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

Introduction

[JOLT reviewers, I left this introduction here since it might be useful in some introductory section of the book, *e.g.*, I think it does a nice job of defining what a "service" is!]

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

1 The Service Configurator Pattern

1.1 Intent

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

1.2 Example

To motivate the Service Configurator pattern, consider the distributed time service shown in Figure 1. A distributed time service [1] provides accurate, fault-tolerant clock synchronization for computers collaborating in networks. Synchronized time services are important in distributed systems that require hosts to maintain accurate global time across a network. For instance, large-scale distributed medical imaging systems [2] require globally synchronized clocks in order to ensure that patient exams are accurately timestamped and analyzed expeditiously by radiologists throughout a distributed health-care delivery system.

The architecture of the distributed time service shown in Figure 1 contains the following components:

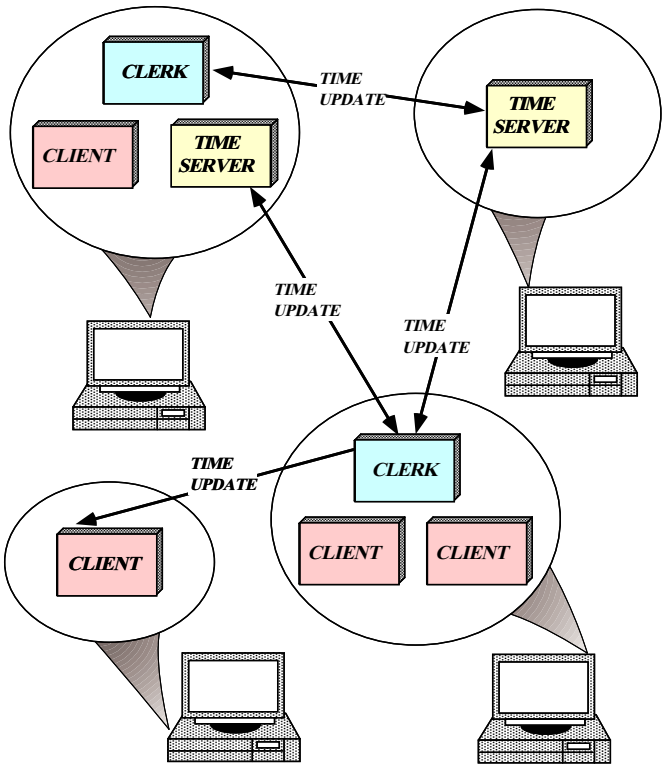

Figure 1: A Distributed Time Service

- *Time Servers* answer Clerk queries about the current time.
- *Clerks* query one or more Time Servers to sample their notion of the current time, calculate the approximate correct time using one of several distributed time algorithms [3, 4], and update their own local system time.
- A *Client* uses the global time information maintained by its Clerk to provide consistency with the notion of time used by Clients on other hosts.

A common way to implement this distributed time service architecture is to statically configure the functionality of Time Servers, Clerks, and Clients into separate processes. In this design, one or more hosts would run Time Server pro-

cesses, which service time update requests from Clerk processes. Each host that requires global time synchronization runs a Clerk process. Clerks periodically update their local system time based on values received from one or more Time Servers. Client processes use the synchronized time reported by their local Clerk. To minimize communication overhead, the current time can be stored in shared memory that is mapped into the address space of the Clerk and all of its Clients on the same host.

However, the statically configured design for the distributed time service outlined above yields an inflexible, and often inefficient, solution. The main problem with static configuration is that it tightly couples the *implementation* of a particular service to the *configuration* of the service with respect to other services in an application or system.

1.3 Context

An application or system where services must be initiated, suspended, resumed, and terminated dynamically.

1.4 Problem

Applications that are composed of component-based services must provide a mechanism to configure these services. The solution to this problem is influenced by the following forces:

Service configuration decisions should be possible late in the development cycle: Early binding of service implementations to service configurations is undesirable since developers may not know in advance (1) which service implementations are most suitable or (2) the best way to collocate or distribute multiple 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. Conversely, in a real-time avionics environment it may be necessary to collocate the Clerk and Server into one process to reduce communication latency.[1] In general, forcing developers to commit prematurely to a particular configuration of service implementations impedes flexibility and can reduce overall system performance and functionality.

The implementation of a service may change: Changes to service behavior or implementation are common in many systems and applications. Therefore, the modification of a service component should have minimal impact on the implementation of applications or other components that use the service. For example, in the real-time avionics environment mentioned in the previous force, a Client and Clerk might be configured to execute in one process to reduce latency. If the distributed time algorithm implemented by the Clerk changes, however, other Clerk code, or the client code that uses the Clerk, should be unaffected by these changes.

Services may need to be reconfigured at run-time: The initial configuration of services in an application or system may not be optimal over time. For instance, better algorithms for a particular service may become available. Likewise, in large-scale systems it may be necessary to distribute services to other processes and hosts since a collocated configuration may not scale efficiently. In addition, platform upgrades may require service reconfiguration. However, for highly available systems, such as telecommunication switches or call centers [6], no disruption in service is acceptable. Therefore, it should be possible to initiate, suspend, resume, and terminate services at run-time, *i.e.*, without affecting currently executing services. For example, if we change the Clerk's implementation of the time algorithm, it may be undesirable to recompile the whole application that contains the Clerk. Likewise, collocating a Client and Clerk in a single process or thread may be less efficient then running them in separate processes or threads.

Services may need to be configured by a single administrative unit: Performing common service administration tasks, such as their configuration, initiation, and control, should be straightforward and service-independent. Moreover, this administrative code should be managed in a central location, rather than being scattered throughout an application. For example, the administrative code for initializing and terminating instances of Clerks and Time Servers should be localized in a set of generic library components that can be linked with application code and invoked a uniform manner, such as via a standard set of hook methods or messages. Likewise, the application code that implements these hook methods or messages should be localized within designated factories in an application.

1.5 Solution

Decouple the behavior of services from the point in time at which these services are configured into an application or system. A `Service` provides a standard interface for configuring and controlling service components. Applications can use this interface to initiate, suspend, resume, and terminate their services, as well as to obtain run-time service information. The services themselves reside within a `Service Repository` and can be added and removed to and from the `Service Repository` statically or dynamically under control of a `Service Configurator`.

Two `Concrete Services`, `Time Server` and `Clerk`, appear in the distributed time service. Each `Concrete Service` provides specific functionality to the distributed time service. The `Time Server` service is responsible for receiving and processing requests for time updates from `Clerks`. The `Clerk` service is responsible for querying one or more `Time Servers` to determine the approximate correct time and update their own local system time.

[1] If the Clerk and the Server are collocated in the same process, the Clerk can optimize communication by (1) eliminating the need to set up a transport connection with the Server and (2) directly accessing the Server's notion of time via shared memory [5].

1.6 Structure

The structure of the Service Configurator pattern is illustrated using OMT notation in Figure 2. The key participants

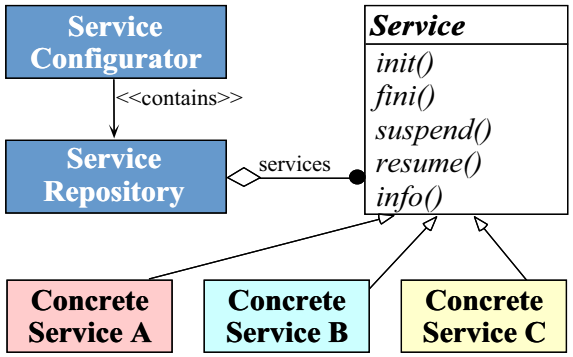

Figure 2: Structure of the Service Configurator Pattern

in the Service Configurator pattern include the following:

Service:

- Defines an interface for controlling a `Service`, *e.g.*, initializing, suspending, resuming, and terminating a `Service` instance.

Concrete Service:

- Implements the `Service` control interface and other `Service`-specific functionality, such as communicating with peers.

Service Repository:

- Maintains a repository of all the `Services` offered by a Service Configurator-enabled application. This allows administrative entities to manage and control the behavior of configured `Services` via a central administrative unit.

Service Configurator:

- This facade [7] manages the other components in the Service Configurator pattern. It is responsible for coordinating the (re)configuration of `Services` via the `Service Repository`.

1.7 Dynamics

Figure 3 depicts the collaborations between components in the following three phases of the Service Configurator pattern:

1. Service initialization: `Service`(s) are initialized by the `Service Configurator`, which dynamically links or launches them if necessary.[2] Once a `Service` has been initialized successfully, the `Service Configurator` adds it to the `Service Repository` and then manages and controls all configured `Services` at run-time.

[2] Section 1.8 describes how parameters can be passed into the `Service`, as well as how the `Service` can be activated.

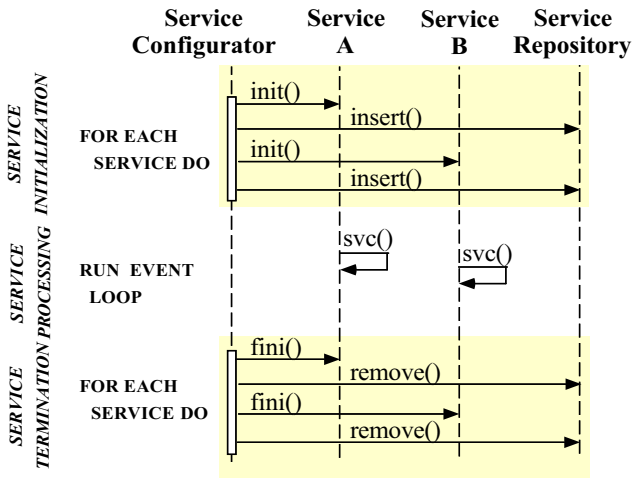

Figure 3: Interaction Diagram for the Service Configurator Pattern

2. Service processing: After being configured into an application, a `Service` performs its processing tasks, *e.g.*, servicing client requests or exchanging messages with peers. While `Service` processing is occurring, the `Service Configurator` can suspend and resume existing `Services`, as well as (re)configure new `Services`.

3. Service termination: A `Service Configurator` shuts down `Services` once they are no longer needed, allowing them an opportunity to clean up their resources before terminating. Once a `Service` is terminated, the `Service Configurator` removes it from the `Service Repository`. If the `Service` was dynamically linked into an application process the `Service Configurator` can unlink it.

The state diagram in Figure 4 illustrates how a `Service Configurator` controls the lifecycle of a `Service`.

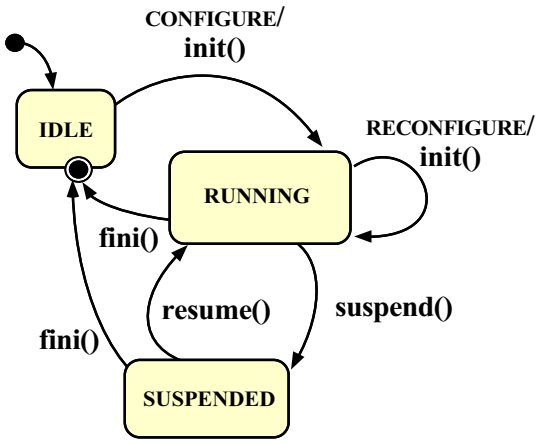

Figure 4: State Diagram of the Service Lifecycle

1.8 Implementation

This section explains the steps involved when implementing the Service Configuration pattern. The implementation described below is influenced by the reusable components provided in the ACE communication software framework [8].

Define the Service control interface: The following is the interface that services should support so the `Service Configurator` can configure and control a service:

- *Service initialization* – provide an entry point into the service and initialize the service.
- *Service finalization* – shutdown a service and cleanup its resources.
- *Service suspension* – temporarily suspend service execution.
- *Service resumption* – resume execution of a suspended service.
- *Service information* – report information that describes the static or dynamic characteristics of a service.

A `Service` control interface can be based on inheritance or message passing.

• **Inheritance-based:** In this strategy, each service inherits from a common base class that contains pure virtual hook methods for each service control operation, 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 **status, size_t len) = 0;
};
```

The `init` method is the entry point into a `Service`. It is used by a `Service Configurator` to initialize a `Service`. The `fini` method allows the `Service Configurator` to shutdown a `Service` and cleanup its resources. The `suspend` and `resume` methods are scheduling hooks used by the `Service Configurator` to suspend and resume the execution of a `Service`. The `info` method allows the `Service Configurator` to obtain `Service`-related information, such as its name, network address, or current status. Together, these methods impose a uniform interface between the `Service Configurator` and the `Services` that it controls.

• **Message-based:** Another way to control communication services is to program each `Service` to respond to a specific set of messages, such as INIT, SUSPEND, RESUME, and FINI. Service developers must write code to process these messages. Using messages, rather than inheritance, makes it possible to implement the Service Configurator pattern in non-OO programming languages, such as C or Ada83, that lack inheritance.

Implement a Service Repository: All `Service` implementations, such as objects, executable programs, or dynamically linked library (DLLs), are maintained by a `Service Repository`. A `Service Configurator` uses the `Service Repository` to access a service when it is configured into or removed from an application or system. Each service's current status, such as whether it is active or suspended, is maintained in the Repository, as well. A `Service Repository` can be implemented as a collection stored in main memory, the file system, or within an OS kernel and maintained using the Manager pattern [9].

The methods commonly defined in the interface of a `Service Repository` are shown below:

```
class Service_Repository
{
public:
  // = Initialization and termination hooks.
  Service_Repository (void);
  // Initialize the repository.

  ~Service_Repository (void);
  // Close down the repository and free up
  // dynamically allocated resources.

  // = Container operations.
  int insert (const char name[],
              const Service *);
  // Insert a new <Service> with a particular <name>.

  int find (const char name[],
            const Service **service);
  // Locate a <Service> with a particular <name>.

  int remove (const char name[]);
  // Remove an existing <Service> with a
  // particular <name>.

  // = Liveness control.
  int suspend (const char name[]);
  // Suspend a <Service>.

  int resume (const char name[]);
  // Resume a <Service>.

private:
  // ...
};
```

Implement a Service Configurator: The `Service Configurator` integrates the other components in the pattern to manage the static and/or dynamic configuration of `Services` into applications or systems. In particular, it is responsible for coordinating the (re)configuration of `Services` via the `Service Repository`. The implementation of a `Service Configurator` involves the following steps:

1. Define the Service Configuration mechanism: A service must be configured before it can be executed. Thus, the `Service Configurator` mechanism should support the following features:

- *An API for specifying per-service attributes* – These attributes supply parameters needed to locate a service's implementation and initialize the service at run-time. Service implementations typically reside in statically linked objects, DLLs, or executable programs. Per-service attributes can be specified in various ways, such

as on the command-line, via environment variables, through a graphical user interface, or in a configuration file.

- *A mechanism for interpreting service configuration attributes* – Regardless of how or where the per-service attributes are specified, an interpreter is required to parse and process the attributes specified to configure each service. This interpreter helps to decouple the configuration-related aspects of a service from its run-time behavior. A per-service attribute interpreter can be developed using the Interpreter pattern [7] or standard parser-generator tools like LEX/YACC [6].

To simplify installation and administration, the `Service Configurator` in our distributed time server example uses a configuration file, referred to as `svc.conf`. This file consolidates service attributes and initialization parameters into a single location that can be managed centrally. For every `Service` to be (re)configured, developers specify a directive in the `svc.conf` file using a simple configuration scripting language defined with the following BNF grammar:

```
<entry> ::= <dynamic> | <static> | <suspend>
                      | <resume> | <remove>
<dynamic> ::= dynamic <svc-location>
                    [ <parameters-opt> ]
<static> ::= static <svc-name>
                  [ <parameters-opt> ]
<suspend> ::= suspend <svc-name>
<resume> ::= resume <svc-name>
<remove> ::= remove <svc-name>
<svc-location> ::= <svc-name> <type>
                   <function-name>
<type> ::= Service_Object '*' | NULL
<function-name> ::= PATHNAME ':' IDENT '(' ')'
<parameters-opt> ::= STRING | NULL
```

The entries in a `svc.conf` file are processed by the Service Configurator's per-service attribute interpreter described above. Each entry starts with a directive that instructs the interpreter what configuration-related operation to perform. Table 1 outlines the purpose of these directives.

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

Table 1: Service Configuration Directives

The following `svc.conf` file illustrates how to configure a Time Server into a Service Configurator-enabled application:

```
# Configure a Time Server.
dynamic Time_Server Service *
   netsvcs.dll:make_Time_Server()
     "-p $TIME_SERVER_PORT"
```

This entry in the `svc.conf` file contains one `dynamic` directive that instructs the interpreter to dynamically link the `netsvcs.dll` DLL into the application's address space and invoke the `make_Time_Server` Factory function. This function dynamically allocates a new `Time Server` instance, as follows:

```
extern "C"
Service *make_Time_Server (void)
{
  // Time_Server inherits from the
  // Service base class.
  return new Time_Server;
}
```

The string parameter at the end of the 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, a pointer to the `Service` is stored in the `Service Repository` under the name `Time_Server`. This name identifies the dynamically configured `Service` so it can be controlled via the `Service Configurator`.

2. Define the Service Configurator interface: The `Service Configurator` is typically implemented as a facade [7] that mediates access to other Service Configurator pattern components, such as the `Service Repository` and the per-service attribute API and interpreter. The following C++ interface is used for our distributed time server example:

```
class Service_Configurator
{
public:
  // = Initialization and termination hooks.
  Service_Configurator (const char svc_conf_file[]);
  // Perform the directives specified in the
  // <svc_conf_file>.

  Service_Repository *service_repository (void);
  // Accessor to the <Service_Repository>.
};
```

Implement the Services and Service execution mechanism: The final step is to implement the `Concrete Services`. In addition to implementing the service logic, this step also requires determining the `Service` execution mechanism. A `Service` that has been configured dynamically by a `Service Configurator` can be executed using various combinations of Reactive [10] and Active Object [11] mechanisms. These alternatives are outlined 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:** In this approach, the configured services execute in their own threads of control within a Service Configurator-enabled process. Services can either spawn new threads "on-demand" or execute within an existing pool of threads.

- **Multi-process Active Objects:** In this approach, the configured services execute in their own processes. Services can either spawn new processes "on-demand" or execute within an existing pool of processes.

1.9 Example Resolved

This section presents an example of the Service Configurator pattern that focuses on the configuration-related aspects of the distributed time service presented in the Example section. The Service Configurator pattern increases the flexibility of the distributed time service by managing the configuration of the Clerk and Time Server service components. In addition, the Service Configurator pattern consolidates the configuration and management of these and other services within a single administrative unit.

Figure 5 illustrates a distributed time service designed according to the Service Configurator pattern. This design

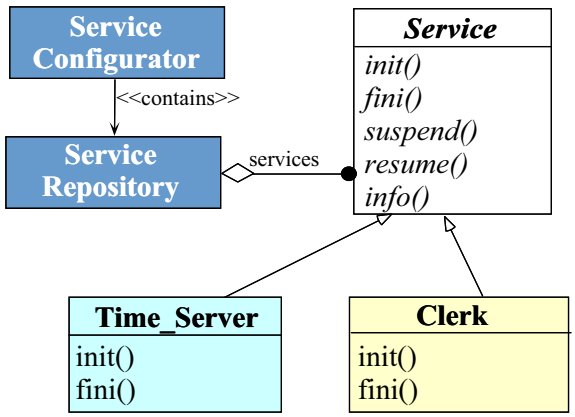

Figure 5: Structure of the Distributed Time Service

uses an inheritance-based strategy for service control. The `Concrete Service` shown in Figure 2 is represented by the `Time Server` and `Clerk` subclasses, which inherit from the `Service` base class. The `Time Server` service is responsible for receiving and processing requests for time updates from `Clerks`. The `Clerk` service is a Connector [12] 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.

This example uses a configuration mechanism based on explicit dynamic linking [6] and a `svc.conf` configuration file to dynamically configure the Clerk and Time Server components of the distributed time service. These features allow Clerk components to change their algorithms for computing local system time *without* affecting the execution of other components configured by the `Service Configurator`. Once an algorithm has been modified, the Clerk component can be reconfigured dynamically by the `Service Configurator`.

The example illustrates how to use C++ to write a generic `main` driver function, which provides a reusable, service-independent entry point into any Service Configurator-enabled application. The service execution mechanism is based on a reactive event handling model [8] within a single thread of control.

1.9.1 The Time Server and Clerk Implementations

Our initial Time Server implementation is based on Cristian's algorithm [4]. In this algorithm, each Time Server is a passive entity that responds to queries made by Clerks. In particular, a Time Server does not actively query other machines to determine its own notion of time.

The `Time Server` class inherits from the `Service` base class defined in Section 1.8, which enables the `Service Configurator` to dynamically link and unlink `Time Server` objects. This design decouples the implementation of the Time Server from its configuration. Therefore, developers can change the implementation of the Time Server's algorithm independently from its configuration.

The C++ interface for the `Time_Server` is shown below:

```
class Time_Server : public Service
{
public:
  // Initialize the service when linked dynamically.
  virtual int init (int argc, char *argv[]);

  // Terminate the service when dynamically unlinked.
  virtual int fini (void);

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

private:
  // ...
};
```

Before storing the `Time Server` service in the `Service Repository`, the `Service Configurator` invokes its `init` hook automatically to perform `Time Server`-specific initialization code. Likewise, the `Service Configurator` calls the `Time Server`'s `fini` hook method automatically shutdown and cleanup a service when it is no longer needed.

Our `Clerk` implementation establishes and maintains connections with `Time Servers` and periodically queries them to calculate the current time. The C++ interface for the `Clerk` is defined below:

```
class Clerk : public Service
{
public:
  // Initialize the service when linked dynamically.
  virtual int init (int argc, char *argv[]);

  // Terminate the service when dynamically unlinked.
  virtual int fini (void);

  // <info>, <suspend>, and <resume> methods omitted.

  // Implements Cristian's clock synchronization
  // algorithm that computes local system time.
  int handle_timeout (void);

private:
  // ...
};
```

The `Clerk` class inherits from the `Service` base class. Therefore, like the `Time Server` above, the `Clerk` can be linked and unlinked dynamically by the `Service Configurator`. Likewise, the `Service`

Configurator can control Clerks by calling their init, suspend, resume, and fini hooks.

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 locally cached time value that has been synchronized with the global notion of time.

1.9.2 Configuring an Application Dynamically

There are two general strategies for configuration a distributed time service application: *collocated* and *distributed*. Each strategy is outlined below to illustrate how a Service Configurator-enabled application can be dynamic (re)configured and run.

Collocated configuration: This configuration that uses a svc.conf file to collocate the Time Server and the Clerk within the same OS process. The following generic main program configures communication services dynamically within the constructor of the Service Config object and then runs the application's event loop, which is based on the Reactor pattern [10].

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

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

The constructor for Service_Configurator interprets the following svc.conf configuration file:

```
# Configure a Time Server.
dynamic Time_Server Service *
   netsvcs.dll:make_Time_Server()
     "-p $TIME_SERVER_PORT"

# Configure a Clerk.
dynamic Clerk Service *
   netsvcs.dll:make_Clerk()
     "-h tango.cs:$TIME_SERVER_PORT"
     "-h perdita.wuerl:$TIME_SERVER_PORT"
     "-h atomic-clock.lanl.gov:$TIME_SERVER_PORT"
     "-P 10" # polling frequency
```

The entries in this svc.conf file specify how to dynamically configure a collocated Time Server and Clerk in the same application process. The Service Configurator dynamically links netsvcs.dll DLL into the application's address space and invokes the appropriate factory function, *i.e.*, make_Time_Server or make_Clerk, to create new Service instances. The factory function for the Time_Server class is shown below:

```
Service *make_Time_Server (void)
{
  // Time_Server must inherit from Service.
  return new Time_Server;
}
```

The factory function for the Clerk class is shown below:

```
Service *make_Clerk (void)
{
  // Clerk must inherit from Service.
  return new Clerk;
}
```

Once each factory function returns successfully, the designated initialization parameters in the svc.conf file are passed to the respective init hook methods, which perform Service-specific initialization.

Distributed configuration: To reduce the memory footprint of an application, we may not want to collocate the Time Server and the Clerk in the same process. Due to the flexibility of the Service Configurator pattern, all we must do to distribute these services is to split the svc.conf file into two parts and run them in separate processes or hosts. One process would contain the Time Server entry and the other process would contain the Clerk entry.

Figure 6 shows what the configuration looks like with the

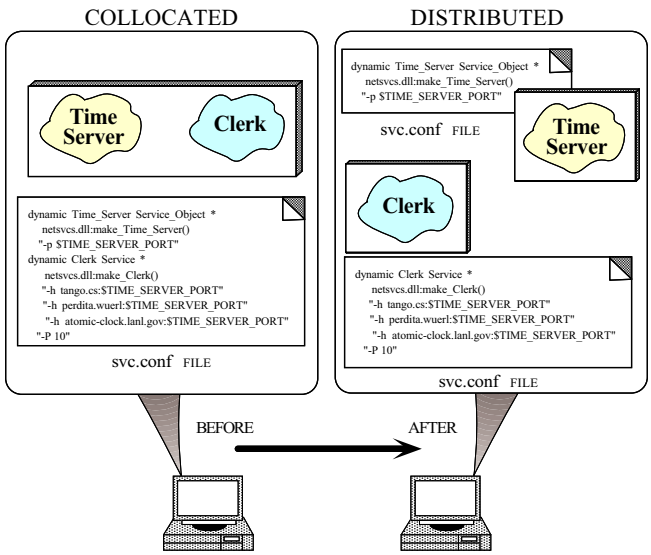

Figure 6: Reconfiguring a Time Server and a Clerk

Time Server and Clerk collocated in the same process, as well as the new configuration after the split. Note that the services themselves need not change by virtue of the fact that the Service Configurator pattern decouples their processing behavior from their configuration.

1.9.3 Reconfiguring an Application's Services

Now consider what happens if we must change algorithms that implement components in the distributed time service.

For example, we may decide to switch from Cristian's algorithm [4] described in Section 1.9.1 to the Berkeley algorithm [3]. In the Berkeley algorithm, the `Time Server` is an active component that polls every machine in the network periodically to ask what time it is there. Based on the responses it receives, it computes an aggregate notion of the correct time and tells all the machines to adjust their clocks accordingly.

Such a change may be necessary to leverage new features in the environment. For instance, if the machine on which the `Time Server` resides has a WWV receiver[3] the `Time Server` can act as a passive entity and Cristian algorithm would be appropriate. Conversely, 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.

Ideally, we should be able to change `Time Server` algorithm implementations without affecting the execution of other services or other components of the time service. Accomplishing this using the Service Configurator pattern requires the following minor modifications to our distributed time service:

1. Modify the existing svc.conf file: We can start by making the following change to the `svc.conf` file:

```
# Shutdown Time_Server
remove Time_Server
```

This directive instructs the `Service Configurator` to shutdown the `Time Server` service and remove it from the configurator's `Service Repository`.

2. Notify the Service Configurator to reinterpret the svc.conf file: The next step is to get the `Service Configurator` to process this directive in the updated `svc.conf` file. This can be done by generating an external event, such as the UNIX SIGHUP signal, an RPC notification, or a Windows NT Registry event. On receipt of this event, the `Service Configurator` consults its `svc.conf` file again and shutdowns the `Time Server` service by calling its `fini` method. Note that the execution of other services should be unaffected during this step.

3. Update the Service implementation: Once the `Time Server` service has been shutdown, it can be modified to implement a different algorithm, such as the Berkeley algorithm. The new code can be recompiled and relinked to form a `new_netsvcs` DLL.

4. Initiate reconfiguration: We can now repeat steps 1 and 2 to configure the modified `Time Server` service back into an application. The `svc.conf` file must be modified with a new directive specifying that the `Time_Server` be linked dynamically, as follows:

```
# Configure a Time Server.
dynamic Time_Server Service *
   new_netsvcs.dll:make_Time_Server()
     "-p $TIME_SERVER_PORT"
```

An external event would then be generated, causing the process to reread the configuration file and add the updated `Time_Server` service component to the repository. This component would begin execution once its `init` method was called by the `Service Configurator`.

During the process of shutting down, removing, and reconfiguring the `Time Server` service, no other active services should be affected. The ease with which new service implementations can be substituted exemplifies the flexibility offered by the Service Configurator pattern.

1.10 Known Uses

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

Modern operating system device drivers: Most modern operating systems, such as Solaris and Windows NT, provide support for dynamically configurable kernel-level device drivers. These drivers can be linked into and unlinked out of the system dynamically via hooks, such as the `init`, `fini`, and `info` functions defined in SVR4 UNIX. These operating systems use the Service Configurator pattern to allow administrators to reconfigure the OS kernel without having to shut it down, recompile and statically relink new drivers, and restart the operating system.

Network daemon management: The Service Configurator pattern has been used in "superservers" that manage network daemons. Two widely available network daemon management frameworks are `inetd` [13] and `listen` [14]. Both frameworks consult configuration files that specify (1) service names, such as the standard Internet services `ftp`, `telnet`, `daytime`, and `echo`, (2) port numbers to listen on for clients to connect with these services, and (3) an executable file to invoke and perform the service when a client connects. Both `inetd` and `listen` contain a master Acceptor [15] process that monitors a set of port numbers 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 preregistered 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): The Windows NT Service Control Manager (`SCM`) allows a master `SCM` process to automatically initiate and control administrator-installed services using the message-based strategy described in the Implementation section. The master `SCM` process automatically initiates and manages system services by passing them various control messages, such as

[3] 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.

PAUSE, RESUME, and TERMINATE, which must be handled by each service. SCM-based services run as separate threads within either a single-service or a multi-service daemon process. Each installed service is individually responsible for configuring itself and monitoring any communication endpoints, which can be more general than socket ports. For instance, the SCM can control named pipes and shared memory.

The ADAPTIVE Communication Environment (ACE) framework: The ACE framework [6] provides a set of C++ mechanisms for configuring and controlling communication services dynamically using the inheritance-based strategy described in the Implementation section. The ACE Service Configurator framework extends the mechanisms provided by inetd, listen, and SCM to automatically support dynamic linking and unlinking of communication services. The mechanisms provided by ACE were influenced by the strategies 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 supports downloading, initializing, starting, suspending, resuming, and terminating applets. It uses the inheritance-based strategy described in the Implementation section by providing virtual methods, such as start and stop, that can be overridden by applications to initiate and terminate threads. A method in a Java applet can access the thread it is running under using Thread.currentThread, and then invoke control methods, such as suspend, resume, and stop, to manage the applet's behavior. An illustration of how the Service Configurator pattern is used for Java applets appears in [16].

1.11 Related Patterns

The intent of the Service Configurator pattern is similar to the Configuration pattern [17]. The Configuration pattern decouples structural issues related to configuring services in distributed applications from the execution of the services themselves. The Configuration pattern has been used in frameworks for configuring distributed systems to support the construction of a distributed system from a set of components. In a similar way, the Service Configurator pattern decouples service initialization from service processing. The primary difference is that the Configuration pattern focuses more on the active composition of a chain of related services, whereas the Service Configurator pattern focuses on the dynamic initialization of service handlers at a particular endpoint.

1.12 Consequences

The Service Configurator pattern offers the following **benefits**:

Uniform configuration and control interfaces: The Service Configurator pattern imposes a uniform configuration and control interface, which allows services to be treated as building blocks that can be integrated as components into a larger application. Enforcing a uniform interface across all the services makes them "look and feel" the same with respect to their configuration process. In turn, this uniformity simplifies application development by promoting the "principle of least surprise."

Centralized administration: 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/closing files and acquiring/releasing locks. In addition, it centralizes the administration of communication services by imposing uniform configuration management operations, such as init, suspend, resume, and fini.

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

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

Increased opportunity for tuning and optimization: The pattern increases the range of service configuration alternatives available to developers by decoupling service functionality from service execution mechanisms. For instance, developers can tune daemon concurrency strategies adaptively to match client demands and available OS processing resources. Common execution alternatives include spawning a thread or process upon the arrival of a client request or pre-spawning a thread or process at service creation time.

The Service Configurator pattern has the following **liabilities**:

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

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

Reduced security or reliability: An application that uses the Service Configurator pattern may be less secure or reliable than an equivalent statically configured application. It may be less secure since impostors can masquerade as services in DLLs. It 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 collocated in the same process.

Increased overhead: The pattern adds extra levels of indirection to execute services. For instance, the `Service Configurator` first initializes services and then loads them into the `Service Repository`. This may incur excessive overhead in time-critical applications. In addition, when dynamic linking is used to implement services many compilers adds extra levels of indirection to method invocations and global variable accesses [18].

Overly narrow common interfaces: The initialization or termination of a service may be too complicated or too tightly coupled with its context to be performed in a uniform manner via common service control interfaces, such as `init` and `fini`.

References

[1] Object Management Group, *CORBA Time Service Specification*, OMG Document orbos/97-12-21 ed., December 1997.

[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] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[6] 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.

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

[8] 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.

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

[10] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[11] 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.

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

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

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

[15] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[16] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the* 3^{rd} *Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

[17] 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.

[18] 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.