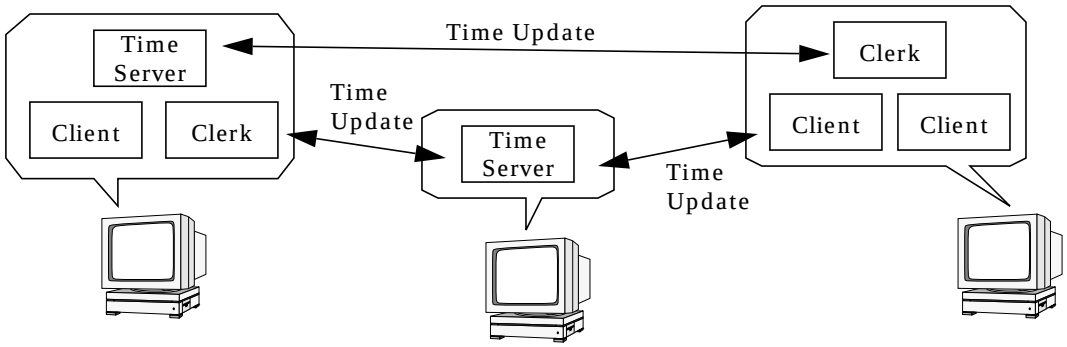
Service Configurator

The *Service Configurator* architecture pattern decouples the behavior of services from the point in time at which service implementations are configured into application processes.

Example A distributed time service [OMG97] provides accurate, fault-tolerant clock synchronization for computers collaborating in networks. A synchronized time service is important for distributed systems where hosts must maintain accurate global time across a network. For instance, large-scale distributed electronic medical imaging systems [PHS96] require globally synchronized clocks that can ensure patient exams are accurately timestamped and analyzed expeditiously by radiologists throughout a distributed health-care delivery system. The architecture of the distributed time service contains the following components:

- *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 [GZ89] [Cris89], and update their own local system time.
- *Clients* use the global time information maintained by their clerks to provide consistent notion of time used by clients on other hosts.



A common way to implement such a distributed time service is to configure the functionality of time servers, clerks, and clients *statically*

into separate monolithic processes. In this design, one or more hosts would run time server processes that service time update requests from clerk processes. A clerk process is run on each host that requires global time synchronization. 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.

The architecture described above has been used in production distributed systems. However, this statically configured, overly process-centric design yields an inflexible, and often time/space inefficient, solution. The main problem with static configuration is that it tightly couples at compile-time the *implementation* of a particular service with the *configuration* of the service with respect to other services in application processes. This tight coupling makes it hard to change service implementations and configurations without having to modify, recompile, and relink existing application processes, as well as shutdown, reconfigure, and restart running processes.

Context An application or system where component-based services must be initiated, suspended, resumed, and terminated flexibly.

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

- It should be possible to make service configuration decisions at any point in the application development cycle. For instance, it should be straightforward to collocate selected services into a single process for particular application use cases, as well as partition them into separate processes—and even separate hosts—to support different use cases. Statically binding service implementations to service configurations reduces this flexibility, however. The problem is that developers often do not know *a priori* the most effective way to collocate or distribute multiple service components into processes and hosts. Improper service configurations can significantly reduce overall system performance and functionality.

➥ The lack of memory resources in wireless computing environments may require client and clerk service implementations to be split into two independent processes running on separate hosts. Conversely, in a real-time avionics environment it may be necessary to collocate the clerk and time server service implementations 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.

- It should be possible to make service implementation decisions at any point in the development cycle. This degree of flexibility is often necessary because changes to service functionality or implementation are common in many systems and applications. In general, modifications to a particular service implementation should have minimal impact on the implementation of applications or other components that use the service.

  ➥ In the real-time avionics environment mentioned in the previous force, client and clerk service implementations might be collocated into one process to reduce latency. If the distributed time algorithm implemented by the clerk changes, however, other clerks and client components that use the clerk should not be affected by these implementation changes. ❑

- It should be possible to change services configurations and implementations at any point in the development cycle. For instance, initial choices may not be optimal over time because better algorithms or architectures be discovered. Likewise, in large-scale systems it may be necessary to distribute services to other processes and hosts because a collocated configuration may not scale efficiently. In addition, platform upgrades for highly available systems, such as telecommunication switches or call centers [SchSu94] may require service reconfiguration without disrupting running services.

  ➥ If we change the clerk's implementation of the time synchronization algorithm, it may be undesirable to recompile,

1. If the clerk and the server are collocated in the same process, the clerk can optimize communication by eliminating the need to set up a transport connection with the server and directly accessing the server's notion of time via shared memory [PRSW+99].

relink, and restart the whole application that contains the clerk. Therefore, it should be possible to initiate, suspend, resume, and terminate the clerk dynamically at run-time, without affecting other currently executing services. ❑

- Performing common service administrative tasks, such as their configuration, initiation, and control, should be straightforward and service-independent. Often, these administrative tasks can be managed more effectively from a central location, rather than being scattered throughout an application or system.

  ➥ The administrative code for initializing and terminating clerks and time servers within server processes 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. ❑

Solution Decouple the behavior of services from the point in time at which these services are configured into application processes. 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 dynamically, as well as to obtain run-time information about each configured service. The services themselves reside within a *service repository* and can be added and removed to and from the service repository under control of a s*ervice configurator.*

Structure The Service Configurator pattern includes four participants:

A *service* defines an interface to control a service. Common control operations include initializing, suspending, resuming, and terminating a service.

Concrete services implement the service control interface and other service-specific functionality, such communicating with peers.

A *service repository* maintains a repository of all the concrete services offered by a Service Configurator-enabled application. This allows administrative entities to manage and control the behavior of configured concrete services via a central administrative unit.

A *service configurator* is responsible for coordinating the (re)configuration of concrete services via the service repository.

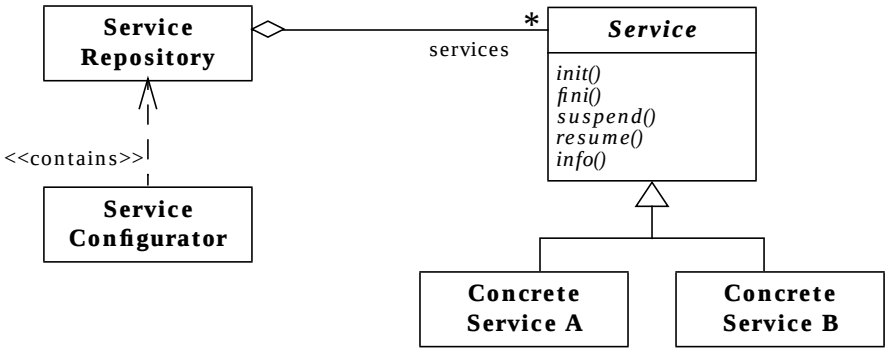
Class Service	***Collaborator***
Responsibility • Defines an interface for configurable components	

Class Concrete Service	***Collaborator***
Responsibility • Implements an application service that should by dynamically configurable	

Class Service Repository	***Collaborator*** • Concrete Services
Responsibility • Maintains the configurated services	

Class Service Configurator	***Collaborator*** • Concrete Services • Service Repository
Responsibility • Configures services into an application process	

The UML class diagram for the Service Configurator pattern is as follows



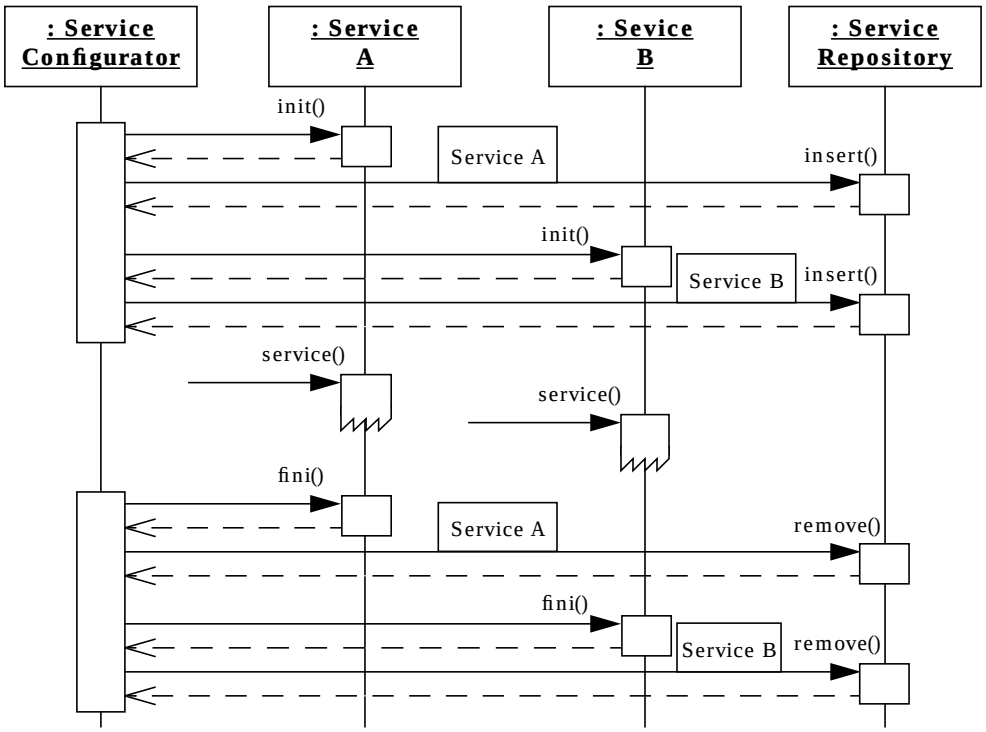
➥ Two concrete services, time server and clerk, appear in the distributed time service. Each concrete service provides specific

functionality to the distributed time service. The time server service receives and processes requests for time updates from clerks. The clerk service queries one or more time servers to determine the approximate correct time and to update its own local system time. ❑

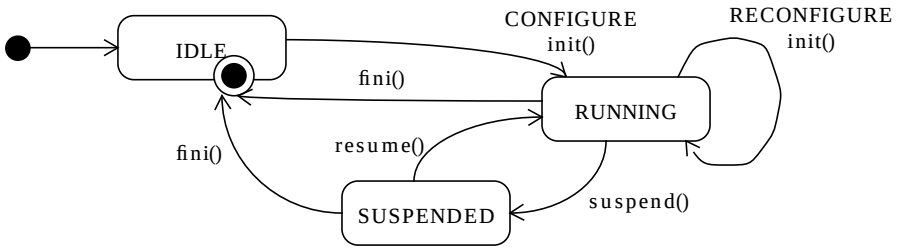
Dynamics The canonical behavior of the Service Configurator pattern includes three phases:

- Service initialization. Services is initialized by the service configurator, which dynamically links the service into an application or launches a process to execute the service 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.
- Service processing. After being configured into an application, a service performs its processing tasks, such as 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.
- 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. Likewise, if the service was launched in a separate process the service configurator can terminate the process.

2. The *Implementation* section describes how parameters can be passed into the service, as well as how the service can be activated.



The following state diagram further illustrates how a service configurator controls the life-cycle of a service.



Implementation The participants in the Service Configurator pattern can be decomposed into two layers: *configuration management infrastructure* and *application components*. Components in the configuration management layer perform general, application-independent strategies for installing, initializing, controlling, and terminating services. Components in the application layer implement service-specific processing.

The implementation guidelines in this section start at the bottom with the configuration management layer and work upwards to the application layer.

1 *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 to 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 attributes of a service.

A service control interface can be based on *inheritance* or *message passing*.

Inheritance-based interface. In this strategy, each service inherits from a common base class that contains pure virtual hook methods for each service control operation.

➥ For instance, the following abstract `Service` class is defined in ACE [Sch94]:

```
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;
};
```
❑

Message-based interface. 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, which are sent to the service by the service configurator framework. Service developers must write code to process these messages, for instance to initialize, suspend, resume, and terminate a service, respectively. 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.

2 *Implement a service repository*. All concrete 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 maintained according to the Manager pattern [PLoP96]. This collection can be stored in main memory, the file system, or shared memory and controlled by a separate process or linked into the application's process.

➥ The methods of `Service_Repository` class defined in ACE [Sch94] are:

```
class Service_Repository {
public:
    // = Initialization and termination operations.
    // Initialize the repository.
    Service_Repository (void);
    // Close down the repository and free up
    // dynamically allocated resources.
    ~Service_Repository (void);

    // = Container operations.
    // Insert a new <Service> with <service_name>.
    int insert (const char service_name[],
                const Service *);
    // Locate <service_name>.
    int find (const char service_name[],
              const Service **service);
    // Remove <service_name>.
    int remove (const char service_name[]);

    // = Liveness control.
    // Suspend <service_name>.
    int suspend (const char service_name[]);
    // Resume <service_name>.
    int resume (const char service_name[]);
private:
    // ...
};
```

❑

3 *Implement the service configuration infrastructure*. The service configurator integrates the other participants in the pattern in order to manage the static and/or dynamic configuration of services into application processes. For instance, it is responsible for coordinating the (re)configuration of services via the service repository. The implementation of a service configurator involves the following steps:

3.1 *Define the service configurator interface*. The service configurator is typically implemented as a Facade [GHJV95] 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 the facade used for our distributed time server example:

```
class Service_Configurator {
public:
     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>.
};
```

All processing of configuration directives is performed in the constructor of `Service_Configurator`.

3.2 *Define the service (re)configuration mechanism*. A service must be configured before it can be executed. Thus, the service configuration mechanism should support the following features:

- *An API for specifying per-service attributes*. Service 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 needed 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 [GHJV95] or standard parser-generator tools like LEX and YACC [SchSu94].

- *A reconfiguration mechanism*. A reconfiguration mechanism allows service implementations and service configurations to evolve dynamically without affecting the execution of other services in an application process. Reconfiguration can be triggered in various ways, such as generating an external event, such as the UNIX SIGHUP signal, or sending an notification via an IPC channel, such as a socket connection or a CORBA operation invocation. On receipt of a reconfiguration event, the service configurator reinterprets any modified service configuration attributes.

➥ To simplify installation and administration, the service configurator in our distributed time server example uses the mechanism provided by ACE [Sch94]. It is based on 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 by developers or administrators. Every service to be (re)configured is specified by 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 '*' | 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 which configuration-related operation to perform during service configuration or reconfiguration. The following table outlines the purpose of these directives.

Directive	Description
`dynamic`	dynamically link and enable a servive

Directive	Description
`static`	Enable a statically linked service
`remove`	Completely remove a service
`suspend`	Suspend a service withot removing it
`resume`	Resume a previously suspended service

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 automatically 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 class.
     return new Time_Server;
}
```

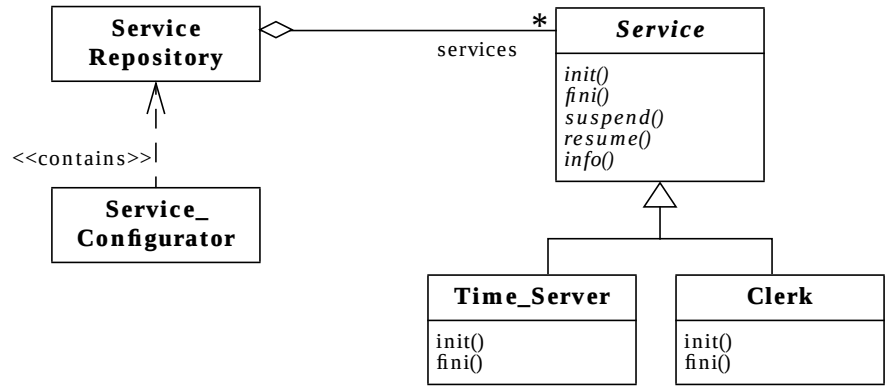
The string parameter at the end of the entry specifies an environment variable containing a port number upon which the time server will accept clerk connections. The service configurator converts this string into an '`argc/argv`'-style vector and passes it to the `init()` hook of the time server. If the `init()` method successfully initializes the service, a pointer to the service is stored in the service repository under the name `Time_Server`. This name identifies the dynamically configured service so that it can be controlled dynamically via the service configurator. ❑

4 *Implement the concrete services and service execution mechanism*. In this step, implement the concrete services and determine the service execution model. For instance, a service that has been configured by a service configurator can be executed using various concurrency models based on the Reactor (71) and Active Object (183) patterns:

- *Reactive execution*. A single thread of control can be used for service configuration processing, as well as the execution of all the services configured by the service configurator.
- *Multi-threaded Active Objects*. In this approach, the configured services execute in their own threads of control within a service sonfigurator-enabled process. For instance, services can spawn new threads 'on-demand' or execute them within a pre-spawned pool of threads.
- *Multi-process Active Objects*. In this approach, the configured services execute in their own processes. For instance, services can spawn new processes 'on-demand' or execute within a pre-spawned pool of processes.

Example Resolved

The Service Configurator-enabled component model for our distributed time server example is defined as follows:



This design uses an inheritance-based strategy for service control. The concrete service participants of the pattern are represented by the `Time_Server` and `Clerk` components, which inherit from the `Service` class. The `Time_Server` service is responsible for receiving and processing requests for time updates from `Clerk`s. The `Clerk` service is a connector factory that is designed according to the Acceptor-Connector pattern (99). It is responsible for creating new connections to time servers, dynamically allocating handlers to send time update requests to connected time servers, receiving these server replies, and then updating the local system time.

Our example implementation uses a configuration mechanism based on explicit dynamic linking [SchSu94] and a `svc.conf` configuration file. This configuration mechanism supports the dynamic configuration of `Clerk` and `Time_Server` components into the distributed time service via *scripting*. Moreover, these features allow `Clerk` components to change how their algorithms compute local system time *without* affecting the execution of other components controlled by the service configurator. Once an algorithm has been modified, the `Service_Configurator` can dynamically reconfigure the `Clerk` component. The service execution mechanism is based on a reactive event handling model within a single thread of control, as described by the Reactor pattern (71).

The time server and clerk implementations. Our initial `Time_Server` implementation is based on Cristian's algorithm [Cris89]. 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` class, 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, allowing developers to change the implementation of the `Time_Server`'s algorithm independently from the time or context where it is configured:

```
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 to shutdown and cleanup the 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:

```
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` 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.

Configuring an application dynamically. There are two general strategies for configuration a distributed time service application: collocated and distributed. We outline each strategy to illustrate how a Service Configurator-enabled application can be dynamic (re)configured and run.

Collocated configuration. This configuration uses a `svc.conf` file to collocate the `Time_Server` and the `Clerk` within the same process. The following generic `main()` program configures services dynamically within the constructor of the `Service_Configurator`

object and then runs the application's event loop, which in our example is based on the Reactor pattern (71).

```
int main (int argc, char *argv[]) {
     // Configure services into the server process.
     // Each service registers itself with the
     // Singleton Reactor.
     Service_Configurator server (argc, argv);

     // Perform service processing and any
     // reconfiguration updates using a Reactor.
     for (;;)
          Reactor::instance ()->handle_events ();
     /* 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 to the `Service_Configurator` 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 to create new service instances. In our example the factory functions are `make_Time_Server()` or `make_Clerk()`, which are defined as follows:

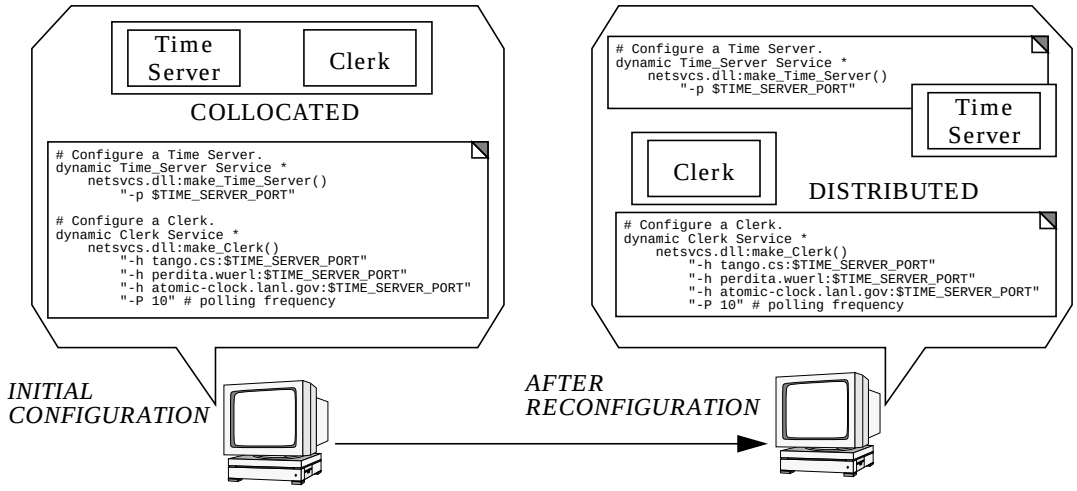
```
Service *make_Time_Server (void) {
     // Time_Server must inherit from Service.
     return new Time_Server;
}

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

Once each factory function returns the new allocated service, 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 want to collocate the `Time_Server` and the `Clerk` in different processes. Due to the flexibility of the Service Configurator pattern, all that is needed 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.

The figure below shows what the configuration looks like with the `Time_Server` and `Clerk` collocated in the same process, as well as the new configuration after the reconfiguration 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.



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 [Cris89] to the Berkeley algorithm [GZ89]. In the Berkeley algorithm, the `Time_Server` is an active component that polls every machine in the network periodically to determine what time it is there. Based on the responses it receives, it computes an aggregate notion of the correct time and instructs 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 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, a socket, a CORBA event callback, 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

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.

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 `Service_Configurator` in the process to reread the configuration file and add the updated `Time_Server` service component to the repository. This component would start executing once its `init()` method was called by the `Service_Configurator`.

While the `Service_Configurator` is 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 dynamically exemplifies the flexibility offered by the Service Configurator pattern.

Known Uses Modern operating system **device drivers**. Most modern operating systems, such as Solaris, Linux, 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 server management. The Service Configurator pattern has been used in 'superservers' that manage network servers. Two widely available network server management frameworks are `inetd` [Ste90]

and `listen` [Rago93]. Both frameworks consult configuration files that specify the following information:

- *Service names*, such as standard `Internet` services like `ftp`, `telnet`, `daytime`, and `echo`,
- *Port numbers* to listen on for clients to connect with these services, and
- *An executable file* to invoke and perform the service when a client connects.

Both `inetd` and `listen` contain a master acceptor (99) 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 pre-registered service handler. This handler performs the service, either reactively (71) or as an active object (183), and returns any results to the client.

The Windows NT Service Control Manager (SCM). The `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 `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 server 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 [Sch97]. ACE 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 is presented in the original version of this pattern [JS97b].

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 for services. This uniformity allows services to be treated as building blocks that can be integrated as components into a larger application. Enforcing a common interface across all services makes them 'look and feel' the same with respect to their configuration activities, which 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 enabling common service initialization and termination activities, such as opening/closing files and acquiring/releasing locks, to be performed automatically. In addition, it centralizes the administration of communication services by ensuring that each service supports the same configuration management operations, such as `init()`, `suspend()`, `resume()`, and `fini()`.

Increased modularity, testability, and reuse. The pattern improves application modularity and reusability by decoupling the implementation of services from manner in which the services are configured into processes. Because all services have a uniform configuration and control interface, monolithic applications can be decomposed more easily into reusable component-based serivices

that can be developed and tested independently. This separation of concerns encourages greater reuse and simplifies development of subsequent services.

Increased configuration dynamism and control. The pattern enables a service to be dynamically reconfigured without modifying, recompiling, or statically relinking existing code. In addition, (re)configuration of a service often can be performed without restarting the service or other active services with which it is collocated.[4] These features help create an infrastructure for user-defined application service configuration frameworks.

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 server concurrency strategies adaptively to match client demands and available operating system 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 and ordering dependencies. 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, because a dynamically configured service may not behave predictably when run with certain other services. For example, a newly configured service may consume excessive CPU cycles, thereby starving out other services and causing them to miss their deadlines.

Reduced 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 because impostors can masquerade as services in DLLs. It may be less reliable because a particular configuration of services may adversely affect the execution of the services. For instance, a faulty

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*.

service may crash, thereby corrupting state information it shares with other services configured into the same process.

Increased run-time overhead and infrastructure complexity. The pattern adds extra levels of abstraction and indirection in order 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 [GLDW87].

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()`.

See Also The intent of the Service Configurator pattern is similar to the Configuration pattern [CMP95]. 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.