# Dynamically Configuring Communication Services with the Service Configurator Pattern

Prashant Jain and Douglas C. Schmidt
pjain@cs.wustl.edu and schmidt@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130, (314) 935-7538

This article appeared in the June '97 C++ Report magazine.

## 1 Introduction

A rapidly growing collection of communication services are now available on the Internet. A communication service is a component in a server that provides capabilities to clients. Services available on the Internet include: WWW browsing and content retrieval services (*e.g.*, Alta Vista, Apache, Netscape's HTTP server); software distribution services (*e.g.*, Castinet), electronic mail and network news transfer agents (*e.g.*, `sendmail` and `nntpd`), file access on remote machines (*e.g.*, `ftpd`), remote terminal access (*e.g.*, `rlogind` and `telnetd`), routing table management (*e.g.*, `gated` and `routed`), host and user activity reporting (*e.g.*, `fingerd` and `rwhod`), network time protocols (*e.g.*, `ntpd`), and request brokerage services (*e.g.*, `orbixd` and RPC `portmapper`).

A common way to implement these services is to develop each one as a separate program and then compile, link, and execute each program in a separate process. However, this "static" approach to configuring services yields inflexible, often inefficient, applications and software architectures. The main problem with static configuration is that it tightly couples the *implementation* of a particular service with the *configuration* of the service with respect to other services in an application.

This article describes the *Service Configurator* pattern, which increases application flexibility (and often performance) by decoupling the behavior of services from the point in time at which these service implementations are configured into applications. This article illustrates the Service Configurator pattern using a distributed time service written in C++ as an example. However, the Service Configurator pattern has been implemented in many ways, ranging from device drivers in modern operating systems (like Solaris and Windows NT), to Internet superservers (like `inetd` and the Windows NT Service Control Manager), as well as Java applets.

## 2 The Service Configurator Pattern

### 2.1 Intent

Decouples the behavior of services from the point in time at which service implementations are configured into an application or system.

### 2.2 Also Known As

Super-server

### 2.3 Motivation

The Service Configurator pattern decouples the implementation of services from the time at which the services are configured into an application or a system. This decoupling improves the modularity of services and allows the services to evolve over time independently of configuration issues (such as whether two services must be co-located or what concurrency model will be used to execute the services).

In addition, the Service Configurator pattern centralizes the administration of the services it configures. This facilitates automatic initialization and termination of services and can improve performance by factoring common service initialization and termination patterns into efficient reusable components.

This section motivates the Service Configurator pattern using a distributed time service as an example.

#### 2.3.1 Context

The Service Configurator pattern should be applied when a service needs to be initiated, suspended, resumed, and terminated dynamically. In addition, the Service Configurator pattern should be applied when service configuration decisions must be deferred until run-time.

To motivate this pattern, consider the distributed time service shown in Figure 1. This service provides accurate, fault-tolerant clock synchronization for computers collaborating in local area networks and wide area networks. Synchronized time services are important in distributed systems that require multiple hosts to maintain accurate global time. For instance, large-scale distributed medical imaging systems [1] require

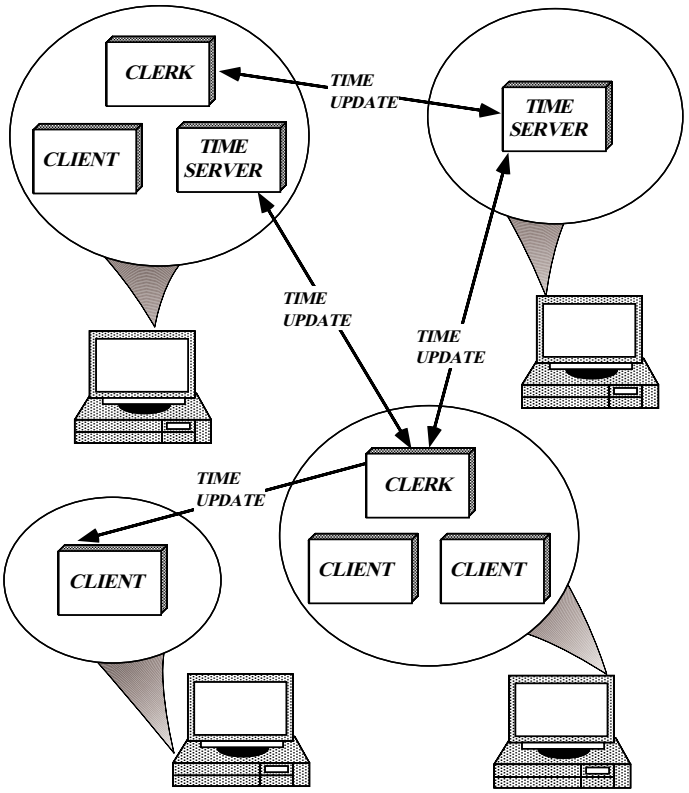


Figure 1: A Distributed Time Service

globally synchronized clocks to ensure that patient exams are accurately timestamped and analyzed expeditiously by radiologists throughout the health-care delivery system.

As shown in Figure 1, the architecture of the distributed time service contains the following Time Server, Clerk, and Client components:

- A *Time Server* answers queries about the time made by Clerks.
- A *Clerk* queries one or more Time Servers to determine the correct time, calculates the approximate correct time using one of several distributed time algorithms [2, 3], and updates its own local system time.
- A *Client* uses the global time information maintained by a Clerk to provide consistency with the notion of time used by clients on other hosts.

## 2.4 A Common Solution

One way to implement a distributed time service is to statically configure the *logical* functionality of Time Servers, Clerks, and Clients into separate *physical* processes. In this approach, one or more hosts would run Time Server processes, which handle time update requests from Clerk processes. The following C++ code fragment illustrates the structure of a statically configured Time Server process:

```
// 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 */
}
```

This program uses the Reactor pattern [4] and the Acceptor pattern [5] to implement a statically configured Time Server process.

Each host that requires global time synchronization would run a Clerk process. The Clerks periodically update their local system time based on values received from one or more Time Servers. The following C++ code fragment illustrates the structure of a statically configured Clerk process:

```
// 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 */
}
```

This program uses the Reactor pattern [4] and the Connector pattern [6] to implement a statically configured Clerk process.

Client processes would use the synchronized time reported by their local Clerk. To minimize communication overhead, the current time could be stored in shared memory that is mapped into the address space of the Clerk and all Clients on the same host. In addition to the time service, other communication services (such as file transfer, remote login, and HTTP servers) provided by the hosts would also execute in separate statically configured processes.

## 2.5 Traps and Pitfalls with the Common Solution

Although the use of patterns like Reactor, Acceptor, and Connector improve the modularity and portability of the distributed time server shown above, configuring communication services using a static approach has the following drawbacks:

• **Service configuration decisions must be made too early in the development cycle:** This is undesirable since developers may not know *a priori* the best way to co-locate or distribute service components. For example, the lack of memory resources in wireless computing environments may force the split of Client and Clerk into two independent processes running on separate hosts. In contrast, in a real-time avionics environment it might be necessary to co-locate the Clerk and Server into one process to reduce communication latency.[1] Forcing developers to commit prematurely to a particular service configuration impedes flexibility and can reduce performance and functionality.

• **Modifying a service may adversely affect other services:** The implementation of each service component is tightly coupled with its initial configuration. This makes it hard to modify one service without affecting other services. For example, in the real-time avionics environment mentioned above, a Clerk and a Time Server might be statically configured to execute in one process to reduce latency. If the distributed time algorithm implemented by the Clerk is changed, however, the existing Clerk code would require modification, recompilation, and static relinking. However, terminating the process to change the Clerk code would terminate the Time Server as well. This disruption in service may not be acceptable for highly available systems (such as telecommunication switches or call centers [7]).

• **System performance may not scale up efficiently:** Associating a process with each service ties up OS resources (such as I/O descriptors, virtual memory, and process table slots). This design can be wasteful if services are frequently idle. Moreover, processes are often the wrong abstraction for many short-lived communication tasks (such as asking a Time Server for the current time or resolving a host address request in the Domain Name Service). In these cases, multi-threaded Active Objects [8] or single-threaded Reactive [4] event loops may be more efficient.

## 2.6 A Better Solution

Often, a more convenient and flexible way to implement distributed services is to use the *Service Configurator pattern*. This pattern decouples the behavior of communication services from the point in time at which these services are

[1] If the Clerk and the Server are co-located in the same process, the Clerk may optimize communication by (1) eliminating the need to set up a socket connection with the Server and (2) directly accessing the Server's notion of time via shared memory.

configured into an application or system. The Service Configurator pattern resolves the following forces:

• **The need to defer the selection of a particular type, or a particular implementation, of a service until very late in the design cycle:** This allows developers to concentrate on a service's functionality (*e.g.*, the clock synchronization algorithm), without committing themselves prematurely to a particular service configuration. By decoupling functionality from configuration, the Service Configurator pattern permits applications to evolve independently of the configuration policies and mechanisms used by the system.

• **The need to build complete applications or systems by composing multiple independently developed services that do not require global knowledge:** The Service Configurator pattern requires all services to have a uniform interface for configuration and control. This allows the services to be treated as building blocks that can be integrated easily as components into a larger application. The uniform interface across all the services makes them "look and feel" the same with respect to how they are configured. In turn, this uniformity simplifies application development by promoting the "principle of least surprise."

• **The need to optimize, control, and reconfigure the behavior of a service at run-time:** Decoupling the implementation of a service from its configuration makes it possible to fine-tune certain implementation or configuration parameters of services. For instance, depending on the parallelism available on the hardware and operating system, it may be either more or less efficient to run multiple services in separate threads or processes. The Service Configurator pattern enables applications to select and tune these behaviors at run-time, when more information may be available to help optimize the services. In addition, adding a new or updated service to a distributed system can often be performed without requiring downtime for existing services.

Figure 2 uses OMT notation to illustrate the structure of the distributed time service designed according to the Service Configurator pattern.

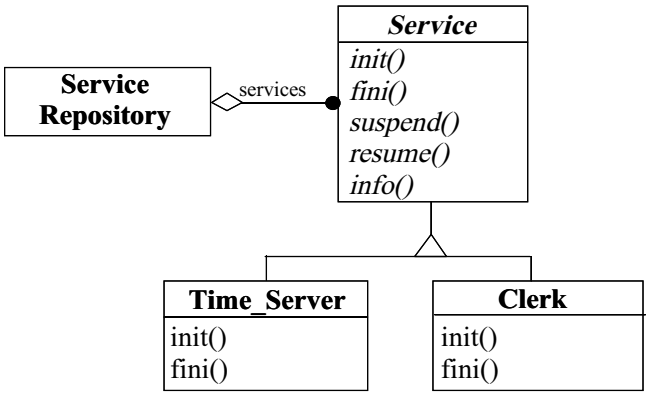


Figure 2: Structure of a Distributed Time Service

The `Service` base class provides a standard interface for configuring and controlling services (such as Time Servers or Clerks). A Service Configurator-based application uses this interface to initiate, suspend, resume, and terminate a service, as well as to obtain run-time information about a service (such as its IP address and port number). The services themselves reside within a `Service Repository` and can be added and removed to and from the `Service Repository` by a Service Configurator-based application.

Two subclasses of the `Service` base class appear in the distributed time service: `Time Server` and `Clerk`. Each subclass represents a concrete `Service` that has specific functionality in the distributed time service. The `Time Server` service is responsible for receiving and processing requests for time updates from `Clerks`. The `Clerk` service is a Connector [6] factory that (1) creates a new connection for every server, (2) dynamically allocates a new handler to send time update requests to a connected server, (3) receives the replies from all the servers through the handlers, and (4) then updates the local system time.

The Service Configurator pattern makes the distributed time service more flexible by managing the configuration of the service components in the time service, thereby decoupling it from the implementation issues. In addition, the Service Configurator provides a framework to consolidate the configuration and management of other communication services under one administrative unit.

# 3 Applicability

Use the Service Configurator pattern when:

- Services must be initiated, suspended, resumed, and terminated dynamically; and
- The implementation of a service may change, but its configuration with respect to related services remains the same and/or the configuration of a group of collocated services may change, but their implementations remain the same; or
- An application or system can be simplified by being composed of multiple independently developed and dynamically configurable services; or
- The management of multiple services can be simplified or optimized by configuring them using a single administrative unit.

Do *not* use the Service Configurator pattern when:

- Dynamic (re)configuration is undesirable due to security restrictions (in this case, static configuration may be necessary[2]); or

[2]It's possible to use the Service Configurator pattern to statically configure services by ensuring that the implementations are statically linked into the main executable program.

- The initialization or termination of a service is too complicated or too tightly coupled with its context to be performed in a uniform manner; or
- A service does not benefit from dynamic configuration since it never changes; or
- Stringent performance requirements mandate the need to minimize the extra levels of indirection incurred by the OS and language mechanisms used for dynamic (re)configuration.

# 4 Structure and Participants

The structure of the Service Configurator pattern is illustrated using OMT notation in Figure 3:

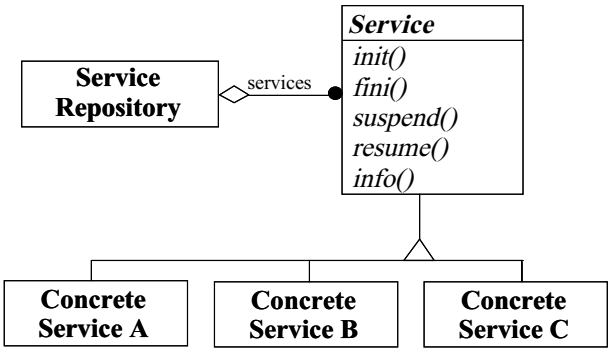


Figure 3: Structure of the Service Configurator Pattern

The key participants in the Service Configurator pattern include the following:

- **Service** (`Service`)
  - Specifies the interface containing hook methods [9] (such as initialization and termination) used by a Service Configurator-based application to dynamically configure the `Service`.
- **Concrete Service** (`Clerk` and `Time Server`)
  - Implements the service's hook methods and other service-specific functionality (such as event processing and communication with clients).
- **Service Repository** (`Service Repository`)
  - Maintains a repository of all the services offered by a Service Configurator-based application. This allows administrative entities to centrally manage and control the behavior of the configured services.

# 5 Collaborations

Figure 4 depicts the collaborations between components in the following three phases of the Service Configurator pattern:

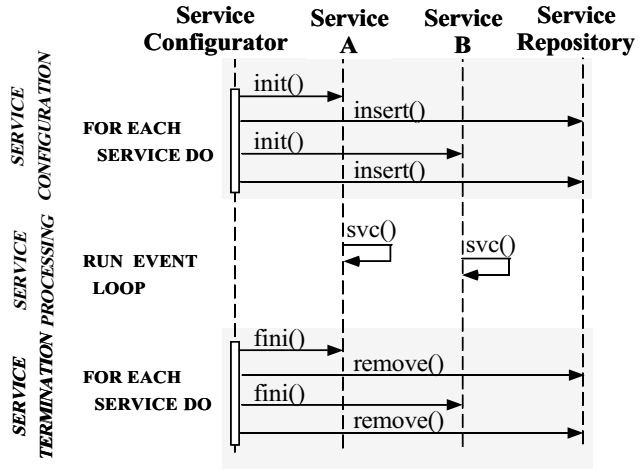


Figure 4: Interaction Diagram for the Service Configurator Pattern

- *Service configuration* – The Service Configurator initializes a `Service` by calling its `init` method. Once the `Service` has been initialized successfully, the Service Configurator adds it to the `Service Repository`, which manages and controls all the `Services`.
- *Service processing* – After it is configured into the system, a `Service` performs its processing tasks (*i.e.*, servicing client requests).[3] While `Service` processing is executing, the Service Configurator can suspend and resume the `Service`.
- *Service termination* – The Service Configurator terminates the `Service` once it is no longer needed by calling the `fini` hook method on the `Service`. This hook allows the `Service` to clean up before terminating. Once a `Service` is terminated, the Service Configurator removes it from the `Service Repository`.

# 6 Consequences

## 6.1 Benefits

The Service Configurator pattern offers the following benefits:

• **Centralized administration:** The pattern consolidates one or more services into a single administrative unit. This helps to simplify development by automatically performing common service initialization and termination activities (such as opening and closing files, acquiring and releasing locks, etc.). In addition, it centralizes the administration of communication services by imposing a uniform set of configuration management operations (such as *initialize*, *suspend*, *resume*, and *terminate*).

[3] Section 7 covers the details of how parameters can be passed into the `Service`, as well as how the `Service` can be activated.

• **Increased modularity and reuse:** The pattern improves the modularity and reusability of communication services by decoupling the implementation of these services from the configuration of the services. In addition, all services have a uniform interface by which they are configured, thereby encouraging reuse and simplifying development of subsequent services.

• **Increased configuration dynamism:** The pattern enables a service to be dynamically reconfigured without modifying, recompiling, or statically relinking existing code. In addition, reconfiguration of a service can often be performed without restarting the service or other active services with which it is co-located.[4]

• **Increased opportunity for tuning and optimization:** The pattern increases the range of service configuration alternatives available to developers by decoupling service functionality from the concurrency strategy used to execute the service. Developers can adaptively tune daemon concurrency levels to match client demands and available OS processing resources by choosing from a range of concurrency strategies. Some alternatives include spawning a thread or process upon the arrival of a client request or pre-spawning a thread or process at service creation time.

### 6.2 Drawbacks

The Service Configurator pattern has the following drawbacks:

• **Lack of determinism:** The pattern makes it hard to determine the behavior of an application until its services are configured at run-time. This can be problematic for real-time systems since a dynamically configured service may not perform predictably when run with certain other services. For example, a newly configured service may consume excessive CPU cycles, thereby starving out other services and causing them to miss their deadlines.

• **Reduced reliability:** An application that uses the Service Configurator pattern may be less reliable than a statically configured application since a particular configuration of services may adversely affect the execution of the services. For instance, a faulty service may crash, thereby corrupting state information it shares with other services. This is particularly problematic if multiple services are configured to run within the same process.

• **Increased overhead:** The pattern adds extra levels of indirection to execute a service. For instance, the Service Configurator first initializes the service and then loads it into the `Service Repository`. This may incur excessive overhead in time-critical applications. In addition, the use of dynamic linking to implement the Service Configurator pattern adds extra levels of indirection to method invocations and global variable accesses.

[4]It is beyond the scope of the Service Configurator pattern to ensure robust dynamic service reconfiguration. Supporting robust reconfiguration is primarily a matter of *protocols* and *policies*, whereas the Service Configurator pattern primarily addresses (re)configuration *mechanisms*.

| Step | Common Alternatives |
|---|---|
| Define the service control interface | • Services inherit from an abstract base class<br>• Services respond to control messages |
| Define a Service Repository | • Maintain a table of service implementations |
| Select a configuration mechanism | • Specify at command line<br>• Specify through a user interface<br>• Specify through a configuration file |
| Determine service execution mechanism | • Reactive execution<br>• Multi-threaded Active Objects<br>• Multi-process Active Objects |

Table 1: Steps Involved in Implementing the Service Configurator Pattern

## 7 Implementation

The Service Configurator pattern may be implemented in many ways. This section explains the steps and alternatives involved when implementing the pattern. These steps and alternatives are summarized in Table 1.

• **Define the service control interface:** The following is the basic interface that services must support to enable the Service Configurator to configure and control a service:

- *Service initialization* - provide an entry point into the service and perform initialization of the service.
- *Service termination* - terminate execution of a service.
- *Service suspension* - temporarily suspend the execution of a service.
- *Service resumption* - resume execution of a suspended service.
- *Service information* - report information (*e.g.*, port number or service name) that describes a service.

There are two basic approaches to defining the service control interface, *inheritance-based* and *message-based*:

- *Inheritance-based* – This approach has each service inherit from a common base class. This is the approach used by the ACE `Service Configurator` framework [7] and Java applets. It works by defining an abstract base class containing a number of pure virtual "hook" methods, as follows:

```
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;
};
```

The `init` method serves as an entry point into a `Service`. It is used by the Service Configurator to initialize the execution of a `Service`. The `fini` method allows the Service Configurator to terminate the execution of a `Service`. The `suspend` and `resume` methods are scheduling hooks used by the Service Configurator to suspend and resume the execution of a `Service`. The `info` method allows the Service Configurator to obtain `Service`-related information (such as its name and network address). Together, these methods impose a uniform contract between the Service Configurator and the `Services` that it manages.

- *Message-based* – Another way to control communication services is to program each `Service` to respond to a specific set of messages. This makes it possible to integrate the Service Configurator into non-OO programming languages that lack inheritance (such as C or Ada83).

  The Windows NT Service Control Manager (`SCM`) uses this scheme. Each Window NT host has a master `SCM` process that automatically initiates and manages system services by passing them various control messages (such as `PAUSE`, `RESUME`, and `TERMINATE`). Each developer of an `SCM`-managed service is responsible for writing code to process these messages.

• **Define a Repository:** A `Service Repository` is used to maintain all the service implementations such as objects, executable programs, or dynamically linked library (DLLs). A Service Configurator uses the `Service Repository` to access a service when it is configured into or removed from the system. Each service's current status (such as whether it's active or suspended) is maintained in the Repository, as well. The Service Configurator stores and accesses the `Service Repository` information in main memory, the file system, or the kernel (*e.g.*, it may use operations that report the status of processes and threads).

• **Select a configuration mechanism:** A service needs to be configured before it can be executed. To configure a service requires specifying attributes that indicate the location of the service's implementation (such as an executable program or DLL), as well as the parameters required to initialize a service at run-time. This configuration criteria can be specified in various ways (such as on the command line, via environment variables, through a user interface, or in a configuration file). A centralized configuration mechanism simplifies the installation and administration of the services in an application by consolidating service attributes and initialization parameters in a single location.

• **Determine the service execution mechanism:** A service that has been dynamically configured by a Service Configurator can be executed using various combinations of Reactive [4] and Active Object [8] schemes. These alternatives are examined briefly below:

- *Reactive execution* - A single thread of control can be used for the Service Configurator, as well the execution of all the services it configures.
- *Multi-threaded Active Objects* - This approach runs the dynamically configured services in their own threads of control within the Service Configurator process. The Service Configurator can either spawn new threads "on-demand" or execute the services within an existing pool of threads.
- *Multi-process Active Objects* - This approach runs the dynamically configured services in their own processes. The Service Configurator can either spawn new processes "on-demand" or execute the services within an existing pool of processes.

# 8 Sample Code

The following code presents an example of the Service Configurator pattern written in C++. The example focuses on the configuration-related aspects of the distributed time service presented in Section 2.3. In addition, this example illustrates the use of other patterns (such as the Reactor pattern [4] and the Acceptor [5] and Connector [6] patterns) that are commonly used to develop communication services and Object Request Brokers.

In the example below, the `Concrete Service` class in the OMT class diagram shown in Figure 3 is represented by the `Time Server` class, as well as the `Clerk` class. The C++ code in this section implements the `Time Server` and the `Clerk` classes.[5] Both classes inherit from `Service`, allowing them to be dynamically configured into an application. In addition, the approach uses a configuration mechanism based on explicit dynamic linking [7] and a configuration file to dynamically configure the Clerk and Server portions of the distributed time service. The service execution mechanism is based on the reactive event handling model within a single thread of control.

The example shows how the Clerk component can change the algorithm it uses to compute the local system time *without* affecting the execution of other components configured by the Service Configurator. Once the algorithm has been modified, the Clerk component is reconfigured dynamically by the Service Configurator.

The code shown below also includes the `main` driver function, which provides the generic entry point into any Service Configurator-based application. The implementation runs on both UNIX/POSIX and Win32 platforms using ACE [7], which can be obtained via the WWW at `www.cs.wustl.edu/~schmidt/ACE.html`.

[5] To save space, most of the error handling code has been omitted.

## 8.1 The Time Server Class

The `Time Server` uses the `Acceptor` class to accept connections from one or more Clerks. The `Acceptor` class uses the Acceptor pattern [5] to create handlers for every connection from Clerks that want to receive requests for time updates. This design decouples the implementation of the Time Server from its configuration. Therefore, developers can change the implementation of the Time Server independently of its configuration. This provides flexibility with respect to evolving the implementation of the Time Server.

The `Time Server` class inherits from the `Service` base class defined in Section 7. This enables the Service Configurator to dynamically link and unlink a `Time Server`. Before loading the `Time Server` service into the `Service Repository`, the Service Configurator invokes its `init` hook. This method performs the `Time Server`-specific initialization code. Likewise, the `fini` hook method is called automatically by the Service Configurator to terminate the service when it is no longer needed.

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

Note that the Service Configurator can also suspend and resume the `Time Server` by calling its `suspend` and `resume` hooks, respectively.

## 8.2 The Clerk Class

The `Clerk` uses the `Connector` class to establish and maintain connections with one or more `Time Servers`. The `Connector` class uses the Connector pattern [6] to create handlers for each connection to a Time Server. The handlers receive and process time updates from the `Time Servers`.

The `Clerk` class inherits from the `Service` base class. Therefore, like the `Time Server`, it can be dynamically configured by the Service Configurator. The Service Configurator can initialize, suspend, resume, and terminate the `Clerk` by calling its `init`, `suspend`, `resume`, and `fini` hooks, respectively.

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

The `Clerk` periodically sends a request for time update to all its connected `Time Servers` by iterating over its list of handlers. Once the `Clerk` receives responses from all its connected `Time Servers`, it recalculates its notion of the local system time. Thus, when Clients ask the Clerk for the current time, they receive a globally synchronized time value.

## 8.3 Configuring an Application

### 8.3.1 Co-located Configuration

The following code illustrates the dynamic configuration and execution of an application that uses a configuration file to co-locate the `Time Server` and the `Clerk` within the same OS process:

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

  // Perform daemon services updates.
  daemon.run_event_loop ();
  /* NOTREACHED */
}
```

| Directive | Description |
|---|---|
| **dynamic** | Dynamically link and enable a service |
| **static** | Enable a statically linked service |
| **remove** | Completely remove a service |
| **suspend** | Suspend service without removing it |
| **resume** | Resume a previously suspended service |

Table 2: Service Configuration Directives

This completely generic `main` program configures communication services dynamically within the constructor of the `Service Config` object. This method consults the following `svc.conf` configuration file:

```
# 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
```

Each entry in the `svc.conf` configuration file is processed by the ACE Service Configurator framework. The framework interprets the `dynamic` directive as a command to dynamically link the designated `Service` into the application process. Table 2 summarizes the available service configuration directives.

For example, the first entry in the `svc.conf` file specifies a service name (`Time_Server`) that is used by the `Service Repository` to identify the dynamically configured `Service`. The `make_Time_Server` is a factory function located in the dynamic link library `netsvcs.dll`. The Service Configurator framework dynamically links this DLL into the application's address space and then invokes the `make_Time_Server` Factory function. This function dynamically allocates a new `Time Server` instance, as follows:

```
Service *make_Time_Server(void)
{
  return new Time_Server;
}
```

The final string parameter in the first entry specifies an environment variable containing a port number that the `Time Server` will listen on for `Clerk` connections. The Service Configurator converts this string into an "`argc/argv`"-style vector and passes it to the `init` hook of the `Time Server`. If the `init` method succeeds, the `Service *` is stored in the `Service Repository` under the name `Time_Server`.

The second entry in the `svc.conf` file specifies how to dynamically configure a `Clerk`. As before, the Service Configurator dynamically links `netsvcs.dll` DLL into the application's address space and invokes the `make_Clerk` factory function to create a new `Clerk` instance. The `init` hook is passed the names and port numbers of three Time Servers (`tango.cs`, `perdita.wuerl`, and `atomic-clock.lanl`). In addition, the `-P 10` option defines how frequently the Clerk will poll the Time Servers.

### 8.3.2 Distributed Configuration

Suppose we did not want to co-locate the `Time Server` and the `Clerk` in order to reduce the memory footprint of an application. Since we're using the Service Configurator pattern, all that's required is to split the `svc.conf` file into two parts. One part contains the `Time Server` entry and the other part contains the `Clerk` entry. The services themselves would not have to change by virtue of the fact that the Service Configurator pattern has decoupled their behavior from their configuration. Figure 5 shows what the configuration looks like with the `Time Server` and `Clerk` co-located in the same process, as well as what the configuration looks like after the split.

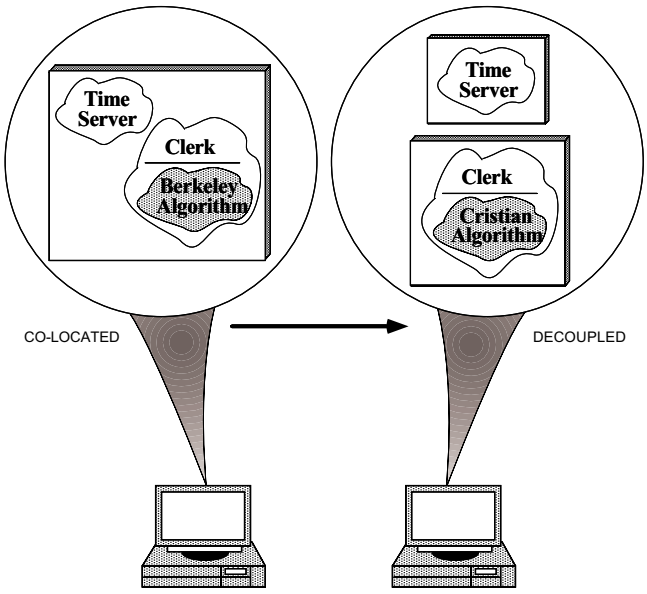


Figure 5: Reconfiguring a Time Server and a Clerk

## 8.4 Reconfiguring an Application

Now suppose we need to change the algorithm implementation of the `Clerk`. For example, we may decide to switch from an implementation of the Berkeley algorithm [2] to an implementation of Cristian's algorithm [3], both of which are outlined below:

- *Berkeley algorithm* – In this approach, the `Time Server` is an active component that polls every machine in the network periodically to ask what time it is there. Based on the responses it receives, it computes an aggregate notion of the correct time and tells all the machines to adjust their clocks accordingly.
- *Cristian's algorithm* – In this approach, the `Time Server` is a passive entity responding to queries made by Clerks. It does not actively query other machines to determine its own notion of time.

A change in the time synchronization algorithm may be necessary in response to the characteristics of the `Time Server`. For instance, if the machine on which the `Time Server` resides has a WWV receiver[6] the `Time Server` can act as a passive entity and Cristian algorithm would be appropriate. On the other hand, if the machine on which the `Time Server` resides does not have a WWV receiver then an implementation of the Berkeley algorithm would be more appropriate.

Figure 5 shows a change in the Clerk implementation (corresponding to the change in clock synchronization algorithm). The change takes place in the process of separating the `Time Server` and the `Clerk`, which were previously co-located.

Ideally, we'd like to change the algorithm implementation without affecting the execution of other services or other components of the time service. Accomplishing this using the Service Configurator simply requires the following modification to the `svc.conf` file:

```
# Terminate Clerk
remove Clerk
```

The only additional requirement is to have the Service Configurator process this directive. This can be done by generating an external event (such as the UNIX SIGHUP signal, an RPC notification, or a Windows NT Registry event). On receipt of this event, the application would consult the configuration file again and terminate the execution of the Clerk service. The Service Configurator would call the `fini` method of the `Clerk` and thereby terminate the execution of the Clerk component. The execution of other services should not be affected.

Once the Clerk service has been terminated, changes can be made to the algorithm implementation. The code can then be recompiled and relinked to form a new `netsvcs` DLL. A similar approach can be taken to add a Clerk service back to the Service Configurator. The configuration file would be modified with a new directive specifying that the Clerk be dynamically linked, as follows:

```
# 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"
```

[6] A WWV receiver intercepts the short pulses broadcasted by the National Institute of Standard Time (NIST) to provide Universal Coordinated Time (UTC) to the public.

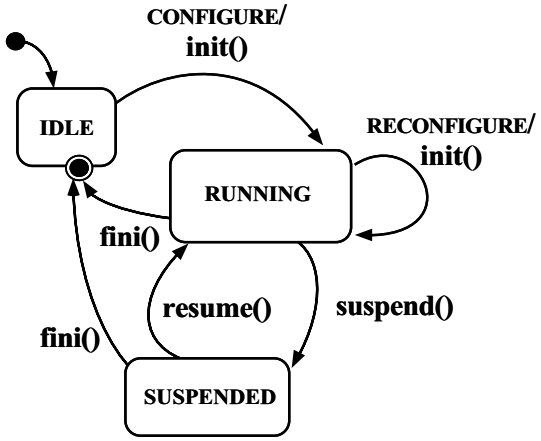

Figure 6: State Diagram of the Service Lifecycle

```
"-P 10" # polling frequency
```

An external event would then be generated, causing the process to reread the configuration file and add the Clerk component to the repository. The Clerk component would begin execution once the `init` method of `Clerk` is called by the Service Configurator framework.

Figure 6 shows a state diagram of the lifecycle of a `Service` such as the Clerk service.
Note that in the entire process of terminating and removing the Clerk service, no other active services are affected by changes to the implementation and reconfiguration of the Clerk service. The ease of being able to substitute a new service implementation further exemplifies the flexibility offered by the Service Configurator pattern.

# 9 Known Uses

The Service Configurator pattern is widely used in system and application programming environments including UNIX, Windows NT, ACE, and Java applets:

• **Modern operating system device drivers:** Most modern operating systems (such as Solaris and Windows NT) provide support for dynamically configurable kernel-level device drivers. These drivers can be linked into and unlinked out of the system dynamically via `init`/`fini`/`info` hooks. These operating systems use the Service Configurator pattern to allow administrators to reconfigure the OS kernel without having to shut it down, recompile and statically relink new drivers, and restart the system.

• **UNIX network daemon management:** The Service Configurator pattern has been used in "superservers" that manage UNIX network daemons. Two widely available network daemon management frameworks are `inetd` [10] and `listen` [11]. Both frameworks consult configuration files that specify (1) service names (such as the standard Internet services `ftp`, `telnet`, `daytime`, and `echo`), (2) port numbers to listen on for clients to connect with these services, and (3) an executable file to invoke and perform the service when a client connects. Both frameworks contain a master Acceptor [5] process that monitors the set of ports associated with the services. When a client connection occurs on a monitored port, the Acceptor process accepts the connection and demultiplexes the request to the appropriate pre-registered service handler. This handler performs the service (either reactively or in an active object) and returns any results to the client.

• **The Windows NT Service Control Manager (SCM):** Unlike `inetd` and `listen`, the Windows NT Service Control Manager (`SCM`) is not a port monitor, per se. That is, it does not provide built-in support for listening to a set of I/O ports and dispatching server processes "on-demand" when client requests arrive. Instead, it provides an RPC-based interface that allows a master `SCM` process to automatically initiate and control (*i.e.*, pause, resume, terminate, etc.) administrator-installed services (such as `remote registry access`). These services would otherwise run as separate threads within either a single-service or a multi-service daemon process. Each installed service is individually responsible for configuring itself and monitoring any communication endpoints, which can be more general than socket ports. For instance, the `SCM` can control named pipes and shared memory.

• **The ADAPTIVE Communication Environment (ACE) framework:** The ACE framework [7] provides a set of C++ mechanisms for configuring and controlling communication services dynamically. The ACE `Service Configurator` extends the mechanisms provided by `inetd`, `listen`, and `SCM` to automatically support dynamic linking and unlinking of communication services. The code contained in Section 8 shows how the ACE framework is used to implement the distributed time service. The mechanisms provided by ACE were influenced by the interfaces used to configure and control device drivers in modern operating systems. Rather than targeting kernel-level device drivers, however, the ACE `Service Configurator` framework focuses on dynamic configuration and control of application-level `Service` objects.

• **Java applets:** – The applet mechanism in Java uses the Service Configurator pattern. Java provides support for downloading, initializing, starting, suspending, resuming, and terminating applets. It does this by providing methods (*e.g.*, `start` and `stop`) that initiate and terminate threads. A method in a Java applet can access the thread it is running under using `Thread.currentThread()`, and then issue control messages to it such as `suspend`, `resume`, and `stop`. [12] presents an example that illustrates how the Service Configurator pattern is used for Java applets.

# 10 Related Patterns

The intent of the Service Configurator pattern is similar to the Configuration pattern [13]. The Configuration pattern

decouples structural issues related to configuring services in distributed applications from the execution of the services themselves. The Configuration pattern has been used in frameworks for configuring distributed systems to support the construction of a distributed system from a set of components. In a similar way, the Service Configurator pattern decouples service initialization from service processing. The primary difference is that the Configuration pattern focuses more on the active composition of a chain of related services, whereas the Service Configurator pattern focuses on the dynamic initialization of service handlers at a particular endpoint. In addition, the Service Configurator pattern focuses on decoupling service behavior from the service's concurrency strategies.

The Manager Pattern [14] manages a collection of objects by assuming responsibility for creating and deleting these objects. In addition, it provides an interface to allow clients access to the objects it manages. The Service Configurator pattern can use the Manager pattern to create and delete Services as needed, as well as to maintain a repository of the Services it creates using the Manager Pattern . However, the functionality of dynamically configuring, initializing, suspending, resuming, and terminating a Service created using the Manager Pattern must be added to fully implement the Service Configurator Pattern.

A Service Configurator often makes use of the Reactor [4] pattern to perform event demultiplexing and dispatching on behalf of configured services. Likewise, dynamically configured services that execute for a long periods of time often use the Active Object pattern [15].

Administrative interfaces (such as configuration files or GUIs) to a Service Configurator-based system provide a Facade [16]. This Facade simplifies the management and control of applications that are executing within the Service Configurator.

The virtual methods provided by the `Service` base class are callback "hooks" [9]. These hooks are used by the Service Configurator to initiate, suspend, resume, and terminate services.

A `Service` (such as the `Clerk` class described in Section 8) may be created using a Factory Method [16]. This allows an application to decide which type of `Service` subclass to create.

## 11 Concluding Remarks

This article describes the Service Configurator pattern and illustrates how it decouples the implementation of services from their configuration. This decoupling increases the flexibility and extensibility of services. In particular, service implementations can be developed and evolved over time independently of many issues related to service configuration. In addition, the Service Configurator patten provides the ability to reconfigure a service without modifying, recompiling, or statically relinking existing code.

The Service Configurator pattern also centralizes the administration of services it configures. This centralization can simplify programming effort by automating common service initialization tasks (such as opening and closing files, acquiring and releasing locks, etc). In addition, centralized administration can provide greater control over the lifecycle of services.

The Service Configurator pattern has been applied widely in many contexts. This article used a distributed time service written in C++ as an example to demonstrate the Service Configurator pattern. The ability to decouple the development of the components of a distributed time service from their configuration into the system exemplifies the flexibility offered by the Service Configurator pattern. This decoupling allows different Clerks to be developed with different distributed time algorithms. The decision to configure a particular Clerk becomes a run-time decision, which yields greater flexibility. The article also showed how the Service Configurator pattern can be used to dynamically reconfigure the distributed time service without having to modify, recompile, or statically relink running servers.

The Service Configurator pattern is widely used in many contexts such as device drivers in Solaris and Windows NT, Internet superservers like inetd, the Windows NT Service Control Manager, and the ACE framework. In each case, the Service Configurator pattern decouples the implementation of a service from the configuration of the service. This decoupling supports both extensibility and flexibility of applications.

## References


[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 Transactions on 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 Development*. Reading, MA: Addison-Wesley, 1994.

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

[11] S. Rago, *UNIX System V Network Programming*. 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* $3^{rd}$ *Conference on Object-Oriented Technologies and Systems*, 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 Systems*, (Austin, TX), ACM, Oct. 1995.

[14] P. Sommerland and F. Buschmann, "The Manager Design Pattern," in *Proceedings of the* $3^{rd}$ *Pattern Languages of Programming Conference*, September 1996.

[15] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Proceedings of the* $2^{nd}$ *Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–7, September 1995.

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