An Object-Oriented Framework for Developing Network Server Daemons

Douglas C. Schmidt† and Paul Stephenson‡
schmidt@ics.uci.edu and ebupsn@ebu.ericsson.se
†Department of Information and Computer Science, University of California, Irvine, CA 92717
‡Ericsson Business Communications, Inc., Cypress, CA 90630

An earlier version of this paper appeared at the C++ World Conference in Dallas, Texas, October 1993.

Abstract

Developing distributed applications that utilize multi-processing and network services is a promising technique for increasing system performance, scalability, and cost effectiveness. However, designing and implementing efficient, robust, and extensible multi-threaded client/server applications is a complex and challenging task. The `Service Configurator` *(*`SVC-CON`*) framework described in this paper provides an object-oriented infrastructure that simplifies the development of dynamically configured, concurrent, multi-service network daemons. The framework integrates mechanisms for (1) local and remote interprocess communication, (2) I/O-based and timer-based event multiplexing, (3) explicit dynamic linking, and (4) multi-threading and multi-processing to aid the creation of network servers that may be updated and extended without modifying, recompiling, relinking, or restarting executing daemons.*

1 Introduction

This paper describes the architectural design and functionality of an object-oriented (OO) framework called the `Service Configurator` (`SVC-CON`). This framework provides a collection of reusable C++ components that simplify the construction of network server daemons by enhancing the modularity, extensibility, reusability, and portability of their interprocess communication (IPC), I/O-based and timer-based event multiplexing, service dispatching, and concurrency mechanisms. The OO techniques and tools described in this paper are currently being applied on a family of client/server applications as part of the Ericsson External Operating Systems (EOS) project. This project employs the `SVC-CON` framework to enhance the configuration flexibility and software component reuse of applications that monitor and manage MD110 [1] private-branch exchanges (PBXs) efficiently and portably across multiple hardware and software platforms.

In addition to describing the general structure and behavior of the `SVC-CON` framework, this paper also explores the process by which the framework's reusable C++ components "emerged" from careful analysis of the common objects and abstractions that exist in the domain of network servers. Since most textbooks and network programming reference guides present function-oriented models for designing network applications, it is not surprising that developers often decompose their server daemons according to functions rather than classes and objects. Therefore, the OO network server design perspective presented in this paper may appear somewhat "counter-intuitive" at first. However, our experience with the strategies and tactics underlying the `SVC-CON` framework offer compelling evidence that the long-term payoffs of applying object-oriented techniques to network programming significantly improves application modularity, extensibility, and component reuse.

This paper is organized in the following manner: Section 2 briefly summarizes the requirements and general architecture of the EOS PBX project; Section 3 describes the `SVC-CON` framework's primary features and reviews related work; Section 4 outlines the C++ design and implementation of the `SVC-CON` framework; Section 5 examines the structure of several EOS applications built using the `SVC-CON` framework; and Section 6 presents concluding remarks.

2 Overview of the Ericsson EOS PBX Project

Ericsson is developing a family of applications that monitor and manage MD110 PBXs. These applications enhance the functionality of a PBX (or cluster of PBXs) by providing end-users with directory management, call center management, and extension manager services. For example, the Directory Management application allows a PBX operator to profile incoming calls diverted from subscriber extensions, handle subscriber messages, and perform other general subscriber database queries (such as accessing visitor information and recording facility conference room location and availability). Likewise, the Call Center Management application allows the staff of a call center (such as an airline reservation center) to assess the performance and quality of the call center by providing real-time graphical displays of system resources such

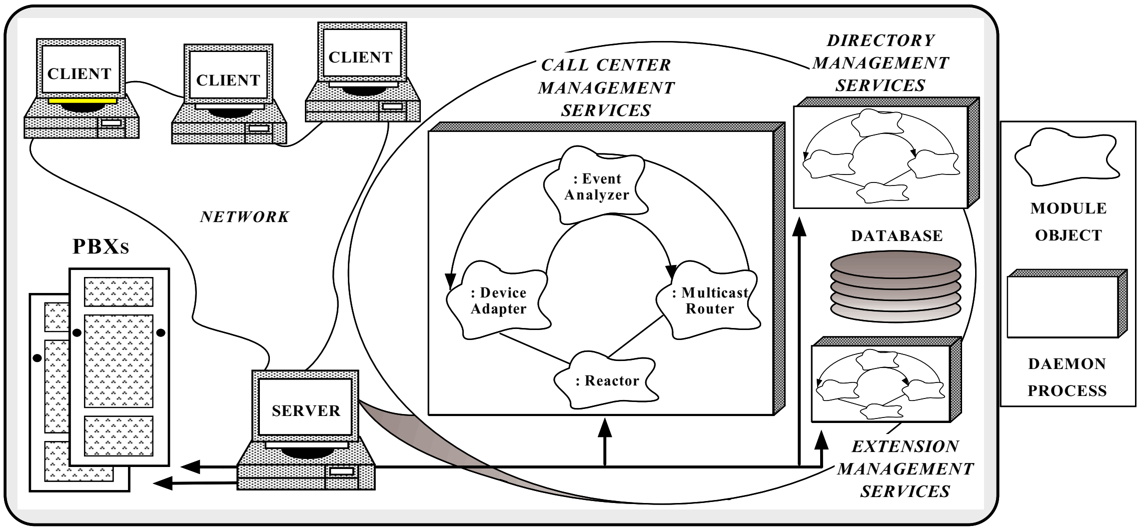

Figure 1: The EOS Client/Server Architecture

as agents and call queues. Finally, the Extension Management application allows a PBX administrator to view, add, modify, or delete PBX extensions interactively.

The architecture, design, and implementation of the Ericsson EOS applications are influenced by several general requirements that necessitate support for (1) *multiple application services*, (2) *platform independence*, (3) *configuration flexibility*, and (4) *efficient performance*. For example, multiple EOS application services must be available simultaneously to multiple remote users, potentially interacting with one or more PBXs. A client/server architecture (illustrated in Figure 1) was selected to support this distributed functionality in a scalable and relatively transparent manner. Each PBX is directly attached to a server host via a serial communication link, and one or more server daemons on the host supply services required by the client applications. The current EOS applications provide MD110 PBX communication services, database services, batch processing services, extension administration services, and asynchronous signal routing services (described in Section refusage).

Platform independence is another key requirement of the EOS project. Applications are targeted for various configurations of host and network platforms, including Windows NT, UNIX, Windows 3.1, and OS/2 running over TCP/IP and Novell IPX/SPX networks. The project relies heavily upon object-oriented design techniques and C++ language features to reduce the overall development effort and to improve software component reuse across platforms and among the family of related applications. In particular, the encapsulation and flexibility offered by C++ classes, abstract base classes, and parameterized types are used extensively to localize platform dependencies.

Two additional system requirements involve configuration flexibility and service performance. Since not all customers require every EOS feature, applications may be deployed with various combinations of services. It would be possible (although highly undesirable) to manually construct and deliver one or more client/server applications that are (1) customized for the services required by a customer and (2) optimized for the level of concurrency available on the host platform [2]. However, such a "statically configured" system would require the application service combination, client/server division of labor, and host platform to be completely fixed during product deployment. Our experience with earlier-generation EOS applications suggests that even if this information is available at the time of deployment, it will certainly change in the future, often upon short notice. Therefore, to maximize installation flexibility and to take advantage of available multi-processing capabilities, the EOS family of applications utilize `SVC-CON` framework features that defer decisions regarding both (1) the set of available services and (2) the partitioning of these services onto processes and/or threads until as late as possible – *e.g.,* initial server startup-time or even during run-time. As described below, the `SVC-CON` framework provides several foundation classes that support the deferred binding of services onto processes and/or threads. Other object-oriented components used by the EOS applications are described elsewhere [3, 4, 5].

3 The Service Configurator Framework

The `SVC-CON` provides an object-oriented framework that simplifies the development, configuration, and reconfiguration of concurrent, multi-service network daemons. A daemon is a statically or dynamically configured process that executes in the "background" (*i.e.,* disassociated from any controlling terminal) on a host computer. The fundamental unit of configuration in the `SVC-CON` framework is the *service.* Network daemons provide communication-related services that resolve distributed name lookups, access network file systems, manage routing tables, and perform other remote services such as printing, login, and file transfer. In the EOS system, network daemons orchestrate the server-side directory management, call center management, and extension administration services.

Depending on configuration policies specified during system installation, the EOS applications run as one or more multi-service network daemons that simultaneously support multiple remote services via one or more OS processes and/or threads. Explicit dynamic linking may be used to dynamically (re)configure (*i.e.,* insert and remove) these services from a network daemon at run-time. Deferring the binding of services to processes and threads until run-time increases the flexibility, extensibility, and performance of network daemons. Moreover, in certain cases, daemons that execute within the `SVC-CON` framework may reconfigure their services *without* being terminated and restarted. In addition, the concurrency level of a network daemon may be fine-tuned during installation or run-time to match client application demands and available OS multi-processing capabilities more efficiently.

The `SVC-CON` framework integrates C++ language features (such as inheritance, dynamic binding, and parameterized types) and advanced OS mechanisms (such as the threads and explicit dynamic linking facilities available in SVR4 UNIX [6] and Windows NT [7]) to facilitate the development of network clients and servers that may be updated and extended *without* modifying, recompiling, relinking, or restarting the running daemons. In addition, it provides a suite of reusable components that extend the functionality of conventional port monitoring and service dispatching tools such as the UNIX System V Release 4 (SVR4) `listen` facility [8] and BSD `inetd` superserver [9].

The framework's components also reduce the effort required to develop network daemons. For instance, the `SVC-CON` framework's components simplify development by consolidating common server activities (such as I/O-based and timer-based event multiplexing, service dispatching, subroutine tracing and status logging, daemonization, service directory functionality, and various process, thread, and linking strategies) into reusable C++ foundation classes. These classes include the IPC_SAP object-oriented transport interface [3] and the `Reactor` I/O-based and timer-based event multiplexing class library [4, 5] (both of which enhance application robustness by accessing OS local and remote IPC mechanisms via type-secure interfaces, rather than the weakly-typed "descriptor-based" underlying system call interfaces).

3.1 Conventional Port Monitoring and Service Dispatching Frameworks

This section describes several conventional frameworks for developing, configuring, and reconfiguring network daemons. Section 3.2 compares the features of these frameworks with those of the `SVC-CON` framework.

3.1.1 Single-Service Daemons

In early versions of UNIX, standard network services such as remote file transfer (`ftp`) and remote login (`telnet` and `rlogin`) ran as single-service daemons that were initiated at OS boot-time [10]. The services offered by these daemons were configured *statically* at compile-time and/or static link-time. As illustrated in Figure 2 (1), a separate program was typically written to implement each service. Each service ran in a separate process, though a master daemon might spawn one or more slave processes to perform certain long-duration services externally in a separate address space on behalf of its clients.

As the number of system daemons grew steadily, however, this "statically configured, single-service per-process" design approach revealed several significant limitations. First, OS process management overhead increased since each single-service daemon consumed a process table slot, even though it was often idle. Second, each daemon redundantly reimplemented the same daemonization and transport endpoint initialization code. Third, the flexibility and extensibility of statically configured daemons was limited since adding or deleting services required modifying, recompiling, and relinking existing code. Moreover, running daemons had to be terminated and restarted explicitly after making any changes. Finally, administering and monitoring the security and performance aspects of each daemon was handled in an *ad hoc* manner [8].

3.1.2 Multi-Service Port Monitor and Service Dispatcher Frameworks

Multi-service port monitor and service dispatcher frameworks were devised to alleviate the limitations with single-service daemons described above. Two widely available frameworks are the Internet superserver `inetd` (which originated with BSD UNIX [9]) and the `listen` port monitor facility (distributed as part of the Service Access Facility with SVR4 UNIX [8]). `Inetd` and `listen` integrate many single-service daemons into one administrative framework in order to (1) reduce unnecessary process overhead by spawning daemons "on-demand," (2) simplify daemon development by automatically performing daemonization and transport endpoint initialization, (3) allow external services to be changed without modifying source code or terminating an executing daemon process, and (4) consolidate the administration of network services via a standard set of configuration files and command-line utilities.

Figure 2 (2) illustrates the general structure of daemon dispatcher tools such as `inetd` and `listen`. Both tools use Internet-domain port numbers to demultiplex client requests and dispatch them to either (1) statically named internal services or (2) statically and/or dynamically named external services (daemon-related terminology is defined more thoroughly in a companion paper available in the 1993 C++ World conference proceedings [11]). For example, the `inetd` superserver operates in the following manner:

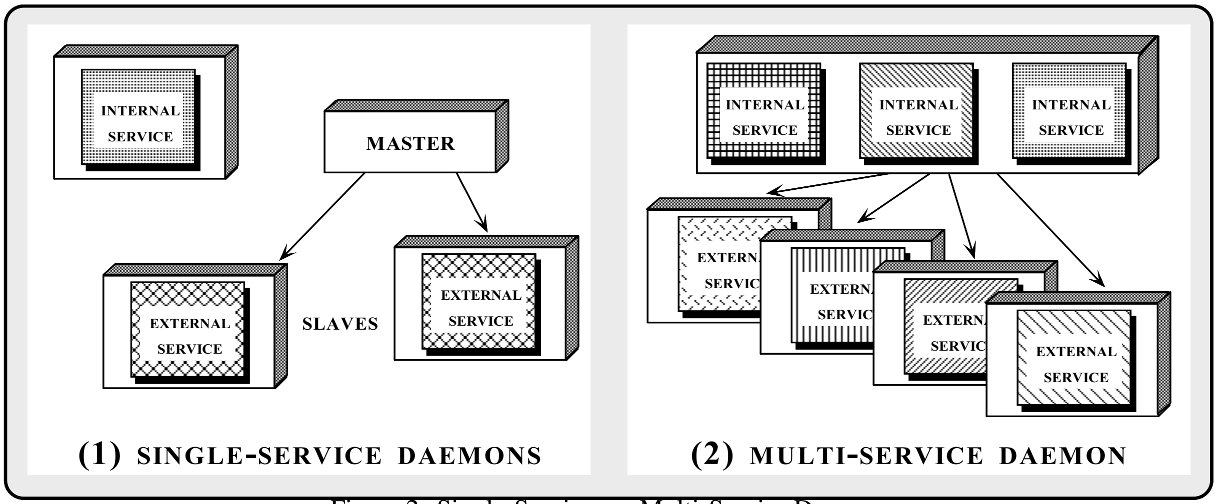

Figure 2: Single-Service vs. Multi-Service Daemons

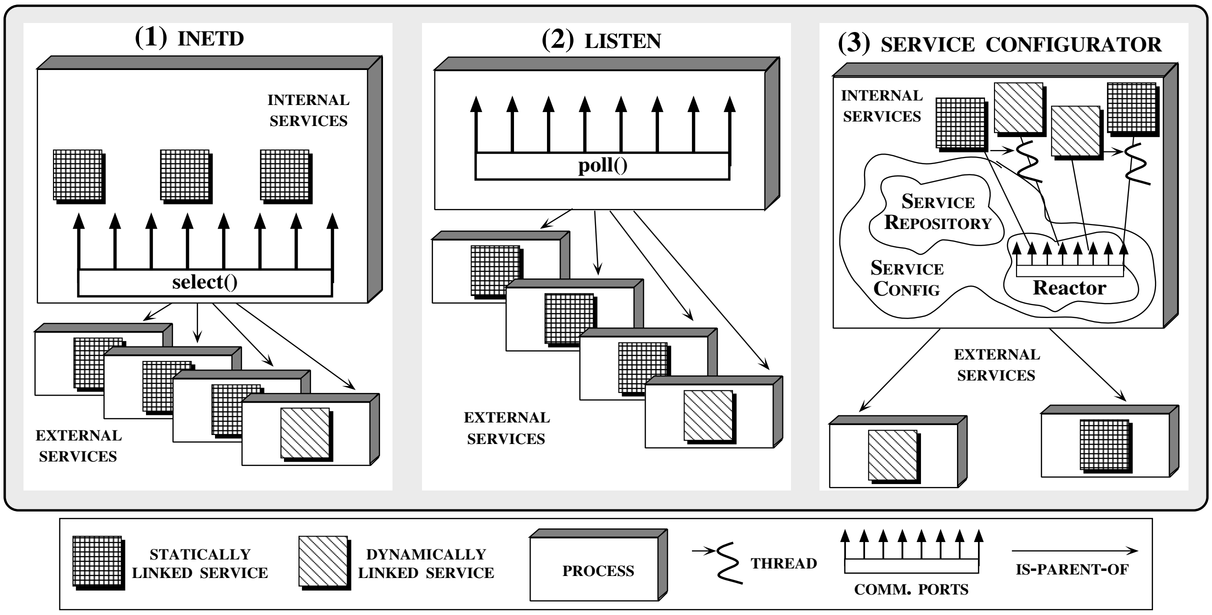

Figure 3: Alternative Port Monitoring and Daemon/Service Dispatching Frameworks

1. When invoked at OS boot-time, inetd reads service configuration information stored in the inetd.conf file.

2. For each service in the configuration file, inetd performs the "socket/bind/listen" socket initialization sequence to register the well-known port number of the service with the OS.

3. Inetd then enters a select loop that waits for one or more client connection requests or datagrams to arrive at the port of any registered services. Datagrams arriving for statically named internal services (such as echo and daytime) are performed internally by the master inetd process. Connection requests arriving for external services (such as ftp or rlogin) are handled by accept'ing the connection, fork'ing a new process, and exec'ing the appropriate executable program to perform the service on behalf of the client.

Although inetd's internal services (such as echo and daytime) are fixed at static link-time, the master inetd daemon permits dynamic reconfiguration of its external services (such as ftp or telnet). For instance, when sent the SIGHUP signal, the inetd daemon re-reads its inetd.conf file and performs the socket/bind/listen sequence for all services listed in that file. However, since inetd does not support dynamic reconfiguration of internal services, any newly listed services must still be processed by spawning slave daemons via fork and exec. Therefore, although inetd and listen[1] overcome many limitations with single-service daemons, they still possess several shortcomings that are addressed by the SVC-CON framework described below.

Another network service management facility that recently become available is the Service Control Manager (SCM) distributed with Windows NT [7]. Unlike inetd and listen, SCM is not a port monitor *i.e.*, it does not provide built-in support for listening to a set of I/O ports and dispatching server processes "on-demand" when client requests arrive. Instead, it provides an RPC-based interface that allows the master SCM process to automatically initiate and control (*i.e.*, pause, resume, terminate, etc.) administrator-installed services (such as ftp and telnet) that typically run as separate threads within either a single-service or a multi-service daemon process. Each installed service is individually responsible for configuring the service and monitoring any communication endpoints (which may be more general than I/O ports, *e.g.,* named pipes). Note that the SVC-CON framework may be utilized within the SCM environment to provide additional support for dynamic daemon configuration, port monitoring, and service dispatching.

[1]The SVR4 listen port monitoring facility is similar to inetd, though it only supports connection-oriented protocols accessed via TLI and STREAMS, and does not provide internal services. However, unlike inetd, listen supports "standing-daemons" by passing initialized file descriptors via STREAM pipes from the listen process to a previously-registered standing-daemon.

3.2 Primary Features of the Service Configurator

This subsection outlines the primary features offered by the SVC-CON framework and compares these features with those provided by inetd and listen. Figure 3 illustrates the major architectural features of the three frameworks. In general, the features of the SVC-CON framework are designed to (1) increase configuration flexibility and daemon extensibility, (2) improve performance, and (3) reduce development effort for concurrent, multi-service network daemons.

3.2.1 Increase Flexibility and Extensibility

The SVC-CON framework enhances configuration flexibility and network daemon extensibility by decoupling and deferring the point at which services are bound to OS processes and/or threads. In particular, services may be configured into the SVC-CON framework either (1) *statically* (at compile-time or link-time) or (2) *dynamically* (when a daemon first begins executing or even while it is running). Moreover, the choice between these two alternatives may be deferred. For example, services may be partitioned and/or migrated between clients and servers during or after installation, thereby enabling a flexible division of labor on the placement of services within a distributed application.

The SVC-CON framework provides an object-oriented interface to OS explicit dynamic linking features. As described in Section 4.1, this interface facilitates the dynamic configuration and reconfiguration of network daemon services, often without requiring the modification, recompilation, or relinking of existing code. Dynamic linking also provides an opportunity to reconfigure services without terminating and restarting a daemon. Inetd and listen, on the other hand, provide a more limited form of dynamic configuration that does not support reconfiguration of internal services at run-time. Instead, adding new internal services requires modifying, recompiling, relinking, and restarting inetd (listen does not support internal services).

3.2.2 Improve Performance

By deferring the binding of services to processes and threads, applications may postpone certain decisions until run-time, when additional information is available to guide the selection of more efficient daemon configurations. For instance, customizing or reconfiguring daemons during or after startup-time helps to account for factors such as (1) the class of service required by applications (*e.g.,* reliable vs. unreliable and real-time vs. non-real-time), (2) the type of traffic generated by applications (*e.g.,* bursty vs. continuous and short-duration vs. long-duration), (3) the class of protocol that implements the application services (*e.g.,* connection-oriented vs. connectionless vs. request-response), (4) certain static and dynamic characteristics of the hardware and operating system architecture (*e.g.,* message passing vs. shared memory, process and thread management overhead, number of CPUs, and current end-system load), and (5) the underlying network environment (*e.g.,* high-speed vs. low-speed and large frame size vs. small frame size) [12].

The SVC-CON framework automates many of the steps required to (re)configure network daemons and helps developers navigate through the diverse set of factors that affect the configuration of network daemons. For example, a "concurrent/multi-service" daemon configuration may be efficient for an OS that effectively utilizes multiple CPUs. In this case, each application service may be mapped onto a separate process or thread. On the other hand, an "iterative/single-service" configuration may be more suitable for certain combinations of OS platform and application service characteristics. For instance, on a uni-processor platform, daemon efficiency may be improved by executing short-duration, request/response services in a single-threaded process, due to the reduction in scheduling and context switching overhead [13].

The SVC-CON framework also employs OS mechanisms such as dynamic linking, multi-threading, and multi-processing to improve performance. Explicit dynamic linking and threads support the (re)configuration of concurrent internal services *without* spawning a new OS process. This helps improve the performance of multi-service daemons that perform short-duration, request-response services. Conversely, `inetd` and `listen` spawn a new process to achieve similar dynamic service invocation functionality. However, this invocation technique may be too costly for short-duration services, due to the overhead of `fork` and `exec`.

Dynamic linking also helps reduce overall host memory utilization, which may improve aggregate end-system performance [14]. For example, dynamically linked services are not fully loaded, resolved, or relocated into the address space of an executing daemon until they are first referenced, which often reduces a daemon's consumption of primary and secondary storage resources. Moreover, to further reduce run-time memory utilization, a dynamically linked service may be shared between multiple network daemons running simultaneously [6]. In addition, services may be dynamically unlinked from daemons when they are no longer required, thereby releasing resources for subsequent use by other applications and daemon services.

3.2.3 Reduce Development Effort via Reusable Components

The SVC-CON framework provides a collection of reusable components that implement the following common foundation services used by network daemons and distributed applications:

• **Event Multiplexing and Service Dispatching:** Network server daemons often multiplex different types of I/O events sent or received simultaneously from one or more clients on multiple communication ports. The SVC-CON framework provides port multiplexing and service dispatching functionality via a C++ class library called the `Reactor` [4, 5]. The `Reactor` provides a set of extensible, reusable, and type-secure C++ classes that portably encapsulate and enhance the `select` and `poll` I/O multiplexing facilities. The `Reactor` integrates the multiplexing of synchronous and asynchronous I/O-based events together with timer-based events. When events occur, the `Reactor` automatically dispatches "call-back" member functions of previously registered objects to perform application-specified services. The `Reactor` enables developers to concentrate on higher-level daemon design and functionality issues, rather than reimplementing the same lower-level event detection and dispatching code for each new network daemon.

• **Automatic Service Configuration:** To help automate many daemon configuration steps, the SVC-CON provides a standard model for installing application services into network daemons. This configuration model leverages off notations and tools that (1) identify the service(s) to activate, (2) statically or dynamically instantiate, link, and initialize C++ object(s) that implement the service(s), (3) notify the underlying OS transport provider to bind communication ports and network addresses for the object(s), (4) register the object(s) with an instance of the `Reactor`, and (5) arrange to run the service via one or more processes and/or threads. The SVC-CON framework's configuration model is flexible enough to support dynamic and static configuration, as well as hybrid approaches that provide both configuration methods simultaneously. Section 4.3.1 examines the SVC-CON framework's service configuration model in detail.

• **Process and Thread Generation Strategies:** Several SVC-CON framework facilities implement *on-demand*, *eager*, and *lazy* process and thread generation strategies. In general, these strategies help to further decouple the services offered by network daemons from the OS processes and threads that execute the services. In particular, they enable daemons to adaptively tune their concurrency levels to match client demands and available OS parallelism. For example, on-demand generation spawns a new process or thread in response to the arrival of client requests. Eager generation pre-spawns one or more OS processes or threads at daemon creation time to reduce service startup overhead and improve response time. Conversely, lazy allocation 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. Only if the timer expires is a new process spawned to continue processing the service concurrently [13].

• **Distributed Logging:** Network daemons are often difficult to develop and debug since diagnostic output appears in different windows and/or on remote host systems. To simplify network daemon debugging, the SVC-CON framework supports a distributed logging facility (described and implemented in [4, 5]). This logging facility coalesces diagnostic output (potentially sent from multiple daemons on multiple hosts) at a designated location in a local and/or wide area network. The distributed logging facility utilizes several levels of "many-to-one" multiplexing. For example, applications send logging records via `named pipes` or `message queues` to a client logging daemon running on their local host machine. Each client daemon timestamps and forwards the logging records via TCP/IP connections to a remote server logging daemon running on a designated server host. This concurrent server daemon processes the logging records and displays them on one or more output devices (such as printers, persistent storage devices, and/or monitoring consoles).

• **Function-Call Tracing:** To further aid debugging, the `SVC-CON` framework provides a function-call `Trace` class that interoperates with the distributed logging facility. The `Trace` class enables developers to monitor the calling sequence of any or all stand-alone subroutines or member functions at run-time. A simple regular-expression-based filter tool automatically instruments application source code with `Trace` object definitions. At run-time, output from the constructor and destructor of `Trace` objects visually indicates the function calling sequence. This output is indented appropriately to illustrate the current call-chain nesting level as functions are entered and exited. The creation and termination semantics of C++ simplify function-call tracing since `Trace` object destructors are automatically invoked regardless of the point that the function returns. In addition, the `SVC-CON` framework enables tracing to be toggled on or off via signals or other asynchronous notification events generated by a user.

• **Daemonization:** The daemonization utility provides network servers with robust capabilities to execute and survive as daemon processes executing "in the background." These daemonized processes do not automatically receive events generated from a terminal nor do they receive hangup indications if/when their parent process exits. As described in [10], daemonization under UNIX 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.

In general, component reuse in the `SVC-CON` framework is enhanced by (1) accessing framework services via extensible object-oriented interfaces written in C++ and (2) separating higher-level application processing *policies* that perform client requests from lower-level daemon *mechanisms* (such as event demultiplexing and dispatching, logging and tracing, daemonization, and various process and thread generation strategies). In contrast, both `inetd` and `listen` allow only course-grain "black-box" reuse of their general service dispatching facilities, without encouraging more fine-grain reuse of their internal components. For example, the standard BSD `inetd` implementation is written in C and is characterized by global variables, lack of information hiding, and a functional decomposition that complicates direct reuse of its internal components.

4 The Server_Daemon Design and Implementation

This section outlines the object-oriented design and implementation of the `SVC-CON` framework's primary components and describes the sequence of steps performed to develop and configure a daemon's services statically and/or dynamically. In addition to examining the interfaces and general functionality of the framework's components, the strategic decisions that yielded the decomposition illustrated in Figure 4 are also discussed.[2]

The `SVC-CON` framework was developed using several object-oriented design techniques and C++ language features. Domain analysis on the typical attributes and operations performed by network daemons yielded the following class components in the `SVC-CON` framework:

- The `Service_Object` inheritance hierarchy (Figure 4 (1)) – this hierarchy ensures that developers specify the information necessary to automate dynamic linking, initialization, port multiplexing, and dispatching of an application service at run-time.
- The `Service_Repository` class (Figure 4 (2)) – this class provides an object manager that coordinates individual and/or collective access to active services in a daemon.
- The `Service_Config` class (Figure 4 (3)) – this "framework integration class" orchestrates the configuration and reconfiguration of statically/dynamically configured, iterative/concurrent, single/multi-service network daemons.

Though difficult to quantify precisely, it appears rather unlikely that a functional design approach would have yielded a set of reusable components that offer such a high degree of modularity and extensibility to distributed applications.

4.1 The Service_Object Class

The `Service_Object` class (illustrated in Figure 4 (1)) forms one part of a multi-level hierarchy of types related by inheritance. This hierarchy decouples the application-specific portions of a network service from the underlying mechanisms provided by the framework that link, register, and dispatch the service at run-time. This separation of concerns minimizes the effort required to add and/or remove of services to and/or from a network daemon. Each class in this hierarchy performs a set of well-delineated tasks for application developers, as described below:

• **The Event_Handler Abstract Base Class:** The root of the inheritance hierarchy is defined by the `Event_Handler`

[2] These components and their relationships are illustrated via Booch notation [15]. Dashed clouds indicate classes and directed edges indicate inheritance relationships between these classes. Solid clouds indicate one or more class objects and undirected edges indicate composition relationships between these objects (*cf.* Figure 6).

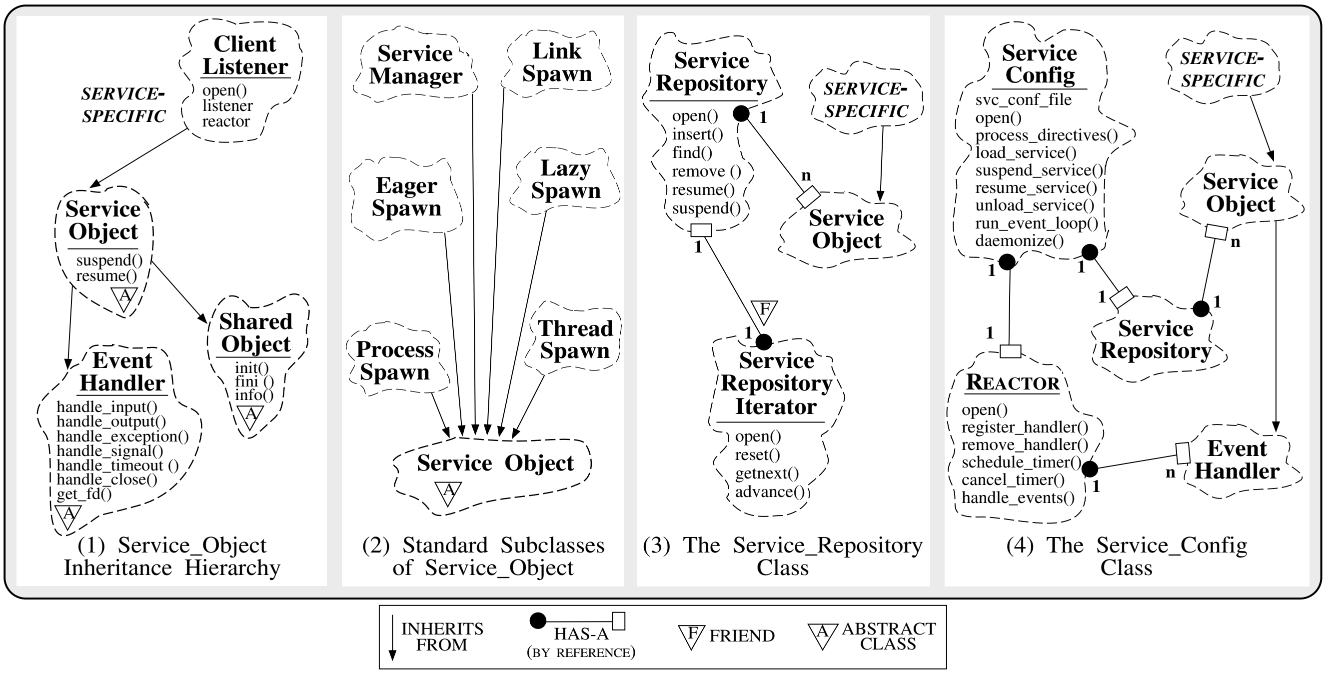

Figure 4: The Server_Daemon Class Components and their Relationships

abstract base class. This base class supplies an event dispatching interface that consists of virtual member functions for (1) synchronous input, output, and exception events and (2) timer-based events. In the SVC-CON framework, application-specific subclasses indirectly inherit and refine this functionality through the Service_Object derived class. This derivation process results in composite objects that are subsequently registered with an instance of the Reactor [5]. The Reactor then extracts the underlying I/O descriptor from the Event_Handler portion of a composite object and passes it along with other descriptors to select or poll I/O demultiplexing system calls. When events associated with a registered object occur at run-time, the Reactor automatically dispatches the appropriate member function(s) of the object, which then perform application-specific services.

• **The Service_Object Abstract Derived Class:** The Service_Object class exports an abstract interface consisting of three *pure virtual functions* [16] that impose a "contract" between the general-purpose foundation classes provided by the SVC-CON and application-specific services utilizing these classes. The use of pure virtual functions ensures that an application service supplies the SVC-CON framework with the appropriate information necessary to link, initialize, identify, and unlink a service at run-time.

During development, application-specific subclasses must implement the init function to perform initialization operations when an instance of a composite Service_Object first comes into existance. Likewise, during service initialization, init serves as the "entry-point" to an application service, (*i.e.,* it is passed a pair of "argc/argv"-style parameters that are similar to those passed to the main function of a stand-alone executable program). The fini member function is called automatically to perform any necessary termination operations when a Service_Object is unlinked and removed from a daemon at run-time. The info member function returns a humanly-readable string that documents the functionality and addressing information of a service.

• **Application-Specific Concrete Derived Subclasses:** The Service_Object and Event_Handler are both "abstract" classes since they contain pure virtual functions. Therefore, developers must derive concrete subclasses (such as the Signal_Router subclass described in Section 5) that define the functions inherited from the abstract base classes and implement the application-specific service functionality. Application-specific classes are also responsible for supplying the necessary "encode-state" and "decode-state" conversion functions necessary to enable service migration [17].

4.2 The Service_Repository Class

The SVC-CON framework supports the configuration of both single-service and multi-service network daemons. To simplify administration, it is often necessary to individually and/or collectively control and coordinate the Service_Objects that comprise a daemon's services. The Service_Repository is an object manager that coordinates local and remote queries and updates involving the services offered by a SVC-CON-based application. 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. As shown in Figure 4 (2), each entry in the Service_Repository

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

contains a pointer to the `Service_Object` portion of an application-specific C++ derived class.

Figure 4 (2) also depicts the member functions that load, enable, disable, reenable, or remove `Service_Objects` from a daemon statically and/or dynamically. For dynamically linked `Service_Objects`, the repository also stores a handle to the underlying shared object. This handle is used to unlink and unload a `Service_Object` from a running daemon when its services are no longer required. In addition, an iterator class is provided to visit every `Service_Object` in the repository without compromising data encapsulation. For example, a complete listing of all currently enabled daemon services may be obtained by calling the `info` virtual function on each enabled entry in the `Service_Repository`. This iterator feature is used by the standard `Service_Directory` service described in Section 4.3.2 below.

4.3 The Service_Config Class

As illustrated in Figure 4 (3), the `Service_Config` class is the central abstraction in the `SVC-CON` framework. This class integrates the other foundation services (such as the `Service_Repository` and the `Reactor`) to facilitate the static and/or dynamic configuration of concurrent, multi-service network daemons. The following subsections outline the configuration and run-time activities performed by `Service_Config` class functions.

4.3.1 Server Daemon Configuration Activities

This subsection briefly describes the standard daemon configuration process supported by the `SVC-CON` framework. Alternative mechanisms for statically or dynamically inserting and/or removing services from a daemon are also examined. In addition, the steps used to implement the various mechanisms are also outlined.

• **The svc.conf File:** The `svc.conf` file is the heart of the `SVC-CON` configuration and reconfiguration process. Each instance of the `Service_Config` class may be associated with a distinct `svc.conf` configuration file that characterizes essential attributes of the services offered by a daemon. This file simplifies both service administration and daemon development. Service administration is simplified by consolidating service installation parameters into a single location. Likewise, daemon development is simplified by decoupling the configuration and reconfiguration mechanisms provided by the framework from the policies specified in the `svc.conf` file. The `svc.conf` file is consulted when a new instance of a daemon is first started. This file is also when a running daemon receives either a pre-designated external signal or IPC request from a remote management facility.

Figure 5 uses extended-Backus/Naur Format (EBNF) to describe the primary syntactical elements of *service config entries* used in a `svc.conf` file. Each line in the file begins with a *service config directive* that indicates the configuration activity to perform (Table 1 summarizes the valid service config directives). For example, the **dynamic** directive is followed by a *service identifier*:

```
dynamic /svcs/Logger.so:_alloc() Logger -p 7001
```

`/svcs/Logger.so:_alloc()` is a service identifier that indicates the pathname of a shared object file to dynamically link (`/svcs/logger`), as well as the name of the associated `Service_Object` (or in this case, a function called `_alloc` that dynamically allocates a `Service_Object`). The remaining contents on the line (`Logger -p 7001`) represent a service-specific set of configuration parameters. These parameters are passed to the `init` function of the service as `argv`-style command-line arguments. The `argv[0]` argument (`Logger`) specifies the service name that will identify the corresponding `Service_Object` within the `Service_Repository`.

Figure 6 illustrates a complete `svc.conf` file used to configure EOS project services (described further in Section 5). This figure also indicates how services may be selectively configured either statically (*e.g.,* `Svc_Directory`, `MML_Svc`, and `GICI_Svc`) or dynamically (*e.g.,* `PBX_Svc`, `XAD_Svc`, and `Client_Muxer`) in the same daemon, depending on the format of the configuration file.

• **Static Configuration:** In a statically configured daemon, all `Service_Objects` are completely specified at daemon installation-time. This limits a daemon to a specific, non-reconfigurable set of services, which may be necessary for secure daemons that contain only "trusted" services. Implementing a daemon composed solely of statically configured services requires developers to derive a subclass from the `Service_Config` base class. This subclass then becomes responsible for pre-initializing the `Service_Repository` to contain only trusted services. The derived class may

```
<service-config-entry>     ::= <service-config-directive>
                               [ <service-identifier> ]
                               SERVICE_NAME
                               [ <optional-parameters> ]
<service-config-directive> ::= DYNAMIC | STATIC | REMOVE | DISABLE | ENABLE
<service-identifier>       ::= SHARED_OBJECT ':' [ <service_initializer> ]
<service_initializer>      ::= OBJECT_NAME | FUNCTION_NAME '(' ')'
```

Figure 5: EBNF Format for a Service Config Entry

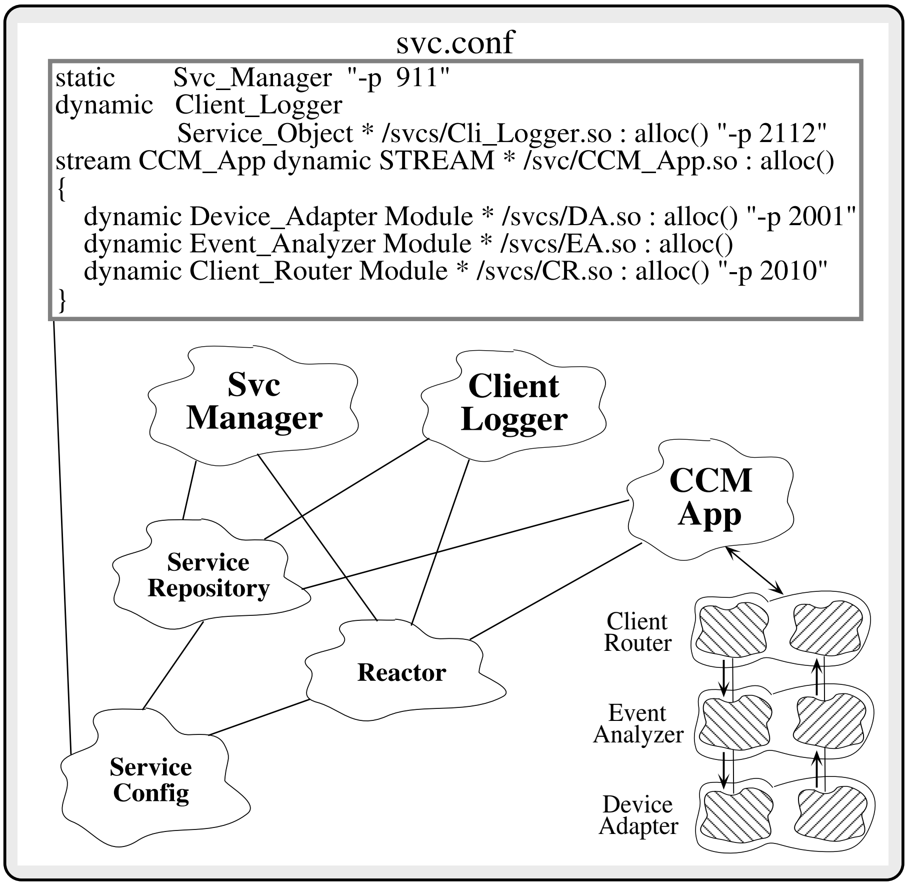

Figure 6: Object Components in the Server_Daemon Framework

also redefine one or more of the virtual functions inherited from the base class so that only the pre-initialized `Service_Repository` is searched to locate a service (the default behavior is to search the symbol table of the dynamically linked shared object to locate the appropriate `Service_Object`). These modifications ensure that any **dynamic** service configuration directives in the `svc.conf` file will be properly ignored. As a further precaution to prevent the use of non-trusted services, derived classes should be configured to use only `Service_Objects` that are fully resolved at static link-time (*i.e.,* no implicit dynamically linked services should be allowed). Clearly, static configuration trades off flexibility for increased security.

• **Dynamic Configuration:** A dynamically configured daemon permits the insertion, modification, or removal of `Service_Objects` during the initial daemon invocation sequence. This behavior is specified by placing the **dynamic** service configuration directive before the service identifier in the `svc.conf` file. Dynamic configuration requires the underlying operating system to support explicit dynamic linking (SVR4 and OSF/1 UNIX and Windows NT all support this feature). In general, dynamic linking simplifies the configuration of network daemons by avoiding the modification, recompilation, relinking, or restarting of running daemon code. Moreover, if every daemon service is dynamically configured, the `svc.conf` file contains all the information necessary to populate the `Service_Repository`. This makes it possible to extend a daemon's services "in the field" without requiring an administrator to have access to the original source code.

Dynamic configuration also helps reduce overall end-system memory utilization by creating instantiations of `Service_Object` derived classes as dynamically linked shared objects. These Service_Objects will not be loaded into a daemon unless the svc.conf file indicates they are actually required. Naturally, developers must carefully consider the subtle trade-offs between flexibility and time/space efficiency when choosing between dynamic and static linking ([6] enumerates many of the trade-offs).

• **Dynamic Reconfiguration:** Dynamic reconfiguration allows the modification of services offered by a network daemon without actually terminating an executing instance of the daemon [18]. Reconfiguration may be triggered by external events that are generated both locally and/or remotely. For example, when an executing SVC-CON receives a pre-designated signal (*e.g.,* SIGHUP) that was generated on the local host machine, the configuration steps are performed again for any services added to or removed from the svc.conf file. Likewise, the Service_Directory service described below in Section 4.3.2 may also be used to initiate reconfiguration across a network via a remote daemon service management facility.

The development and administrative steps used to add a service to a SVC-CON are straight-forward. First, a developer writes a new service that inherits from the interface offered by the Service_Object/Event_Handler class hierarchy. In general, services may be arbitrarily complex, though many standard network services (such as ftp and telnet) do not require the retention of persistent state information between consecutive service invocations (these "stateless" services are often simpler to configure and reconfigure reliably). Next, an object of the derived class is instantiated, linked into the daemon, and inserted into the Service_Repository (this sequence of steps may be performed by the SVC-CON either *statically* at compile-time or *dynamically* at run-time). The svc.conf file is then updated manually or via an administrative tool to contain an additional entry that identifies the location of the new service and specifies its command-line configuration parameters, which indicate the arguments to pass the init function of the specified Service_Object and whether to use static or dynamic linking. At this point, the developer either starts, restarts, or sends a pre-designated signal to a SVC-CON to initiate configuration or reconfiguration.

The configuration steps performed internally by a network daemon are initiated when the application calls the open function of the SVC-CON class. This function parses command-line arguments to enable daemon options, opens a channel to the distributed logging service, invokes the daemonization code, dynamically creates an instance of the Reactor, and calls the process_directives function to process the daemon's configuration file.

The process_directives function processes the svc.conf file line-by-line. It first converts each line into an argv-style vector of arguments. Then it carries out the specified service configuration directive. For example, if the **dynamic** directive appears, the load_service function is called to (1) dynamically link the appropriate Service_Object into the address space of the daemon and (2) insert the address of the object into the Service_Repository. Likewise, if the **remove** directive is specified, the unload_service function is called to gracefully close down and delete the service from the Service_Repository. If the service was dynamically linked, the shared object file is unloaded from the executable daemon.

The enable_service function is invoked if the configuration directive is **static** or **enable**. This function queries the Service_Repository to determine the appropriate instance of the statically or dynamically configured Service_Object that is currently bound to the associated service name. After the instance is located, its init function is called and the remaining argv arguments are passed as a parameter. If init returns the REGISTER_SVC value, the Event_Handler portion of the newly initialized Service_Object is registered with the Reactor automatically.

If the **disable** directive appears at the beginning of a line, the disable_service function is invoked. This function temporarily restricts access to the named service *without* actually unlinking and fully removing it from the daemon. Temporarily disabling services is useful during maintenance periods when certain services may be inaccessible, but their existing non-persistent state information must be retained until the service is reactived. A subsequence reconfiguration may be performed to re-enable the service without performing the entire sequence of initialization steps again.

4.3.2 Server Daemon Run-Time Activities

When configuration activities are complete, an application calls the Service_Config's run_event_loop function. This function enters an endless loop that continuously calls the Reactor's handle_events service dispatch function, which blocks awaiting the occurrence of events such as I/O from clients or timer expiration. As these events occur, the Reactor automatically dispatches previously-registered application-specific handler(s) to perform the designated services.

Run-time activities may be influenced by a set of Service_Object subclasses (illustrated in Figure 7) that perform the following standard daemon foundation services:

• **Eager and Lazy Process and Thread Generation:** Two standard subclasses of Service_Object implement the "eager" and "lazy" process and thread generation techniques discussed in Section 3.2.3 above. The Eager_Gen subclass pre-spawns one or more processes or threads to form a pool that minimizes service startup overhead when requests arrive. The Lazy_Gen subclass, on the other hand, only spawns a new process if an executing request does not finish within a certain time interval. Application services that use these techniques may inherit from either the Eager_Gen or Lazy_Gen subclasses.

• **Lightweight Dynamic Service Spawning:** Two subclasses of Service_Object implement service spawning

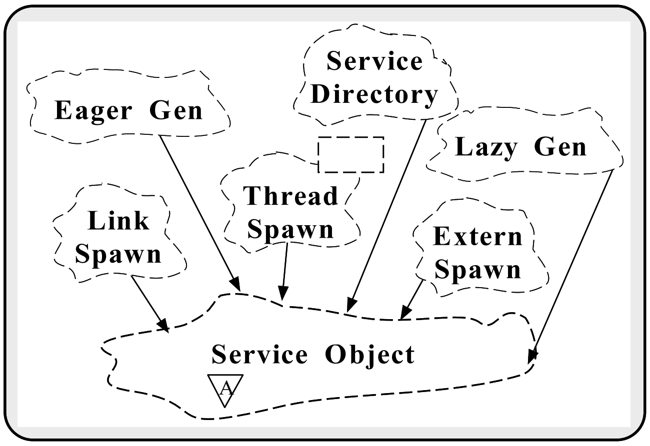

Figure 7: Standard Service_Object Subclasses

techniques that are typically "lighter-weight" than the process invocation method used by `inetd` and `listen`. For example, rather than use `fork` and `exec` to create new processes that perform service requests externally, the `Link_Spawn` subclass dynamically links and executes a new service internally. Moreover, services derived from `Link_Spawn` are loaded and unloaded on demand. This contrasts with the default dynamic configuration behavior obtained by specifying the **dynamic** service config directive, which pre-loads services during daemon initialization. The `Link_Spawn` subclass is implemented by (1) dynamically linking an object file, (2) obtaining the entry-point of the appropriate `Service_Object` in this file, and (3) invoking the service to perform the client request. Upon completion, the service installed by `Link_Spawn` is automatically removed by closing down the `Service_Object` and unlinking the object file from the daemon's address space.[3]

The `Thread_Spawn` subclass provides another technique for handling service requests internally. It creates a separate thread on-demand and each thread carries out the service to completion. However, unlike the `Eager_Gen` subclass, these threads are not pre-spawned and cached. The use of threads is typically less time consuming than using `fork` and `exec` [20]. On the other hand, the `fork/exec` approach may be preferable in situations where the owner of the child process must differ from the parent for security reasons, which is typically the case with remote login and file access services. Moreover, spawning separate threads may be less robust than spawning separate processes since all threads share resources in a process and global data structures may be corrupted if errors occur. In general, developers must consider their application requirements carefully when selecting an appropriate service execution agent.

• **Service Directory:** The `Service_Directory` subclass provides local and/or remote clients with access to daemon administration commands that report 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_Directory` object may be registered at a well-known communication port accessible by clients using the following entry in the `svc.conf` file (the `Service_Directory` service is statically linked into the `SVC-CON` framework):

```
static Service_Directory -d -p 9000
```

When clients request a summary of a daemon's active services, the `Service_Repository` iterator is automatically invoked by the `Service_Directory`. This iterator transfers a complete listing of the developer-supplied information for each enabled service back to the client. This listing indicates both the address format and the transport protocol to use to contact a given service, and provides a brief explanation of each service. The `Service_Directory` may also be used to trigger reconfiguration requests from remote sites.

• **Internet Superserver Emulation:** The external service dispatching semantics of `inetd` are provided via a subclass of `Service_Object` called `Extern_Spawn`. This class spawns processes on-demand to handle client requests as external services. By default, `Extern_Spawn` utilizes the standard `inetd.conf` file and serves as a replacement for `inetd`.

5 Using the Server_Daemon for EOS Applications

This section describes how the `SVC-CON` framework forms the basis for implementing the primary services that comprise the Ericsson EOS application family. The boxes enclosing certain collections of services in Figure 8 indicate the default binding of services to processes (the thread bindings are described further below). The `SVC-CON` framework's configuration techniques, tools, and resources simplify the task of modifying these default bindings in response to performance enhancements and additional application requirements. The following paragraphs describe the EOS services and indicate the communication protocol, service, and concurrency dimensions associated with each service:

• **GICI Manager Services:** The GICI (General Information Computer Interface) Manager provides services that exchange low-level, real-time status information with a PBX. The GICI protocol operates over an RS-232 serial link, exchanging signals (represented as short sequences of ASCII characters) between an external computer and the *Information Computer Unit* (ICU) port on the PBX. The GICI Manager provides services that transmit signals to the PBX upon request of a client and asynchronously receive signals generated by the PBX. To increase throughout and reduce latency, this service is implemented internally via a single separate thread within the Directory Management application process.

[3]This technique was inspired by the command-line interpreter mechanisms used to invoke programs in Multics [19].

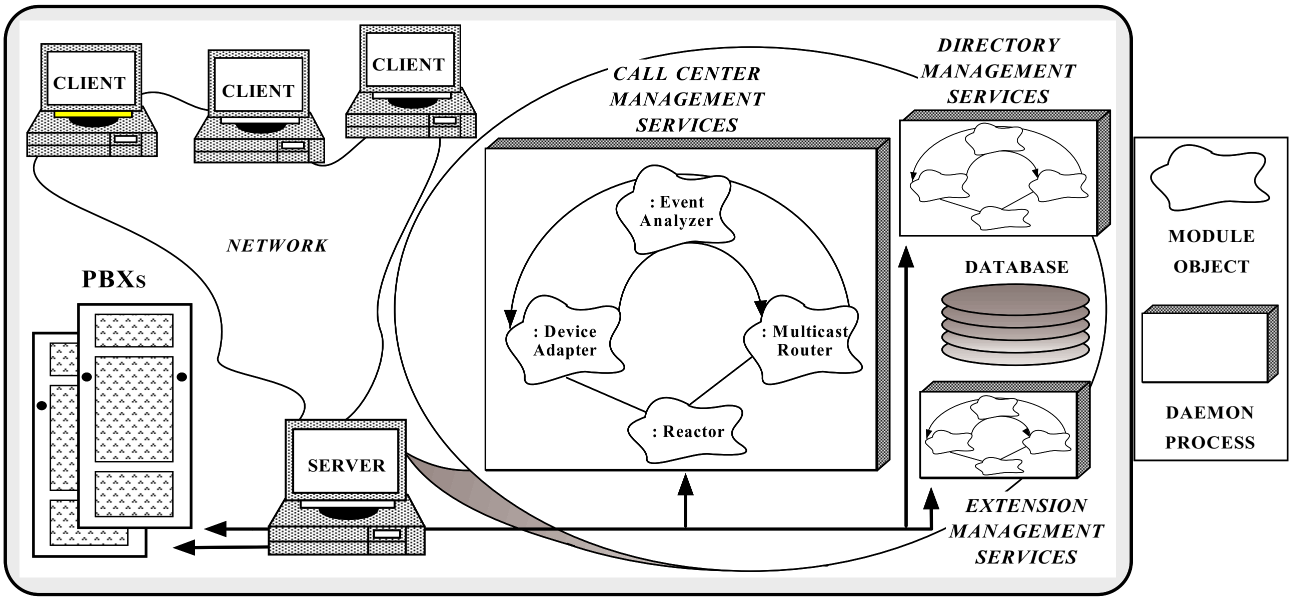

Figure 8: Services Offered by EOS Applications

• **MML Manager Services:** The MML (Man-Machine Language) Manager provides services that perform low-level, static configuration operations on the PBX. The MML protocol involves the synchronous, request-response transmission of an MML command from an external computer to the *I/O Processor Unit* (IPU) of a PBX via an RS-232 serial link. An MML command is a string of ASCII characters containing a command identification code and associated parameters. The response from the PBX is returned to the external computer via the same RS-232 link. The MML Manager is implemented in the Extension Management application as an internal service via a separate thread that serializes multiple clients accessing a PBX.

• **Signal Router Services:** The Signal Router services provide capabilities for demultiplexing GICI signals passed from the GICI Manager monitoring a PBX to the proper client(s) that have registered to receive the generated signals. This service is implemented internally via a single separate thread within the Call Center Management application (the thread uses a multicast protocol [21] to forward signals to interested clients).

• **PBX Manager Services:** The PBX Manager provides services for high-level PBX management operations such as call profiling, diversion management, and message management. This service is implemented internally within the Directory Management application and is controlled by a Client Muxer.

• **Client Muxer:** A Client Muxer enables one or more services to communicate concurrently with multiple clients via connection-oriented or request-response protocols (such as TCP or RPC, respectively). A separate thread is maintained for each client connection. Depending on configuration parameters, threads may be allocated from pool spawned by `Eager_Gen` or on-demand via `Thread_Spawn`.

• **Extension Administration (XAD) Services:** The Extension Administration services provide high-level PBX operations such as adding, deleting, and modifying extensions. In addition, services are provided to download PBX extension configuration information, which is mirrored in a database on the server to improve response time and off-load redundant processing from the PBX. This service is implemented as an internal service that interacts with clients via a Client Muxer communicating over a connection-oriented protocol.

• **Batch Manager Services:** The Batch Manager provides scheduling services for queueing and executing extension administration requests at a pre-determined time. Services are provided to insert new batch requests, delete batch requests (that have not yet been executed), or query the completion status of batch requests that have been executed. This service is implemented as an external service running in a separate process invoked periodically via a service dispatcher driven by an external system clock (such as the UNIX `cron` facility).

The layering of the services in Figure 8 illustrates the *uses* relations between the various services in each application.[4] In addition to reusing the foundation classes provided by the `SVC-CON` the EOS applications also reuse several of the services described above. For example, the GICI Manager service is shared by the Call Center Management and Directory Management applications. Likewise, the Client Muxer service is reused by the Directory Management and Extension Management applications. In general, services may be implemented as shared objects to reduce primary and secondary

[4]Currently, the layered application services interoperate via *ad hoc* communication techniques (such as message queues, shared memory, and parameter passing). Future versions of the `SVC-CON` will incorporate a user-level communication framework known as *u*Streams [22] to handle hierarchically-organized services [23] more elegantly and efficiently (*e.g.,* by reducing context switching and data copying overhead [24]).

storage consumption.

Note that the SVC-CON framework tries to make as few assumptions as possible regarding the structure of the client (and even the server). Basically, the primary contribution of the SVC-CON is to provide a set of object-oriented interfaces and standard mechanisms for automatically configuring a set of (practically arbitrary) services into a server application (actually, the same approach could also be used for the client, though that is somewhat less common). The term "practically arbitrary" indicates that the current version assumes services will be communicating via an I/O descriptor that is capable of being select'd or poll'd. Therefore, it is feasible to integrate the svc functionality "underneath" an RPC communication model, though it might require some quasi-portable assumptions to extract the underlying descriptor from a given RPC toolkit.

We are currently evaluating the performance of the configuration depicted in Figure 8 to determine whether to incorporate other SVC-CON features such as Link_Spawn and Lazy_Gen. We are also investigating service reconfiguration policies to formulate guidelines that ensure the dynamic modification of a daemon does not corrupt or seriously disrupt existing services. A more ambitious extension involves using the SVC-CON mechanisms to experiment with service migration policies that relocate certain services dynamically to reduce overall system workload.

6 Concluding Remarks

The SVC-CON is an integration framework that supports static and dynamic configuration of internal and external network services the execute within one or more OS processes and threads. The long-term goals of this project are (1) to produce an extensible environment that coordinates reusable abstractions and components to support families of distributed applications and (2) to devise techniques and tools for developing distributed systems that are efficient, cost-effective, modular, scalable, extensible, and easily configured and installed. To help achieve these goals, the general principles underlying the SVC-CON framework involve (1) separating policies from mechanisms via object-oriented class abstractions, inheritance, dynamic binding, and parameterized types in order to enhance the reuse of common network daemon components, (2) decoupling the binding of OS processes and threads from the application services to improve flexibility and performance, and (3) utilizing dynamic linking and threads to improve extensibility and permit fine-grained time/space tradeoffs.

The existing prototype implementation described in this paper fulfills many of the project's goals. We are currently using the SVC-CON framework to configure, install, and administer a suite of concurrent network services for the Ericsson EOS client/server PBX management applications. Thus far, the primary benefits of the framework center around enabling developers to (1) enhance network daemon functionality and reliability and (2) fine-tune performance without extensive redevelopment and reinstallation effort. For example, debugging a faulty service typically involves reinstalling a functionally equivalent service containing additional instrumentation that helps isolate the source of erroneous behavior. The utility of certain features remain to be seen. For example, the Link_Spawn service may be less applicable for network servers running on multi-threaded platforms, compared with the Thread_Spawn service. In addition, we are experimenting with certain reconfiguration and service migration mechanisms offered by the SVC-CON to determine circumstances where they may be applied reliably. We are also developing a suite of tools that reduce the effort required to administer daemon configuration files (which are currently managed manually).

An implementation of a public domain subset of the SVC-CON framework described in this article is available via anonymous ftp from ics.uci.edu in the gnu/C++_wrappers.tar.Z file. This file also contains the source code, documentation, and examples for the IPC_SAP and Reactor utilities described in [3, 4, 5].

References

[1] Ericsson Business Communications, Tyreso, Sweden, *MD110 Operations and Maintenance*, 1992.

[2] 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.

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

[4] D. C. Schmidt, "The Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing (Part 1 of 2)," *C++ Report*, vol. 5, February 1993.

[5] D. C. Schmidt, "The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2)," *C++ Report*, vol. 5, September 1993.

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

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

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

[9] S. J. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

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

[11] D. C. Schmidt, "Object-Oriented Techniques for Developing Extensible Network Servers," in *Proceedings of the Second C++ World Conference*, (Dallas, Texas), SIGS, Oct. 1993.

[12] D. C. Schmidt, B. Stiller, T. Suda, A. Tantawy, and M. Zitterbart, "Language Support for Flexible, Application-Tailored Protocol Configuration," in *Proceedings of the* 18^{th} *Conference on Local Computer Networks*, (Minneapolis, Minnesota), pp. 369–378, Sept. 1993.

[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] W. W. Ho and R. Olsson, "An Approach to Genuine Dynamic Linking," *Software: Practice and Experience*, vol. 21, pp. 375–390, Apr. 1991.

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

[16] Bjarne Stroustrup and Margret Ellis, *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[17] M. Herlihy and B. H. Liskov, "A Value Transmission Method for Abstract Data Types," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 527–551, October 1982.

[18] 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.

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

[20] 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.

[21] S. E. Deering and D. R. Cheriton, "Multicast routing in datagram internetworks and extended LANs," *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 85–110, May 1990.

[22] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[23] D. Batory and S. W. O'Malley, "The Design and Implementation of Hierarchical Software Systems Using Reusable Components," *ACM Transactions on Software Engineering and Methodology*, vol. 1, Oct. 1992.

[24] N. C. Hutchinson and L. L. Peterson, "The *x*-kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, January 1991.