第6章 主动对象（Active Object）：用于并发编程的对象行为模式

R. Greg Lavender Douglas C. Schmidt

摘 要

本论文描述主动对象（Active Object）模式。该模式使方法执行与方法调用去耦合，以简化对驻留在它自己的线程控制中的对象的同步访问。主动对象模式允许一或多个交错访问数据的独立执行的线程被建模为单个对象。这一并发模式能良好地适用于广泛的生产者/消费者和读者/作者应用类。该模式通常用于需要多线程服务器的分布式系统中。此外，客户应用，比如窗口系统和网络浏览器，采用主动对象来简化并发和异步的网络操作。

6.1 意图

主动对象设计模式使方法执行与方法调用去耦合，以增强并发、并简化对驻留在它自己的线程控制中的对象的同步访问。

6.2 别名

并发对象和Actor。

6.3 例子

为演示主动对象模式，考虑一个通信网关[1]的设计。网关使协作的组件去耦合，并允许它们进行交互，而无需彼此直接依赖[2]。图6-1中所示的网关在分布式系统中将来自一或多个供应者进程的消息路由到一或多个消费者进程[3]。

在我们的例子中，网关、供应者和消费者在面向连接的协议TCP[4]之上进行通信。因此，当网关软件尝试向远地消费者发送数据时，可能会遇到来自TCP传输层的流控制。TCP使用流控制来确保快速的生产者或网关不会过快地生产出数据，以致慢速消费者或拥挤的网络不能缓冲和处理这些数据。

为了改善所有供应者和消费者的端到端服务质量（QoS），整个网关进程不能在任何到消费者的连接上阻塞以等待流控制缓解。此外，当供应者和消费者的数目增加时，网关还必须能高效地扩展，

防止阻塞并提高性能的一种有效的方法是在网关设计中引入并发。并发应用允许执行对象O的方法的线程控制与调用O的方法的线程控制去耦合。而且，在网关中使用并发还使TCP连接被流控制的线程的阻塞不会阻碍TCP连接未被流控制的线程的执行。

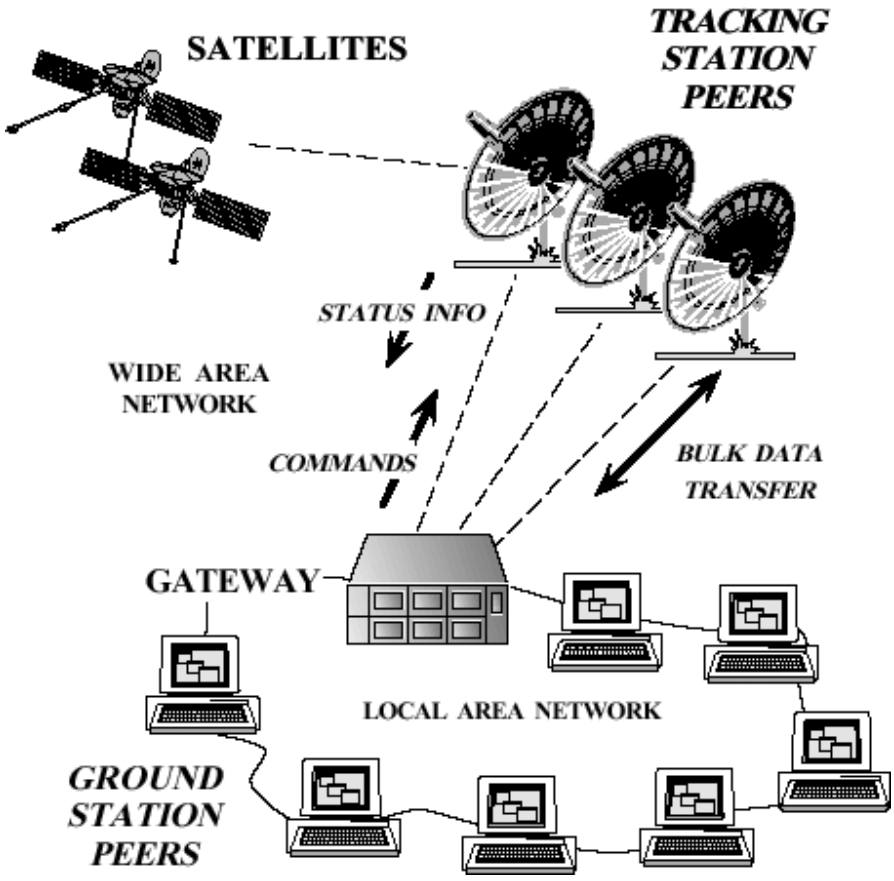


图6-1 通信网关

6.4 上下文

对运行在相互分离的线程控制中的对象进行访问的客户。

6.5 问题

许多应用受益于使用并发对象来改善它们的QoS，例如，通过允许应用并行地处理多个客户请求。并发对象驻留在它们自己的线程控制中，而不是使用单线程*被动对象*索这些对象在调用其方法的客户的线程控制中执行它们的方法。但是，如果对象并发执行，且这些对象被多个客户线程共享，我们必须同步对它们的方法和数据的访问。在存在这样的问题时，会产生三种压力：

1. **对对象方法的并发调用不应阻塞整个进程，以免降低其他方法的**QoS**：**例如，如果在我们的网关例子中，一个外出的到消费者的TCP连接因为流控制而阻塞，网关进程仍应该能在等待流控制缓解的同时，排队新的消息。同样地，如果其他外出的TCP连接*没有*被流控制，它们应该能独立于任何阻塞连接发送消息给它们的消费者。
2. **对共享对象的同步访问应该很简单：**如果开发者必须显式地使用低级同步机制，比如像获取和释放互斥锁（mutex），常常难于对网关这样的应用进行编程。一般而言，当对象被多个客户线程访问时，需进行同步约束的方法应该被透明地序列化。
3. **应用应设计为能透明地利用硬件/软件平台上可用的并行机制：**在我们的网关例子中，发往不同消费者的消息应该被网关并行地在不同的TCP连接上发送。但是，如果整个网关被编写为仅在单个线程控制中运行，性能瓶颈就不可能通过在多处理器上运行网关而被透明地克服。

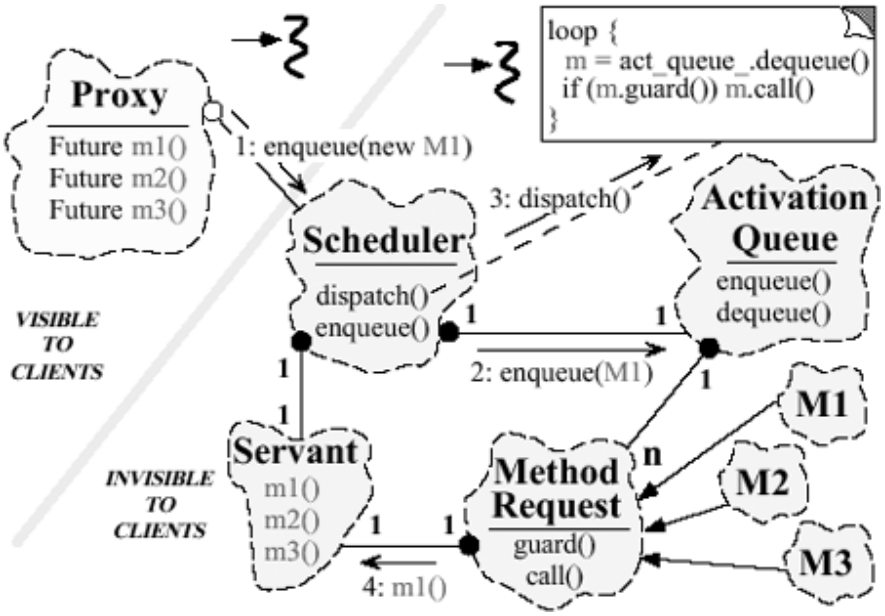
6.6 解决方案

对于每个需要并发执行的对象，使对对象方法的请求与方法执行去耦合。这样的去耦合被设计用于使客户线程看起来像是调用一个平常的方法。该方法被自动转换为方法请求对象，并传递给另一个线程控制，在其中它又被转换回方法，并在对象实现上被执行。

主动对象由以下组件组成：*代理*（Proxy）[5, 2]表示对象的接口，*仆人*（Servant）提供对象的实现。代理和仆人运行在分离的线程中，以使方法调用和方法执行能并发运行：代理在客户线程中运行，而仆人在不同的线程中运行。在运行时，代理将客户的方法调用（Method Invocation）转换为*方法请求*（Method Request），并由*调度者*（Scheduler）将其存储在*启用队列*（Activation Queue）中。调度者持续地运行在与仆人相同的线程中，当启用队列中的方法请求变得可运行时，就将它们出队，并分派给实现主动对象的仆人。客户可通过代理返回的“*期货*”（future）获取方法执行的结果。

6.7 结构

主动对象模式的结构在下面的Booch类图中演示：



在主动对象模式中有六个关键的参与者：

代理（Proxy）

- 代理提供一个接口，允许客户使用标准的强类型程序语言特性，而不是在线程间传递松散类型的消息，来调用主动对象的可公共访问的方法。当客户调用代理定义的方法时，就会在调度者的启用队列上触发方法请求对象的构造和排队；所有这些都发生在客户的线程控制中。

方法请求（Method Request）

- 方法请求用于将代理上的特定方法调用的*上下文信息*，比如方法参数和代码，从代理传递给运行在分离线程中的调度者。抽象方法请求类为执行主动对象方法定义接口。该接口还包含*守卫*（guard）方法，可用于确定何时方法请求的同步约束已被满足。对于代理提供的每个主动对象方法（它们在其仆人中需要同步的访问），抽象方法请求类被子类化，以创建具体的方法请求类。这些类的实例在其方法被调用时由代理创建，并包含了执行这些方法调用和返回任何结果给客户所需的特定的上下文信息。

启用队列（Activation Queue）

- 启用队列维护一个有界缓冲区，内有代理创建的待处理的方法请求。该队列跟踪哪些方法请求将要执行。它还使客户线程与仆人线程去耦合，以使两个线程能并发运行。

调度者（Scheduler）

- 调度者运行在与其客户不同的线程中，它管理待处理的方法请求的启用队列。调度者决定下一个出队的方法请求，并在实现该方法的仆人上执行。这样的调度决策基于各种标准，比如像方法被插入到启用队列中的顺序；以及同步约束，例如特定属性的满足或特定事件的发生，比如在有界数据结构中有新的条目空间变得可用。调度者通常使用方法请求守卫来对同步约束进行求值。

仆人（Servant）

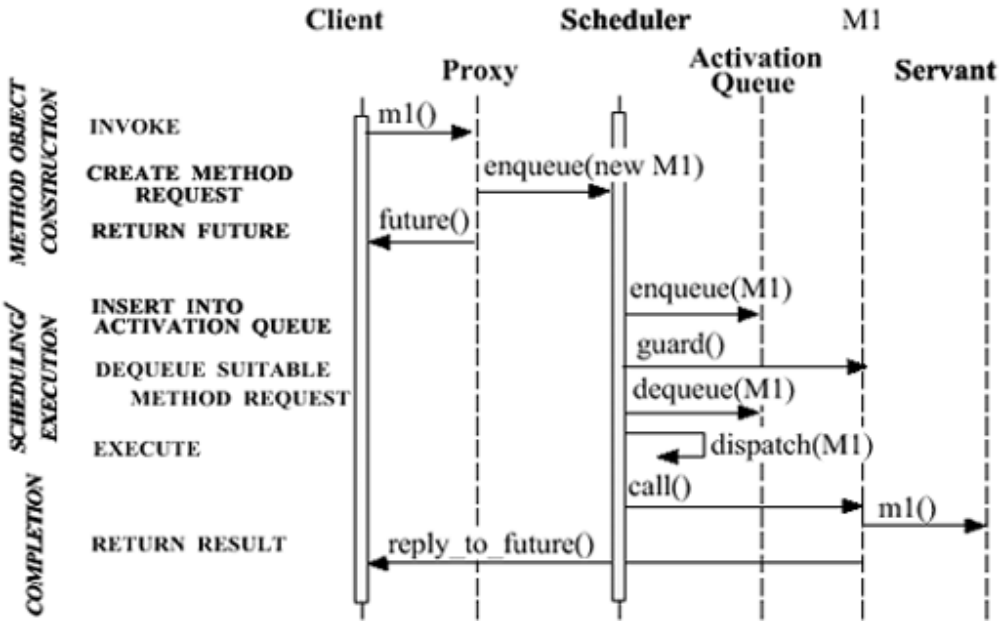
- 仆人定义被建模为主动对象的行为和状态。它实现在代理中定义的方法及相应的方法请求。仆人方法在调度者执行其相应的方法请求时被调用；因而，仆人在调度者的线程控制中执行。仆人还可提供其他方法，由方法请求用于实现它们的守卫。

- 期货[7, 8]允许客户在仆人结束方法的执行后获取方法调用的结果。当客户通过代理调用方法时，期货被立即返回给客户。期货为被调用的方法保留空间，以存储它的结果。当客户想要获取这些结果时，它可以阻塞或者轮询，直到结果被求值和存储到期货中，然后与期货“会合”。

6.8 动力特性

下图演示主动对象模式中的协作的三个阶段：

1. **方法请求构造和调度：**在此阶段，客户调用代理上的方法，从而触发方法请求的创建；方法请求维护方法的参数绑定，以及其他任何执行方法和返回结果所需的绑定。代理随后将方法请求传递给调度者，后者将其放入启用队列中。如果方法被定义为“*两路*”（two way）[6]的，一个期货的绑定被返回给调用该方法的客户。如果方法被定义为“*单路*”（oneway）的，就没有期货被返回，也就是，它没有返回值。
2. **方法执行：**在此阶段，调度者在与其客户不同的线程中持续运行。在此线程中，调度者监控启用队列，并确定哪些方法请求已成为可运行的，例如，当它们的同步约束已被满足时。当方法请求成为可运行的，调度者就使其出队，绑定到仆人，并分派仆人上的适当方法。当此方法被调用时，它可以访问/更新它的仆人的状态并创建它的结果。
3. **完成：**在最后的阶段中，结果（如果有的话）被存储在期货中，而调度者持续地监控启用队列中，看是否有可运行的方法请求。在一个两路方法完成后，客户可以通过与期货会合来获取它的结果。一般而言，任何与期货会合的客户都可以获取它的结果。当方法请求和期货不再被引用时，它们就被删除或被垃圾回收。



6.9 实现

这一部分解释使用主动对象模式构建并发应用所涉及的步骤。使用主动对象模式实现的应用是6.3网关的一部分。图6-2演示该例子的结构和参与者。这一部分中的例子使用ACE构架[9]的可复用组件。ACE提供了一组丰富的可复用C++包装和构架组件，可跨越广泛的OS平台执行常见的通信软件任务。

1. **实现仆人：**仆人定义被建模为主动对象的行为和状态。客户可通过代理来访问仆人所实现的方法。此外，仆人还可包含其他方法，方法请求可以用这些方法来实现守卫，以允许调度者对运行时同步约束进行求值。这些约束决定调度者分派方法请求的顺序。

   在我们的网关例子中，仆人是一个消息队列，缓冲待处理的递送给消费者的消息。对于每一个远地消费者，都有一个Consumer Handler（消费者处理器），其中含有一个到消费者进程的TCP连接。此外，Consumer Handler含有一个被建模为主动对象的消息队列，并通过MQ_Servant来实现。当从供应者传递到网关的消息在等待被发送到它们的远地消费者时，每个Consumer Handler的主动对象消息队列就存储这些消息。下面的类提供了这个仆人的接口：

```
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.

};
```

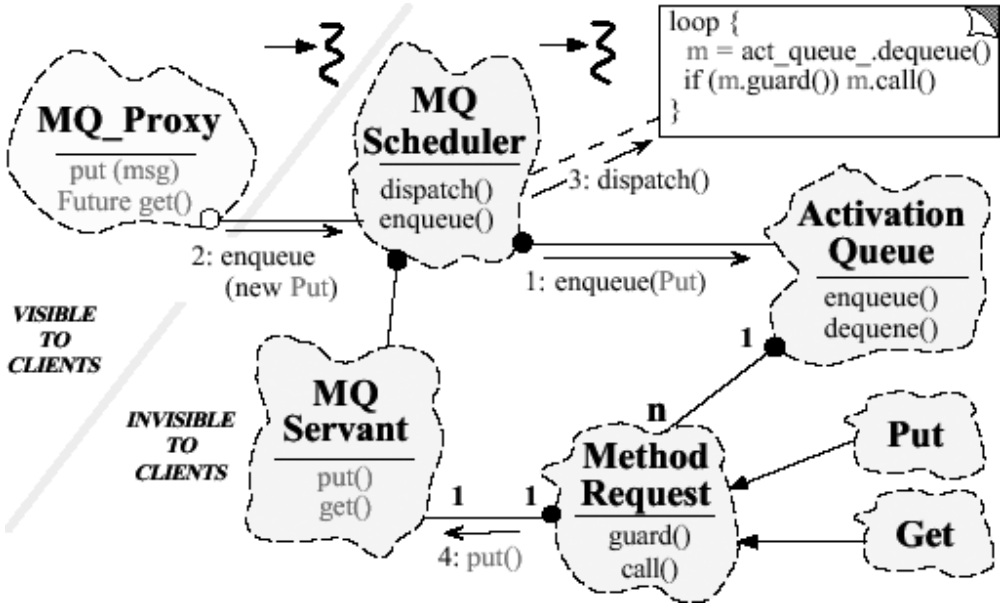


图6-2 将消费者处理器的消息队列实现为主动对象

put_i和get_i方法分别实现队列的插入和移除操作。此外，仆人还定义了两个断言（Predicate）：empty_i和full_i，可区分三种内部状态（1）空，（2）满，以及（3）既不为空也不为满。这些断言用于方法请求的看守方法的实现，后者允许调度者强制实施运行时同步约束；这些同步约束规定仆人的put_i和get_i的调用顺序。

注意MQ_Servant类是怎样设计，以使同步机制始终外在于仆人。例如，在我们的网关例子中，MQ_Servant类中的方法并不包括任何实现同步的代码。该类仅仅提供实现仆人功能和检查它的内部状态的方法。这样的设计避免了“*继承异常*”[10, 11, 12, 13]问题；如果子类需要不同的同步策略，该问题将会妨碍仆人实现的复用。因而，对主动对象的同步约束的改变不需要影响它的仆人实现。

2. **实现代理和方法请求：**代理为客户提供仆人方法的接口。对于客户的每一次方法调用，代理都会创建一个方法请求。方法请求是方法上下文的抽象。该上下文通常包括方法参数、到将要应用此方法的仆人的绑定、结果期货，以及方法请求的call方法的代码。

在我们的网关例子中，MQ_Proxy提供步骤1中定义的MQ_Servant的抽象接口。该消息队列被Consumer Handler用于排队递送给消费者的消息，如图6-2所示。此外，MQ_Proxy还是一个工厂，它构造方法请求的实例，并将它们传递给调度者，后者将它们排队，用于后面在分离的线程中的执行。MQ_Proxy的C++实现如下所示：

```
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 = 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_;

};
```

MQ_Proxy的每个方法都将它的调用转换为方法请求，并将其传递给它的MQ_Scheduler，后者将请求入队，用于后续的启用。Method_Request基类定义虚guard和call方法，分别被它的调度者用

于决定方法请求是否可被执行和在它的仆人上执行方法请求。如下所示：

```
class Method_Request

{

public:

// Evaluate the synchronization constraint.

virtual bool guard (void) const = 0;


// Implement the method.

virtual void call (void) = 0;

};
```

该类中的方法必须被子类定义，代理中定义的每个方法都有一个相应的Method_Request子类。定义这两个方法的原因是为调度者提供一个统一接口来计算和执行具体Method_Request。因而，调度者就得以与怎样计算同步约束、或是触发具体Method_Request执行的特定知识去耦合。

例如，在我们的网关例子中，当客户调用代理上的put方法时，该方法被转换为Put子类的实例；该子类继承自Method_Request，并含有指向MQ_Servant的指针。如下所示：

```
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_;

};
```

注意guard方法怎样使用MQ_Servant的full_I断言来实现同步约束，以允许调度者确定Put方法请求何时可以执行。当Put方法请求可被执行时，调度者调用它的call挂钩方法。该方法使用它的MQ_Servant运行时绑定来调用仆人的put_i方法。put_i方法在仆人的上下文中执行，并且不需要任何显式的序列化机制，因为调度者通过方法请求guard来强制实施所有必要的同步约束。

代理还将get方法转换为Get类的实例；Get类定义如下：

```
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_;

};
```

对于代理中所有返回值的两路方法，比如在我们的网关例子中的get_i方法，Message_Future被返回给调用该方法的客户线程，如下面的实现步骤4所示。客户可以选择立即对Message_Future的值进行求值，在这样的情况下，客户会阻塞、直到方法请求被调度者执行为止。相反，对主动对象方法调用的返回结果的求值也可被延期，在这样的情况下，客户线程和执行该方法的线程可以异步地执行。

3. **实现启用队列：**每个方法请求都被放入启用队列中。启用队列通常实现为线程安全的有界缓冲区，由客户线程与调度者及仆人的线程共享。启用队列还提供一个迭代器，允许调度者能依照迭代器模式[5]遍历它的元素。

   下面的C++代码演示Activation_Queue是怎样被用于网关中的：

```
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.

// ...

};
```

enqueue和dequeue方法提供一种“有界缓冲区生产者/消费者”并发模式，允许多个线程同时插入和移除Method_Request，而不会破坏Activation_Queue的内部状态。一或多个客户线程扮演生产者角色，通过代理将Method_Request入队。调度者线程扮演消费者角色，当Method_Request的guard方法求值为“真”时，将它们出队，并调用它们的call挂钩来执行仆人方法。

Activation_Queue被设计为使用条件变量和互斥体[14]的有界缓冲区。因此，当试图从空的Activation_Queue中移除Method_Request时，调度者将会阻塞msec_timeout长度的时间。同样地，当试图在满的Activation_Queue中（也就是，当前Method_Request数目等于其高水位标的队列）插入时，客户线程将会阻塞最多msec_timeout长度的时间。如果enqueue方法超时了，控制会返回给客户线程，而方法没有被执行。

4. **实现调度者：**调度者维护启用队列，并执行同步约束已满足的待处理方法请求。调度者的公共接口通常为代理提供一个方法，用于将方法请求放入启用队列中；还有另一个方法，用于在仆人上分派方法请求。这些方法运行在分离的线程中，也就是，代理运行在与调度者和仆人不同的线程中，而后两者运行在同一线程中。

   在我们的网关例子中，我们定义MQ_Scheduler类如下：

```
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);

};
```

调度者在与它的客户线程不同的线程控制中执行它的dispatch方法。这些客户线程驱使代理将方法请求放入调度者的Activation_Queue中。调度者在它自己的线程中监控它的Activation_Queue，选择其guard求值所得为“真”（也就是，同步约束已被满足）的Method_Request。于是这个Method_Request就通过调用它的call挂钩方法被执行。注意多个客户线程可以共享同一个代理。代理方法无需是线程安全的，因为调度者和启用队列会处理并发控制。

例如，在我们的网关例子中，MQ_Scheduler的构造器初始化Activation_Queue，并派生一个新线程来运行MQ_Scheduler的dispatch方法。如下所示：

```
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);

}
```

这个新线程执行svc_run静态方法，后者仅仅是一个调用dispatch方法的适配器。如下所示：

```
void *MQ_Scheduler::svc_run (void *args)

{

MQ_Scheduler *this_obj = reinterpret_cast<MQ_Scheduler *> (args);

this_obj->dispatch ();

}
```

dispatch方法基于底层的MQ_Servant断言empty_i和full_i来决定Put和Get方法请求的处理顺序。这些断言反映仆人的状态，比如消息队列是否为空、满，或都不是。通过经由方法请求的guard方法对这些断言约束进行求值，调度者可以确保对MQ_Servant的公平的共享访问。如下所示：

```
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;

}

}

}

}
```

在我们的网关例子中，MQ_Scheduler类的dispatch的实现持续地执行下一个其guard求值为真的Method_Request。但是，调度者实现还可以更为成熟，并且可以含有表示仆人同步状态的变量。例如，要实现一个多读者/单作者的同步策略，可在调度者中存储若干计数器变量，以跟踪读和写请求的数目。调度者可使用这些计数器来确定一个单个的作者何时可以继续执行，也就是，当目前的读者数目为0，而目前又没有其他作者在运行时。注意计数器的值独立于仆人的状态，因为后者仅仅被调度者用来代表仆人强制实施正确的同步策略。

5. **决定会合和返回值策略：**会合策略决定客户怎样从在主动对象上调用的方法那里获取返回值。之所以需要会合策略，是因为主动对象仆人并不在调用它们的方法的客户的线程里执行。主动对象的实现通常从以下的会合和返回值策略中进行选择：

    1. *同步的等待*：在代理中同步地阻塞客户线程，直到方法请求被调度者分派、结果被计算并存储在期货中。
    2. *同步的超时*：仅阻塞有限数量的时间，如果在指定的时间段里两路调用的结果没有返回，方法调用就会失败。如果timeout为零，客户线程就进行“轮询”，也就是，如果调度者不能立即分派方法请求，客户线程就返回调用者、而不使方法请求入队。
    3. *异步的*：排队方法请求，将控制立即返回给客户线程。如果该方法是要产生结果的两路调用，就必须使用某种形式的期货机制，以提供对结果值的同步的访问（或是错误状态，如果方法调用失败的话）。

    期货构造允许进行两路异步调用，它们返回值给客户。当仆人完成方法执行时，它获取期货上的写锁，用结果值更新期货。任何正阻塞等待该结果值的线程都被唤醒，并可以并发地访问结果值。期货对象可在作者和所有读者都不再引用它后被垃圾回收。在像C++这样不直接支持垃圾回收的语言里，期货对象可以在它们不再被使用时通过像计数器指针[2]这样的习语来进行回收。

    在我们的网关例子中，在MQ_Proxy上调用get方法最终导致Get::call方法被MQ_Scheduler分派，如上面的步骤2所示。因为MQ_Proxy get方法返回一个值，当客户调用它时会返回一个Message_Future。Message_Future被定义如下：

```
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 ();

};
```

Message_Future使用计数指针习语[2]来实现。通过使用引用计数的Message_Future_Rep *体*（通过Message_Future *句柄*单独进行访问），这种习语简化了动态分配的C++对象的内存管理。

一般而言，客户可通过下面两种方法中的一种来从Message_Future对象那里获取消息的结果值：

- **立即求值：**客户可以选择立即对Message_Future进行求值。例如，运行在分离线程中的网关的Consumer Handler可以选择进行阻塞，直到来自供应者的新消息到达。如下所示：

```
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);
```

- **延期求值：**对主动对象的方法请求的返回值的求值可以被延期。例如，如果消息不是立即可用，Consumer Handler可存储来自mq的Message_Future返回值，并执行其他的“簿记”任务，比如交换*keep-alive消息*，以确保它的消费者仍然是活动的。当Consumer Handler完成这些任务后，它可以进行阻塞，直到来自供应者的消息到达。如下所示：

```
// 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);
```

6.10 例子解答

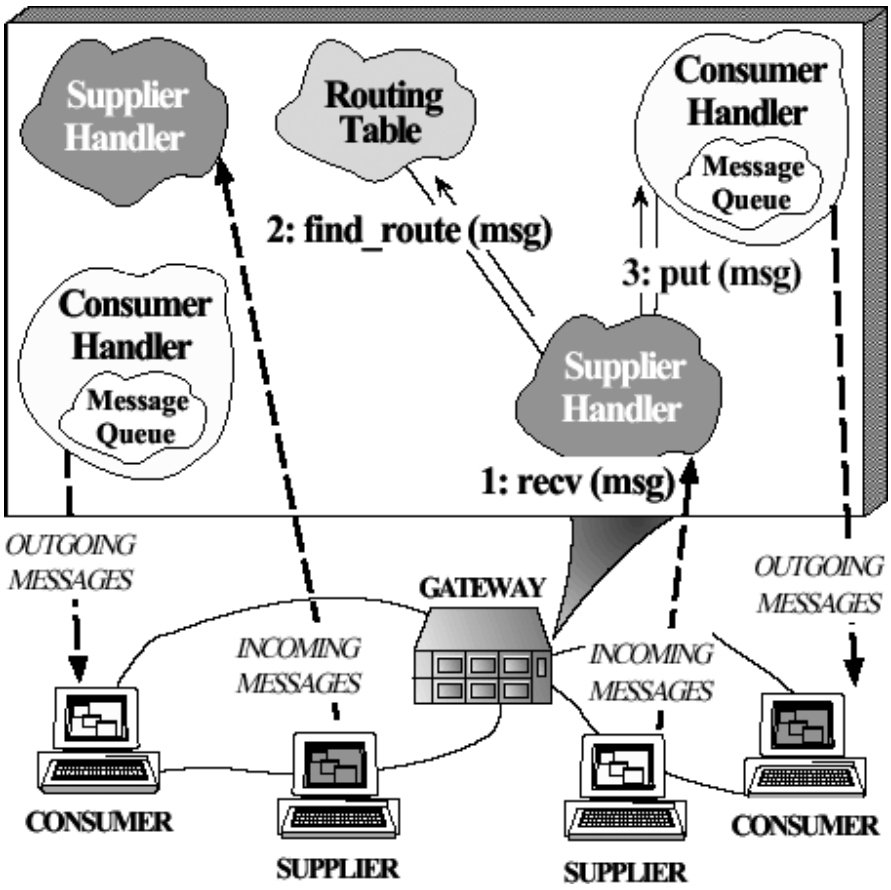


图6-3 通信网关

网关软件在内部包含有Supplier和Consumer Handler，分别用作远地供应者和消费者的本地代理[2, 5]。如图6-3所示，Supplier Handler接收来自远地供应者的消息，检查消息中的地址域，并将该地址用作Routing Table中的关键字；该关键字标识哪一个远地消费者应该接收该消息。Routing Table维护Consumer Handler的一个映射表；每一个Consumer Handler负责在各自的TCP连接上递送消息给它的远地消费者。

为在多个TCP连接上处理流控制，每个Consumer Handler都含有一个使用6.9中描述的主动对象来实现的消息队列。Consumer_Handler类被定义如下：

```
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);

};
```

在它们自己的线程中运行的Supplier Handler将消息放进适当的Consumer Handler的消息队列中。如下所示：

```
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);

};
```

为处理放置在消息队列中的消息，每个Consumer_Handler在它的构造器中都派生出一个单独的线程控制。如下所示：

```
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);

}
```

这个新线程执行svc_run方法，取得由Supplier Handler线程放置在队列中的消息，并在TCP连接上将它们发送给消费者。如下所示：

```
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);
}
}
```

因为消息队列被实现成主动对象，send操作可以在任何给定的Consumer_Handler对象中阻塞，而不会影响其他Consumer_Handler的服务质量。

6.11 变种

下面是主动对象模式的一些变种：

集成调度者：为减少实现主动对象模式所需组件的数目，代理和仆人的角色常常被集成进调度者组件中，尽管仆人仍然在与代理不同的线程中执行。而且，将方法调用转换为方法请求也可被集成进调度者。例如，下面是使用集成调度者来实现消息队列的另一种方法：

```
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_;

// ...

};
```

通过使生成方法请求的地方集中化，可以简化模式的实现，因为组件变少了。当然，缺点是调度者必须知道仆人和代理的类型，这使得开发者难以将调度者复用于不同类型的主动对象。

消息传递：一种更为精致的集成调度者变种是将代理和仆人一起去除，并在客户线程和调度者线程间使用直接的*消息传递*。如下所示：

```
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_;
// ...
};
```

在此设计中没有代理，于是客户简单地创建适当类型的Message_Request，并调用enqueue，将请求插入到Activation_Queue。同样地，也没有仆人，于是运行在调度者线程中的dispatch方法简单地使下一个Message_Request出队，并根据它的类型来进行处理。

一般而言，开发一种消息传递机制要比开发主动对象容易，因为需要开发的组件较少。但是，消息传递通常更为麻烦而易错，因为应用开发者要负责对代理和仆人逻辑进行编程，而不是让主动对象开发者来编写此代码。

多态期货：多态期货[15]允许对期货所代表的最终结果类型进行参数化，并会采取必要的同步。特别地，多态期货结果值提供一次写、多次读的同步。客户是否会阻塞在期货上取决于结果值是否已被计算。因此，多态期货部分地是一种读者/作者条件同步模式，部分地是生产者/消费者同步模式。

下面的类演示一种C++多态期货模板：

```
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.
};
```

客户可以这样来使用多态期货：

```
// 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);
```

分布式主动对象：在此变种中，在代理和调度者之间存在着分布式的边界，而不是传统的主动对象模式中的线程边界。因此，客户代理扮演*stub*的角色，负责将方法参数整编（marshal）为方法请求的参数格式；后者可跨越网络进行传输，并由在单独的地址空间中的仆人执行之。此外，该变种通常还引入服务器端的*skeleton*概念，在将方法请求的参数传递给服务器中的仆人方法之前对它们进行去整编。

线程池：*线程池*是一种泛化的主动对象，支持每个主动对象有多个仆人。这些仆人可以提供相同的服务，以提高吞吐量和响应能力。所有仆人都运行在自己的线程中，并在为当前作业作好准备时主动要求调度者分配新的请求。于是一旦有新的作业可做，调度者就会分配一个新作业。

6.12 已知应用

下面是已知的主动对象模式的特定使用：

CORBA ORB：主动对象模式已被用于实现并发ORB中间件构架，比如CORBA[6]和DCOM[16]。例如，TAO ORB[17]为它的缺省并发模型[18]实现了主动对象模式。在此设计中，CORBA stub对应主动对象模式的代理，前者将远地的操作调用转换为CORBA请求。TAO ORB核心的Reactor（反应堆）是调度者，而ORB核心中的socket队列对应着启用队列。开发者创建仆人，它在服务器的上下文中执行方法。客户可以进行同步的两路调用，它们会阻塞调用线程，直到操作返回；或是进行异步的方法调用，它们会返回一个Poller期货对象，可随后进行求值[19]。

ACE**构架：**ACE构架[9]提供了主动对象模式中的Method Request、Activation Queue和Future组件的可复用实现。这些组件已被用于实现许多分布式系统产品。

Siemens MedCom：主动对象模式还被用于Siemens MedCom构架，它为电子医学系统[20]提供了一种面向组件的黑盒子构架。MedCom采用主动对象模式和命令处理器模式（Command Processor Pattern）来简化在多个医学服务器上访问病人信息的客户窗口应用。

西门子呼叫中心管理系统：该系统使用主动对象模式的线程池变种。

Actor（**行动者**）：主动对象模式已被用于实现Actor[21]。一个Actor含有一组实例变量和behavior（行为），响应其他Actor发送的消息。发送给Actor的消息被放置在它的消息队列中。在Actor模型里，消息以到达的顺序被“当前” behavior执行。每个behavior提名一个替换behavior来执行下一个消息，该执行有可能是发生在进行提名的behavior完成执行之前。基本的Actor模型的变种允

许消息队列中的消息基于不同于到达顺序的其他标准来被执行[22]。当主动对象模式用于实现Actor时，调度者对应于Actor调度机制，方法请求对应于为Actor定义的behavior，而仆人是共同表示Actor状态的一组实例变量[23]。代理仅仅是一种强类型的用于传递消息给Actor的机制。

6.13 效果

主动对象模式提供以下好处：

增强应用并发性并简化同步复杂性：并发性是通过允许客户线程和异步方法同时运行来增强的。同步复杂性则通过调度者来简化；后者对同步约束进行求值，以依照它们的状态来保证对仆人的序列化访问。

透明而有效地利用可用的并发：如果硬件和软件平台能高效地支持多CPU，该模式可以允许多个主动对象依据它们的同步约束并行地执行。

方法执行序可与方法调用序不同：异步调用的方法基于它们的同步约束来执行，其顺序可与它们被调用的顺序不同。

但是，主动对象模式也有以下缺点：

性能开销：取决于调度者怎样实现（例如，在用户空间还是内核空间），当调度和执行主动对象方法调用时，可能会产生上下文切换、同步和数据移动等开销。一般而言，主动对象最适用于粒度相对较粗的对象。相反，如果对象的粒度非常细，与其他像管程（Monitor）这样的并发模式相比，主动对象的性能开销可能会很过分。

复杂的调试：由于调度者的并发性和非决定论，有可能会难以调试含有主动对象的程序。而且，许多调试器都不能充分地支持并发应用。

6.14 参见

管程模式确保某一时刻在被动对象中只有一个方法在执行，而不管并发调用此对象的方法的数目有多少。管程通常比主动对象要更为高效，因为它们带来的上下文切换和数据移动的开销较少。但是，使用管程模式，在客户和服务器之间增加分布式边界要更难。

反应堆模式[24]负责处理多个事件处理器的多路分离和分派；处理器在可以开始操作而不会阻塞时被触发。该模式常常用于替代主动对象模式，以调度对被动对象的回调操作。它还可以与主动对象模式结合使用，以构成在下一段中描述的半 - 同步/半 - 异步（Half-Sync/Half-Async）模式。

半 - 同步 / 半异步模式[25]是一种体系结构模式，用于使系统中的同步I/O和异步I/O去耦合，以简化并发编程工作，而又不影响执行效率。该模式通常使用主动对象模式来实现同步任务层，使用反应堆模式[24]来实现异步任务层，以及使用生产者/消费者模式来实现排队层。

命令处理器（Command Processor）模式[2]与主动对象模式类似。它的意图是分离请求的发出与执行。命令处理器对应于调度者，它维护待处理的服务请求，后者被实现为命令（Command）[5]。它们在对应于仆人的供应者那里执行。但是，命令处理器模式并不特别关心并发，而且客户、命令处理器和供应者驻留在同一线程控制中。因而，没有代理来向客户代表仆人。客户创建命令，并将它们直接传递给命令处理器。

Broker模式[2]有着许多与主动对象模式一样的组件。特别地，客户通过代理来访问Broker，而服务器通过仆人来实现远地对象。Broker模式和主动对象模式之间的主要差异是：在Broker模式中的代理和仆人之间是分布式的边界，而在主动对象模式中的代理和仆人之间是线程边界。

互斥模式[26]是一种简单的锁定机制，常常用于在实现并发被动对象（也称为管程）时代替主动对象。互斥模式可以略微不同的形式出现，比如回旋锁或信号量。互斥模式可有多种语义，比如递归互斥体和优先级继承互斥体。

感谢

主动对象模式由Greg Lavender首创。感谢Frank Buschmann、 Hans Rohnert、 Martin Botzler、 Michael Stal、 Christa Schwanninger和Greg Gallant提出大量意见，极大地改进了这一版本的模式描述的形式和内容。

参考文献

[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 6th 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 2nd 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.