scheduler monitors its activation queue and determines which method request(s) have become runnable, for example, when their synchronization constraints are met. When a method request becomes runnable, the scheduler dequeues it, binds the request to the servant, and dispatches the appropriate method on the servant. When this method is called, it can access and update the state of its servant and create its result, if it's a two-way method.

Completion. In the final phase, the result, if any, is 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 result via the future. In general, any clients that rendezvous with the future can obtain its result. The method request and future are explicitly deleted or garbage collected when they are no longer referenced.



Implementation The following steps illustrate how to implement the Active Object pattern.

1 *Implement the servant.* A servant defines the behavior and state that are being modeled as an Active Object. The methods it implements are accessed indirectly by clients via a proxy. In addition, the servant may contain other methods used by method requests to implement guards that allow the scheduler to evaluate run-time synchronization constraints. These constraints determine the order in which a scheduler dispatches method requests.

➥ Our gateway example uses a number of servants of type `MQ_Servant` to buffer messages that are pending delivery to consumers. For each remote consumer, there is a *consumer handler* that contains a TCP connection to a consumer process. In addition, a consumer handler contains a message queue modeled as an Active Object and implemented with an MQ_Servant. Each consumer handler's Active Object message queue stores messages passed from suppliers to the gateway while they are waiting to be sent to the remote consumer.

The following class illustrates the interface for MQ_Servant:

```
class MQ_Servant {
public:
     // Constructor
     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 message 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: empty, full, and neither empty nor full. These predicates are used in the implementation of the method request *guard* methods, which allow a 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 the 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* problem [MWY91], 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[2] of the 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 executing the method request.

➥ In our gateway example, the MQ_Proxy provides an abstract interface to the MQ_Servant defined in step 1. In addition, it 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.

```
class MQ_Proxy {
public:
    // Bound the message queue size.
    const int MQ_MAX_SIZE = 100;

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

    // Schedule <put> to execute on the Active Object.
    void put (const Message &msg) {
        Method_Request *method_request =
            new Put (servant_, msg);
        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 =
            new Get (servant_, result);
        scheduler_->enqueue (method_request);
        return result;
    }

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

2. The context is often called a *closure* of a method.

```
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 the scheduler, which enqueues it for subsequent activation. A Method_Request base class defines virtual guard() and call() methods that are used by the 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;

    // Execute 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 which allows to decouple them from specific knowledge of how to evaluate the synchronization constraints or trigger the execution of concrete method requests.

Note that multiple client threads in a process can share the same proxy. For instance, several supplier handlers can access the proxy that belongs to a specific consumer handler. The proxy methods need not be thread-safe since the scheduler and activation queue handle serialization.

➥ When a client invokes the put() method on the proxy in our gateway example, this method call 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 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. Then, its scheduler invokes the method request's `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 method request `guard()` methods. Likewise, the proxy transforms the `get()` method into an instance of the `Get` class:

```
     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 ();
          }

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

     private:
          MQ_Servant *servant_;
          Message_Future result_;
     };
```

For every two-way method in the proxy that returns a value, such as the `get()` method in our gateway example, a `Message_Future` is returned to the client thread that calls it (see implementation step 4). This `Message_Future` is implemented using the Counted Pointer idiom [POSA1], which uses a reference counted pointer to a dynamically allocated `Message_Future_Rep` *body* that is accessed via the `Message_Future` *handle*. ❑

A 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 both proceed asynchronously.

3 *Implement the activation queue.* Each method request is enqueued on an activation queue. This queue 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 provides an iterator that allows the scheduler to traverse its elements in accordance with the Iterator pattern [GHJV95].

➥ For our gateway example we specify a class `Activation_Queue` as follows:

```
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
     // specificied 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 and enqueue `Method_Requests` via a proxy. The scheduler thread plays the role of a consumer, dequeueing `Method_Requests` when their `guards` evaluate to 'true' and invoking their `call()` hooks to execute Servant methods.

The `Activation_Queue` is also designed using concurrency control patterns like Monitor (113) and synchronization mechanisms, such as condition variables and mutexes [Ste97]. Therefore, the scheduler thread will block for up to `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 into a full `Activation_Queue`, that is, 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 request 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, that is, the proxy runs in a different thread 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 a 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 of control.
     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);
};
```
❏

As we have said, a 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 the activation queue in its own thread, selecting a method request whose *guard* evaluates to 'true', that is, whose synchronization constraints are met. This method request is then executed by invoking its `call()` hook method.

➥ 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 `MQ_Scheduler::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 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> method 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
                    act_queue_->dequeue (mr);
                    mr->call ();
                    delete mr;
                }
                // ... Other scheduling activities can go
                // here, e.g., to handle the case where no
                // Method_Requests in the Activation_Queue
                // have guard() methods that evaluate
                // to true.
            }
        }
    }
```
❏

In our example, the `MQ_Scheduler::dispatch()` implementation 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 a prospective writer will call 'write' on the proxy, passing the data to write. Likewise, readers will call 'read' and obtain a future as return value. The scheduler maintains several counter variables that keep track of the number of read and write requests. In addition, it maintains knowledge about the identity of the prospective writers. 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 write request from a different writer is currently pending execution. When such a write request arrives, the scheduler may choose to dispatch this writer to enhance fairness. In contrast, when read requests arrive and the servant can satisfy these read requests, that is, it is not empty, the scheduler can block all writing activity and dispatch read requests first.

The counter variable values described 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. The servant focuses solely on its task to temporarily house client-specific application data, whereas the scheduler focuses on coordinating multiple readers and writers. Thus, modularity and reusability is enhanced.

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:

- *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.
- *Synchronous timed wait*. Block only for a bounded amount of time and fail if the scheduler does not dispatch the method request within that time period. If the timeout is zero the client thread

'polls,' that is, it returns to the caller without queueing the method request if the scheduler cannot dispatch it immediately.

- *Asynchronous*. Queue the method call and return control to the client immediately. If the method is a two-way that produces a result then some form of future 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 its result. Any client threads that are currently blocked waiting for the result are awakened and can access the result concurrently. A future 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, futures can be reclaimed when they are no longer in use via idioms like Counted Pointer [POSA1].

➥ 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 implementation step 2, page 95. Since the `MQ_Proxy::get()` method returns a value, a `Message_Future` is returned to the client that 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 ();
private:
    // Message_Future_Rep uses the "Counted Pointer"
    // idiom.
    Message_Future_Rep *future_rep_;
};
```

The Message_Future is implemented using the Counted Pointer idiom [POSA1]. 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 result value from a future either by *immediate* evaluation or by *deferred* evaluation.

➥ For example, a consumer handler in our gateway example running in a separate thread may choose to block until new messages arrive from suppliers, as follows:

```
MQ_Proxy message_queue;
// ...

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

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

Conversely, if messages are not available immediately, a consumer handler can store the Message_Future return value from message_queue 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:

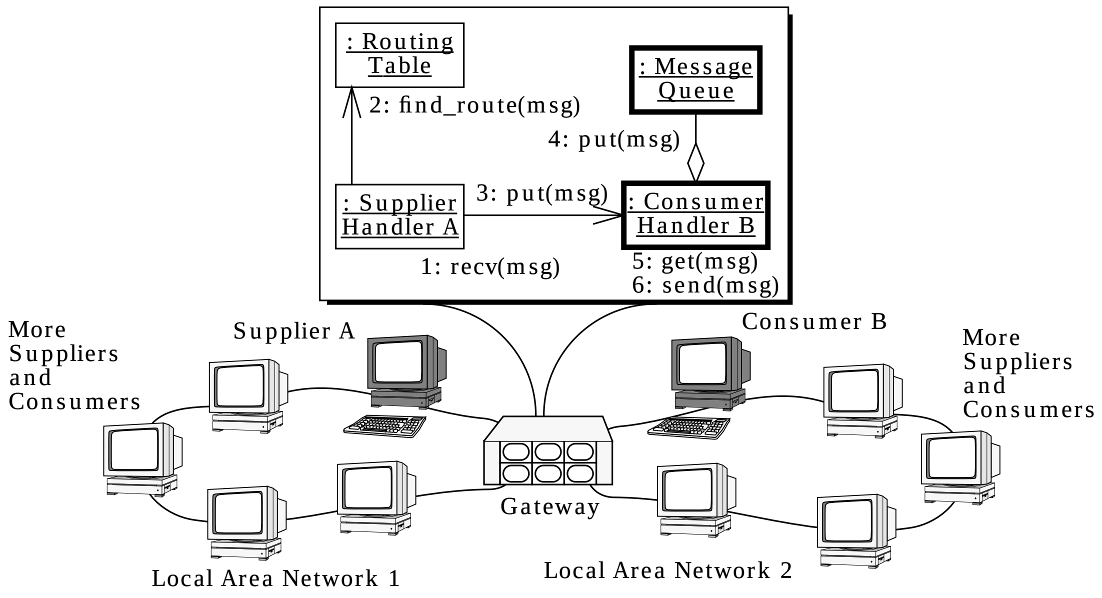
```
// Obtain a future (does not block the client).
Message_Future future = message_queue.get ();

// Do something else here...

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

Example Resolved

Internally, the Gateway software contains supplier and consumer handlers that act as local proxies [POSA1] for remote suppliers and consumers, respectively.



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 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 pattern, as described in the *Implementation* section. 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 inserted 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 entry point, 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 over the
            // TCP connection.
            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`.

Variants *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 on the proxy into a method request can also be integrated into the scheduler. For instance, the following is another way to implement the message queue example:

```
class MQ_Scheduler
public:
    MQ_Scheduler (size_t size)
        :   servant_ (new MQ_Servant (size)),
            act_queue_ (new Activation_Queue (size)) {}
    // ... other constructors/destructors, etc.,

    void put (const Message m) {
        Method_Request *method_request =
            new Put (servant_, m);
        act_queue_->enqueue (method_request);
    }
    Message_Future get (void) {
        Message_Future result;

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

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

By centralizing where method requests are generated, the Active Object 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.,

    void enqueue (Message_Request *message_request) {
        act_queue_->enqueue (message_request);
    }

    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 variant, there is no proxy, so clients themselves 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 also 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 [LK95]. A polymorphic future allows parameterization of the eventual result type represented by the future and enforces the necessary synchronization. In particular, a polymorphic future describes a typed future result value that 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 for 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> representation.
    void operator = (const Future<T> &r);

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

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

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

private:
    // ...
};
```

A client can use a polymorphic future as follows:

```
// Obtain a future (does not block the client).
Future<Message> future = message_queue.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 just a threading boundary, as with the Active Object pattern described earlier. Therefore, the solution is similar to the Broker pattern [POSA1] where 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 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 the Active Object pattern 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 indicates to the scheduler when it is ready with its current job. The scheduler then assigns a new method request to that servant as soon as one is available.

Additional variants of Active Objects can be found in [Lea96], chapter 5: *Concurrency Control* and chapter 6: *Services in Threads*.

Known Uses **Object Request Broker**. The Active Object pattern has been used to implement concurrent ORB middleware frameworks, such as CORBA [OMG98b] and DCOM [Box97]. For instance, the TAO ORB [POSA4] implements the Active Object pattern for its default concurrency model. In this design, each client's CORBA stub corresponds to an Active Object pattern's proxy, which transforms remote operation invocations into CORBA `Requests`. These requests are sent to the server process where they are inserted into socket queues. The TAO ORB Core's `Reactor` is the scheduler and the socket queues in the operating system correspond to the activation queues. Developers create a servant that executes 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 [OMG98b].

ACE Framework [Sch97]. Reusable implementations of the method request, activation queue, and future components in the Active Object pattern are provided in the ACE framework. The corresponding classes are called `ACE_Method_Request`, `ACE_Activation_Queue`, and `ACE_Future`. These components have been used to implement many production distributed systems.

Siemens MedCom [JWS98]. The Active Object pattern is used in the Siemens MedCom framework, which provides a blackbox component-oriented framework for electronic medical systems. MedCom employs the Active Object pattern in conjunction with the Command Processor pattern [POSA1] to simplify client windowing applications that access patient information on various medical servers.

Siemens FlexRouting - Automatic Call Distribution [Flex98]. This call center management system uses the Thread Pool variant of the Active Object pattern. Services which a call center offers are implemented as applications of their own. For example, there may be a hotline application, an ordering application, and a product information application—dependent of the types of service offered. These applications support agents[3] in serving the various possible customer requests. Each instance of these applications is a separate servant component. A 'FlexRouter' component, which corresponds to the scheduler, automatically dispatches incoming customer requests to agent applications that are able to serve these requests.

Actors [Agha86]. The Active Object pattern has been used to implement Actors. 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

3. In the context of this paragraph an agent is a human person.

messages in the message queue to be executed based on criteria other than arrival order [ToSi89]. When the Active Object pattern is used to implement Actors, the scheduler corresponds to the Actor scheduling mechanism, method requests 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 [KML92]. The proxy is simply a strongly-typed mechanism used to pass a message to an Actor.

Consequences The Active Object pattern provides the following **benefits**:

Enhances application concurrency and simplifies 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.

Transparent leveraging of 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, for example in user-space versus 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 related concurrency patterns such as Monitors (113).

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.

See Also The Monitor pattern (113) 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 (59) 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 in order to schedule callback operations to passive objects. It can also be used in conjunction with the Reactor pattern to form the Half-Sync/Half-Async pattern (113).

The Half-Sync/Half-Async pattern (113) 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 (59) to implement the asynchronous task layer, and a Producer/Consumer pattern [Lea96], such as a variant of the Pipes and Filters architecture [POSA1] to implement the queueing layer.

The Command Processor pattern [POSA1] separates issuing requests from their execution. A command processor, which corresponds to the Active Object pattern's scheduler, maintains pending service requests, which are implemented as Commands [GHJV95]. These are executed on suppliers, which correspond to servants. The Command Processor pattern does not focus on concurrency, however, since clients, the command processor and suppliers reside in the same thread of control. There are no proxies that represent the servants to clients. Clients create commands and pass them directly to the command processor.

The Broker pattern [POSA1] 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 is a distribution boundary between proxies and servants in the Broker pattern versus a threading boundary between proxies and servants in the Active Object pattern.

The Mutual Exclusion (Mutex) pattern [McK95] is a simple locking pattern that can occur in slightly different forms, such as a spin lock or a semaphore. Mutexes are often used by an Active Object implementation to serialize access to its Activation Queue. The Mutex pattern can have various semantics, such as recursive mutexes and priority mutexes.

Credits The genesis for the Active Object pattern originated with Greg Lavender. Ward Cunningham helped us with shaping the [PLoP95] version of Active Object. Bob Laferriere provided useful hints for improving the clarity of the pattern's implementation section.