Service Configurator

A Pattern for Dynamic Configuration and Reconfiguration of Communication Services

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 paper appeared in the Proceedings of the 3^{rd} Annual Pattern Languages of Programming Conference, Allerton Park, Illinois.

Abstract

This paper explores the Service Configurator pattern, which decouples the implementation of communication services from their configuration. This pattern increases the flexibility and extensibility of communication services by enabling their configuration and reconfiguration at any point in time without affecting other services. The Service Configurator pattern is widely used in system and application programming environments including UNIX, Windows NT, and Java.

1 Intent

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

2 Also Known As

Super-server

3 Motivation

3.1 Context

To illustrate the need for the Service Configurator pattern, consider the ever-increasing range of communication services offered in the Internet:

- WWW browsing and content retrieval services (*e.g.,* `httpd`);
- 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`).

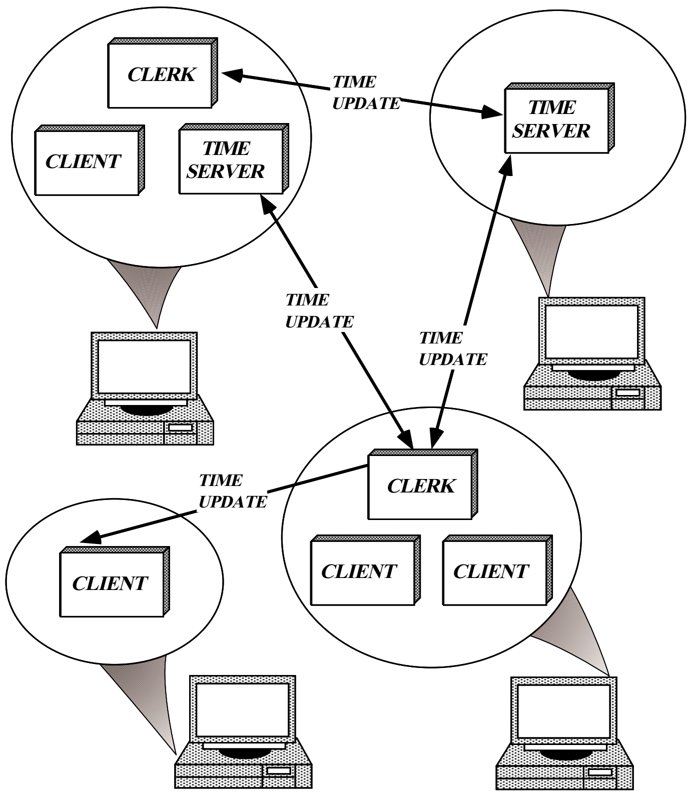

Figure 1: A Distributed Time Service

- Routing table management (*e.g.,* `gated` and `routed`);
- Host and user activity reporting (*e.g.,* `fingerd`);
- Network time protocols (*e.g.,* `ntpd`);
- Object 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 running in a separate process. As a concrete example, 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 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.

3.2 Common Traps and Pitfalls

One way to implement the distributed time service is to hard-code the *logical* functionality of Time Servers, Clerks, and Clients in separate *physical* processes. In this approach, one or more hosts would run Time Server processes, which handle time update requests from Clerk processes. Hosts that require global time synchronization would run a Clerk process to periodically update their local system time based on values received from Time Servers. Client processes would then use the synchronized time reported by their local Clerk. In addition, other communication services (such as file transfer, remote login, and WWW access) provided by the hosts would also execute in separate hard-coded processes.

However, implementing and configuring communication services in this manner 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 may 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 described above, a Clerk and a Time Server might be hard-coded 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 relinking. However, terminating the process to change the Clerk would terminate the Time Server as well. This disruption in service may not be acceptable for highly available distributed systems (such as telecommunication switches or call centers [4]).

• **System performance may not scale up efficiently:** Associating a process for each service ties up OS resources (such as descriptors, virtual memory, and process table slots). This 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, a thread or a reactive event loop may be more efficient.

3.3 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 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 the functionality of a service, 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 easily integrated as components in a large application. The uniform interface across all the services makes them "look and feel" the same with respect to how they are configured, thereby simplifying application development.

• **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 more or less efficient to run one or more 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

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

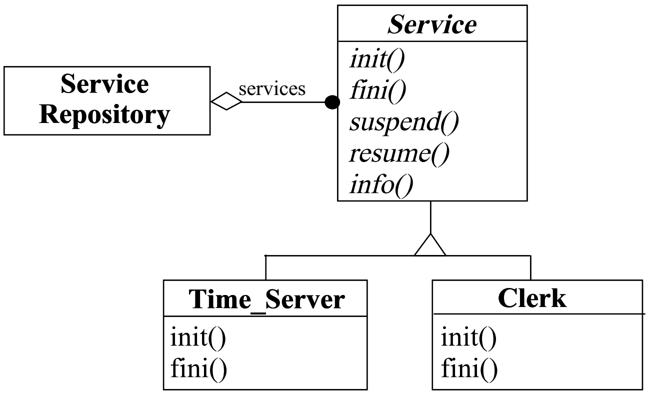

Figure 2: Structure of a Distributed Time Service

service to a distributed system should not require 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. 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 (such as service access point) about the service.

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 [5] 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 of 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.

4 Applicability

Use the Service Configurator pattern when:

- Services must be initiated, suspended, resumed, and terminated dynamically;
- The implementation of a service may change, but its configuration with respect to related services remains the same and/or the configuration of services may change, but their implementations remain the same;

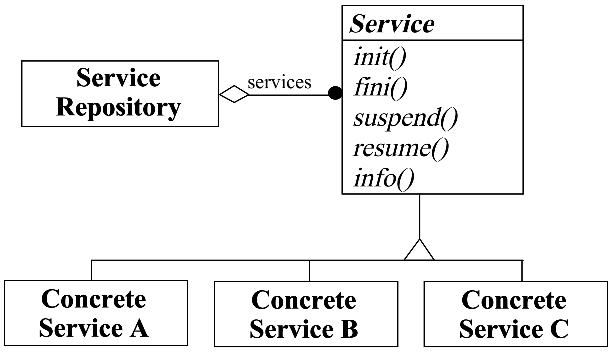

Figure 3: Structure of the Service Configurator pattern

- An application or system can be simplified by being composed of multiple independently developed and dynamically configurable services;
- The management of multiple services can be simplified or optimized by configuring and reconfiguring 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);
- The initialization or termination of a service is too complicated or too tightly coupled to its context to be performed in a uniform manner;
- A service does not benefit from dynamic configuration or reconfiguration since it never changes;
- 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.

5 Structure and Participants

The structure of the Service Configurator pattern is illustrated using OMT notation in Figure 3. The key participants in the Service Configurator pattern include the following:

- **Service** (`Service`)
  - Specifies the interface that contains hooks used by a Service Configurator-based application to dynamically configure and reconfigure the `Service`.
- **Concrete Service** (`Clerk` and `Time Server`)
  - Implements the service hooks (such as initialization and termination) and other service-specific functionality (such as event processing, and communication with clients and other services).
- **Service Repository** (`Service Repository`)

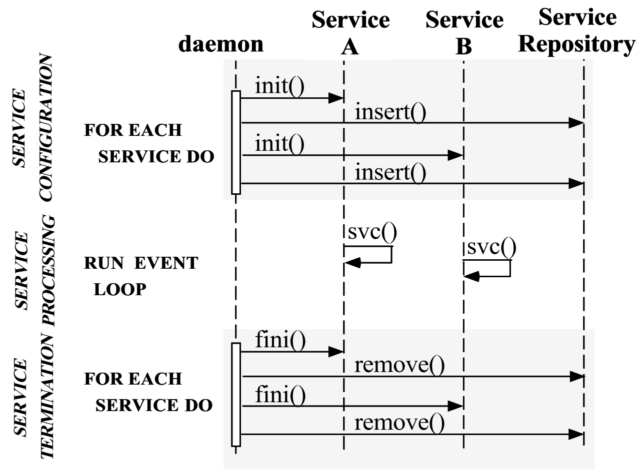

Figure 4: Interaction Diagram for the Service Configurator Pattern

  – 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 services.

6 Collaborations

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

- *Service configuration* – The Service Configurator initializes a `Service` by calling the `init` method on it. Once the `Service` has been initialized successfully, the Service Configurator adds it to the `Service Repository`. The `Service Repository` is used by the Service Configurator to manage and control all the `Services`.
- *Service processing* – Once a `Service` has been configured into the system it is executed. While the `Service` 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. The Service Configurator calls the `fini` method on the `Service` to allow it to clean up before terminating. Once a `Service` is terminated, it is removed from the `Service Repository`.

7 Consequences

7.1 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 over them (such as *initialize*, *suspend*, *resume*, and *terminate*).

• **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.

• **Increased configuration dynamism:** The pattern enables a service to be dynamically reconfigured without modifying, recompiling, or 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.[2]

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

7.2 Drawbacks

The Service Configurator pattern has the following drawbacks:

• **Lack of determinism:** The pattern makes it hard to determine the behavior of a service until run-time. This is particularly problematic for real-time systems since a dynamically configured service may not perform predictably when run with certain other services. For example, one service may consume excessive CPU cycles, thereby starving out other services.

• **Reduced reliability:** An application that uses the Service Configurator pattern may be less reliable 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.

[2] It is beyond the scope of this 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*.

• **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 be undesirable or an unnecessary overhead in time-critical applications. In addition, the use of dynamic linking often adds extra levels of indirection to call functions and to access global variables [6].

8 Implementation

The Service Configurator pattern may be implemented in many ways. This section explains the steps involved when implementing the pattern.

• **Define the service control interface:** The following is the core interface that services must support to enable the Service Configurator to configure and control the 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* - obtain information about a service to determine its characteristics.

There are two basic approaches to defining the service control interface. One is to have each service inherit from a common base class. This is the approach used by the ACE `Service Configurator` framework [4]. 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 and execute a `Service`. The `fini` method allows the Service Configurator to terminate the execution of a `Service`. The `suspend` and `resume` methods serve as scheduling hooks and are used by the Service Configurator to suspend and then 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).

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

Table 1: Steps Involved in Implementing the Service Configurator Pattern

Together, these methods impose a contract between the Service Configurator and the `Services` that it manages.

A different way to control communication services involves programming each `Service` to respond to a specific set of messages. This makes it possible to integrate the Service Configurator into non-OO programming languages. The Windows NT Service Control Manager (`SCM`) [7] 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 is active or suspended) is typically 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, through a user interface, or through 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 [8] and Active Object [9] 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.

Table 1 summarizes the steps involved in implementing the Service Configurator pattern along with the options available at each step.

9 Sample Code

The following code illustrates an example of the Service Configurator pattern. The example focuses on the configuration-related aspects of the distributed time service described in Section 3. In addition, this example illustrates the use of other patterns (such as the Reactor pattern [8] and the Acceptor and Connector patterns [5]) that are commonly used to develop communication services.

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.[3] 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 [4] 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 model with 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 forms the generic entry point into any Service Configurator-based application. The implementation runs on both UNIX and Win32 platforms using ACE [4], which can be obtained via the WWW at `www.cs.wustl.edu/~schmidt/ACE.html`.

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

9.1 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 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 over time.

The `Time Server` class inherits from the `Service` base class defined in Section 8. 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` is called automatically by the Service Configurator to terminate the service when it is no longer needed.

```
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 Reactor pattern)
    acceptor_->open (port_);
  }

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

  // info(), suspend(), and resume() methods omitted.

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

  // Acceptor used to accept all connections.
  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.

9.2 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 to create handlers for every 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.

```
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);

    // Use the Iterator pattern to set up
    // connections.
    Clerk_Handler **handler = 0;

    for (ITERATOR iterator (handler_set_);
         iterator.next (handler) != 0;
         iterator.advance ()) {
      connector_.connect (*handler);
  }

  // Terminate the service when dynamically unlinked.
  virtual int fini (void) {
    // Disconnect all time servers
    Clerk_Handler **handler = 0;
    for (ITERATOR iterator (handler_set_);
         iterator.next (handler) != 0;
         iterator.advance ())
      (*handler)->close ();
  }

  // info(), suspend(), and resume() methods omitted.

  // Called periodically to compute
  // local system time.
  int update_time (long t) {
    // Periodically query the servers by
    // iterating over the handler set and obtaining
    // time update from every Time Server.
    // Implement Clock Synchronization algorithm
    // here to compute local system time.
  }
private:
  // Parse command line arguments or those
  // specified by the configuration file and
  // create Clerks for every server specified.
  int parse_args (int argc, char *argv[]);

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

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

  // Connector used to set up connections
  // to all servers.
  Connector connector_;
};
```

The `Clerk` periodically sends a request for time update to all its connected `Time Servers` by iterating over the 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 value.

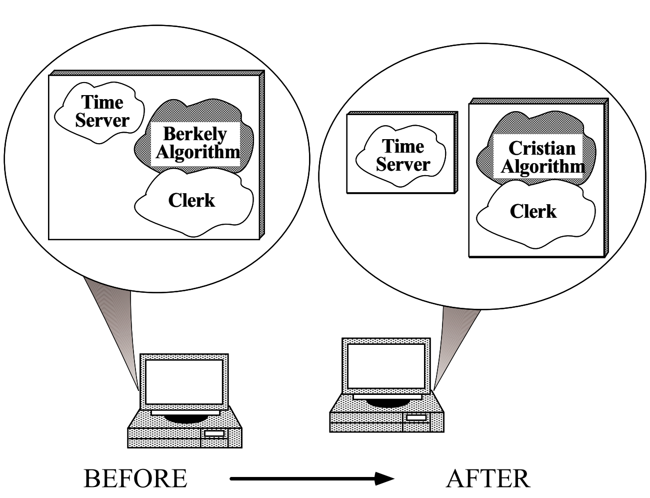

Figure 5: Reconfiguring a Time Server and a Clerk

9.3 Configuring an Application

9.3.1 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 */
}
```

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 *
   libnet_svcs_a.dll:make_Time_Server()
     "-p $TIME_SERVER_PORT"

# Configure a Clerk.
dynamic Active_Time_Server_Clerk Service *
   libnet_svcs_a.dll:make_Clerk()
     "-h tango.cs:$TIME_SERVER_PORT"
     "-h perdita.wuerl:$TIME_SERVER_PORT"
     "-h atomic-clock.lanl.gov:$TIME_SERVER_PORT"
```

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.

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 `libnet_svcs_a.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`.

9.3.2 Distributed Configuration

If we did not wanted to co-locate the `Time Server` and the `Clerk` all that is required is to split the `svc.conf` file into two parts. One part would contain the `Time Server` entry and the other part would contain 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 would look like with the `Time Server` and `Clerk` co-located in the same process and what the configuration would look like after the split.

9.4 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` acts as an active component, polling every machine periodically to ask what time it is there. Based on the responses it receives, it computes its notion of the correct time and then 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 algorithm may be needed in response to the characteristics of the `Time Server`. If the machine on which the `Time Server` resides has a WWV receiver[4] the `Time Server` can act as a passive entity and Cristian

[4] 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.

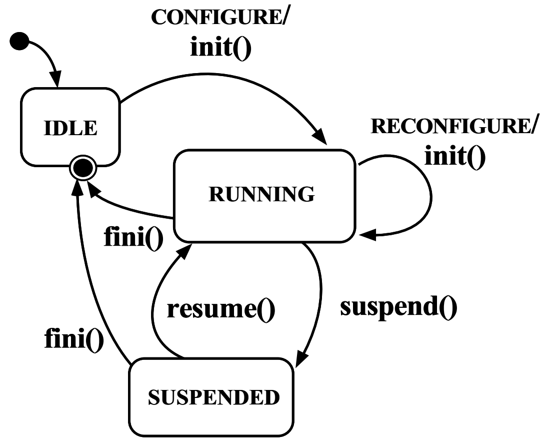

Figure 6: State Diagram of the Service Lifecycle

algorithm would be best suited. 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` that were previously co-located.

Ideally, we want to perform this change in 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 Time_Server_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 or an RPC notification). 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 will 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 add a Clerk service back to the Service Configurator, a similar approach as above can be taken. The configuration file would be modified with a new directive specifying dynamically linking the Clerk, as follows:

```
# Reconfigure a new Clerk.
dynamic Passive_Time_Server_Clerk Service *
   libnet_svcs_p.dll:make_Clerk()
     "-h tango.cs:$TIME_SERVER_PORT"
     "-h perdita.wuerl:$TIME_SERVER_PORT"
     "-h atomic-clock.lanl.gov:$TIME_SERVER_PORT"
```

An external event would then be generated, causing the configuration file to be re-read and the Clerk component to be added to the repository. The Clerk component would begin execution with the `init` method of `Clerk` getting called.

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.

10 Known Uses

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

• **Modern operating system device drivers:** Most modern operating systems (such as Solaris [10] and Windows NT [7]) provide support for dynamically configurable kernel-level device drivers. These drivers can be linked into and unlinked out of the system dynamically. This makes it possible to reconfigure the operating system without having to shut it down, recompile and relink new drivers into the kernel, and then 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` [11] and `listen` [12]. Both frameworks consult configuration files that specify () 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. These frameworks contain a master Acceptor [5] process that reactively 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:** Unlike `inetd` and `listen`, the Windows NT Service Control Manager (`SCM`) [7] is not a port monitor. 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 may be more general than I/O ports, *e.g.,* named pipes or shared memory).

• **The ADAPTIVE Communication Environment (ACE) framework:** The ACE framework [13] provide a set of C++ mechanisms for configuring and controlling communication services dynamically. This `Service Configurator` framework extends the mechanisms provided by `inetd`, `listen`, and `SCM` to automatically support dynamic linking and unlinking of communication services. The code contained in Section 9 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`, `stop`, and destroy) that control the behavior of 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`.

11 Related Patterns

A Service Configurator often makes use of the Reactor [8] pattern to perform event demultiplexing and dispatching on behalf of configured services. Likewise, dynamically configured services that execute for a long-duration often use the Active Object pattern [14].

Administrative interfaces (such as configuration files or GUIs) to a Service Configurator-based system provide a Facade [15]. 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" [16]. 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 9) may be created using a Factory Method [15]. This allows an application to decide which type of `Service` subclass to create.

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," in *Proceedings of the* 2^{nd} *Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada), USENIX, June 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 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.

[5] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Proceedings of the* 1^{st} *European Pattern Languages of Programming Conference*, July 1996.

[6] R. Gingell, M. Lee, X. Dang, and M. Weeks, "Shared Libraries in SunOS," in *Proceedings of the Summer 1987 USENIX Technical Conference*, (Phoenix, Arizona), 1987.

[7] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[8] 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.

[9] 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.

[10] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

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

[12] S. Rago, *UNIX System V Network Programming*. Reading, MA: Addison-Wesley, 1993.

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

[14] 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.

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

[16] S. Berczuk, "A Pattern for Separating Assembly and Processing," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.