The Service Configurator Framework

An Extensible Architecture for Dynamically Configuring Concurrent, Multi-Service Network Daemons

Douglas C. Schmidt and Tatsuya Suda
schmidt@ics.uci.edu and suda@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine, CA 92717, (714) 856-4105

An earlier version of this paper appeared in the proceedings of the IEEE Second International Workshop on Configurable Distributed Systems, Pittsburgh, PA, March 1994.

Abstract

Developing extensible, robust, and efficient network daemons is a challenging task. This paper describes an object-oriented framework consisting of automated tools and reusable components that simplifies the task of developing, configuring, and reconfiguring concurrent, multi-service network daemons. These daemons may contain multiple communication-related services that execute in one or more processes or threads. This framework uses object-oriented design techniques and C++ language features to enhance operating system mechanisms that provide interprocess communication, communication port demultiplexing, explicit dynamic linking, and parallel processing. In addition to describing the object-oriented architecture of the daemon control framework, this paper presents an example that illustrates how the framework supports the development of network software whose services may be updated and extended without modifying, recompiling, relinking, or restarting active daemons.

1 Introduction

This paper describes the structure and functionality of the `Service Configurator` framework. This framework contains automated tools and reusable components that help to simplify the development, configuration, and reconfiguration of concurrent, multi-service network daemons. A daemon is an operating system (OS) process that executes application services on a host machine in the "background" (*i.e.,* disassociated from any controlling terminal) [1]. A service is a portion of a daemon that offers a single processing capability to communicating entities. Particular emphasis is given in this paper to concurrent, multi-service network daemons. These daemons contain multiple communication-related services that execute in one or more processes or threads.

An increasingly wide range of network daemons have become available throughout the Internet. These daemons provide communication-related services that resolve distributed name binding requests (`named` and `rpcbind`), access network file systems (`nfsd`), manage routing tables (`gated` and `routed`), report host and user activities (`rwhod` and `fingerd`), and consolidate multiple network services such as terminal access (`rlogin` and `telnet`) and file transfer (`ftp`) together into a single "superserver" framework (`inetd` and `listen`). The `Service Configurator` framework we have developed provides an integrated set of tools that automate many activities commonly performed to configure these types of network daemons.

The `Service Configurator` framework performs the following configuration-related activities:

- Determining the service(s) that will be advertised by the daemon. This may be accomplished by statically coding this information into an application or by dynamically consulting a configuration file or service database.
- Requesting the distributed operating environment to bind the communication ports and network addresses that identify each advertised service. This process may require interactions with distributed name servers.
- Registering a handler for each service with an event dispatcher. This handler is responsible for statically or dynamically linking the service's implementation into the daemon's address space when the service is activated.

After the initial set of services have been configured, an application typically enters an event loop that waits for events (such as connection requests or data messages) to arrive from clients. When these events occur, the `Service Configurator` framework executes the appropriate service handler using one or more processes or threads. If the service associated with the handler has not yet activated, the `Service Configurator` framework will automatically activate it (dynamically linking it into the daemon's address space if necessary).

To simplify these configuration and run-time activities, the `Service Configurator` framework leverages off an integrated collection of reusable C++ components. These components enhance the modularity, extensibility, and portability of advanced OS mechanisms that provide interprocess communication (IPC), event demultiplexing, explicit dynamic linking, and concurrency. In addition, the `Service Configurator` framework's components may be used to enhance the performance of network daemons by deferring until installation-time the choice of execution agent (*e.g.,* one or more processes or threads) used to run network services. This enables more effective use of the multi-processing capabilities available on the target OS platform [2].

The `Service Configurator` framework also coordinates OS mechanisms to automate the reconfiguration of daemon services at run-time. In many circumstances, this allows daemons executing within the `Service Configurator` framework to dynamically update their functionality without being completely shutdown and restarted. Support for dynamic reconfiguration is essential for highly available distributed applications such as mission-critical systems that perform on-line transaction processing, real-time remote process control, or global personal communication systems based on low earth orbit satellites. For these types of applications, it is often necessary to phase new versions of a service into a daemon without disrupting currently executing services [3].

This paper is organized in the following manner: Section 2 reviews related background material; Section 3 outlines the primary features in the `Service Configurator` framework; Section 4 describes the object-oriented architecture of the `Service Configurator` framework; Section 5 illustrates how the `Service Configurator` framework components are applied to configure a distributed logging facility used in a commercial on-line transaction processing system; and Section 6 presents concluding remarks.

2 Background Material

Various strategies and tactics for developing configurable distributed application frameworks have emerged in several domains. The `Service Configurator` framework is influenced by research that addresses policies for (1) reliably guiding the reconfiguration of executing applications [4] and (2) representing application state attributes as abstract data types to facilitate service (re)configuration [5], communication [3], and migration in heterogeneous and homogeneous [6] environments.

Another influential branch of research involves daemon control frameworks. Daemon management frameworks provide mechanisms that automate many tedious and error-prone activities associated with configuring and reconfiguring network daemons. These activities include (1) performing daemonization[1] operations; (2) binding transport endpoints to communication ports; (3) demultiplexing events received on these ports; and (4) dispatching the appropriate handlers to process the events.

Two widely available daemon management frameworks are `inetd` [1] and `listen` [7], which are both distributed with System V Release 4 UNIX. `Inetd` and `listen` are multi-service daemon management frameworks that utilize a master dispatcher process to monitor a set of communication ports. Each port is associated with a communication-related service (such as the standard Internet services `ftp`, `telnet`, `daytime`, and `echo`). When a service request arrives on a monitored port, the dispatcher process demultiplexes the request to the appropriate pre-registered service handler. This handler performs the service and returns any results to the client requestor. Long-duration external services[2] (such as `ftp` and `telnet`) are executed concurrently in separate slave processes. In addition, `inetd` may be configured to execute short-duration internal services (such as `daytime` and `echo`) iteratively within its master dispatcher process address space (note that `listen` does not provide this type of functionality).

Both `inetd` and `listen` have proven to be quite useful in practice. However, these daemon management frameworks were developed without adequate consideration of object-oriented techniques (such as class-based encapsulation, inheritance, dynamic binding, and parameterized types) and advanced OS mechanisms (such as explicit dynamic linking and multi-threading). This fact complicates component reuse and limits functionality. For example, the standard version of `inetd` is written in C and its implementation is characterized by a proliferation of global variables, a lack of information hiding, and an algorithmic decomposition that deters fine-grained reuse of its internal components. Furthermore, neither `inetd` nor `listen` provide automated support for (1) dynamically linking services into the address space of their master dispatcher processes at run-time or (2) executing these services concurrently via one or more threads. Therefore, developers who want their services to benefit from these advanced OS mechanisms must manually program them into their network daemons.

3 Overview of the Service Configurator Framework

Figure 1 illustrates the primary architectural components in `inetd`, `listen`, and the `Service Configurator` framework. Each of these daemon management frameworks support the automated configuration and reconfiguration of multi-service daemons that execute their external services concurrently in separate process address spaces. In addition, the `Service Configurator` framework offers added support for concurrent, multi-service network daemons by increasing configuration flexibility, enhancing daemon run-time extensibility, and improving performance. The

[1] Daemonization typically involves (1) dynamically spawning a new process, (2) closing all unnecessary file descriptors, (3) changing the current working directory to the root directory, (4) resetting the file access creation mask, (5) disassociating from the controlling process group and the controlling terminal, and (6) ignoring terminal I/O-related signals [1].

[2] An *external service* is executed in a different process address space than the master dispatcher process that received the request. An *internal service*, on the other hand, is executed within the same address space as the dispatcher process.

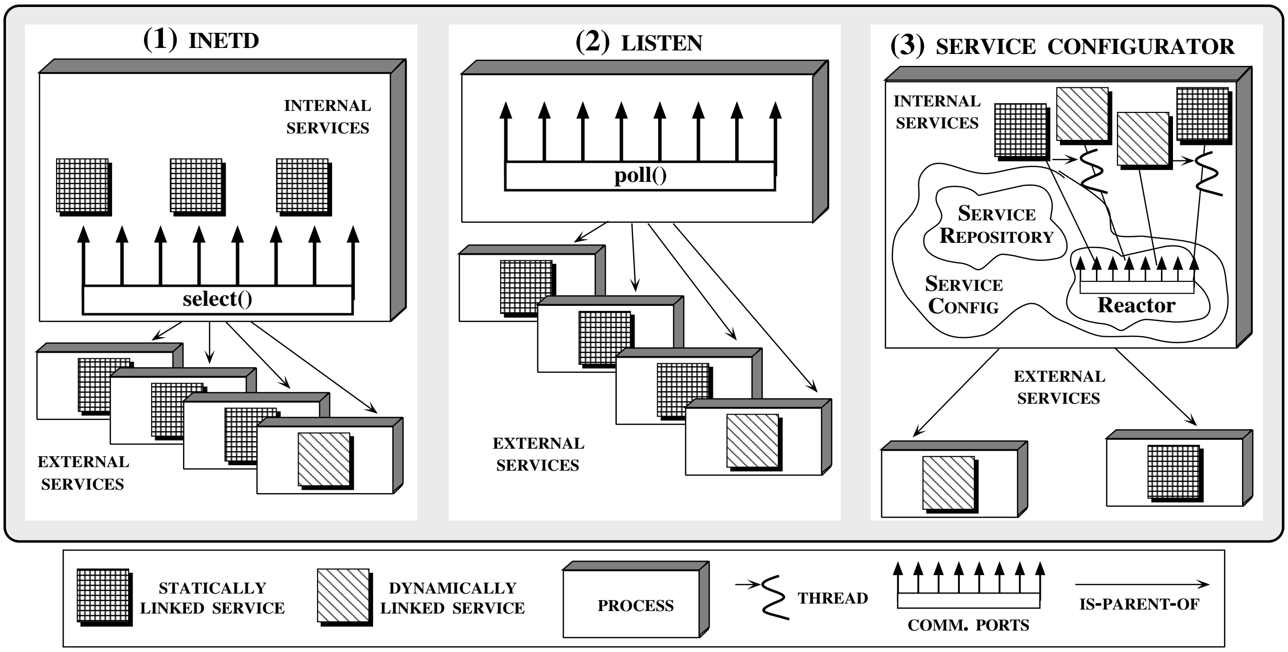

Figure 1: Alternative Daemon Management Frameworks

remainder of this section examines the features provided by the `Service Configurator` framework and illustrates how these features overcome the limitations with `inetd` and `listen` discussed earlier.

The `Service Configurator` framework increases the flexibility of configuring network daemons by decoupling service-specific processing policies from the following development activities and mechanisms:

• *The type and number of services associated with each daemon process:* The `Service Configurator` framework permits daemons to consolidate a number of services into one administrative unit. This multi-service approach to configuring network daemons helps to (1) simplify development and reuse code by performing common service initialization activities automatically, (2) reduce the consumption of OS resources (such as process table slots) by spawning service handlers "on-demand," (3) allow daemon services to be updated without modifying existing source code or terminating an executing dispatcher process, and (4) consolidate the administration of network services via a uniform set of configuration management operations.

• *The point of time at which a service is configured into a daemon:* Both `inetd` and `listen` support the activities listed in the previous bullet. However, they are less flexible with respect to the point of time at which certain types of services may be configured into a network daemon. For example, `inetd` only allows external services to be reconfigured at run-time. Internal services, on the other hand, must be configured into the master `inetd` daemon *statically* (*i.e.,* at compile-time or static link-time). In order to add a new internal service, the master `inetd` dispatcher process must be modified, recompiled, relinked, and restarted (the `listen` daemon control framework does not support internal services).

In contrast, the `Service Configurator` framework provides an extensible high-level object-oriented interface (described in Section 4.2) that automates the use of OS mechanisms for explicit dynamic linking.[3] Dynamic linking enhances the extensibility of network daemons by permitting internal services to be configured into a daemon *dynamically* (*i.e.,* when a daemon first begins executing or while it is running). This feature enables a network daemon's services to be dynamic reconfigured *without* requiring the modification, recompilation, relinking, or restarting of active services.

In the `Service Configurator` framework, the choice between static or dynamic configuration may selected on a per-service basis. Furthermore, this choice may be deferred until a daemon begins execution. Neither `inetd` nor `listen` provide built-in support for explicit dynamic linking, which limits the range of daemon design alternatives that they are capable of configuring automatically.

• *The type of execution agents:* In the `Service Configurator` framework, services may be performed at run-time via several different types of process and thread execution agents (discussed in Section 4.3.1). By decoupling service functionality from the execution agent used to invoke the service, the `Service Configurator` framework increases the range of daemon configuration alternatives available to developers.

An efficient daemon configuration often depends upon certain service requirements and platform characteristics. For

[3]Explicit dynamic linking allows a daemon to obtain, utilize, and/or remove the run-time address bindings of services defined in shared object files [8]. Widely available explicit dynamic linking facilities include the `dlopen/dlsym/dlclose` routines in SVR4 UNIX and the `LoadLibrary/GetProcAddress` routines in the WIN32 subsystem of Windows NT.

example, a process-based configuration may be appropriate for daemons that implement long-duration services (such as the Internet `ftp` and `telnet`) that base their security mechanisms on process ownership. In this case, each service (or each active instance of a service) may be mapped onto a separate process and executed in parallel on a multiprocessor platform. Different configurations may be more suitable in other circumstances, however. For instance, it is often simpler and more efficient to implement cooperating services (such as those found in an end-system of a distributed database engine) in separate threads since they frequently reference common data structures. In this approach, each service may be executed on a separate thread within the same process to reduce the overhead of scheduling and context switching [2].

As before, the range of daemon design alternatives supported by `inetd` and `listen` are limited since they do not provide built-in support for thread-based service execution. Instead, these daemon management frameworks utilize a less flexible and often less efficient invocation mechanism that spawns a separate process to execute a service dynamically. The overhead of `fork` and `exec` makes dynamic service invocation prohibitively expensive for certain types of services, particularly short-duration services (such as resolving the Ethernet number of an IP address or retrieving a disk block from a network file server).

• *The order in which hierarchically-related services are combined into an application:* Each service is represented as an interconnected series of independent service objects that communicate by passing messages. These objects may be joined together in essentially arbitrary configurations to satisfy applications requirements and enhance component reuse.

• *The I/O descriptor-based and timer-based event demultiplexing mechanisms:* These mechanisms are used to dispatch incoming connection requests and data onto a preregistered service-specific handler. Within the `Service Configurator` framework, a class category called the `Reactor` [9] portably encapsulates both the `select` and `poll` I/O demultiplexing system calls. `Select` and `poll` are UNIX system calls that detect the occurrence of different types of input and output events on one or more I/O descriptors simultaneously. As discussed in Section 4.1, the `Reactor` class category integrates the demultiplexing of I/O descriptor-based events together with timer-based and signal-based events via an extensible and type-safe object-oriented interface.

In contrast, `inetd` and `listen` do not provide timer-based event demultiplexing functionality. Moreover, `inetd` is tightly coupled to the `select` interface and `listen` is tightly coupled to the `poll` interface, which limits their portability.

• *The underlying IPC mechanisms:* Services use IPC mechanisms to exchange data with participating communication entities on local or remote end-systems. The `Service Configurator` framework provides a collection of C++ wrappers [10] that encapsulate standard OS mechanisms for network IPC (such as sockets and the Transport Layer Interface (TLI)). Unlike the weakly-typed, "descriptor-based" socket and TLI interfaces, however, these C++ wrappers enable services to access the underlying OS IPC mechanisms via a type-safe, portable interface. Conversely, neither `inetd` nor `listen` provide any standard support for type-safe, platform-independent IPC interfaces.

• *The client and server partitioning:* Conventional daemon management frameworks are not involved with client-side issues. In contrast, the `Service Configurator` framework provides mechanisms for partitioning and composing the services of an application into the client-side or server-side of a distributed system.

The increased flexibility provided by the `Service Configurator` framework facilitates the development and use of reusable components. In general, component reuse is improved in the framework by separating the higher-level service-specific policies from the lower-level mechanisms (such as network communication, event demultiplexing and service dispatching, and execution agents). Conversely, both `inetd` and `listen` allow only coarse-grain reuse of their general service dispatching facilities, without encouraging more fine-grain reuse of their internal components.

4 The Object-Oriented Architecture of the Service Configurator Framework

The `Service Configurator` framework was developed using several key object-oriented design techniques and C++ language features. This section outlines the object-oriented architecture of the `Service Configurator` framework. The architectural components discussed below include the `Reactor` class category (Figure 2), the `Service Object` inheritance hierarchy (Figure 3 (1)), the standard subclasses of the `Service Object` class (Figure 3 (2)), the `Service Repository` class (Figure 3 (3)), and the `Service Config` class (Figure 3 (4)).

Booch notation [11] is used to illustrate relationships between these `Service Configurator` framework components. Solid clouds indicate objects; nesting indicates composition relationships between objects; and undirected edges indicate some type of link exists between two objects. Dashed clouds indicate classes; directed edges indicate inheritance relationships between classes; and an undirected edge with a solid bullet at one end indicates a composition relation between two classes. Solid rectangles indicate class categories, which combine a number of related classes into a common name space.

4.1 The Reactor Class Category

The components in the `Reactor` class category are responsible for *demultiplexing* temporal events generated by a timer-driven callout queue, I/O events received on communication

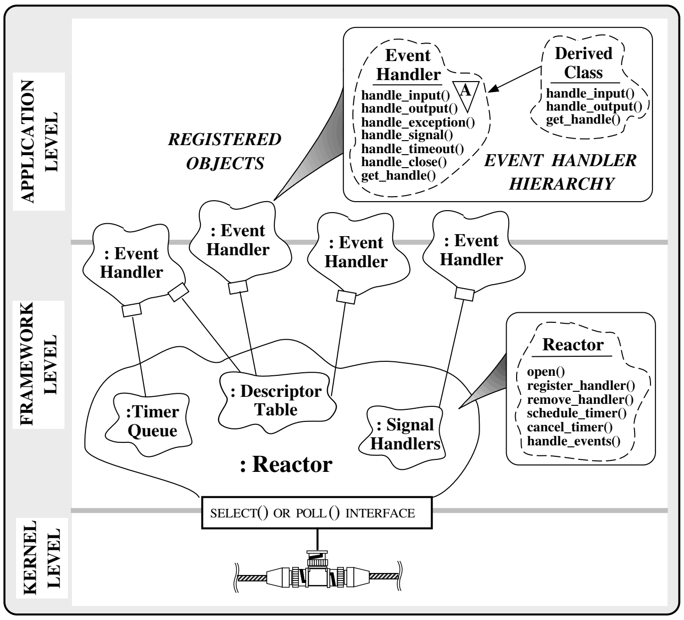

Figure 2: The `Reactor` Class Category

ports, and signal-based events and *dispatching* the appropriate pre-registered handler(s) to process these events. Within the `Service Configurator` framework, multi-service network daemons are structured internally via an event loop that monitors a set of communication ports. Each communication port is associated with a service-specific handler that implements a particular type of functionality on behalf of clients. When a client request arrives on a port monitored by the `Reactor`, the associated service handler is dispatched to perform the request. To consolidate and automate these common request processing activities, the `Service Configurator` framework uses the event demultiplexing and service dispatching mechanisms provided by the `Reactor`.

The `Reactor` class category contains a set of methods illustrated in Figure 2. These methods provide a uniform interface for managing objects that implement various types of service handlers. Certain methods register, dispatch, and remove I/O descriptor-based objects from the `Reactor`. Other methods schedule, cancel, and dispatch timer-based and/or signal-based objects. As shown in Figure 2, these handler objects all derive from the `Event Handler` abstract base class.[4] This class specifies an interface for event demultiplexing and service handler dispatching.

The `Reactor` class category uses the pure virtual methods in the `Event Handler` interface to integrate the demultiplexing of I/O descriptor-based events together with timer-based and signal-based events. I/O descriptor-based events are dispatched via the `handle_input`, `handle_output`, and `handle_exceptions` methods of a handler object. Likewise, timer-based events are dispatched via the `handle_timeout` method and signal-based events are dispatched via the `handle_signal` method. Subclasses of `Event Handler` may augment this interface by defining additional methods and data members. In addition, virtual methods in the interface may be selectively overridden to implement application-specific functionality. When all the pure virtual methods have been defined, a daemon may create an instance of the resulting composite service handler object.

When a daemon instantiates and registers a composite service handler object, the `Reactor` class category extracts the underlying I/O descriptor from the object. This descriptor is stored in a table along with descriptors from other registered objects. When the daemon subsequently invokes its main event loop, these descriptors are passed as arguments to the underlying OS event demultiplexing system call (*e.g.,* `select` or `poll`). As events associated with a registered composite object occur at run-time, the `Reactor` automatically detects these events and dispatches the appropriate method(s) of the service handler object associated with the event. This handler object then becomes responsible for performing its service-specific functionality.

4.2 The Service Configurator Class Category

The primary unit of configuration in the `Service Configurator` framework is the *service*. Services may be simple (such as returning the current time-of-day) or highly complex (such as a distributed, real-time router for PBX event traffic [12]). To provide a consistent environment for defining, configuring, and using network daemons, all network daemon services are derived from the `Service Object` inheritance hierarchy (illustrated in Figure 3 (1)).

The `Service Object` class is the focal point of a multi-level hierarchy of types related by inheritance. The standard interfaces provided by the abstract classes in this type hierarchy may be selectively overridden by service-specific subclasses in order to collaborate with features offered by the `Service Configurator` framework. These framework features provide transparent dynamic linking, service handler registration, event demultiplexing, and service dispatching. By decoupling the service-specific portions of a handler object from the underlying `Service Configurator` framework mechanisms, the effort necessary to insert and remove services from a running network daemon is significantly reduced.

The `Service Object` inheritance hierarchy consists of the `Event Handler` and `Shared Object` abstract base classes, as well as the `Service Object` abstract derived class. The `Event Handler` class was described above in the `Reactor` Section 4.1. The behavior of the other classes is outlined below.

[4] An abstract class in C++ provides an interface that contains at least one *pure virtual method*. A pure virtual method provides only an interface declaration, without any accompanying definition. Subclasses of an abstract class must provide definitions for all its pure virtual methods before any objects of the class may be instantiated.

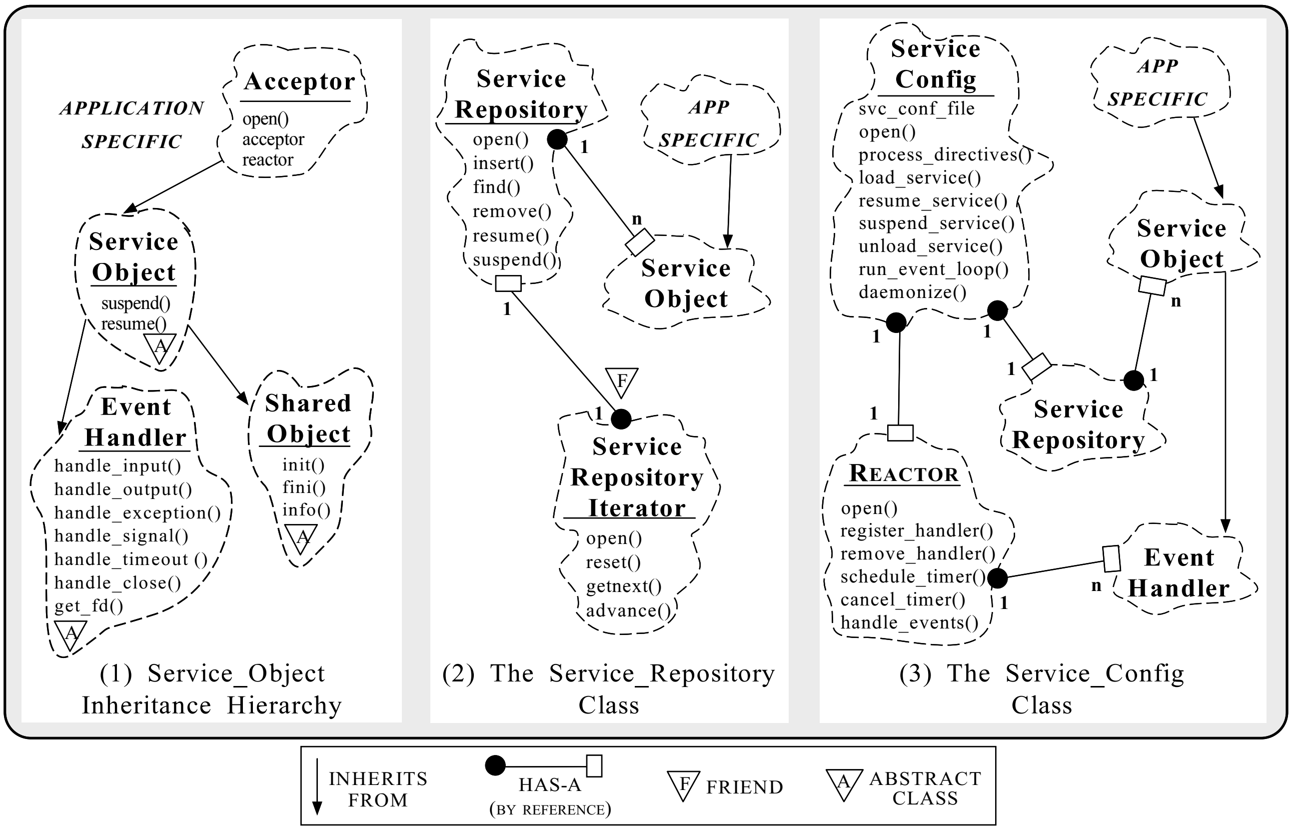

Figure 3: Component Relationships in the `Service Configurator` Framework

4.2.1 The Shared Object Abstract Base Class

The `Shared Object` class specifies an interface for dynamically linking service handler objects into a daemon's address space. This abstract base class exports three pure virtual methods: `init`, `fini`, and `info`. These functions impose a contract between the reusable components provided by the `Service Configurator` framework and service-specific objects that utilize these components. By using pure virtual methods, the `Service Configurator` framework ensures that a service handler implementation honors its obligation to provide certain configuration-related information. This information is subsequently used by the `Service Configurator` framework to automatically link, initialize, identify, and unlink a service at run-time.

The `Shared Object` base class is defined independently from the `Event Handler` class to clearly separate their two orthogonal sets of concerns. For example, certain applications (such as a compiler or text editor) might benefit from dynamic linking, though they might not require communication port event demultiplexing. Conversely, other applications (such as an `ftp` server) require event demultiplexing, but might not require dynamic linking. By separating these interfaces into two base classes, applications are able to select a subset of `Service Configurator` mechanisms without incurring unnecessary performance or storage costs.

4.2.2 The Service Object Abstract Derived Class

In general, support for dynamic linking, event demultiplexing, and service dispatching is necessary to automate the dynamic configuration and reconfiguration of network daemon services. Therefore, the `Service Configurator` framework defines the `Service Object` class, which inherits the interfaces from both the `Event Handler` and the `Shared Object` abstract base classes. The resulting abstract derived class supplies a composite interface that developers use as the basis for implementing and configuring a service into the `Service Configurator` framework. The `Service Object` class acts as an envoy between its two abstract base class components. These two abstract base classes have no knowledge of each other at all.

During development, service-specific subclasses *must* implement the following four pure virtual methods inherited (but not defined) by the `Service Object` subclass:

• `int init (int argv, char **argv);` The `init` method serves as the entry-point to a service handler during run-time initialization. This method is responsible for performing service-specific initialization when an instance of a composite `Service Object` is dynamically linked. When invoked, `init` is passed a pair of "`argc/argv`"-style parameters. Within `init`, these parameters may be treated like arguments passed to the `main` function of a stand-alone executable C++ program and are used to control object initialization.

• `int fini (void);` The `fini` method is called automatically by the `Service Configurator` framework when a `Service Object` is unlinked and removed from a daemon at run-time. This method typically performs termination operations that release dynamically allocated resources (such as memory, synchronization locks, or I/O descriptors).

• `int info (char **string, int length);` The `info` method returns a humanly-readable string that concisely reports service addressing information and documents service functionality. Clients may query a daemon to retrieve this information and use it to contact a particular service running in the daemon.

• `int get_handle (void);` The `get_handle` method is used by the `Reactor` class category to extract the underlying I/O descriptor from a composite service object. This I/O descriptor typically identifies a transport endpoint that may be used to accept connections or receive data from clients.

4.2.3 Service-Specific Concrete Derived Subclasses

`Service Object` is an abstract class since its interface contains the pure virtual methods inherited from the `Event Handler` and `Shared Object` abstract base classes. Therefore, developers must supply concrete subclasses that (1) define the four pure virtual methods described above and (2) implement service-specific functionality. To accomplish the latter task, subclasses typically override the virtual methods exported by the `Service Object` interface. For example, the `handle_input` method is often overridden to accept connections or data that are received from clients.

The `Acceptor` class depicted in Figure 3 (1) is an example of a service-specific subclass that accepts connection requests as part of a distributed logging facility. This class is described further in the example presented in Section 5.

4.3 Standard Subclasses of Service Object

The `Service Configurator` framework contains a library of standard components that inherit from the `Service Object` class. These standard components perform the service invocation and service directory mechanisms described below. Services that want to use these mechanisms may inherit from the `Eager Spawn`, `Lazy Spawn`, `Process Spawn`, `Thread Spawn`, or `Service Manager` subclasses illustrated in Figure 3 (2).

4.3.1 Service Invocation Mechanisms

Several standard subclasses of `Service Object` implement invocation mechanisms that are useful for fine-tuning network daemon performance. The alternatives discussed below enable developers to adaptively tune daemon concurrency levels to match client demands and available OS processing resources. In general, the different approaches tradeoff decreased startup overhead for increased consumption of OS resources (such as process table slots and virtual memory).

• `Eager Spawn` – This subclass pre-spawns one or more processes or threads at daemon creation time. These "warm-started" execution agents form a pool that helps improve response time by reducing service startup overhead when requests arrive from clients. Depending on factors such as number of available CPUs, current machine load, or the length of a client request queue, this pool may be expanded or contracted dynamically.

• `Lazy Spawn` – This subclass does not immediately spawn a process when a client request is received. Instead, a timer is set and the request is handled iteratively by the daemon. However, if the timer expires a new slave process is spawned automatically to continue processing the service independently from the master dispatcher process (which itself continues to wait for subsequent requests) [13].

• `Process Spawn` – This subclass implements the external service process invocation functionality provided by `inetd` and `listen`. It operates by spawning a new slave process "on-demand" in response to the arrival of client requests. The slave process then performs the client request in its own separate address space – terminating when the request is complete. Spawning a process on-demand helps to reduce the consumption of OS resources, at the expense of higher costs for initially starting a service.

• `Thread Spawn` – This subclass implements service spawning techniques that are often more efficient than the process invocation method used by `inetd` and `listen`. Rather than using `fork` and `exec` to create a separate process on a per-request basis, the `Thread Spawn` class creates a separate thread. This thread executes its associated service to completion and then exits.

• `Link Spawn` – The `Link Spawn` subclass dynamically links and executes a new service *without* spawning a new process or thread. This allows services to be loaded and unloaded on demand, rather than being pre-loaded during daemon initialization. The `Link Spawn` subclass is implemented by (1) dynamically linking an object file, (2) obtaining the entry-point of the approriate `Service Object` in this file, and (3) invoking the service to perform the client request.[5] Upon completion, the service installed by `Link Spawn` may be automatically removed by closing the `Service Object` and unlinking the object file from the daemon's address space.

In general, threads are more efficient than processes since they maintain minimal state information, require less overhead to spawn and synchronize, and may communicate via shared memory rather than inter-process message passing [15]. However, since all threads in a process share its global

[5]This approach is similar to the way that programs were invoked in Multics [14].

resources (such as virtual memory, descriptor tables, and signal handlers), one faulty service may corrupt global data accessed by services running in other threads within the process. Therefore, the `Service Configurator` framework provides the flexible set of service invocation mechanisms described above in order to allow developers to select the most appropriate techniques for managing concurrency that satisfy their application requirements.

4.3.2 Service Manager Mechanisms

The `Service Manager` subclass provides local and remote clients with access to daemon administration commands that publish and manage the services currently offered by a network daemon. These commands "externalize" certain internal service attributes in an active network daemon. During daemon configuration, a `Service Manager` object is typically registered at a well-known communication port (*e.g.,* port 911) accessible by clients. If a client requests a list of active daemon services, the `handle_input` method in the `Service Manager` invokes the `Service Repository` iterator (described in Section 4.4 below). This iterator is used to generate a listing of developer-supplied information that describes each active service. This listing is transferred back to the client to indicate both the address format and the transport protocol required to contact daemon services. The `Service Manager` is also be used to handle reconfiguration requests that are triggered by remote sites.

4.4 The Service Repository Class

The `Service Configurator` framework supports the configuration of both single-service and multi-service network daemons. Therefore, to simplify run-time administration, it may be necessary to individually and/or collectively control and coordinate the `Service Objects` and Streams that comprise a daemon's currently active services. The `Service Repository` is an object manager that coordinates local and remote queries and updates involving the services offered by a `Service Configurator`-based daemon. A search structure within the object manager binds service names (represented as ASCII strings) with instances of composite `Service Objects` (represented as C++ object code). A service name uniquely identifies an instance of a `Service Object` stored in the repository.

Each entry in the `Service Repository` contains a pointer to the `Service Object` portion of an service-specific C++ derived class (shown in Figure 3 (3)). This enables the `Service Configurator` framework to load, enable, suspend, resume, or unload `Service Objects` from a daemon statically or dynamically. For dynamically linked `Service Objects`, the repository also maintains a handle to the underlying shared object file where the object code is stored. This handle is used to unlink and unload a `Service Object` from a running daemon when the service it offers is no longer required.

```
<svc-config-entries> ::=
        svc-config-entries svc-config-entry
        | NULL
<svc-config-entry> ::= <dynamic> | <static>
         | <suspend> | <resume> | <remove>
         | <stream> | <remote>
<dynamic> ::= DYNAMIC <svc-location>
                    [ <parameters-opt> ]
<static> ::= STATIC <svc-name>
                  [ <parameters-opt> ]
<suspend> ::= SUSPEND <svc-name>
<resume> ::= RESUME <svc-name>
<remove> ::= REMOVE <svc-name>
<stream> ::= STREAM <stream_ops>
             '{' <module-list> '}'
<stream_ops> ::= <dynamic> | <static>
<remote> ::= STRING '{' <svc-config-entry> '}'
<module-list> ::= <module-list> <module>
                | NULL
<module> ::= <dynamic> | <static>
           | <suspend> | <resume> | <remove>
<svc-location> ::= <svc-name> <type>
                   <svc-initializer> <status>
<type> ::= SERVICE_OBJECT '*' | MODULE '*'
         STREAM '*' | NULL
<svc-initializer> ::= <object-name>
                    | <function-name>
<object-name> ::= PATHNAME ':' IDENT
<function-name> ::= PATHNAME ':' IDENT '(' ')'
<status> ::= ACTIVE | INACTIVE | NULL
<parameters-opt> ::= STRING | NULL
```

Figure 4: EBNF Format for a Service Config Entry

An iterator class is also supplied along with the `Service Repository`. This class is used to visit every `Service Object` in the repository without unduly compromising data encapsulation. As described in Section 4.3.2, the standard `Service Manager` implementation employs this iterator feature to obtain a complete listing of all daemon services that are currently active. It generates this listing by repeatedly calling the `info` method associated with each entry in the `Service Repository`.

4.5 The Service Config Class

The `Service Config` class is the unifying component in the `Service Configurator` framework. As illustrated in Figure 3 (4), this class integrates the other `Service Configurator` framework components (such as the `Service Repository` and the `Reactor`) to automate the static and/or dynamic configuration of concurrent, multi-service network daemons.

The `Service Config` class uses a configuration file (known as `svc.conf`) to guide its configuration and reconfiguration activities. Each daemon may be associated with a distinct `svc.conf` configuration file. Likewise, a set of daemons may be described by a single `svc.conf` file. Figure 4 describes the primary syntactical elements in a `svc.conf` file using extended-Backus/Naur Format (EBNF). Each *service config entry* in the file begins with a *service config directive* that specifies the configuration activity to perform. Table 1 summarizes the valid service config directives.

Symbol	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
stream	Configure a Stream into a daemon

Table 1: Service Config Directives

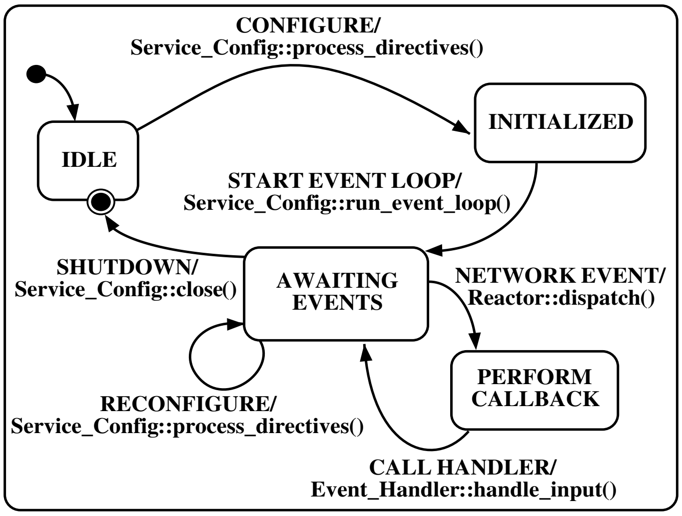

Figure 5: State Transition Diagram for Service Configuration, Execution, and Reconfiguration

Each service config entry contains certain attributes that indicate the location of the shared object file for each dynamically linked service, as well as the parameters required to initialize a service at run-time. By consolidating service attributes and initialization parameters into a single configuration file, the installation and administration of the services in an application is significantly simplified. The `svc.conf` file helps to decouple the structure of an application from the behavior of its services. This decoupling also separates the "lazy" configuration and reconfiguration of mechanisms provided by the framework from the application-specific attributes and parameters specified in the `svc.conf` file.

Figure 5 depicts the state transition diagram illustrating the `Service Config` methods that are invoked in response to events occurring during service configuration, execution, and reconfiguration. For example, when the CONFIGURE or the RECONFIGURE events occur at run-time, the `Service Config` class calls its `process_directives` method. This method consults the `svc.conf` file. This file is first consulted when a new instance of a daemon is initially configured. The file is consulted again whenever a daemon reconfiguration is triggered upon receipt of a pre-designated external event (such as the UNIX SIGHUP signal).

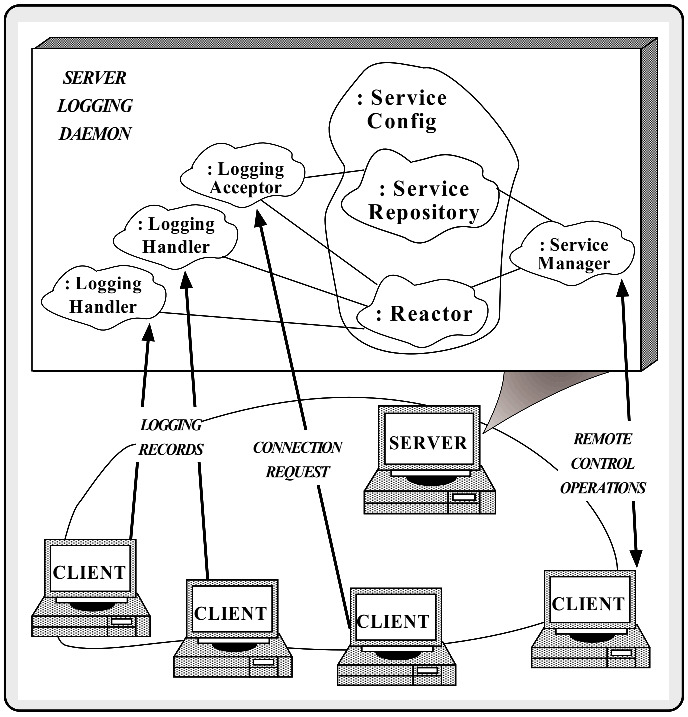

Figure 6: Run-time Configuration of the Server Logging Daemon

5 Distributed Logging Example

Debugging distributed applications is frequently challenging since diagnostic output appears in different windows and/or on different remote host systems. To illustrate how the `Service Configurator` framework is used in practice to simplify distributed application development and administration, the following section examines the architecture of the distributed logging facility shown in Figure 6. This facility is currently used in a commercial on-line transaction processing system [12] to provide logging services for workstations and database engines in a high-speed network environment.

As shown in Figure 6, the distributed logging facility allows applications running on multiple client hosts to send logging records to a server logging daemon running on a designated server host. The remainder of this section focuses on the object-oriented `Service Configurator`-based architecture and configuration of the server daemon portion of the logging facility. The complete design and implementation of the distributed logging facility is described in [9].

5.1 Server Logging Daemon

The server logging daemon is a concurrent, multi-service daemon that processes logging records received simultaneously from one or more client hosts. The object-oriented design of the server logging daemon is decomposed into several modular components (shown in Figure 7) that perform well-defined tasks. The application-specific components (`Logging`

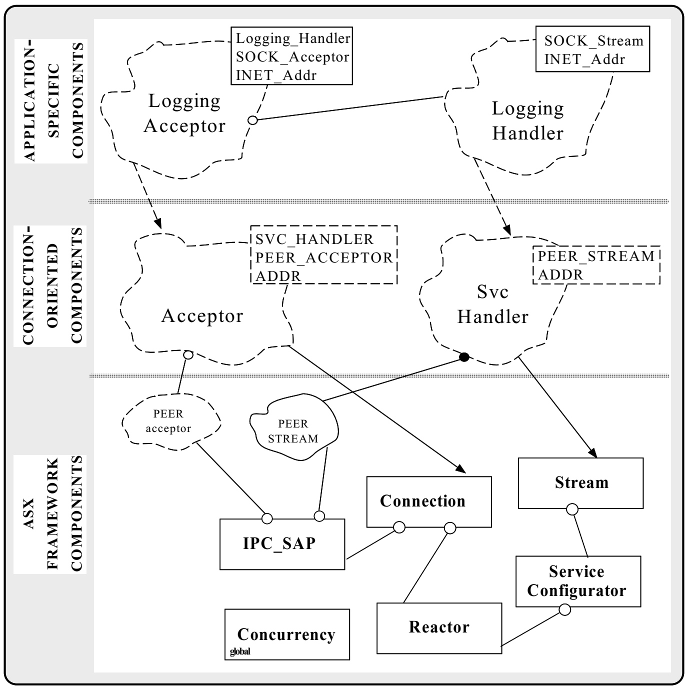

Figure 7: Class Components in the Server Logging Daemon

Acceptor and Logging IO) are responsible for processing logging records received from clients. The connection-oriented application components (Acceptor and Client IO) are responsible for accepting connection requests and data from clients. Finally, the application-independent Service Configurator framework components (such as the Reactor, Service Configurator, and IPC SAP class categories) are responsible for performing IPC, explicit dynamic linking, event demultiplexing, service dispatching, and concurrency control.

The Logging IO subclass is a parameterized type that is responsible for processing logging records sent to the server logging daemon from participating client hosts. Its communication mechanisms may be instantiated with either the SOCK SAP or TLI SAP C++ wrappers, as follows:

```
class Logging_IO : public Client_IO <
#if defined (MT_SAFE_SOCKETS)
                   SOCK_Stream,
#else
                   TLI_Stream,
#endif /* MT_SAFE_SOCKETS */
                   INET_Addr>
{
  /* ... */
};
```

The Logging IO class inherits from Event Handler (indirectly via Client IO) rather than Service Object since it is not dynamic linked into the server logging daemon.

When logging records arrive from the client host associated with a particular Logging IO object, the Reactor automatically dispatches the object's handle input method. This method formats and displays the records on one or more output devices (such as the printer, persistent storage, and/or console devices illustrated in Figure 6).

The Logging Acceptor subclass is also a parameterized type that is responsible for accepting connections from client hosts participating in the logging service:

```
class Logging_Acceptor :
  public Client_Acceptor<Logging_IO,
#if defined (MT_SAFE_SOCKETS)
                         SOCK_Acceptor,
#else
                         TLI_Acceptor,
#endif /* MT_SAFE_SOCKETS */
                         INET_Addr>
{
  /* ... */
};
```

Since the Logging Acceptor class inherits from Service Object (indirectly via its Acceptor base class), it may be dynamically linked into the server logging daemon and manipulated at run-time via the server logging daemon's svc.conf configuration file. Likewise, since Logging Acceptor indirectly inherits from the Event Handler interface, its handle_input method will be invoked automatically by the Reactor when connection requests arrive from clients. When a connection request arrives, the Logging Acceptor subclass allocates a Logging IO object and registers this object with the Reactor.

The object-oriented decomposition illustrated in Figure 7 significantly enhances the modularity, reusability, and configurability of the distributed logging facility by decoupling the functionality of connection establishment and logging record reception into two separate classes. This decoupling allows the Acceptor class to be reused for different types of connection-oriented services (*e.g.,* file transfer, remote login, video-on-demand, etc.). In particular, to provide completely different processing functionality, only the behavior of the Logging IO portion of the server logging daemon would need to be reimplemented. Furthermore, by using other object-oriented language features (such as templates and/or inheritance and dynamic binding) or design patterns (such as Factory Method or Abstract Factory [16]), it is possible to completely parameterize the type of service offered by an application [17]. Furthermore, the use of parameterized types decouples the reliance on a particular type IPC mechanism.

5.2 Configuring the Server Logging Daemon

The Service Configurator framework supports two types of configuration: *static* and *dynamic*. As discussed below, both types are used to configure the server logging daemon.

5.2.1 Dynamic Configuration

A dynamically configured daemon allows the insertion, modification, or removal of Service Objects at run-time.

The following entry in the `svc.conf` file is used to dynamically configure the logging service into the server logging daemon:

```
dynamic Logger Service_Object *
        ./Logger.so:_alloc() "-p 7001"
```

The <svc-name> token (`Logger`) specifies a service name that is used by the `Service Repository` to identify a `Service Object`. The `_alloc()` function is located in the shared object file indicated by the pathname `./Logger.so`. The `Service Configurator` framework locates and dynamically links this shared object file into the daemon's address space and then invokes the `_alloc()` function. The `_alloc()` function returns a `Service Object` pointer, which is inserted into the `Service Repository`. The service location also specifies the name of the service-specific subclass derived from `Service Object`. In this case, the `_alloc` function is used to dynamically allocate a new `Logging Acceptor` object. The remaining contents on the line (`"-p 7001"`) represent a service-specific set of configuration parameters. These parameters are passed to the `init` function of the service as `argc/argv`-style command-line arguments. The `init` method for the `Logging Acceptor` class interprets `"-p 7001"` as the port number where the server logging daemon listens for client connection requests.

5.2.2 Static Configuration

A statically configured service is one that is always available to a daemon when it first begins execution. For example, the `Service Manager` is a standard `Service Configurator` framework component that clients use to obtain a listing of active daemon services. The following entry in the `svc.conf` file is used to statically configure the `Service Manager` service into the server logging daemon during initialization:

```
static Service_Manager "-p 911"
```

In order for the `static` directive to work, the object code that implements the `Service Manager` service must be statically linked together with the main daemon driver executable program. In addition, the `Service Manager` object must be inserted into the `Service Repository` before dynamic configuration takes place (this is done automatically by the `Service Config` constructor). Due to these constraints, a statically configured service may not be reconfigured at run-time without being removed from the `Service Repository` first.

5.3 Server Logging Daemon Run-Time Activities

The main driver program for the server logger daemon is implemented by the following code:

```
int
main (int argc, char *argv[])
{
  Service_Config loggerd;

  /* Configure server logging daemon */
  if (loggerd.open (argc, argv) == -1)
    return -1;

  /* Perform logging service */
  for (;;)
    if (loggerd.run_event_loop () == -1
        && errno == EINTR)
      break;

  return 0;
}
```

Figure 8 depicts the run-time interaction between the various framework and service-specific objects that collaborate to provide the logging service. Daemon configuration is performed in the `Service Config::open` method. This method consults the following `svc.conf` file, which specifies the services to configure into the daemon:

```
static Service_Manager "-p 911"
dynamic Logger Service_Object *
        ./Logger.so:_alloc() "-p 7001"
```

Each of the service config entries in the `svc.conf` file is processed by loading the designated `Service Object` into the `Service Repository`, dynamically linking it if necessary, and registering the `Event Handler` portion of the service object handler with the `Reactor`.

When all the configuration activities have been completed, the main driver program shown above invokes the `Service Config::run_event_loop` method. This method enters an event loop that continuously calls the `Reactor::handle_events` service dispatch function. As shown in Figure 5, this dispatch function blocks awaiting the occurrence of events (such as connection requests or data from clients). As these events occur, the `Reactor` automatically dispatches previously-registered service-specific event handlers to perform the designated services.

The `Service Configurator` framework also responds to external events that trigger daemon reconfiguration at run-time. The dynamic configuration steps outlined in Section 5.2.1 are performed whenever an executing `Service Configurator` daemon receives a pre-designated external signal (such as the UNIX SIGHUP signal). Depending on the updated contents of the `svc.conf` file, services may be inserted, suspended, resumed, or removed from the daemon.

The `Service Configurator` framework's dynamic reconfiguration mechanisms enable developers to modify server logging daemon functionality or fine-tune performance without extensive redevelopment and reinstallation effort. For example, debugging a faulty implementation of the logging service simply requires the dynamic reinstallation of a functionally equivalent service that contains additional instrumentation to help isolate the source of erroneous behavior. Note that this reinstallation process may be performed without modifying, recompiling, relinking, or restarting the currently executing server logging daemon.

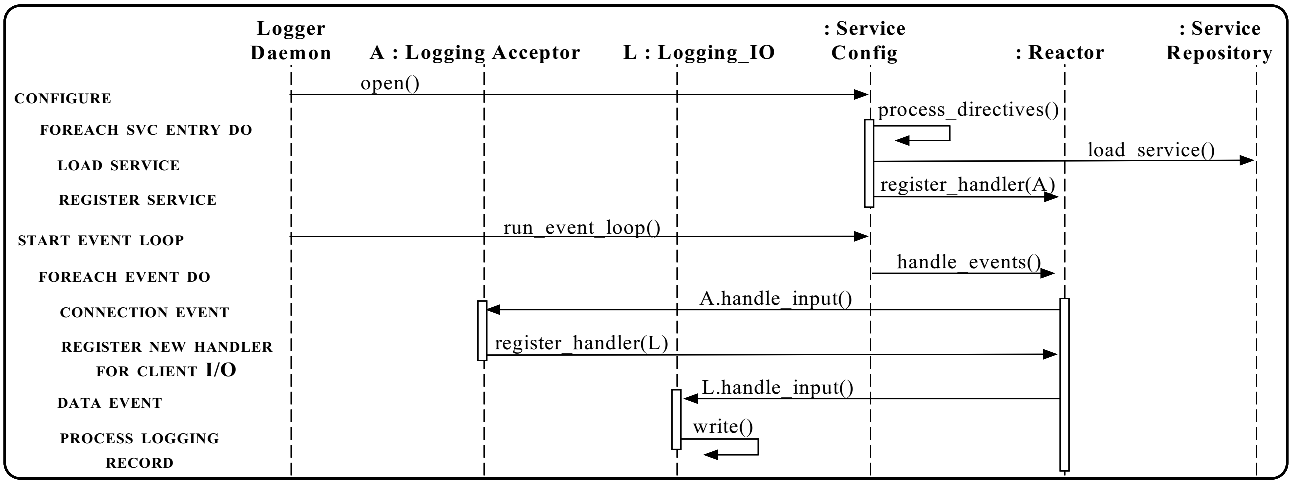

Figure 8: Interaction Diagram for the Server Logging Daemon

6 Concluding Remarks

The `Service Configurator` framework is an integrated collection of automated tools and reusable components that extend the functionality of conventional daemon management frameworks such as `inetd` and `listen`. The components in the `Service Configurator` framework implement basic mechanisms commonly used by network daemons. These components include C++ wrappers for local and remote IPC mechanisms [10]; frameworks for event demultiplexing and service dispatching [9]; tools for automating service configuration and reconfiguration; and C++ subclasses that encapsulate and enhance various dynamic linking and concurrency mechanisms. By allowing the `Service Configurator` framework to perform the lower-level IPC, event demultiplexing, and service dispatching mechanisms, developers are freed to concentrate on higher-level network daemon design and service functionality issues. In addition, the advanced OS mechanisms (such as explicit dynamic linking and multi-threading) employed by the `Service Configurator` framework allow developers to experiment with daemon design alternatives in a flexible manner to determine efficient daemon service configurations.

The general design principles underlying the `Service Configurator` framework may be characterized succinctly as (1) separating policies from mechanisms to enhance the reuse of common network daemon components, (2) decoupling the binding of processes and threads from the daemon services to improve flexibility and performance, and (3) utilizing inheritance, dynamic binding, and dynamic linking to improve extensibility. Together, these principles facilitate the development of network services that may be updated and extended without modifying, recompiling, relinking, or restarting existing daemons.

The techniques and tools described in this paper are currently being applied in several commercial and research environments. For example, the `Service Configurator` framework is being used by Ericsson Communications to configure and administer concurrent network services that implement a family of PBX management products on UNIX and Windows NT platforms [12]. The `Service Configurator` framework is also being used in the ADAPTIVE Communication Environment (ACE) [18]. ACE facilitates the development and experimentation with various aspects of communication subsystems (such as flexible process architectures for multi-processor-based communication protocol stacks [2] and adaptive protocol reconfiguration techniques [19]) and distributed applications (such as extensible frameworks for concurrent event demultiplexing [9]). The public domain ACE implementation of the `Service Configurator` framework is available via anonymous ftp from `ics.uci.edu` in the `gnu/C++_wrappers.tar.Z` file.

References

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

[2] D. C. Schmidt and T. Suda, "Measuring the Performance of Parallel Message-based Process Architectures," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (Boston, MA), IEEE, April 1995.

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

[4] C. R. Hofmeister and J. M. Purtilo, "Dynamic Reconfiguration of Distributed Programs," in *Proceedings of the 11^{th} International Conference on Distributed Computing Systems*, IEEE, 1991.

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

[6] F. Douglis and J. Ousterhout, "Process Migration in the Sprite Operating System," in *Proceedings of the 7^{th} International Conference on Distributed Computing Systems*, (Berlin, West Germany), pp. 18–25, IEEE, Sept. 1987.

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

[8] 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.

[9] 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.

[10] D. C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

[11] G. Booch, *Object Oriented Analysis and Design with Applications (2^{nd} Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

[12] D. C. Schmidt and T. Suda, "Experiences with an Object-Oriented Architecture for Developing Extensible Distributed System Management Software," in *Proceedings of the Conference on Global Communications (GLOBECOM)*, (San Francisco, CA), pp. 500–506, IEEE, November/December 1994.

[13] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol III: Client – Server Programming and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1992.

[14] E. Organick, *The Multics System – An Examination of Its Structure*. M.I.T. Press, 1972.

[15] 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.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

[17] D. C. Schmidt and T. Suda, "The ADAPTIVE Service eXecutive: an Object-Oriented Architecture for Configuring Concurrent Distributed Applications," in *Proceedings of the* 8^{th} *International Working Conference on Upper Layer Protocols, Architectures, and Applications*, Barcelona, Spain: North-Holland, June 1994.

[18] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment," *Journal of Concurrency: Practice and Experience*, vol. 5, pp. 269–286, June 1993.

[19] H. K. Huang, T. Suda, G. Takeuchi, and Y. Ogawa, "Protocol Reconfiguration: a Study of Error Handling Mechanisms," in *Proceedings of the* 2^{nd} *International Conference on Computer Communication Networks*, (San Diego, California), ISCA, June 1993.