Service Configurator
A Pattern for Dynamic Configuration of Services

Prashant Jain and Douglas C. Schmidt
Department of Computer Science
Washington University

Service Configurator

A Pattern for Dynamic Configuration of 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[1]

Abstract

This paper describes the Service Configurator pattern, which decouples the implementation of services from the time when they are configured. This pattern increases the flexibility and extensibility of applications by enabling their constituent services to be configured at any point in time. The Service Configurator pattern is widely used in application environments (e.g., to configure Java applets into WWW browsers), operating systems (e.g., to configure device drivers), and distributed systems (e.g., to configure standard Internet communication services).

1 Introduction

A rapidly growing collection of services is now available on the Internet. The term *service* has several generally accepted meanings: (1) a single capability offered by a server (such as the `echo` service provided by the `inetd` superserver), (2) a collection of capabilities offered by a server (such as the `inetd` superserver itself), and (3) a collection of servers that cooperate to achieve a common task (such as a collection of `rwho` daemons in a LAN that periodically broadcast and receive status reports on user and host activities). Unless otherwise indicated, this paper uses the first definition of service, *i.e.,* an identifiable component in a server that offers a single capability to communicating entities.

The range of services available on the Internet include: WWW browsing and content retrieval services, software distribution services, electronic mail and network news transfer agents, file access on remote machines, remote terminal access, routing table management, host and user activity reporting, network time protocols, and object request brokerage services.

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, and often inefficient, applications and software architectures. The main problem with this static approach 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 or system.

[1]This research is supported in part by a grant from Siemens Medical Engineering.

This paper 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 an application or system. The examples in this paper illustrate the Service Configurator pattern using Java applets. 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).

This paper is organized as follows: Section 2 describes the Service Configurator pattern using a variant of the GoF pattern format [1] and Section 3 presents concluding remarks.

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 modularity of the services and allows the services to evolve over time independently of configuration issues, such as whether or not two services must be co-located or what concurrency model will be used to execute the services.

In addition, the Service Configurator pattern provides centralized administration of all the services it configures. This facilitates automatic initialization and termination of the services and can optimize performance by performing common service initialization and termination activities.

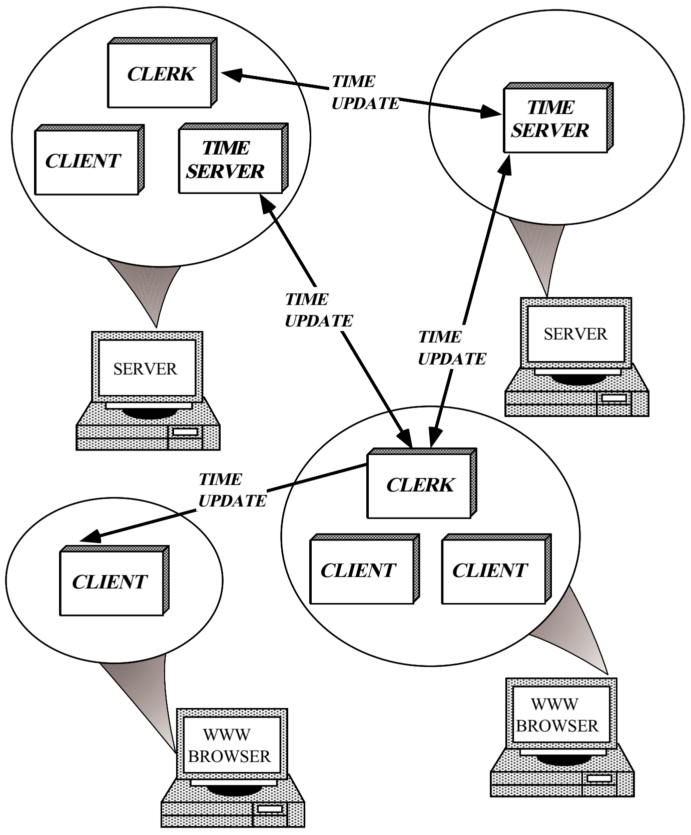

Figure 1: A Distributed Time Service

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 illustrate 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. A synchronized time service is important in distributed systems that require multiple hosts to maintain accurate global time. For instance, large-scale distributed medical imaging systems [2] 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 components:

- *Time Server* – which answers queries about the time made by Clerks.
- *Clerk* – which queries one or more Time Servers to determine the correct time, calculates the approximate correct time using one of several distributed time algorithms [3, 4], and updates its own local system time.
- *Client* – which uses the global time information maintained by a Clerk to provide consistency with the notion of time used by clients on other hosts.

2.3.2 Common Traps and Pitfalls

One way to implement the distributed time service is to statically configure the *logical* functionality of Time Servers, Clerks, and Clients into separate *physical* stand-alone processes. In this static approach, one or more hosts would run Time Server processes, which handle time update requests from Clerk processes. 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. Client processes would then use the synchronized time reported by their local Clerk.[2]

In addition to the time service, other services (such as file transfer, remote login, and HTTP servers) provided by the hosts would also execute in separate statically configured processes.

However, implementing and configuring services in the static manner shown above has the following drawbacks:

• **Service configuration decisions must be made too early in the development cycle:** This early binding is undesirable since developers may not know *a priori* the best way to co-locate or distribute service components. For example, minimal 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 the Time Server into one process to reduce communication latency. Forcing developers to commit prematurely to a particular service configuration impedes flexibility and can reduce performance and functionality.

• **Modifying or terminating a service may adversely affect other services:** In the static approach, 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, the existing Clerk code would require modification, recompilation, and relinking. However, terminating the process to change the Clerk code would also terminate the Time Server. This disruption in service availability may not be acceptable for mission critical distributed systems (such as telecommunication switches or call centers [5]).

[2]For platforms that support shared memory, communication overhead can be minimized by storing the current time into shared memory that is mapped into the address space of the Clerk and all Clients on the same host.

• **System performance may not scale up efficiently:** Associating a process for each service ties up valuable OS resources (such as I/O descriptors, virtual memory, and process table slots). This can be wasteful if services are frequently idle. Moreover, processes are often the wrong concurrency model 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 multi-threaded Active Object [6] or a single-threaded Reactive [7] event loop may be more efficient.

2.3.3 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 services from the point in time at which the service implementations 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:** Dynamic configuration allows developers to concentrate on the functionality of a service, without committing themselves prematurely to a particular configuration of services. 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 modular building blocks that can be integrated easily as components in a larger application. Enforcing a uniform interface for all services makes them "look and feel" the same with respect to how they are configured, thereby simplifying application development.

• **The need to optimize and control the behavior of a service at run-time:** Decoupling implementation details of a service from configuration decisions 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 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 additional information (such as the number of CPUs or the OS version) is available to help optimize the services. In addition, adding a new or updated service to a distributed system may 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.

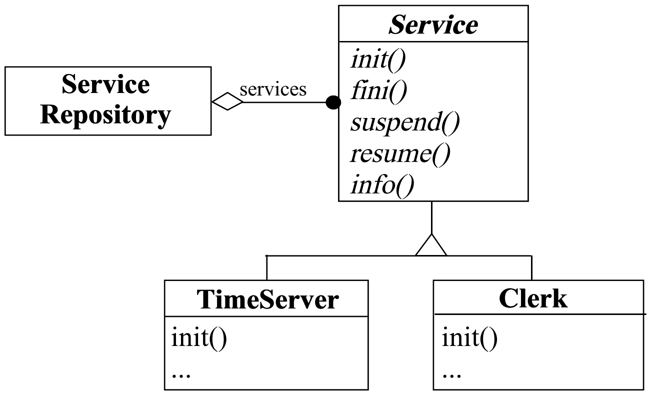

Figure 2: Structure of a Distributed Time Service

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 service-specific information (such as the service name, host address, and port number). Services reside within a `Service Repository` and can be added to and removed from the `Service Repository` by Service Configurator-based applications.

Two subclasses of the `Service` base class appear in the distributed time service: `TimeServer` and `Clerk`. Each subclass represents a concrete `Service`, which has specific functionality in the distributed time service. The `TimeServer` service is responsible for receiving and processing requests for time updates from `Clerks`. The `Clerk` service is a Connector [8] factory that performs the following tasks:

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;
4. Updates the local system time based on an average of all `TimeServer` responses.

The Service Configurator pattern improves the flexibility of the distributed time service by managing the configuration of service components in the time service. Thus, configuration decisions (such as whether or not to co-locate the `TimeServer` and `Clerks`) are decoupled from implementation details (such as the algorithm used by a Clerk to update its notion of time). In addition, implementations of the Service Configurator pattern can provide a framework that consolidates the configuration and management of application services in one administrative unit.

2.4 Applicability

Use the Service Configurator pattern when:

- Services must be initiated, suspended, resumed, and terminated dynamically; and
- 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 configuration is undesirable due to security restrictions (in this case, static configuration of trusted services may be necessary); or
- The initialization or termination of a service is too complicated or too tightly coupled to its context to be performed in a uniform manner; or
- Stringent performance requirements mandate the need to minimize the extra levels of indirection used by the the OS and language mechanisms for dynamic configuration.

2.5 Structure and Participants

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

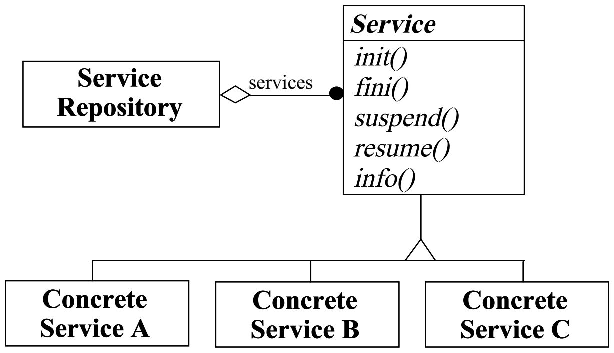

Figure 3: Structure of the Service Configurator Pattern

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

- **Service** (`Service`)
  - Specifies the interface that contains the abstract *hook methods* [9] (such as methods for initialization and termination) used by a Service Configurator-based application to dynamically configure each `Service`.
- **Concrete Service** (`Clerk` and `TimeServer`)
  - Implements the service hook methods and other service-specific functionality (such as event processing and communication with clients and other services).
- **Service Repository** (`ServiceRepository`)
  - Maintains a repository of all services offered by a Service Configurator-based application. This allows an administrative entity to centrally manage and control the behavior of application services.

2.6 Collaborations

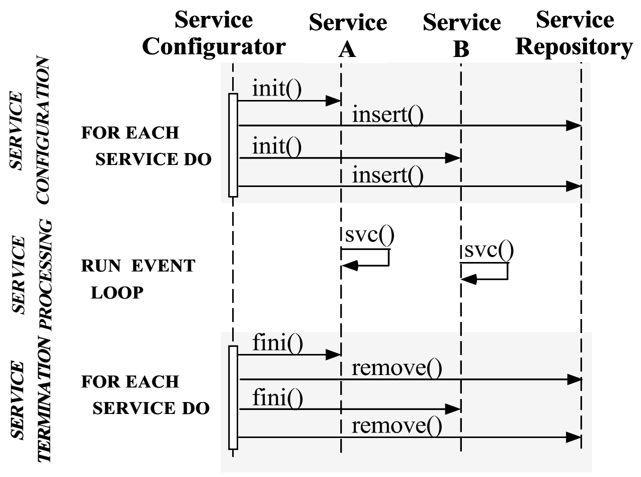

Figure 4: Interaction Diagram for the Service Configurator Pattern

Figure 4 depicts the collaborations in following three phases between components of 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 `ServiceRepository`. The `ServiceRepository` is used by the Service Configurator to manage and control all `Services` that are installed.
- *Service processing* – A `Service` is executed once it has been configured into the system. Once a `Service` is executing, the Service Configurator can suspend and resume the `Service`.
- *Service termination* – The Service Configurator terminates a `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 `ServiceRepository`.[3]

[3]Not all systems support service termination. For example, the Java run-time environment that implements the Service Configurator pattern provides no way to terminate an applet or unload it once it has been loaded into the run-time environment (*e.g.,* a WWW browser).

2.7 Consequences

2.7.1 Benefits

The Service Configurator pattern offers the following benefits:

• **Centralized administration of services:** The pattern consolidates one or more services into a single administrative unit. This simplifies 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 services by enforcing a uniform set of configuration management operations on them (such as *initialize*, *suspend*, *resume*, and *terminate*).

• **Increased modularity and reuse:** The pattern improves the modularity and reusability of 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 opportunity for tuning and optimization:** The pattern increases the range of service configuration alternatives available to developers by decoupling service functionality from the concurrency models (*e.g.,* threads or processes) used to invoke 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 models. Some alternatives include spawning a thread or process upon receipt of a client request or pre-spawning a thread or process at service creation time.

2.7.2 Drawbacks

The Service Configurator pattern has the following drawbacks:

• **Lack of determinism:** The pattern makes it hard to determine the behavior of a service and/or application until run-time. This is particularly problematic for real-time systems since a dynamically configured service may perform unpredictably when run with certain services. For example, if consumers in a real-time Event Service do not obey their periodic processing constraints, other real-time services will miss their deadlines and the system will not behave predictably.

• **Reduced reliability:** An application that uses the Service Configurator pattern may be less reliable than one that does not because 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 co-located services. This is particularly problematic for open systems [10], such as Java applets within WWW browsers that configure and execute multiple services 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 be undesirable or an unnecessary overhead in time-critical applications. In addition, the Service Configurator pattern often configures services via dynamic linking, which adds extra indirection to invoke functions and access global variables [11].

• **Lack of generality:** If services are tightly coupled, it may not be possible to dynamically configure them in arbitrary ways using the Service Configurator pattern. For example, it may be necessary to configure two services in a specific order or it may be necessary to always co-locate two services. The Service Configurator pattern only provides the mechanism of decoupling service implementation from service configuration – it does *not* dictate any policy by which services are to be configured. Therefore, the Service Configurator is a building block in a "pattern language" of strategies for dynamically configuring and reconfiguring services.

2.8 Implementation

The Service Configurator pattern has been implemented in many contexts, ranging from device drivers in operating systems like Solaris and Windows NT, Internet superservers like inetd, and Java applets in WWW browsers. 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 core interface that a service should support to enable the Service Configurator to configure and control the service:

- *Initialization* – Provides an entry point into the service and performs initialization of the service;
- *Termination* – Terminates execution of a service and provides a hook to cleanup application resources;
- *Suspension* – Temporarily suspends the execution of a service;
- *Resumption* – Resumes execution of a suspended service;
- *Information* – Obtains information about a service to determine its identity and behavioral characteristics.

There are two ways to define the service control interface: *inheritance-based* and *message-based*, as described below:

- *Inheritance-based service control interface* – In this approach, each service inherits from a common base class. This approach is used by the ACE `Service Configurator` framework [5] and Java applets, which defines abstract base classes that contain pure virtual "hook" methods. The following shows the `Service` interface similar to the one provided in ACE:

```
class Service
{
```

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 an in-memory table of service implementations • Maintain a persistent database of service implementations
Select a configuration mechanism	• Specify at command line • Specify through a configuration file • Specify through a user interface
Determine service execution mechanism	• Reactive execution • Multi-threaded Active Objects • Multi-process Active Objects

Table 1: Steps Involved in Implementing the Service Configurator Pattern

```
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 is the entry point hook into a `Service`. It is used by the Service Configurator to initialize and execute a `Service`. The `fini` method is a hook that 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 subsequently 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 contract between the Service Configurator and the `Service` objects that it manages.

- *Message-based service control interface* – Another way to control services is to program them to respond to a specific set of messages. This makes it possible to integrate the Service Configurator into non-OO programming languages (such as C). The Windows NT Service Control Manager (`SCM`) [12] uses this scheme. Each Windows NT host has a master `SCM` process that automatically initiates and manages system services by passing them control messages such as PAUSE, RESUME, and TERMINATE. Each developer of an `SCM`-managed service must write code to process these messages and perform the intended actions.

• **Define a Repository:** A `Service Repository` maintains all the `Service` implementations in the form of objects, executable programs, and/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. In addition, the Repository maintains the current status of each service (*e.g.,* whether a service is active or suspended). This information may reside in main memory, a file system, or the kernel.

• **Select a configuration mechanism:** A service must be configured before it can execute. Configuring 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 information can be specified in various ways (*e.g.,* on the command line, through a user interface, or through a configuration file). A centralized configuration mechanism (such as the NT Registry or `inetd.conf` file) 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 concurrency model:** A service that has been dynamically configured by a Service Configurator can be executed using various combinations of Reactive [7] and Active Object [6] schemes. These alternatives are briefly outlined below:

- *Reactive execution* - This approach uses a single thread of control to execute the Service Configurator and 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.

2.9 Sample Code

The following code shows an example of the Service Configurator pattern in the context of Java *applets*. An applet is a Java class that can be loaded and run by a Java application (such as a Web browser, an applet viewer, or an application). The example below focuses on the configuration-related aspects of the distributed time service described in Section 2.3. In addition, this example illustrates how other patterns (such as the Active Object pattern [6] and the Acceptor and Connector patterns [8]) are commonly used in conjunction with the Service Configurator pattern to develop flexible communication infrastructure and services.

In the example, the `Concrete Service` class in the OMT class diagram shown in Figure 3 is represented by the `TimeServer` class and the `Clerk` class. The Java code in this section implements the `TimeServer` and the `Clerk` classes.[4] Both classes inherit from `java.applet.Applet`. This allows them to be downloaded (*e.g.*, from an HTTP server) and dynamically configured (*e.g.*, into a Java interpreter within a WWW browser).

The WWW server's file system serves as the Service Repository for the Java applets. In addition, the `java.applet.Applet` class provides hook methods that allow dynamic (1) configuration of a service (`init`), (2) suspension of a service (`stop`), and (3) resumption of a service (`start`). Note that the `java.applet.Applet` class does not provide a termination method equivalent of `fini` described in Section 2.8. The Service Configurator pattern remains at the heart of the Java applets, however, by allowing their implementation to be decoupled from their dynamic configuration.

2.9.1 The TimeServer Class

The `TimeServer` uses the `Acceptor` class to accept connections from one or more Clerks. The `Acceptor` class uses the Acceptor pattern [8] to create handlers for each Clerk connection that wants to receive requests for time updates [13]. This design decouples the implementation of the `TimeServer` from its configuration. Therefore, developers can change the implementation of the `TimeServer` independently of its configuration. This design provides flexibility with respect to evolving the implementation of the `TimeServer` class.

The `TimeServer` class inherits from the standard `java.applet.Applet` class, which enables a `TimeServer` to be dynamically loaded into a running Java application. Once the `TimeServer` applet has been downloaded and verified, the Java run-time system invokes its `init` hook. This method performs the `Time Server`-specific initialization code.

The `TimeServer` class implements the `Runnable` interface. This allows it to become an active object and run in its own thread of control. Running `TimeServer` as an active object is useful if the applet's main thread of control must perform other tasks (such as responding to user GUI events and methods called by the system).

```
import java.applet.Applet;

public class TimeServer extends Applet
                        implements Runnable
{
  // Initialize the TimeServer when loaded.  This
  // may include synchronizing server clock with
  // an atomic clock.  This method corresponds
  // to the init() hook method of the Service
  // Configurator pattern.

  public void init ()
  {
    // Initialize.
  }

  // (Re)start the TimeServer.  Note that this method
  // gets called after init() when the applet first
  // starts up in the context of Java run-time
  // system and also when the applet is restarted
  // after being temporarily stopped.  The method
  // spawns off a new thread to handle Clerk
  // connections if a thread is not already running.
  // Otherwise it resumes the currently suspended
  // thread.  This method corresponds to the resume()
  // hook method of the Service Configurator pattern.

  public void start ()
  {
    if (serverThread_ == null) {
      serverThread_ = new Thread (this);
      serverThread_.start ();
    }
    else
      // Resume the server thread.
      serverThread_.resume ();
  }

  // Temporarily stop/suspend the TimeServer.
  // This method suspends the thread that handles
  // Clerk connections.  This method corresponds
  // to the suspend() hook method of the Service
  // Configurator pattern.

  public void stop ()
  {
    if (serverThread_ != null &&
        serverThread_.isAlive ()) {
      // Suspend the server thread.
      serverThread_.suspend ();
    }
  }

  // Return information about the TimeServer
  // by overriding the method defined in the
  // java.applet.Applet class.  This method
  // corresponds to the info() hook method of
  // the Service Configurator pattern.
  public String getAppletInfo ()
  {
    // Return a String containing information
    // about this applet. This may include the
    // name of the host, the version number, etc.
    return new String ( ... );
  }

  // This method serves as the entry point for
  // the Time Server thread.  It is called
  // when the thread starts.
  public void run ()
  {
    // Set the connection acceptor_ endpoint into
    // listen mode (using the Acceptor pattern).
    acceptor_.open (port_);

    // Now use the acceptor_ to accept
    // connections from Clerks.
    // ...
  }

  // Acceptor used for Clerk connections.
  protected Acceptor acceptor_;

  // Port the TimeServer listens on.
  private int port_ = SERVER_PORT_NUMBER;

  // The Server Thread
  private Thread serverThread_ = null;

  // ...
}
```

The Java run-time system can suspend and resume the `TimeServer` by calling its `stop` and `start` hooks, respectively. In addition, it can call `getAppletInfo` method to obtain useful information about the service, such as the version number or the name of the author.

[4]To save space, most of the detailed Java code and exception handling code has been omitted.

2.9.2 The Clerk Class

The `Clerk` uses the Connector pattern [8] to establish and maintain connections with one or more `TimeServers`. The Connector pattern creates a handler for every connection to a TimeServer. The handlers receive and process time updates from the `TimeServers`.

The `java.applet.Applet` base class is the parent of the `Clerk` class. Therefore, like the `TimeServer`, a `Clerk` can be dynamically configured by the Java run-time system acting in the role of Service Configurator. The Java run-time system can initialize, suspend, resume, and obtain information about the `Clerk` by calling its `init`, `stop`, `start`, and `getAppletInfo` hooks, respectively.

```
import java.applet.Applet;

public class Clerk extends Applet
                   implements Runnable
{
  // Initialize the Clerk when loaded.  This
  // may include initializing the algorithm
  // implementation to be used to compute the
  // Clerk's notion of time.  This method
  // corresponds to the init() hook method of
  // the Service Configurator pattern.
  public void init ()
  {
    // Initialize.
  }

  // (Re)start the Clerk.  Note that this method
  // gets called after init() when the applet first
  // starts up in the context of Java run-time
  // system and also when the applet is restarted
  // after being temporarily stopped. The method
  // spawns off a new thread to setup connections
  // with the TimeServers if a thread is not already
  // running. Otherwise it resumes the currently
  // suspended thread.  This method corresponds to
  // the resume() hook method of the Service
  // Configurator pattern.

  public void start ()
  {
    if (clerkThread_ == null) {
      clerkThread_ = new Thread (this);
      clerkThread_.start ();
    }
    else
      // Resume the Clerk thread.
      clerkThread_.resume ();
  }

  // Temporarily stop/suspend the Clerk.  This
  // method suspends the thread that handles
  // connection to TimeServers.  This method
  // corresponds to the suspend() hook method
  // of the Service Configurator pattern.

  public void stop ()
  {
    if (clerkThread_ != null &&
        clerkThread_.isAlive ()) {
      // Suspend the Clerk thread.
      clerkThread_.suspend ();
    }
  }

  // Return information about the Clerk by
  // overriding the method defined in the
  // java.applet.Applet class.  This method
  // corresponds to the info() hook method of
  // the Service Configurator pattern.
  public String getAppletInfo ()
  {
    // Return a String containing information about
    // this applet. This may include the name of
    // the host, the version number, etc.
    return new String ( ... );
  }

  // This method serves as the entry point for
  // the Clerk thread. It is called when the
  // thread starts.
  public void run ()
  {
    // Use the connector to set up connections
    // to all the TimeServers.  Then use the
    // updateTime() method to send periodic requests
    // to the TimeServers for time updates, receive
    // the requests from the TimeServers, and compute
    // the local notion of time.
    // ...
  }

  // Called periodically to compute the local
  // system time.
  protected void updateTime (long t)
  {
    // Implement Clock Synchronization algorithm
    // here to compute local system time.
  }

  // Connect to TimeServers.
  protected Connector connector_;

  // The Clerk Thread.
  private Thread clerkThread_ = null;
}
```

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

2.9.3 Lifecycle of a Service

Figure 5 shows a state diagram of the lifecycle of a `Service` (such as the Clerk service).

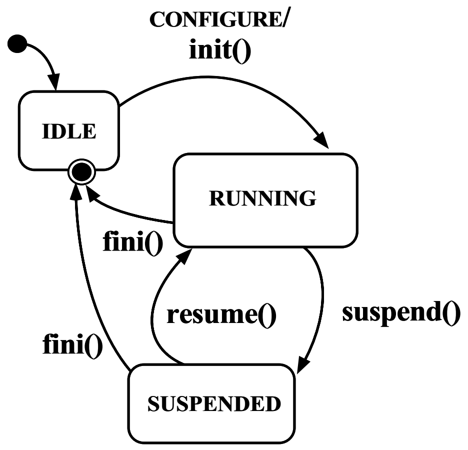

Figure 5: State Diagram of a Service Lifecycle in the Service Configurator Pattern

Initially the service is idle. Depending upon requirements, the user can choose from various implementations dynamically, without having to focus on configuration issues.

For instance, different Clerk services may exist corresponding to different algorithm implementations. Thus, the user may either select a Clerk service that implements the Berkeley algorithm [3] or a Clerk service that implements Cristian's algorithm [4]. The choice may depend upon the characteristics of the `TimeServer`. If the machine on which the `TimeServer` resides has a WWV receiver[5] the `TimeServer` can act as a passive entity and Cristian 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.

Once a Clerk service has been selected, it can be easily configured by loading it into the Java run-time environment (such as a Web browser, an applet viewer, or an application). The following HTML fragment shows how the Clerk applet can be loaded in an applet viewer or a Web browser:

```
<APPLET code="Clerk.class">
<PARAM name=configFile value="svc.conf">
<PARAM name=pollTime value="10">
</APPLET>
```

The `APPLET` tag specifies an applet to be run within a Web browser or an applet viewer. The `PARAM` tag specifies named parameters to be passed to the applet. An applet can look up the value of a parameter specified in a `PARAM` tag with the `Applet.getParameter` method. In the example above, the Clerk applet is passed the name of a service configuration file and a `pollTime` of 10 seconds. This configuration file (`svc.conf`) contains the hostnames and port numbers of all the Time Servers the Clerk will connect to. The `pollTime` indicates how frequently the Clerk will poll the Time Servers.

To reduce communication latency, The Clerk service can be co-located with a Time Server service. The following HTML fragment shows how the Clerk applet can be loaded in an applet viewer or a Web browser together with a co-located Time Server applet:

```
<APPLET code="Clerk.class">
<PARAM name=configFile value="svc.conf">
<PARAM name=pollTime value="10">
</APPLET>
<APPLET code="Server.class">
<PARAM name=port value="7734">
</APPLET>
```

In this example, the Time Server class will listen at port 7734.

Figure 6 shows the Clerk running independently as well as running co-located with a Time Server. This configuration decision need not affect the implementation of the various time services. Note, however, that if the Clerk and the Time Server are co-located in the same process, the Clerk may optimize communication by (1) eliminating the need to set up a communication channel with the Server and (2) directly accessing the Server's local notion of time via shared memory. In general, the decoupling between a service implementation and its configuration exemplifies the flexibility offered by the Service Configurator pattern.

[5] 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.

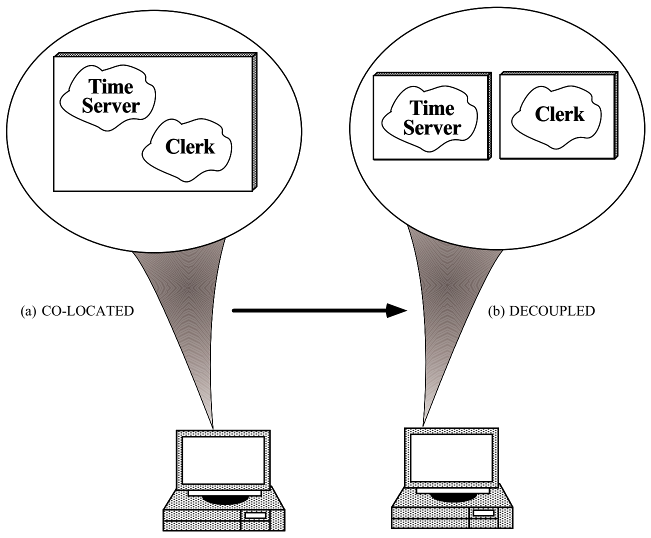

Figure 6: (a) Clerk co-located with a Time Server; (b) Clerk running independently

2.10 Known Uses

The Service Configurator pattern has been used in a wide range of operating system and application programming environments including Java applets, UNIX, Windows NT, and ACE:

• **Java applets:** The applet mechanism in Java uses the Service Configurator pattern. Java supports dynamically downloading, initializing, starting, suspending and resuming applets. For instance, it defines methods (*e.g.,* `stop` and `start`) that suspend and resume applet threads. A method in a Java applet can access the thread it is running in using `Thread.currentThread()`. In addition, threads can control each other by invoking methods like `stop` and `start`.

• **Modern operating system device drivers:** Modern operating systems (such as Solaris [14] and Windows NT [12]) support dynamically configurable kernel-level device drivers. For instance, Solaris drivers can be linked into and unlinked out of the system dynamically via `init`/`fini`/`info` hooks. 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` [15] and `listen` [16]. 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. These frameworks contain a master

Acceptor [8] 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 (SCM):** Unlike `inetd` and `listen`, the Windows NT Service Control Manager (`SCM`) [12] 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 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 [17] provides a set of C++ mechanisms for configuring and controlling services dynamically. The ACE `Service Configurator` extends the mechanisms provided by `inetd`, `listen`, and `SCM` to automatically support dynamic linking and unlinking of services. 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.

2.11 Related Patterns

The intent of the Service Configurator pattern is similar to the Configuration pattern [18]. 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 (such as Regis [19] and Polylith [20]) 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 also focuses on decoupling service behavior from the service's concurrency strategies.

The Manager Pattern [21] 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 [7] pattern to perform event demultiplexing and dispatching on behalf of configured services. Likewise, dynamically configured services that run for a long periods of time often execute using the Active Object pattern [22].

Administrative interfaces (such as configuration files or GUIs) to a Service Configurator-based system provide a Facade [1]. 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" [23] or "hook methods" [9]. These hooks are used by the Service Configurator to initiate, suspend, resume, and terminate services.

A `Service` (such as the `Clerk` class) may be created using a Factory Method [1]. This allows an application to decide the type of `Service` subclass to create.

3 Concluding Remarks

This paper 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.

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 paper used Java applets to demonstrate the application of the Service Configurator pattern in the Java run-time system. The ability to decouple the development of Java applets from their configuration into the Java run-time system exemplifies the flexibility offered by the Service Configurator pattern. This decoupling allows different applets to be developed in accordance with different service implementations. The decision to configure a particular applet into the Java run-time system becomes a run-time decision, which yields greater flexibility.

The Service Configurator pattern is also widely used in other 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.

4 Availability

The ADAPTIVE Communication Environment (ACE) provides an implementation of the Service Configurator pattern. ACE is freely available via the WWW at `www.cs.wustl.edu/~schmidt/ACE.html`.

References

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

[2] 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.

[3] 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.

[4] F. Cristian, "Probabilistic Clock Synchronization," *Distributed Computing*, vol. 3, pp. 146–158, 1989.

[5] 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.

[6] 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.

[7] 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.

[8] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1996.

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

[10] D. Lea and J. Marlowe, "PSL: Protocols and Pragmatics for Open Systems," in *Proceedings of the* 9th *European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, August 1995.

[11] 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.

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

[13] P. Jain and D. Schmidt, "Experiences Converting a C++ Communication Software Framework to Java," *C++ Report*, vol. 9, January 1997.

[14] 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.

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

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

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

[18] 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.

[19] J. Magee, N. Dulay, and J. Kramer, "A Constructive Development Environment for Parallel and Distributed Programs," in *Proceedings of the* 2nd *International Workshop on Configurable Distributed Systems*, (Pittsburgh, PA), pp. 1–14, IEEE, Mar. 1994.

[20] J. M. Purtilo, "The Polylith Software Toolbus," *ACM Transactions on Programming Languages and Systems*, 1994.

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

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

[23] 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.