第10章 服务配置器（Service Configurator）模式：通过服务配置器模式动态配置通信服务

Prashant Jain　　Douglas C. Schmidt

10.1 介绍

一系列迅速增长的通信服务正出现在Internet上。通信服务是为客户提供功能的服务器中的组件。在Internet可用的服务包括：WWW浏览和内容获取服务（例如，Alta Vista、Apache、Netscape的HTTP服务器）、软件发布服务（例如，Castinet）、电子邮件和网络新闻传输代理（例如，sendmail和nntpd）、远程文件访问（例如，ftpd）、远程终端访问（例如，rlogind和telnetd）、路由表管理（例如，gated和routed）、主机和用户活动报告（例如，fingerd和rwhod）、网络时间协议（例如，ntpd），以及请求代理服务（例如，orbixd和RPC portmapper），等等。

实现这些服务的常用方法是将每个服务开发成为单独的程序，随后编译、链接，并在单独的进程中执行每个程序。但是，这样的“静态”配置服务方法生成不灵活、常常也是低效的应用和软件体系结构。静态配置的主要问题是它相对于应用中的其他服务，将特定服务的*实现*和服务的*配置*紧耦合在一起。

本文描述*服务配置器*（Service Configurator）模式，它通过使服务的行为与服务实现被配置进应用的时间点去耦合来增强应用的灵活性（常常还有性能）。本文使用用C++编写的分布式时间服务作为例子来演示服务配置器模式。但是，服务配置器已经以许多方式被实现，范围从现代操作系统（像Solaris和Windows NT）中的设备驱动程序，到Internet超级服务器（像inetd和Windows NT服务控制管理器），以及Java applets。

10.2 服务配置器模式

10.2.1 意图

使服务的行为与服务实现被配置进应用或系统的时间点去耦合。

10.2.2 别名

超级服务器。

10.2.3 动机

服务配置器模式使服务的实现与服务被配置进应用或系统的时间点去耦合。这样的去耦改善了服务的模块性，并允许服务独立于配置问题（比如两个服务是否必须驻留在一起，或是采用何种并发模型执行服务）而持续发展。

此外，服务配置器模式使它配置的服务的管理集中化。这便利了服务的自动初始化和终止，并且可以通过将常用的服务初始化和终止模式分解进高效的可复用组件而提高性能。

这一部分使用分布式时间服务作为例子来说明服务配置器模式的动机。

10.2.3.1 上下文

当服务需要进行动态发起、挂起、恢复和终止时，应该采用服务配置器模式。此外，如果服务配置决策必须被推迟至运行时，也应该采用服务配置器模式。

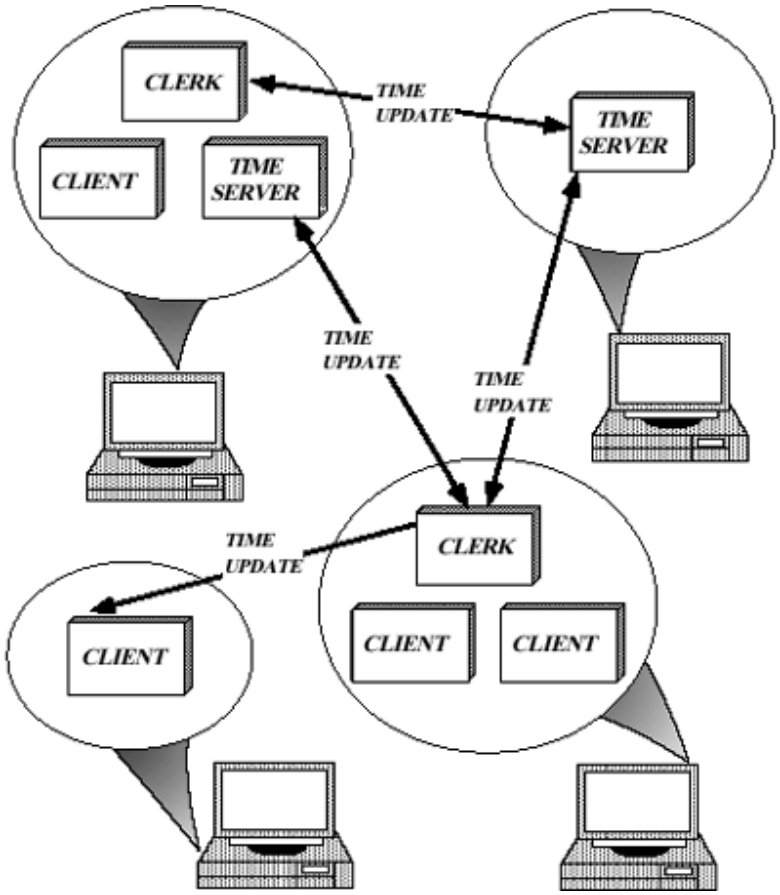

图10-1 分布式时间服务

为说明该模式的动机，考虑图10-1中所示的分布式时间服务。该服务为在局域网和广域网中协作的计算机提供准确、容错的时钟同步。在要求多个主机维护精确的全局时间的分布式系统中，同步时间服务是很重要的。例如，大型分布式医学成像系统[1]需要全局同步的时钟，以确保病人的检查被精确地记录时间，并由放射科医生通过健康保健递送系统迅捷地加以分析。

如图10-1所示，该分布式时间服务的体系结构含有下列时间服务器（Time Server）、事务员（Clerk）和客户（Client）组件：

- *时间服务器*回答事务员作出的关于时间的查询。
- *事务员*查询一或多个时间服务器，以确定正确的时间、使用若干分布式时间算法[2, 3]中的一种来计算近似正确的时间，并且更新它自己的本地系统时间。
- *客户*使用事务员维护的全局时间信息来提供与其他主机上的客户所用的时间观念的一致性。

10.2.4 常用解决方案

一种实现分布式时间服务的方法是将时间服务器、事务员和客户的*逻辑*功能静态地配置进分离的*物理*进程。在该方法中，可有一或多个主机运行时间服务器进程；后者处理来自事务员进程的时间更新请求。下面的C++代码段演示静态配置的时间服务器进程的结构：

```
// The Clerk_Handler processes time requests

// from Clerks.

class Clerk_Handler :

public Svc_Handler <SOCK_Stream>

{

public:

// This method is called by the Reactor

// when requests arrive from Clerks.

virtual int handle_input (void)

{

       // Read request from Clerk and reply

       // with the current time.

}



// ...

};



// The Clerk_Acceptor is a factory that

// accepts connections from Clerks and creates
```

```
// Clerk_Handlers.

typedef Acceptor<Clerk_Handler, SOCK_Acceptor> Clerk_Acceptor;


int main (int argc, char *argv[])

{

// Parse command-line arguments.

Options::instance ()->parse_args (argc, argv);


// Set up Acceptor to listen for Clerk connections.

Clerk_Acceptor acceptor(Options::instance ()->port ());


// Register with the Reactor Singleton.

Reactor::instance ()->register_handler(&acceptor, ACCEPT_MASK);


// Run the event loop waiting for Clerks to

// connect and perform time queries.

for (;;)

       Reactor::instance ()->handle_events ();


/* NOTREACHED */

}
```

该程序使用反应堆（Reactor）模式[4]和接受器（Acceptor）模式[5]来实现静态配置的时间服务器进程。

每个需要全局时间同步的主机都可以运行事务员进程。事务员基于接收自一或多个时间服务器的值周期性地更新它们的本地系统时间。下面的C++代码段演示静态配置的事务员进程的结构：

```
// This class communicates with the Time_Server.

class Time_Server_Handler { /* ... */ };


// This class establishes connections with the

// Time Servers and periodically queries them for

// their latest time values.

class Clerk : public Svc_Handler <SOCK_Stream>

{

public:
```

```
// Initialize the Clerk.
Clerk (void)
{
       Time_Server_Handler **handler = 0;

       // Use the Iterator pattern and the
       // Connector pattern to set up the
       // connections to the Time Servers.
       for (ITERATOR iterator (handler_set_);
              iterator.next (handler) != 0;
              iterator.advance ())
       {
              connector_.connect (*handler);

              Time_Value timeout_interval (60);

              // Register a timer that will expire
              // every 60 seconds. This will trigger
              // a call to the handle_timeout() method,
              // which will query the Time Servers and
              // retrieve the current time of day.
              Reactor::instance ()->schedule_timer
                            (this, timeout_interval);
       }
}

       // This method implements the Clock Synchronization
// algorithm that computes local system time. It
// is called periodically by the Reactor's timer
// mechanism.
int handle_timeout (void)
{
       // Periodically query the servers by iterating
       // over the handler set and obtaining time
       // updates from each Time Server.
       Time_Server_Handler **handler = 0;
```

```
        // Use the Iterator pattern to query all
        // the Time Servers
        for (ITERATOR iterator (handler_set_);
                iterator.next (handler) != 0;
                iterator.advance ())
        {
                Time_Value server_time = (*handler)->get_server_time ();

                // Compute the local system time and
                // store this in shared memory that
                // is accessible to the Client processes.
        }
}

private:
typedef Unbounded_Set <Time_Server_Handler *>HANDLER_SET;
typedef Unbounded_Set_Iterator<Time_Server_Handler *> ITERATOR;

// Set of Clerks and iterator over the set.
HANDLER_SET handler_set_;

// The connector_ is a factory that
// establishes connections with Time Servers
// and creates Time_Server_Handlers.
Connector<Time_Server_Handler, SOCK_Connector>connector_;
};

int main (int argc, char *argv[])
{
// Parse command-line arguments.
Options::instance ()->parse_args (argc, argv);

// Initialize the Clerk.
Clerk clerk;
```

```
// Run the event loop, periodically

// querying the Time Servers to determine

// the global time.

for (;;)

       Reactor::instance ()->handle_events ();


/* NOTREACHED */

}
```

该程序使用反应堆（Reactor）模式[4]和连接器（Connector）模式[6]来实现静态配置的事务员进程。

客户进程可以使用它们的本地事务员报告的同步的时间。为最少化通信开销，当前时间可被存储在共享内存中，后者被映射到事务员和同一主机上所有客户的地址空间中。除了时间服务，这些主机提供的其他通信服务（比如文件传输、远程登录和HTTP服务器）也可以在分离的静态配置的进程中执行。

10.2.5 常用解决方案的陷阱和缺陷

尽管像反应堆、接受器和连接器这样的模式的使用改善了上面所示的分布式时间服务器的模块性和可移植性，使用静态方法来配置通信服务有以下缺点：

- **在开发周期中必须过早地作出服务配置决策：**这是不合需要的，因为开发者可能无法预先知道使服务组件驻留在一起或分布驻留的最佳方式。例如，在无线计算环境中、内存资源的匮乏可能会迫使客户和事务员被划分进运行在分离的主机上的两个独立进程。相反，在实时的航空控制环境中，可能必须使事务员和服务器驻留进一个进程，以降低通信响应延迟。迫使开发者过早地采用特定的服务配置会阻碍灵活性，并可能降低性能和功能。
- **修改某服务可能会对其他服务产生不利影响：**每个服务组件的实现都与它的初始配置紧耦合在一起。这导致开发者难以在不影响其他服务的情况下修改某个服务。例如，在上面提到的实时航空控制环境中，可以静态地配置事务员和时间服务器、使它们在一个进程中执行，以降低响应延迟。但是如果事务员所实现的分布式时间算法被改变，现有的事务员代码就有可能也需要修改、重编译和静态重链接。而终止进程以改变事务员代码可能也会终止时间服务器。对于高度可用的系统（比如电信交换机或呼叫中心[7]），这样的服务中断可能是不能接受的。
- **系统性能不能高效地伸缩：**将每个服务与一个进程关联在一起占死了OS资源（比如I/O描述符、虚拟内存和进程表槽口）。如果服务常常空闲，这样的设计可能是很浪费的。而且，对于许多短期生存的通信任务（比如向时间服务器请求当前时间，或在域名服务中解析主机地址请求）来说，进程常常是错误的抽象。在这些情况下，多线程主动对象（Active Object）（8）或单线程反应式[4]事件循环可能更为高效。

10.2.6 更好的解决方案

更为方便和灵活的实现分布式服务的方法常常是使用*服务配置器*模式。该模式使通信服务的行为与这些服务被配置进应用或系统的时间点去耦合。服务配置器模式消除了以下需求的压力：

- **延缓对服务的特定类型或特定实现的选择，直至设计周期中非常迟后的阶段：**这允许开发者集中考虑服务的功能（例如，时间同步算法），而不用过早地卷入特定的服务配置。通过使功能与配置去耦合，服务配置器模式允许应用独立于系统所用的配置策略和机制而发展。
- **通过编写多个独立开发的、不需要全局知识的服务来构建完整的应用或系统：**服务配置器模式要求所有服务都拥有用于配置和控制的统一接口。这使得服务可被当作积木、很容易地作为组件集成进更大的应用中。贯穿所有服务的统一接口使得它们在怎样配置上有着同样的“外观和感受”（“look and feel”）。继而，这样的统一性通过促进“最少惊讶法则”（“principle of least surprise”）而简化了应用开发。
- **在运行时优化、控制和重配置服务的行为：**使服务的实现与它的配置去耦合使得调谐服务的特定实现或配置参数成为可能。例如，取决于硬件和操作系统上可用的并行性，在分离的线程或进程中运行多个服务可能更高效，也可能更低效。当有更多的信息可用于帮助优化服务时，服务配置器模式使应用能够在运行时选择和调整这些行为。此外，在分布式系统中增加新服务或更新服务常常可以无需停止现有服务就得以完成。

图10-2使用OMT表示法来演示根据服务配置器模式设计的分布式时间服务的结构。

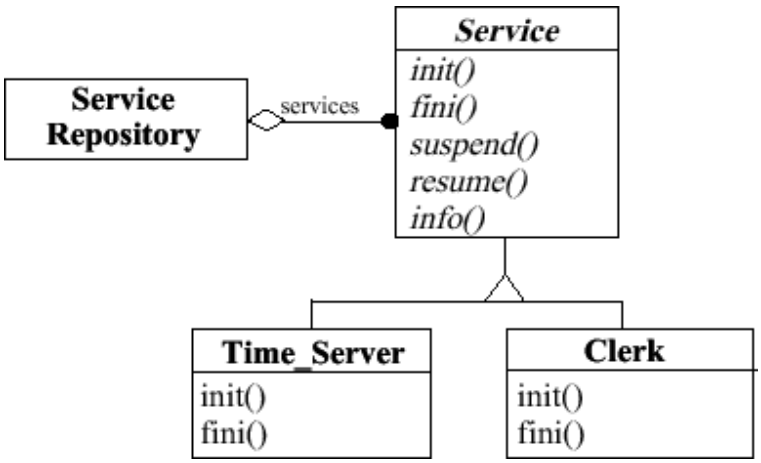

图10-2 分布式时间服务的结构

Service基类为配置和控制服务（比如时间服务或事务员）提供标准接口。基于服务配置器的应用使用该接口来发起、挂起、恢复和终止服务，以及获取关于服务的运行时信息（比如它的IP地址和端口号）。服务自身驻留在Service Repository（服务仓库）中，可由基于服务配置器的应用来在Service Repository中增加或移除。

Service基类的两个子类出现在分布式时间服务中：Time Server和Clerk。每个子类表示一个在分布式时间服务中有着特定功能的具体Service。Time Server服务负责接收和处理来自Clerk的时间更新请求。Clerk服务是连接器（Connector）[6]工厂，负责（1）为每个服务器创建新连接，（2）动态分配新的处理器，以向相连的服务器发送时间更新请求，（3）通过处理器接收来自所有服务器的回复，以及（4）随后更新本地系统时间。

通过管理时间服务中的服务组件的配置，服务配置器模式使分布式时间服务变得更为灵活，并从而使它与实现问题分离开来。此外，服务配置器提供了构架来将其他通信服务的配置和管理合并在一个管理单元中。

10.3 适用性

当有以下情况时使用服务配置器模式：

- 服务必须被动态地发起、挂起、恢复及终止；以及
- 服务的实现可能会改变，但是它相对于有关服务的配置保持不变；并且/或者一组驻留在一起的服务可能会改变，但是它们的实现保持不变；或是
- 通过编写多个独立开发和可动态配置的服务，应用或系统可以被简化；或是
- 通过使用单一管理单元进行配置，多个服务的管理可以被简化或优化。

当有以下情况时*不要*使用服务配置器模式：

- 由于安全限制，动态（重）配置不合需要（在这种情况下，可能必须使用静态配置）；或是
- 服务的初始化或终止太过复杂，或与上下文的耦合过于紧密，以致于不能以统一的方式来完成；或是
- 因为从不变动，服务不能从动态配置中受益；或是
- 对于动态（重）配置使用的OS和语言机制所带来的额外的间接层次，苛刻的性能需求要求使其最小化。

10.4 结构和参与者

在图10-3中使用OMT表示法演示了服务配置器模式的结构：

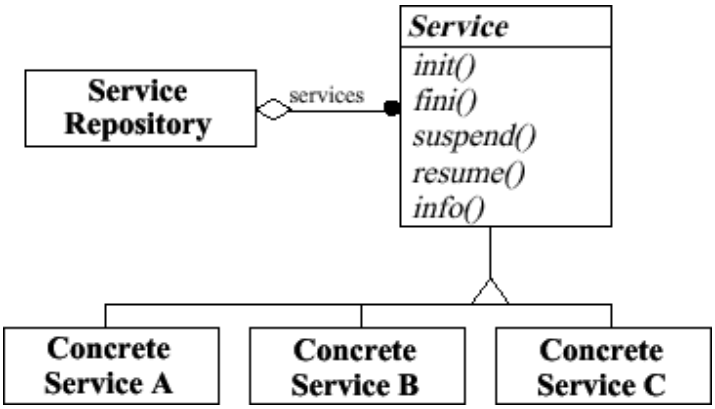

图10-3 服务配置器模式的结构

服务配置器模式中的关键参与者包括：

- **服务（**Service**）：**规定含有挂钩方法[9]（比如初始化和终止）的接口，由基于服务配置器的应用用于动态配置Service。
- **具体服务（**Clerk**和**Time Server**）：**实现服务的挂钩方法及其他的服务特有的功能（比如事件处理和与客户的通信）。
- **服务仓库（**Service Repository**）：**维护基于服务配置器的应用所提供的所有服务的仓库。这使得管理实体可以对被配置的服务的行为进行集中管理和控制。

10.5 协作

图10-4描述下面的服务配置器模式三个阶段中的组件间协作：

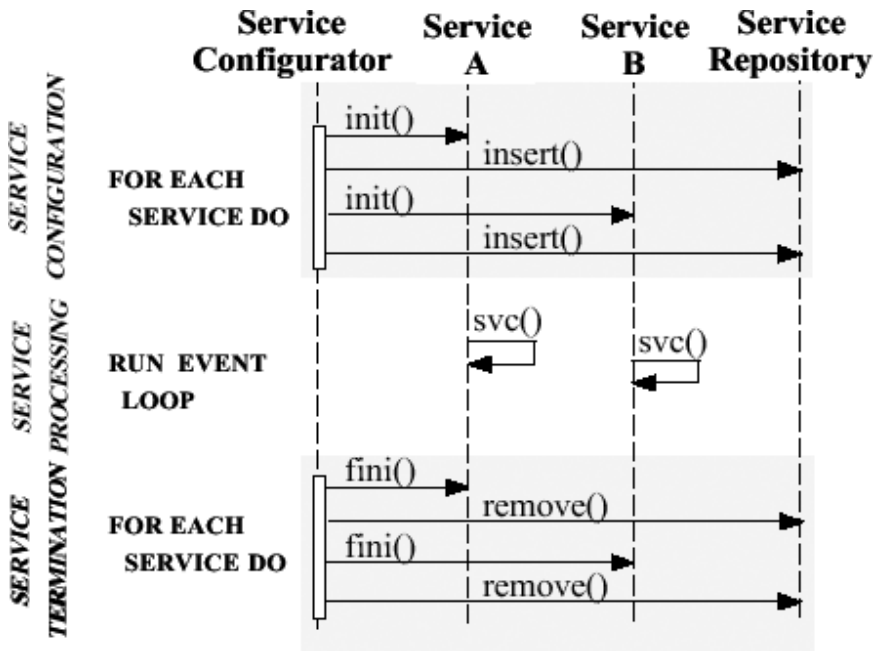

图10-4 服务配置器模式的交互图

- *服务配置*：服务配置器通过调用Service的init方法来对其进行初始化。一旦Service被成功初始化，服务配置器将其增加到Service Repository；后者管理和控制所有的Service。
- *服务处理*：在被配置进系统后，Service执行其处理任务（也就是，为客户请求服务）。当Service的处理在执行时，服务配置器可以挂起和恢复Service。
- *服务终止*：一旦不再需要，服务配置器通过调用Service的fini挂钩方法来将其终止。该挂钩允许Service在终止前进行清理。一旦Service终止，服务配置器将其从Service Repository中移除。

10.6 效果

10.6.1 好处

服务配置器模式提供以下好处：

- **集中式管理：**该模式将一或多个服务合并进单一的管理单元中。通过自动完成通用的服务初始化和终止活动（比如打开和关闭文件、获取和释放锁，等等），这有助于开发的简化。此外，它还利用一组统一的配置管理操作（比如*初始化*、*挂起*、*恢复*和*终止*）来使通信服务的管理集中化。
- **增强模块性和复用：**该模式通过使服务的实现与服务的配置去耦合而改善了通信服务的模块性和可复用性。此外，所有服务都有统一的接口，并通过它来进行配置，从而鼓励了复用、并简化了后续服务的开发。
- **增强配置动态性：**该模式使服务能够被动态地重配置，而不用修改、重编译，或静态地重新链接已有代码。此外，服务的重配置常常可以无需重启该服务、或其他与之驻留一处的服务就得以完成。
- **增加调整和优化的机会：**通过使服务功能与用于执行服务的并发策略去耦合，该模式增加了开发者可用的服务配置选择的范围。通过从一系列并发策略中进行选择，开发者可以适应性地调整看守的并发水平，以配合客户的需求和可用的OS处理资源。可选项包括在客户请求到达时派生线程或进程、或在服务创建时预先派生线程或进程，等等。

10.6.2 缺点

服务配置器模式有以下缺点：

- **缺乏确定性：**该模式导致在应用的服务被在运行时配置之前，难以确定该应用的行为。对于实时系统来说，这可能是成问题的，因为在与其他特定的服务一起运行时，动态配置的服务可能无法可预测地执行。例如，一个新配置的服务可能会消耗过多的CPU周期，从而饿死其他服务，并导致它们错过它们的最终期限。
- **可靠性降低：**使用服务配置器模式的应用比静态配置的应用可能要更不可靠，因为服务的特定配置可能会有害地影响服务的执行。例如，有毛病的服务可能会崩溃，从而破坏它与其他服务共享的状态信息。如果多个服务被配置成在同一进程中运行，这就会特别地成问题。
- **增加开销：**该模式增加了服务执行的额外的间接层次。例如，服务配置器首先初始化服务，并将其装载进Service Repository。这在时间紧急的应用中可能会带来过多的开销。此外，使用动态链接来实现服务配置器模式增加了方法调用和全局变量访问的额外的间接层次。

10.7 实现

服务配置器模式可以通过许多方式实现。这一部分解释实现该模式时所涉及的步骤和可选方案。表10-1中总结了这些步骤和可选方案。

步骤	常用可选方案
定义服务控制接口	• 服务从抽象基类继承 • 服务响应控制消息
定义服务仓库	• 维护服务实现表
选择配置机制	• 在命令行指定 • 通过用户接口指定 • 通过配置文件指定
确定服务执行机制	• 反应式执行 • 多线程主动对象 • 多进程主动对象

表10-1 实现服务配置器模式所涉及的步骤

- **定义服务控制接口：**下面是服务必须支持的的基本接口，用以使服务配置器能够配置和控制服务：

    - *服务初始化*：提供服务的入口，并执行服务的初始化。
    - *服务终止*：终止服务的执行。
    - *服务挂起*：暂时挂起服务的执行。
    - *服务恢复*：恢复挂起服务的执行。
    - *服务信息*：报告描述服务的信息（例如，端口号或服务名）。

    定义服务控制接口有两种基本的方法，*基于继承的*和*基于消息的*：

    - *基于继承的*：该方法使每个服务都从公共的基类继承。这是ACE Service Configurator构架[7]和Java applets所用的方法。它通过定义含有许多纯虚“挂钩”方法的抽象基类来工作，如下所示：

```
class Service

{

public:

// = Initialization and termination hooks.
```

```
virtual int init (int argc, char *argv[]) = 0;

virtual int fini (void) = 0;


// = Scheduling hooks.

virtual int suspend (void);

virtual int resume (void);


// = Informational hook.

virtual int info (char **, size_t) = 0;

};
```

> init方法用作Service的入口。它被服务配置器用于初始化Service的执行。fini方法允许服务配置器终止Service的执行。suspend和resume方法是调度挂钩，由服务配置器用于挂起和恢复Service的执行。info方法允许服务配置器获取与Service相关的信息（比如它的名字和网络地址）。这些方法一起给出了服务配置器和由它管理的Service之间的统一约定。

- *基于消息的*：另一种控制通信服务的方法是编写每个Service，使之响应一组特定的消息。这使得开发者有可能将服务配置器集成进缺乏继承的非OO编程语言中（比如C或Ada83）。

> Windows NT服务控制管理器（SCM）使用这种方案。每个Windows NT主机都拥有主SCM进程，通过向系统服务传递多种控制信息（比如PAUSE、RESUME和TERMINATE）来自动对其进行初始化和管理。每个SCM管理的服务的开发者都要负责编写代码来处理这些消息。

- **定义服务仓库：**Service Repository被用于维护所有的服务实现，比如对象、可执行程序，或是动态链接库（DLL）。当服务被配置进系统、或从中移除时，服务配置器使用Service Repository来对其进行访问。每个服务的当前状态（比如它是活动的还是挂起的）也在仓库中维护。服务配置器存储并访问在主内存、文件系统或内核（例如，它可以使用报告进程和线程状态的操作）中的Service Repository信息。
- **选择配置机制：**服务需要在执行前被配置。配置服务需要指定一些属性，这些属性指示服务实现（比如可执行程序或DLL）的位置，以及在运行时初始化服务所需的参数。这样的配置条件可以通过多种方式来指定（比如在命令行上、经由环境变量、通过用户接口，或是在配置文件中）。通过将服务属性和初始化参数合并在一处，集中式配置机制可以简化应用中服务的安装和管理。
- **确定服务执行机制：**已由服务配置器动态配置的服务可以使用反应式[4]和主动对象[8]方案的多种组合来执行。下面简要地检查这些可选方案：

- *反应式执行*：可为服务配置器和它配置的所有服务的执行使用单线程控制。
- *多线程主动对象*：该方法使动态配置的服务运行在服务配置器进程里它们自己的线程控制中。服务配置器可以“按需”派生新线程，或是在已有线程池中执行这些服务。
- *多进程主动对象*：该方法使动态配置的服务运行在它们自己的进程中。服务配置器可以“按需”派生新进程，或是在已有进程池中执行这些服务。

10.8 示例代码

下面的代码介绍一个用C++写成的服务配置器模式的例子。该例子聚焦于10.2.3中介绍的分布式时间服务的与配置有关的方面。此外，它还演示了其他模式（比如反应堆模式[4]、接受器[5]和连接器[6]模式）的使用，它们通常被用于开发通信服务和对象请求代理（Object Request Broker）。

在下面的例子中，图10-3所示的OMT类图中的Concrete Service类由Time Server类以及Clerk类来表示。这一部分中的C++代码实现Time Server和Clerk类。两个类都继承自Service，使得它们可被动态配置进应用中。此外，该方法使用基于显式动态链接[7]和配置文件的配置机制来动态地配置分布式时间服务的事务员和服务器部分。服务执行机制则基于在单线程控制中的反应式事件处理模型。

下面的例子显示事务员组件可以怎样改变它用于计算本地系统时间的算法、而又*不*影响服务配置器配置的其他组件的执行。一旦算法被修改，事务员组件就由服务配置器进行动态重配置。

下面所示的代码还包括main驱动函数，它提供任何基于服务配置器的应用的通用入口。使用ACE[7]，该实现可运行在UNIX/POSIX和Win32平台上；ACE可通过WWW在http://www.cs.wustl.edu/~schmidt/ACE.html获取。

10.8.1 Time Server类

Time Server使用Acceptor类来接受来自一或多个事务员的连接。Acceptor类使用接受器模式[5]来为所有来自事务员的连接创建处理器；这些事务员想要接收时间更新请求。该设计使Time Server的实现与它的配置去耦合，因此，开发者可以独立于Time Server的配置改变它的实现。这为改进时间服务器的实现提供了灵活性。

Time Server类继承自在10.7中定义的Service基类。这使得服务配置器能够动态地链接Timer Server，以及解除其链接。在将Time Server服务装载进Service Repository之前，服务配置器调用它的init挂钩。该方法执行Time Server特有的初始化代码。同样地，当服务不再被需要时，fini挂钩方法被服务配置器自动调用，以将其终止。

```
// The Clerk_Handler processes time requests

// from Clerks.

class Clerk_Handler :

public Svc_Handler <SOCK_Stream>

{

// This is identical to the Clerk_Handler
```

```
// defined in the second section.
};

class Time_Server : public Service
{
public:
// Initialize the service when linked dynamically.
virtual int init (int argc, char *argv[])
{
       // Parse command line arguments to get
       // port number to listen on.
       parse_args (argc, argv);

       // Set the connection acceptor endpoint into
       // listen mode (using the Acceptor pattern).
       acceptor_.open (port_);

       // Register with the Reactor Singleton.
       Reactor::instance ()->register_handler
              (&acceptor_, ACCEPT_MASK);
}

// Terminate the service when dynamically unlinked.
virtual int fini (void)
{
       // Close down the connection.
       acceptor_.close ();
}

// Other methods (e.g., info(), suspend(), and
// resume()) omitted.

private:
// Parse command line arguments or those
// specified by the configuration file.
int parse_args (int argc, char *argv[]);
```

```
// Acceptor is a factory that accepts

// connections from Clerks and creates

// Clerk_Handlers.

Acceptor<Clerk_Handler, SOCK_Acceptor>acceptor_;


// Port the Time Server listens on.

int port_;

};
```

注意服务配置器也可以分别通过调用Time Server的suspend和resume挂钩来将其挂起和恢复。

10.8.2 Clerk类

Clerk使用Connector类来建立和维护与一或多个Time Server的连接。Connector类使用连接器模式[6]来为每个到时间服务器的连接创建处理器。处理器接收并处理来自Time Server的时间更新。

Clerk类继承自Service基类。因此，像Time Service一样，它也可以被服务配置器动态地配置。服务配置器可以分别通过调用Clerk的init、suspend、resume和fini挂钩来将其初始化、挂起、恢复和终止。

```
// This class communicates with the Time_Server.

class Time_Server_Handler

: public Svc_Handler <SOCK_Stream>

{

public:

// Get the current time from a Time Server.

Time_Value get_server_time (void);


// ...

};


// This class establishes and maintains connections

// with the Time Servers and also periodically queries

// them to calculate the current time.
```

```
class Clerk : public Service
{
public:
// Initialize the service when linked dynamically.
virtual int init (int argc, char *argv[])
{
       // Parse command line arguments and for
       // every host:port specification of server,
       // create a Clerk instance that handles
       // connection to the server.
       parse_args (argc, argv);

       Time_Server_Handler **handler = 0;

       // Use the Iterator pattern and the
       // Connector pattern to set up the
       // connections to the Time Servers.
       for (ITERATOR iterator (handler_set_);
              iterator.next (handler) != 0;
              iterator.advance ())
       {
              connector_.connect (*handler);

              Time_Value timeout_interval (60);

              // Register a timer that will expire
              // every 60 seconds. This will trigger
              // a call to the handle_timeout() method,
              // which will query the Time Servers and
              // retrieve the current time of day.
              Reactor::instance ()->schedule_timer
                            (this, timeout_interval);
       }
}

// Terminate the service when dynamically unlinked.
```

```
virtual int fini (void)
{
	Time_Server_Handler **handler = 0;

	// Disconnect from all the time servers.
	for (ITERATOR iterator (handler_set_);
		iterator.next (handler) != 0;
		iterator.advance ())
			(*handler)->close ();

	// Remove the timer.
	Reactor::instance ()->cancel_timer (this);
}

// info(), suspend(), and resume() methods omitted.
// The handle_timeout method implements the
// Clock Synchronization algorithm that computes
// local system time. It is called periodically
// by the Reactor's timer mechanism.
int handle_timeout (void)
{
	// Periodically query the servers by iterating
	// over the handler set and obtaining time
	// updates from each Time Server.
	Time_Server_Handler **handler = 0;

	// Use the Iterator pattern to query all
	// the Time Servers
	for (ITERATOR iterator (handler_set_);
		iterator.next (handler) != 0;
		iterator.advance ())
	{
		Time_Value server_time = (*handler)->get_server_time ();

		// Compute the local system time and
		// store this in shared memory that
```

```
            // is accessible to the Client processes.
        }
}

private:
// Parse command line arguments or those
// specified by the configuration file and
// create Clerks for every specified server.
int parse_args (int argc, char *argv[]);

typedef Unbounded_Set <Time_Server_Handler *>HANDLER_SET;
typedef Unbounded_Set_Iterator<Time_Server_Handler *> ITERATOR;

// Set of Clerks and iterator over the set.
HANDLER_SET handler_set_;

// Connector used to set up connections
// to all servers.
Connector<Time_Server_Handler, SOCK_Connector>connector_;
};
```

通过遍历其处理器列表，Clerk周期性地发送时间更新请求给所有与其相连的Time Server。一旦Clerk接收到来自所有与其相连的Time Server的响应，它就重新计算它的本地系统时间。因而，当客户向事务员请求当前时间时，它们会接收到全局同步的时间值。

10.8.3 配置应用

10.8.3.1 共驻式配置

下面的代码演示应用的动态配置和执行，使用配置文件来使Time Server和Clerk共同驻留在同一OS进程中：

```
int main (int argc, char *argv[])
{
// Configure the daemon.
```

```
Service_Config daemon (argc, argv);


// Perform daemon services updates.

daemon.run_event_loop ();


/* NOTREACHED */

}
```

这个完全通用的main程序在Service Config对象的构造器中动态地配置通信服务。该方法查询下面的svc.conf配置文件：

```
# Configure a Time Server.

dynamic Time_Server Service*

netsvcs.dll:make_Time_Server()

       "-p $TIME_SERVER_PORT"


# Configure a Clerk.

dynamic Clerk Service*

netsvcs.dll:make_Clerk()

       "-h tango.cs:$TIME_SERVER_PORT"

       "-h perdita.wuerl:$TIME_SERVER_PORT"

       "-h atomic-clock.lanl.gov:$TIME_SERVER_PORT"

       "-P 10" # polling frequency
```

ACE服务配置构架处理svc.conf配置文件中的每一条目。该构架将dynamic指令解释为将指定Service动态链接进应用进程的命令。表10-2总结了可用的服务配置指令。

指令	**描述**
dynamic	动态链接和启用服务
static	启用静态链接的服务
remove	完全地移除服务
suspend	挂起服务，而不将其移除
resume	恢复先前挂起的服务

表10-2 服务配置指令

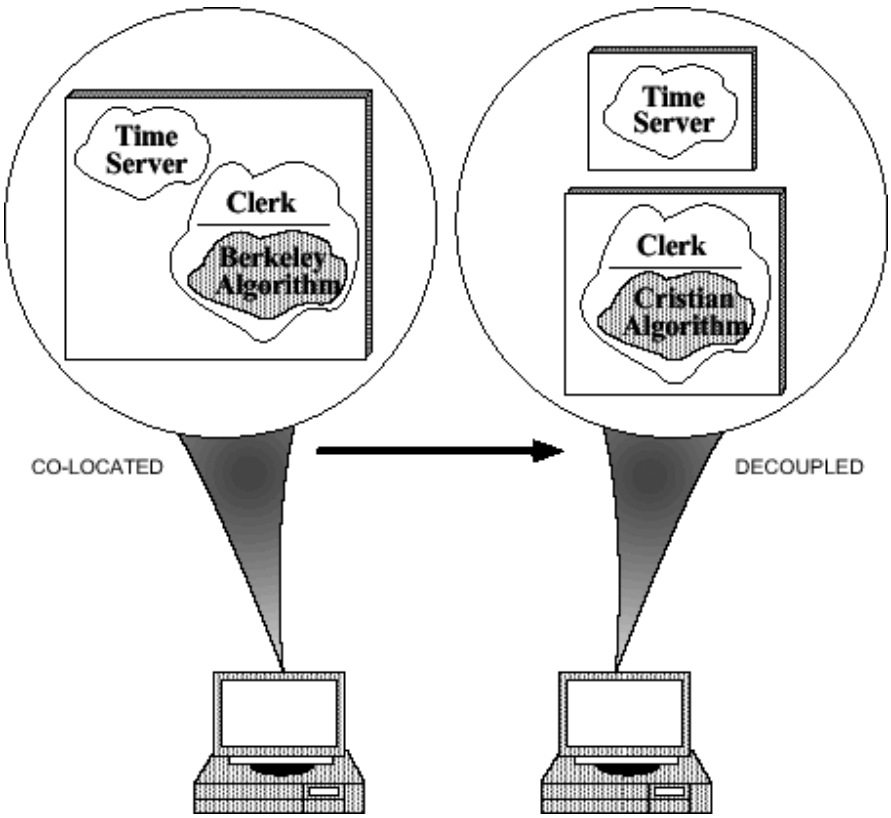

图10-5 重配置时间服务器和事务员

例如，svc.conf文件中的第一个条目指定一个服务名（Time_Server），由Service Repository用于标识动态配置的服务。make_Time_Server是一个工厂函数，位于动态链接库netsvcs.dll中。服务配置器构架动态地将此DLL链接进应用的地址空间，然后调用make_Time_Server工厂函数。该函数动态地分配新的Time Server实例，如下所示：

```
Service *make_Time_Server(void)

{

return new Time_Server;

}
```

在第一个条目中最后的字符串参数指定含有端口号的环境变量，Time Server将在该端口上侦听Clerk连接。服务配置器将此字符串转换为“argc/argv”风格的向量，并将其传递给Time Server的init挂钩。如果init方法成功执行，Service * 就被存储在Service Repository中的Time_Server名下。

svc.conf文件中的第二个条目指定怎样动态配置Clerk。和前面一样，服务配置器将netsvcs.dll动态地链接进应用的地址空间，并调用make_Clerk工厂函数来创建新的Clerk实例。三个时间服务器

（tanggo.cs、perdita.wuerl和atomic-clock.lanl）的名字和端口号被传递给init挂钩。此外，-P 10选项定义事务员轮询时间服务器的频度。

10.8.3.2 分布式配置

假设我们不想将Time Server和Clerk放在一起，以减少应用的内存占用。因为我们正在使用服务配置器模式，所有要做的就是将svc.conf文件划分为两部分。一部分含有Time Server条目，而另一部分含有Clerk条目。因为服务配置器模式有将服务的行为与配置去耦合这一优点，服务自身可以无需改变。图10-5显示在Time Server和Clerk驻留在同一进程时、以及在划分后配置看起来是怎样的。

10.8.4 重配置应用

现在假设我们需要改变Clerk的算法实现。例如，我们可能决定从Berkeley算法[2]的实现切换到Cristian的算法[3]的实现。下面对它们作一概述：

- *Berkeley算法*：在该方法中，Time Server是主动的组件，周期性地轮询网络中的全部机器，询问那里的时间。基于接收到的响应，它计算正确时间的合计值，并告诉所有机器相应地调整它们的时间。
- *Cristian算法*：在该方法中，Time Server是响应事务员作出的查询的被动实体。它不会主动地查询其他机器来确定它自己的时间。

要回应Time Server的特性，可能必需改变时间同步算法。例如，如果Time Server所驻留的机器上有WWV接收器，Time Server可被用作被动的实体，而Cristian算法也可能就是适合的。在另一方面，如果Time Server所驻留的机器上没有WWV接收器，那么Berkeley算法的实现可能就更为适合。

图10-5显示事务员实现中的改变（相应于时钟同步算法中的改变）。改变发生在分离Time Server和Clerk的过程中，它们先前是驻留在一起的。

理想地，我们想要改变算法实现，而又不影响其他服务或时间服务的其他组件的执行。使用服务配置器来实现此要求只需要简单地对svc.conf文件作出下面的修改：

```
# Terminate Clerk

remove Clerk
```

仅有的额外要求是让服务配置器处理该指令。这可以通过生成外部事件（比如UNIX SIGHUP信号、RPC通知，或是Windows NT Registry事件）来完成。收到该事件，应用将再次查询配置文件，并终止Clerk服务的执行。服务配置器将调用Clerk的fini方法，并从而终止事务员组件的执行。其他服务的执行不会受影响。

一旦事务员服务被终止，就可以对算法实现进行改变。随后代码就可以被重编译和重链接，以形成新的netsvcs DLL。也可以采取类似的方法来将事务员服务增加回服务配置器。可以修改配置文件，用新指令指定事务员需被动态链接，如下所示：

```
# Reconfigure a new Clerk.

dynamic Clerk Service*

netsvcs.dll:make_Clerk()

       "-h tango.cs:$TIME_SERVER_PORT"

       "-h perdita.wuerl:$TIME_SERVER_PORT"

       "-h atomic-clock.lanl.gov:$TIME_SERVER_PORT"
```

随后生成一个外部事件，致使进程重新读入配置文件，并将事务员组件增加到仓库中。一旦Clerk的init方法被服务配置器构架调用，事务员组件就将开始执行。

图10-6显示像事务员服务这样的Service的生存期状态图。

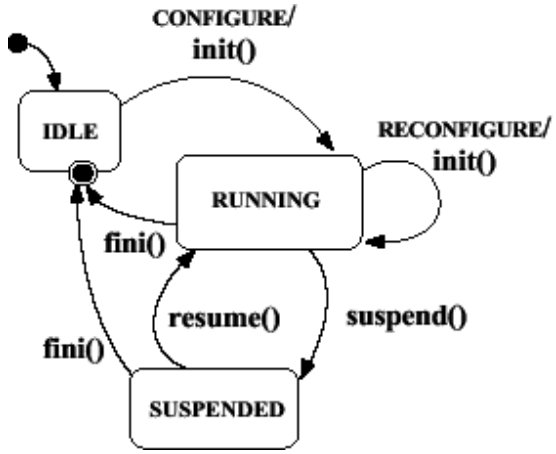

图10-6 服务的生命周期状态图

注意在终止和移除事务员服务的整个过程中，没有其他活动服务受到实现改变和事务员服务重配置的影响。替换新的服务实现的容易程度进一步例证了服务配置器模式所提供的灵活性。

10.9 已知应用

服务配置器模式被广泛用于系统和应用编程环境中，包括UNIX、Windows NT、ACE和Java applets：

- **现代操作系统设备驱动程序：**大多数现代操作系统（比如Solaris和Windows NT）支持动态可配置的内核级设备驱动程序。这些驱动程序可以经由init/fini/info挂钩动态地链接进系统，或从系统中解除链接。这些操作系统使用服务配置器模式来使管理员能够重配置OS内核，而又不必将其关闭、重编译和静态地重新链接新驱动程序，以及重启系统。
- UNIX**网络看守管理：**服务配置模式已被用于管理UNIX网络看守的“超级服务器”（Superserver）中。两个广泛可用的网络看守管理构架是inetd[10]和listen[11]。两个构架都查询配置文件，后者指定（1）服务名（比如标准Internet服务ftp、telnet、daytime和echo），（2）端口号，用于在其上侦听连接到这些服务的客户，以及（3）可执行文件，在客户连接时调用并执行服务。两个构架都含有主接受器[5]进程，它监控一组与这些服务相关联的端口。当客户连接在被监控端口上发生时，接受器进程接受该连接，并将请求多路分离给适当的预登记的服务处理器。该处理器执行服务（反应式地或在主动对象中），并将任何结果返回给客户。
- Windows NT**服务控制管理器（SCM）：**不像inetd和listen，Windows NT服务控制管理器（SCM）本身不是端口监控器。就是说，它不提供内建支持来侦听一组I/O端口、并在客户请求到达时“按需”分派服务器进程。相反，它提供基于RPC的接口，允许主SCM进程自动发起和控制（也就是，暂停、恢复、终止，等等）管理员安装的服务（比如远程注册表访问）。这些服务将另外作为单服务或多服务看守进程中的分离线程运行。每个被安装的服务都单独负责配置它自己，监控任意通信端点（可以比socket端口更广泛）。例如，SCM可以控制命名管道和共享内存。
- ACE**自适配通信环境构架：**ACE构架[7]为动态配置和控制通信服务提供了一组C++机制。ACE Service Configurator扩展了inetd、listen和SCM所提供的机制，以自动支持通信服务的动态链接和解除链接。10.8中包含的代码显示了怎样将ACE构架用于实现分布式时间服务。ACE所提供的机制受到了现代操作系统中用于配置和控制设备驱动程序的接口的影响。但是， ACE Service Configurator构架聚焦于应用级Service对象的动态配置和控制，而非针对内核级设备驱动程序。
- Java applets：Java中的applet机制使用了服务配置器模式。Java支持下载、初始化、启动、挂起、恢复和终止applet。它通过提供发起和终止线程的方法（例如，start和stop）来提供这样的支持。Java applet中的方法可以使用Thread.currentThread()来访问它在其中运行的线程，并随后向它发出控制消息，比如suspend、resume和stop。[12]介绍了一个例子，演示服务配置器模式是怎样被用于Java applets的。

10.10 相关模式

服务配置器模式的意图与配置（Configuration）模式 [13]相类似。配置模式使分布式应用中相关于服务配置的、结构上的问题与服务自身的执行去耦合。它被用于在一些构架中配置分布式系统，以支持从一组组件构造一个分布式系统。以类似的方式，服务配置器模式使服务初始化与服务处理去耦合。主要的区别是配置模式更多地聚焦于一系列相关服务的主动合成，而服务配置器模式聚焦于在特定端点上的服务处理器的动态初始化。此外，服务配置器模式还聚焦于使服务行为与服务的并发策略去耦合。

管理器模式（Manager Pattern）[14]通过承担创建和删除一组对象的责任来对其进行管理。此外，它还提供一个接口，以允许客户访问它管理的对象。服务配置器模式可以使用管理器模式来按照需要创建和删除服务，以及维护它使用管理器模式创建的服务的仓库。但是，必须为使用管理器模式创建的服务增加动态配置、初始化、挂起、恢复和终止功能，才能全面地实现服务配置器模式。

服务配置器常常利用反应堆[4]模式来为被配置的服务完成事件多路分离和分派。同样地，长执行周期的、动态配置的服务常常使用主动对象模式[15]。

基于服务配置器的系统的管理接口（比如配置文件或GUI）提供了一个外观（Fa?ade）[16]。该外观简化了在服务配置器中执行的应用的管理和控制。

Service基类提供的虚方法是一些回调“挂钩”[9]。这些挂钩被服务配置器用于发起、挂起、恢复和终止服务。

可以使用工厂方法（Factory Method）[16]来创建Service。这允许应用决定创建何种类型的Service。

10.11 结束语

本文描述服务配置器模式，并阐释了它怎样使服务的实现与它们的配置去耦合。这样的去耦合增强了服务的灵活性和可扩展性。特别地，服务实现可以独立于许多相关于服务配置的问题而持续开发和改进。此外，服务配置器还提供了重配置服务、而又无需修改、重编译或静态重链接已有代码的功能。

服务配置器还使其配置的服务的管理集中化。通过使通用的服务初始化任务自动化（比如打开和关闭文件、获取和释放锁，等等），这样的集中化可以简化编程工作。此外，集中式管理还对服务的生存期提供了更大的控制。

服务配置器模式已被广泛地用于许多环境中。本文使用了一个用C++编写的分布式时间服务作为例子来演示服务配置器模式。使分布式时间服务的组件的开发与它们被配置进系统的时间点去耦合的能力例证了服务配置器模式所提供的灵活性。这样的去耦合允许开发者采用不同的分布式时间算法开发不同的事务员。配置特定事务员的决策成为运行时决策，从而带来了更大的灵活性。本文还显示了怎样将服务配置器模式用于动态重配置分布式时间服务，而又无须修改、重编译或静态重链接运行中的服务器。

服务配置器模式被广泛用于许多环境中，比如Solaris和Windows NT的设备驱动程序、像inetd这样的Internet超级服务器，Windows NT服务控制管理器，以及ACE构架。在每种情况下，服务配置器都使服务的实现与服务的配置得以去耦合。这样的去耦合对应用的可扩展性和灵活性都提供了支持。

参考文献

[1] I. Pyarali, T. H. Harrison, and D. C. Schmidt, “Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging,” *USENIX Computing Systems*, vol. 9, November/December 1996.

[2] R. Gusella and S. Zatti, “The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD,” *IEEE Transactionson Software Engineering*, vol. 15, pp. 847－853, July 1989.

[3] F. Cristian, “Probabilistic Clock Synchronization,” *Distributed Computing*, vol. 3, pp. 146－158, 1989.

[4] 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.), Reading, MA: Addison-Wesley, 1995.

[5] D. C. Schmidt, “Design Patterns for Initializing Network Services: Introducing the Acceptor and Connector Patterns,” *C++ Report*, vol. 7, November/December 1995.

[6] D. C. Schmidt, “Connector: a Design Pattern for Actively Initializing Network Services,” *C++ Report*, vol. 8, January 1996.

[7] D. C. Schmidt and T. Suda, “An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems,” *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280－293, December 1994.

[8] R. G. Lavender and D. C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming,” in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[9] W. Pree, *Design Patterns for Object-Oriented Software Developmen*t. Reading, MA: Addison-Wesley, 1994.

[10] W. R. Stevens, *UNIX Network Programming, First Editio*n. Englewood Cliffs, NJ: Prentice Hall, 1990.

[11] S. Rago, *UNIX System V Network Programmin*g. Reading, MA: Addison-Wesley, 1993.

[12] P. Jain and D. C. Schmidt, “Service Configurator: A Pattern for Dynamic Configuration of Services,” in *Proceedings of the* 3rd *Conference on Object-Oriented Technologies and System*s, USENIX, June 1997.

[13] S. Crane, J. Magee, and N. Pryce, “Design Patterns for Binding in Distributed Systems,” in *The OOPSLA ’95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented System*s, (Austin, TX), ACM, Oct. 1995.

[14] P. Sommerland and F. Buschmann, “The Manager Design Pattern,” in *Proceedings of the* 3rd *Pattern Languages of Programming Conferenc*e, September 1996.

[15] R. G. Lavender and D. C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming,” in *Proceedings of the* 2nd *Annual Conference on the Pattern Languages of Program*s, (Monticello, Illinois), pp. 1－7, September 1995.

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