The ADAPTIVE Service Executive: An Object-Oriented Architecture for Configuring Concurrent Distributed Communication Systems

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 [1]

An earlier version of this paper appeared in the proceedings of the 8^{th} IFIP International Working Conference on Upper Layer Protocols, Architectures, and Applications in Barcelona, Spain, June 1994.

Abstract

The ADAPTIVE Service eXecutive (`ASX`*) is an object-oriented framework that enhance the development of distributed applications across a range of operating system platforms. The components in* `ASX` *were developed using object-oriented design techniques and C++ language features in order to simplify the use of OS mechanisms that provide interprocess communication, communication port demultiplexing, explicit dynamic linking, and concurrency. In addition, the* `ASX` *components automate many system configuration and reconfiguration steps by dynamically linking network services into applications at run-time and arranging to execute these services on one or more processes or threads. This paper describes the structure and functionality of* `ASX` *and presents several examples illustrating key* `ASX` *features.*

1 Introduction

Developing communication systems that effectively utilize multi-processing and network services is a promising technique for increasing system performance, scalability, and cost effectiveness. However, complex distributed communication systems (such as on-line transaction processing systems, manufacturing process controllers, distributed object managers, and global mobile communication systems) typically exhibit reliability, functionality, efficiency, and portability requirements that are challenging to satisfy simultaneously. To meet these challenges, developers must address many topics that are not relevant or are less problematic for stand-alone applications, including (1) local and remote interprocess communication (IPC) facilities, (2) system configuration management techniques that permit the flexible insertion, modification, and removal of services from applications at installation-time and during run-time, (3) process and thread creation, synchronization, communication, and termination mechanisms, and (4) debugging and monitoring support for tracking application behavior.

Object-oriented design and implementation techniques offer a variety of principles, methods, and tools that help to alleviate complexity related to developing distributed communication systems. This complexity emanates from factors such as non-type-secure, non-portable, and non-extensible library and system call interfaces, as well as a lack of efficient higher-level network programming abstractions that leverage off the increasing availability of advanced OS mechanisms such as explicit dynamic linking and multi-threading. To illustrate how object-oriented techniques are being successfully applied in several commercial and research projects, this paper examines the structure and functionality of the ADAPTIVE Service eXecutive (`ASX`) framework. This framework facilitates the development, configuration, and experimentation with concurrent, multi-service distributed communication systems composed of singleton and/or hierarchically-related services [1]. The `ASX` framework leverages off a collection of C++ components that (1) support dynamic configuration of application services, (2) consolidate common distributed application activities (such as connection management, external data conversion, reliable data transfer, I/O-based and timer-based event demultiplexing, service dispatching, content-based message routing, status logging, and inter-connection of hierarchically-related communication services) within reusable C++ classes and frameworks, and (3) take advantage of available OS multi-threading and multi-processing mechanisms in a flexible manner.

This paper is organized as follows: Section 2 reviews relevant background material; Section 3 describes the primary features and object-oriented architecture of the `ASX` framework; Section 4 outlines several research projects and commercial communication systems that utilize components in the `ASX` framework; and Section 5 presents concluding remarks.

[1] This material is based upon work supported by the National Science Foundation under Grant No. NCR-8907909. This research is also supported in part by grants from the University of California MICRO program, Nippon Steel Information and Communication Systems Inc. (ENICOM), Hitachi Ltd., Hitachi America, and Tokyo Electric Power Company.

2 Research Background

Various strategies and tactics for developing highly configurable communication systems and distributed application frameworks have emerged in several research domains. The `ASX` framework incorporates concepts from several modular communication frameworks including System V STREAMS [2], the *x*-kernel [3], and the Conduit framework [4] from the Choices object-oriented OS (a survey of these and other communication frameworks appears in [5]). These frameworks all contain features that support the flexible configuration of communication systems (such as those based upon the Internet or ISO OSI reference models) by inter-connecting "building-block" protocol and service components. In general, these frameworks encourage the development of standard communication-related components (such as message managers, timer-based event dispatchers, demultiplexors [3], and assorted protocol functions [6]) by decoupling processing functionality from the surrounding framework infrastructure. As described below, the `ASX` framework contains additional features that further decouple system functionality from the concurrency mechanisms used to provide parallelism.

Another influential branch of research addresses broader policies for (1) reliably guiding the reconfiguration of executing communication systems and (2) representing application state attributes as abstract data types to facilitate flexible service configuration [7], communication [8], and migration in heterogeneous and homogeneous [9] environments.

Another influential branch of research involves daemon control frameworks..[2] Daemon control frameworks provide mechanisms that automate many tedious and error-prone activities associated with configuring and reconfiguring network daemons. These activities include (1) performing daemonization 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 control frameworks are `inetd` [10] and `listen` [11], which are both distributed with System V Release 4 UNIX. `Inetd` and `listen` are multi-service daemon control 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[3] (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 control 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 The ADAPTIVE Service eXecutive (ASX) Framework

Object-oriented application frameworks are becoming increasingly popular as a means to simplify and automate the development and configuration process associated with complex domains such as graphical user interfaces, databases, and distributed communication systems. An application framework is characterized by an integrated collection of components that cooperate to define a reusable architecture for a family of related applications [12]. Frameworks are distinguished from conventional class libraries in several ways:

- The components constituting a framework are integrated together to address a particular problem domain. In contrast, class library components (such as classes for Strings, complex numbers, arrays, bitsets, etc.) are often developed to be domain independent
- Complete communication systems may be formed by inheriting from and/or customizing existing framework components, rather than simply invoking methods provided in a class library. Inheritance enables the features of a framework class to be shared automatically by its descendant classes. It also allows the framework to be extended transparently without affecting the original code. Developers often interact with an application framework by inheriting basic functionality from its existing scaffolding and overriding certain virtual methods to perform application-specific processing.

[2] A daemon is a single operating system (OS) process that executes on a host machine in the "background" (*i.e.,* disassociated from any controlling terminal) [10].

[3] 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.

- At run-time, the framework is usually responsible for managing the event-loop(s) that provide the default flow of control within an application. The framework determines which set of framework-specific and application-specific methods to invoke in response to external events (such as messages arriving on communication ports).

The ADAPTIVE Server eXecutive (`ASX`) is an object-oriented framework that is specifically targeted for the distributed application domain. In particular, this framework simplifies the construction of distributed communication systems by improving the modularity, extensibility, reusability, and portability of both the application-specific services and the underlying OS concurrency, IPC, and demultiplexing mechanisms that these services rely upon. The primary components in the `ASX` framework consist of C++ classes and other class categories that may be combined flexibly via inheritance, template instantiation, and object composition. The following section describes the key features of the `ASX` framework and outlines its primary architectural components.

3.1 The `ASX` Framework Features

The `ASX` framework provides the features described below.

3.1.1 Reusable, Application-Independent Components

The `ASX` framework integrates a collection of reusable communication-related components [13] that handle common distributed application activities such as port monitoring; message buffering, queueing, and demultiplexing; service dispatching; local/remote interprocess communication; concurrency control; and application configuration, installation, and run-time service management. The use of object-oriented techniques and C++ features enhance the reusability and extensibility of these `ASX` components.

To implement these features, the `ASX` framework leverages off the multi-threading and explicit dynamic linking facilities available in operating systems such as UNIX and Windows NT [14]. When combined with the use of C++ language features such as templates, inheritance, and dynamic binding, the reusable `ASX` components facilitate the development of clients and servers that often may be updated and extended without modifying, recompiling, relinking, or even restarting existing applications. For example, as illustrated in Section 4, distributed communication systems may be developed incrementally by inheriting, composing, and customizing the existing suite of `ASX` components that support the dynamic configuration and concurrent execution of application services [1].

3.1.2 Support for Highly-Decoupled System Architectures

The `ASX` framework enhances the flexibility and extensibility of distributed communication systems by decoupling application service functionality from the following system characteristics:

- **System Structure:**
  - The type and number of services associated with each process. In particular, the `ASX` framework supports both single-service and multi-service applications.
  - The point of time at which the service(s) are configured into an application. A class category within `ASX` called the `Service Configurator` [1] is used to encapsulate OS explicit dynamic linking mechanisms. This enables services to be configured into `ASX`-based applications either (1) *statically* (at compile-time or link-time) or (2) *dynamically* (when an application first begins executing or even while it is running). Moreover, the choice between static and dynamic configuration may be deferred until installation-time.
  - The order in which hierarchically-related services are combined into an application. Each service is represented as a distinct set of independent objects that communicate by passing messages. These objects may be joined together in essentially arbitrary configurations to satisfy applications requirements and enhance component reuse.

- **Communication Mechanisms:**
  - The underlying IPC mechanisms used to communicate with participating clients and servers. The `IPC_SAP` [15] class library encapsulates the socket, TLI, STREAM pipe, and named pipe mechanisms via an object-oriented interface.
  - The I/O-based and timer-based event demultiplexing mechanisms used to dispatch incoming connection requests and data onto the appropriate application-specified service. A sub-framework within `ASX` called the `Reactor` [16] portably encapsulates both the `select` and `poll` I/O demultiplexing system calls via an object-oriented interface.

- **Execution Agents:**
  - The type and number of execution agents and process architectures used to perform services at run-time. Developers may select between user-level and kernel-level process and thread execution agents.
  - The `ASX` framework provides a set of classes and tools that enable flexible selection from among several message-based and task-based *process architectures* [5]. A process architecture binds units of application service processing (such as layers, functions, connections, messages, etc.) with one or more CPUs [17]. The choice of process architecture significantly affects key sources of distributed application performance overhead (such as memory-to-memory copying and data manipulation,

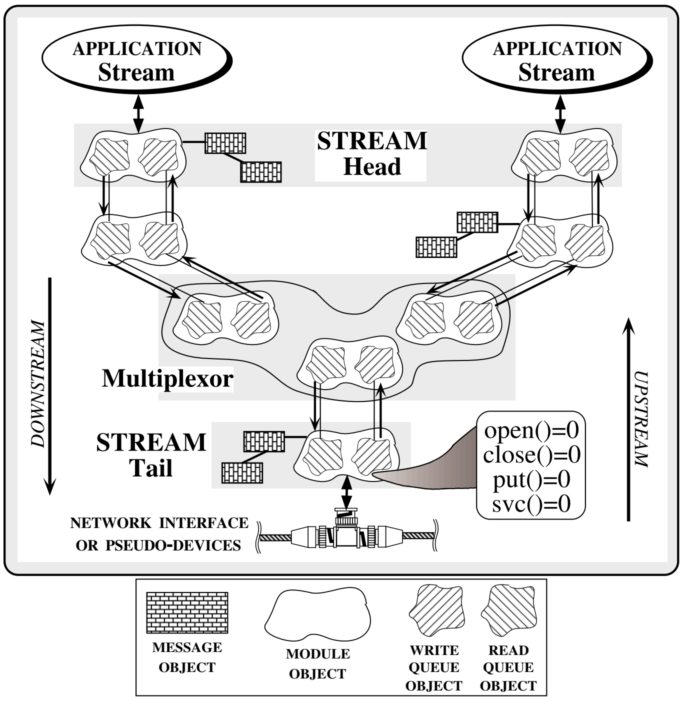

Figure 1: C++ Components in the ASX Framework

context switching, scheduling, and synchronization), and also influences demultiplexing strategies and protocol programming techniques [3]. The process architecture components in ASX also support *multiple service invocation semantics*, where application service processing may be performed concurrently via either synchronous and/or asynchronous techniques. Section 3.3 describes the process architecture support in the ASX framework in greater detail.

The ASX framework enables applications to avoid prematurely committing to the structural, communication mechanism, and execution agent system charactistic described above until late in the development cycle (*i.e.,* during installation-time or run-time). By deferring these decisions, application portability, reusability, and extensibility is enhanced. For example, decoupling design and implementation choices until sufficient information is available helps tailor distributed communication system service configurations to specific application requirements, OS platform characteristics, and network conditions.

3.2 The Architecture of the ASX Framework

This section briefly describes the primary C++ components in the ASX framework (illustrated in Figure 1). To avoid gratuitously renaming familiar terminology, many C++ class names in the ASX framework correspond closely with functionally equivalent components available in System V STREAMS. In addition, most ASX class interfaces are also relatively similar to their STREAMS counterparts, though C++ inline accessor/mutator functions are generally used in lieu of accessing class data fields directly. However, the implementation techniques used in ASX are significantly different, with an emphasis on supporting flexible service configuration and concurrency control for distributed communication systems running on multi-processor platforms. For instance, to reduce the likelyhood of deadlock and to simplify intra-Stream flow control, the ASX framework's process architecture components completely re-engineer the coroutine-based, "weightless" service processing mechanisms used in STREAMS [18]. A weightless process does not execute on its own separate run-time stack. Therefore, it may not suspend execution to wait for resources to become available or events to occur, which greatly complicates programming and increases the potential for deadlock.

The remainder of this section discusses the primary components of the ASX framework in detail.

• **The STREAM Class:** In ASX, the STREAM class provides applications with a get/put-style interface for sending and receiving data and control messages on a Stream. The STREAM class is also the primary unit of application service configuration for a particular instance of a Stream. This class implements the interconnection logic required to insert and remove service processing Modules into and from a Stream concurrently and correctly at run-time.

• **The Module Class:** The Module class is the primary unit of interconnection for clustering one or more application services together in a Stream. By default, two standard Modules (the Stream_Head and the Stream_Tail) are installed automatically when a Stream is opened. These Modules interpret standard control messages that circulate through a Stream at run-time. For incoming messages, the Stream_Tail class typically transforms packets from network devices or pseudo-devices into a canonical message format recognized by other components in a Stream (it performs the opposite transformation for outgoing messages). Likewise, the Stream_Head class provides message creation and buffering capabilities between an application and a Stream. I/O between an application and a Stream is synchronous when the Stream_Head Module appears at the top of a Stream. However, if the Stream_Head is omitted, messages percolating up a Stream are delivered into the address space of an application asynchronously. Each instance of a Module contains two Tasks: one handles "read-side" processing for incoming messages and the other handles "write-side" processing for outgoing messages.

• **The Task Class:** Each Task contains (1) a Message_List that enables queueing of messages, (2) a pointer to its adjacent Task on a Stream, (3) a back-pointer to its enclosing parent Module (which enables it to locate its sibling), and (4) a number of standard utility methods that maintain and mediate access to the internal state of a Task. Several methods in the Task class are defined as *pure vir-*

tual functions [4], which ensures that derived subclasses will provide the requisite data structures and application service functionality. For example, derived subclasses must supply `open` and `close` methods that perform application-specific `Task` initialization and termination activities (such as allocating and releasing per-session control blocks and private synchronization objects). Likewise, subclasses must also define a `put` method, which performs service processing synchronously when a message arrives from an adjacent `Task`. In addition, a subclass may optionally provide a `svc` method to handle service processing asynchronously. Selecting between these alternatives depends on certain installation-time and run-time factors such as the choice of process architecture and the current availability of chronically scarce shared resources like `Message_Blocks`.

• **The Message_List Class:** The `Message_List` class provides a thread-safe message buffering facility built upon the underlying `Message_Block` class (which itself is a linked list of one or more `Message_Block` objects that form a complete message). A "simple" message contains a single `Message_Block` and a "composite" message contains multiple `Message_Blocks` linked together. The overhead resulting from passing `Message_Blocks` between `Tasks` is minimized by passing pointers to messages rather than copying data. Each `Message_List` contains a spin lock that prevents race conditions when `Message_Blocks` are enqueued and dequeued concurrently by multiple threads. In addition, each `Message_List` contains a pair of condition variables (named `notfull` and `notempty`) that implement flow control between adjacent `Tasks`. When one `Task` attempts to insert a `Message_Block` into a neighboring `Task` that has reached its high watermark, the `wait` operation it performs on the `notfull` condition variable atomically relinquishes the CPU and sleeps awaiting flow control conditions to abate. When the number of bytes in the flow controlled `Message_List` falls below its low watermark, the blocked `Task` is automatically signaled to resume its execution.

• **The Multiplexor Class:** The `Multiplexor` class provides mechanisms that enable layered application services to demultiplex messages between one or more `Modules` in a Stream. `Multiplexors` are implemented as a C++ template class parameterized by an external identifier. This external identifier is used to instantiate a `Map_Manager` template that performs efficient intra-Stream message routing. Each `Map_Manager` object contains a set of `Modules` that may be linked above and below a `Multiplexor` in essentially arbitrary configurations. Although layered multiplexing and demultiplexing is generally disparaged for high-performance communication systems, most conventional communication models involve some form of multiplexing, so the `ASX` framework provides mechanisms that support it.

[4] Pure virtual functions are a C++ feature that provide only abstract interfaces, without any accompanying definitions [19]. Subclasses *must* subsequently provide these definitions before objects may be instantiated.

• **The Service Configurator Class Category:** The `ASX` framework uses the `Service Configurator` described in [1]. The `Service Configurator` is centered around the `Service_Config` class illustrated in Figure 2 (3)[5], which integrates the `Service_Object`, `Service_Repository`, and `Reactor` components as described below:

- **The Service_Object Abstract Base Class** – The `Task` class is derived from the `Service_Object` class, which provides interfaces that allow developers to specify the information necessary to support automatic dynamic linking and service initialization at run-time [1]. As shown in Figure 2 (1), the `Service_Object` class itself inherits from the `Event_Handler` base class. When used in conjunction with the `Reactor` described below, the `Event_Handler` base class provides automatic I/O port demultiplexing and service dispatching for application-specific `Tasks` that communicate with external processes and/or devices.
- **Standard Subclasses of Service_Object** – The `Service Configurator` contains a library of standard components that inherit from `Service_Object`. 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`, `Link_Spawn`, or `Service_Manager` subclasses illustrated in Figure 2 (2).

  The `Eager_Spawn` subclass pre-spawns one or more processes or threads at application 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.

  The `Lazy_Spawn` 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 application. However, if the timer expires a new slave process is automatically spawned to continue processing the service independently from the master dispatcher process [21].

  The `Process_Spawn` 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

[5] These components and their relationships are illustrated via Booch notation [20]. Dashed clouds indicate classes and directed edges indicate inheritance relationships between these classes. Solid clouds indicate one or more class objects and nesting indicate composition relationships between these objects (*cf.* Figure 1).

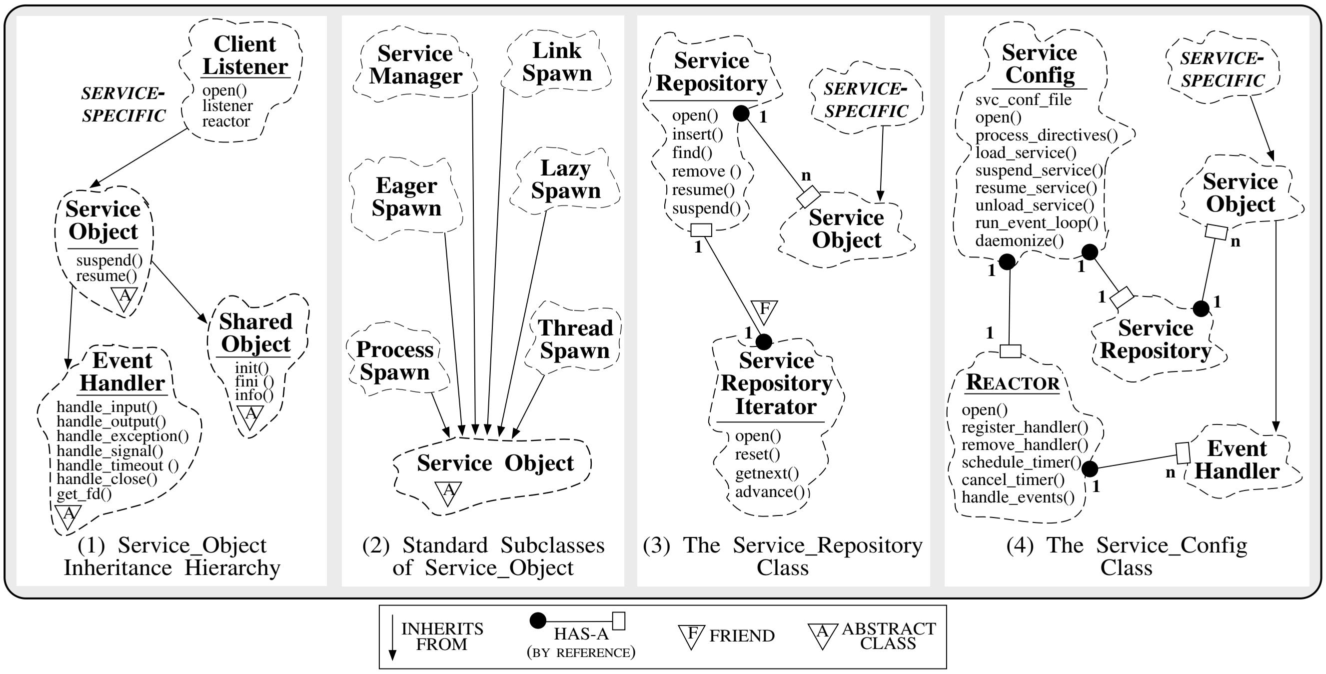

Figure 2: Component Relationships for the Service Configurator Framework

service 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.

The Thread_Spawn 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.

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 appropriate Service_Object in this file, and (3) invoking the service to perform the client request. 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.

The Service_Manager subclass enables local and remote clients to determine which services are currently offered by a network application. During application configuration, a Service_Manager object may be registered at a well-known communication port accessible by clients. When a client requests a list of enabled application services, the handle_input method in the Service_Manager invokes the iterator for the Service_Repository class (described below). This iterator is used to generate a complete listing of developer-supplied information that describes each enabled service. This listing is transferred back to the client to indicate both the address format and the transport protocol required to contact application services.

- **The Service_Repository Class** – To simplify administration of single-service and multi-service applications, it is necessary to individually and/or collectively control and coordinate the Service_Objects that comprise an application's current suite of active services. The Service_Repository is an object manager that coordinates local and remote queries and updates involving the services offered by an ASX-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 illustrated in Figure 2 (3), the Service_Repository also contains methods that load, (re)enable, disable, or remove Modules and Multiplexors from an application statically and/or dynamically, based upon notification from external events (such as signals or control messages).

- **The Reactor Class Category** – The `Reactor` is an extensible event demultiplexing and service dispatching framework that portably encapsulates and enhances the functionality of the UNIX `select` and `poll` I/O demultiplexing mechanisms. An instance of the `Reactor` is provided within the `ASX` framework to automate the registration and dispatching of services within one or more Streams. Typically, these services interact with external I/O devices (such as network controllers and serial-line drivers). The `Reactor` integrates the demultiplexing of I/O descriptor-based events together with timer-based events via the uniform service dispatching interface provided by the `Event_Handler` base class. Application-specific subclasses define composite objects by inheriting and refining this interface. As shown in Figure 2 (4), the `Reactor` also contains a set of methods that register, remove, schedule, and expire I/O-based and timer-based objects. When an object is registered by an application, the `Reactor` extracts the underlying I/O descriptor from the object and stores it (along with descriptors from other registered objects) into a data structure passed to `select` or `poll`. When events associated with registered objects occur at run-time, the `Reactor` automatically dispatches the appropriate method(s) of the activated objects, which then perform application-specific services.
- **The Service_Config Class** – The `Service_Config` class shown in Figure 2 (4) integrates the other components in the `Service Configurator` to facilitate the static and/or dynamic configuration of concurrent, multi-service communication systems.

Figure 3 illustrates several representative architectural configurations of `ASX` components within a distributed environment. Figure 3 (1) depicts a client application that utilizes `ASX` components to manipulate I/O messages sent and received on one or more network devices. Figure 3 (2) portrays a server application composed of several inter-connected, *layered* services. Figure 3 (3) illustrates an `rwho` daemon configuration involving *singleton* services that are not related hierarchically nor inter-connected.

3.3 Flexible Process Architecture Support

The `ASX` framework is designed to decouple operations that implement service functionality from the (1) execution agents, (2) concurrency control mechanisms, and (3) service invocation semantics used to implement particular process architectures such as message-based parallelism and task-based parallelism [22]. The following techniques are used by the `ASX` framework to accomplish this decoupling:

• **Multi-level Concurrency Model:** The `ASX` framework leverages off the multi-level concurrency model provided by the underlying SunOS 5.3 process and thread management facilities [14]. The semantics of the SunOS synchronization objects (such as mutex and condition variables, counting semaphores, and readers/writer locks) offer equivalent semantics when used within threads spawned via one of the following two modes:

1. *Bound threads* – which map 1-to-1 directly onto kernel threads. Bound threads permit independent services to execute in parallel on multiple CPUs. However, a context switch is generally required to reschedule bound threads and most synchronization operations require OS kernel intervention.
2. *Unbound threads* – which are multiplexed in an n-to-m manner atop one or more kernel threads by a hybrid user/kernel-level thread run-time library. This library schedules, dispatches, and suspends unbound threads, while attempting to minimize kernel involvement. Depending upon the number of kernel threads that an application and/or library associates with a process, one or more unbound threads may execute on multiple CPUs in parallel.

The `ASX` framework employs this multi-level concurrency model to flexibly support several flavors of parallelism with minimal impact on a process architecture's overall structure. For example, the parallelism obtained directly via bound threads is useful for simultaneously performing presentation layer conversions on multiple messages using multiple CPUs. These operations benefit significantly from direct parallelism since they involve almost no inter-thread communication or synchronization [23]. Conversely, maintaining a pool of unbound threads that shepard messages throughout a stack of services may benefit from the reduced kernel involvement associated with the multiplexed flavor of parallelism provided by unbound threads.

• **Configurable Concurrency Control Classes:** By default, the core `ASX` C++ classes described in Section 3.2 are implemented with minimal internal locking to avoid over-constraining the granularity of a process architecture's synchronization strategies. In particular, only framework mechanisms that would not function correctly in a multi-threaded environment (such as enqueueing `Message_Blocks` onto a `Message_List` or resolving internal `Module` addresses stored in a `Multiplexor` within a Stream) are protected by synchronization objects. More sophisticated concurrency control schemes may be created by selectively instrumenting services with `ASX` synchronization wrappers [13]. These wrappers utilize several C++ features such as (1) inheritance, (2) parameterized types, and (3) conditional compilation to select from a pre-defined library of C++ synchronization components. In general, this approach decouples the protocol processing functionality from the mutual exclusion code that synchronizes interactions between objects within a particular process architecture.

• **Alternative Service Invocation Semantics:** Messages that arrive at the head or the tail of a Stream are shepharded

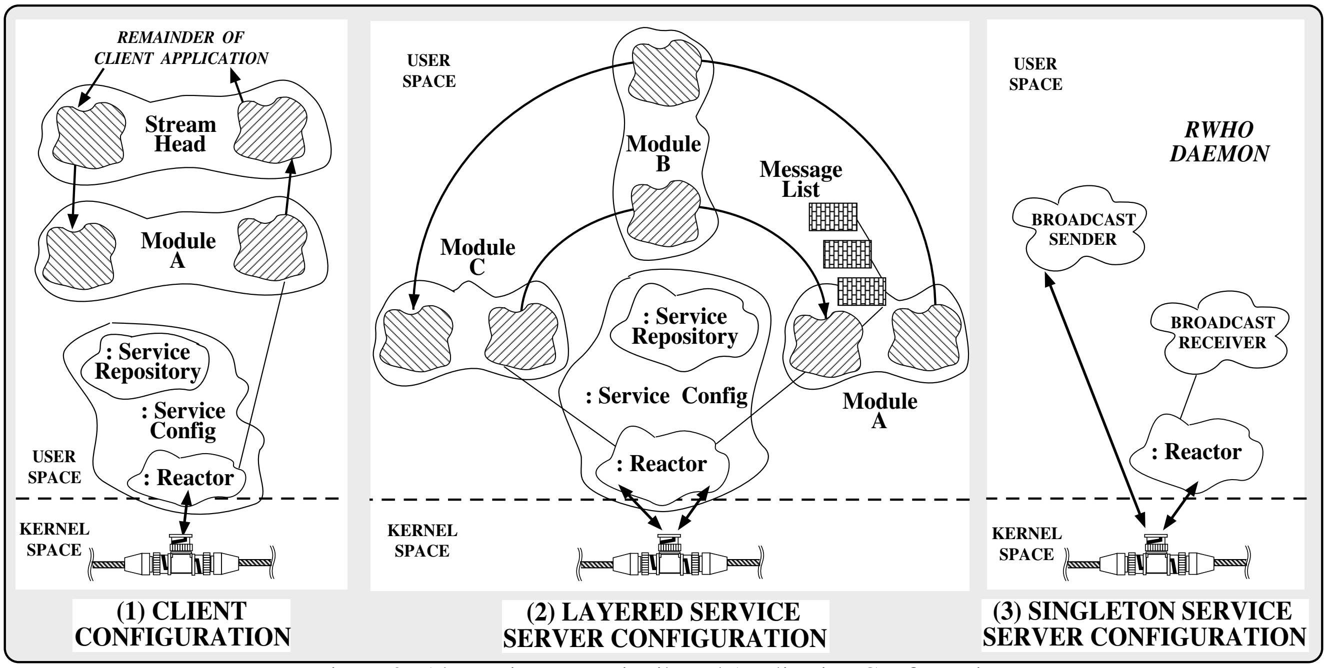

Figure 3: Alternative ASX Distributed Application Configurations

through a series of inter-connected `Tasks` by repeatedly calling their `put` and/or `svc` methods via either synchronous and/or asynchronous invocation mechanisms [24], as follows:

- *Synchronous Invocation* – This mechanism borrows the thread of control from the entity that passed a message via the `Task::put` method. A thread of control generally originates "upstream" from an application process, "downstream" from an I/O device interrupt, or internally from an event dispatching mechanism (such as a timer-driven callout queue used to trigger retransmissions for connection-oriented transport protocols).
- *Asynchronous Invocation* – This mechanism typically emanates from one or more threads associated with a `Task` or `Module`. A thread executes a service's `Task::svc` method, which runs an event loop that continuously waits for messages to arrive on the `Task`'s `Message_List`. When messages arrive, the `svc` routine performs the necessary service processing operations. Messages are forwarded to the next `Task` in a Stream by invoking the `Task::putnext` method, which subsequently calls the `put` method in the adjacent `Task`. This `put` routine will either borrow the thread of control from the `Task` that invoked it or it will enqueue the message for subsequent processing in its corresponding `svc` routine.

The `ASX` framework provides the basic service invocation mechanisms described above. However, an application is responsible for selecting an appropriate set of process architecture policies that combine these synchronous and/or asynchronous invocation mechanisms [1]. In general, selecting between these different mechanisms involves trade-offs between efficiency, ease of programming, and deadlock avoidance. For example, depending on application characteristics, available OS parallelism, and/or developer constraints (such as cross-platform portability), a `Task`, `Module`, and/or entire Stream may be bound to one or more threads of control in a flexible manner determined during Stream configuration.

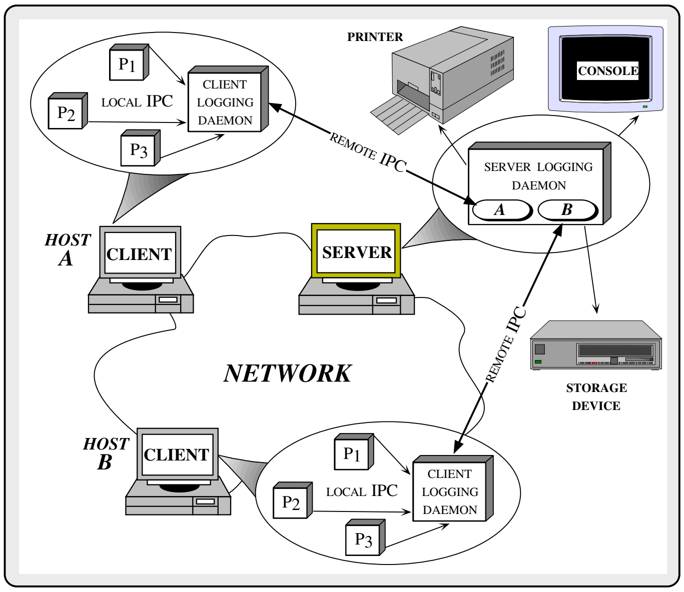

Figure 4: The Distributed Logging Facility

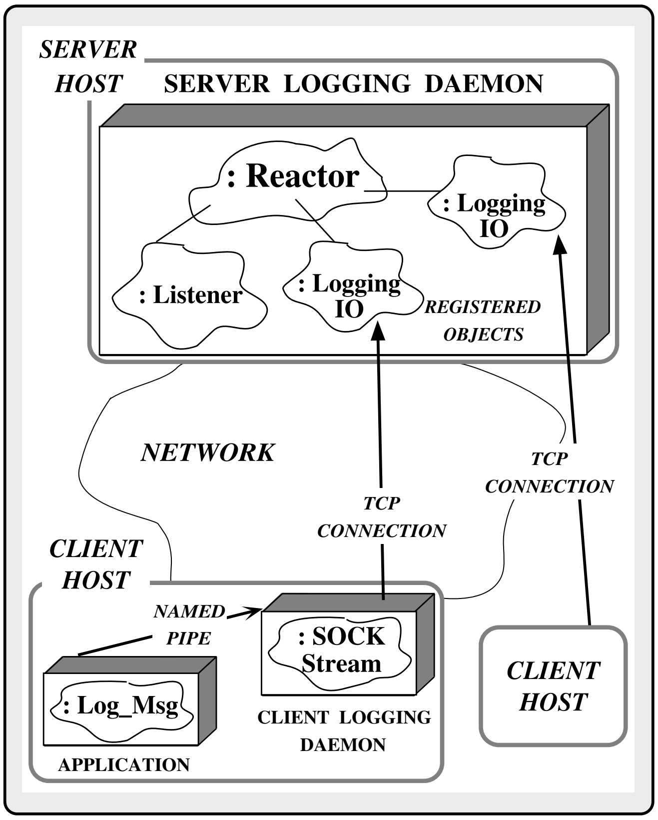

Figure 5: ASX Components in the Distributed Logging Facility

4 ASX Examples

The ASX framework components are currently being used in several research projects [17] and commercial projects [1] to enhance the configuration flexibility and software component reuse of distributed communication systems that operate efficiently and portably across multiple hardware and software platforms. The remainder of this section examines the architecture of two existing communication systems, a distributed logging facility and a device monitoring system for telecommunications devices. Note that the logging facility is an example of a *monolithic* service and the PBX monitoring system is an example of *layered* services operating within the ASX framework.

4.1 Distributed Logging Example

The ADAPTIVE Communication Environment (ACE) provides a distributed logging facility that simplifies distributed application debugging and run-time tracing. Debugging distributed communication systems may be quite challenging since diagnostic output appears in different windows and/or on remote host systems. As shown in Figure 4, the ACE distributed logging facility consists of several interoperating components located on multiple machines throughout an internetwork (the complete design and implementation of the distributed logging facility is described in detail in [25, 16]).

Application processes (*e.g.,* P_1, P_2, P_3) running on client hosts use the `Log_Msg` C++ class to generate various types of logging records (such as LOG_ERROR and LOG_DEBUG). The `Log_Msg::log` method provides a `printf`-style interface that timestamps logging records and sends them via the `FIFO_SAP` C++ wrapper for named pipes. The named pipe communicates with a *client logging daemon* running on the local host machine. This client logging daemon receives the logging records in "priority order" and uses another C++ wrapper (the `SOCK_Stream` class) to forward the records via a TCP/IP connection to a remote *server logging daemon* residing at a designated server host in a local and/or wide area network. The server logging daemon displays these records on one or more output devices (such as printers, persistent storage devices, and/or monitoring consoles).

The *server logging daemon* is a single-threaded concurrent server [26] that is built upon various ASX components such as the `Reactor`, the `Logging_Listener` class, and the `Logging_IO` class. The relationships between these components are illustrated in Figure 5. `Logging_Listener` is a template subclass that inherits from the `Service_Object` class (which enables it to be dynamically linked into the server logging daemon and initialized) and is parameterized by the `SOCK_Listener` class and the `Logging_IO` class (which itself is another template class that inherits from `Event_Handler` and is parameterized by the `SOCK_Stream` class). `Logging_Listener` is responsible for establishing connections with clients by dynamically creating `Logging_IO` objects and registering them with the `Reactor`. The `Logging_IO` class is responsible for displaying logging records sent from multiple client logging daemons on multiple client hosts. Decoupling the connection establishment and data transmission functionality into these two parameterized classes helps to improve the modularity, reusability, and configurability of the distributed logging components.

4.2 PBX Device Monitor

The ASX framework also facilitates the flexible configuration of communication systems containing layered services. Figure 6 illustrates the client/server architecture of a PBX monitoring system. This system is implemented using ASX components. In this example, the server host acts as an intermediate router, forwarding status information generated by one or more PBX devices to client monitoring hosts attached to a network. PBX devices may be attached to a server daemon through some form of communication link (*e.g.,* a serial-line or network connection) that interacts with the `Device_Adapter` class. Likewise, clients attach to the server by (1) establishing a connection with the `Client_Adapter` module and (2) indicating which type(s) of PBX signals they are interested in monitoring. The write-side of the `Client_Adapter` class accepts connection requests from clients and also forwards client-based control messages through the inter-connected write-side `Tasks` of the Stream to the appropriate PBX.

The read-side of the `Device_Adapter` class is responsible for parsing and transforming incoming device signals into a canonical event message format built atop the `Message_Block` class. These messages are then passed along to the read-side of the `Signal_Router` class, which identifies the client(s) that should receive the message based upon addressing tables maintained in the `Task`. Once the proper destination(s) are known, the read-side of the `Client_Adapter` class uses separate connections to transmit messages to all clients that have registered previously to receive these particular types of event messages.

An instance of `Service_Config` is used in this example to control the initialization and termination of the `Task` and `Module` components configured at installation-time and during run-time. Likewise, within the `Service_Config` object, an instance of the `Reactor` is used to dispatch incoming client messages to the appropriate `Client_IO` handler. The `Client_Listener` class handles connection requests from clients and the `Client_IO` class handles data transfer between the server and its clients. Control messages arriving from clients are sent down the write-side of the Stream, starting with the `Client_Adapter` and continuing through to the write-side of the `Device_Adapter`. Likewise, incoming signals from devices are sent to the read-side of the Stream, starting with the `Device_Adapter`.

The following C++ code fragment illustrates how the PBX monitoring application configures its hierarchically-related services:

```
Module *da = new Module ("Device_Adapter",
         new Device_Adapter, new Device_Adapter);
Module *ea = new Module ("Event_Analyzer",
         new Event_Analyzer, new Event_Analyzer);
Module *mr = new Module ("Multicast_Router",
         new Multicast_Router, new Multicast_Router);

STREAM PBX_mon;

/* Push the modules onto the application stream */

if (PBX_app.push (da) == -1)
  Log_Msg::log (LOG_ERROR "%p\n", sa->get_name ());
else if (PBX_app.push (ea) == -1)
  Log_Msg::log (LOG_ERROR "%p\n", ea->get_name ());
else if (PBX_app.push (mr) == -1)
  Log_Msg::log (LOG_ERROR "%p\n", ca->get_name ());
```

Earlier non-object-oriented incarnations of the PBX monitoring application used *ad hoc* techniques (such as linked lists, parameter passing, and shared memory) to exchange messages between the various related application services. In contrast, the `ASX` framework provides standard queueing mechanisms that compose and layer the hierarchically-related services together to form a complete server application. Using multiple inter-connected processing modules greatly simplifies portability and configurability. For example, it is relatively simple to migrate certain processing functionality from the server to the clients. Moreover, the `ASX` framework also shields the majority of the PBX application code from knowledge of the client/server interactions and the particular choice of communication protocols. In addition, it is also straight forward to reconfigure the binding of threads onto `Tasks` and `Modules` in order to reduce programming effort, ensure correct behavior, and improve performance.

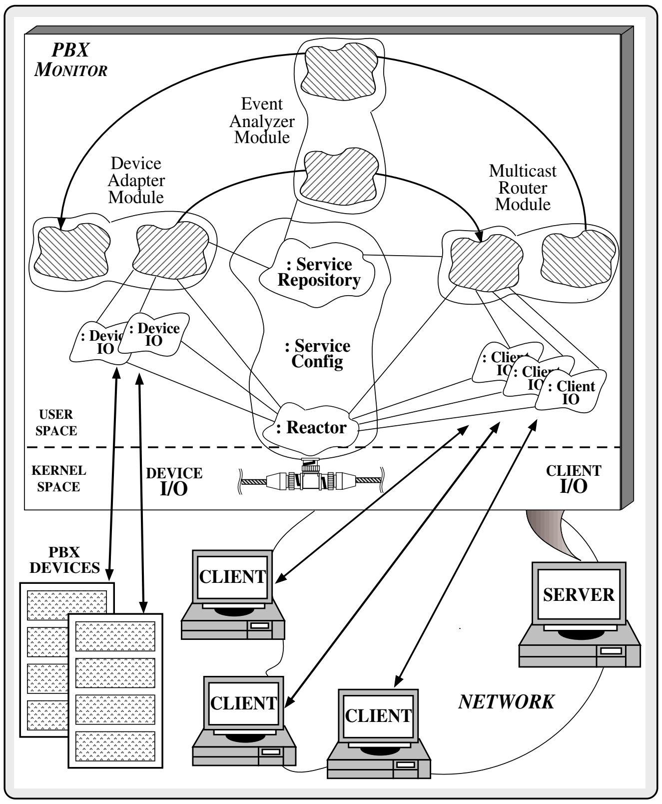

Figure 6: `ASX` Components for the PBX Monitor Application

5 Concluding Remarks

The `ASX` framework is an integral part of the ADAPTIVE Communication Environment (ACE) [17]. The goal of the ACE project is to produce an extensible framework that simplifies the development of concurrent, multi-service distributed communication systems composed of services tailored for particular application, OS platform, and network characteristics. To help achieve this goal, the `ASX` framework employs a variety of advanced operating system mechanisms, object-oriented design techniques, and C++ language features. In general, object-oriented techniques and C++ features enhance software quality factors (such as robustness, ease of use, portability, reusability, and extensibility), whereas OS features improve functionality and performance. In particular, the `ASX` framework automates many steps involved with configuring and reconfiguring services into distributed communication systems [1]; encapsulates local and remote IPC [15], distributed logging [25], I/O port demultiplexing and service dispatching [16]; and enables flexible invocation methods for intra- and inter-service concurrency [17].

The ASX components described in this paper are available via anonymous ftp from ics.uci.edu in the file gnu/C++_wrappers.tar.Z. This distribution contains complete source code, documentation, and example test drivers for the C++ components developed as part of the ACE project [17] at the University of California, Irvine. Components in the ASX framework have been ported to both UNIX and Windows NT and are currently being used in a number of commercial products including the Bellcore Q.port ATM signaling software product, the Ericsson EOS family of PBX monitoring applications, and the network management portion of the Motorola Iridium mobile communications system.

References

[1] D. C. Schmidt and T. Suda, "The Service Configurator Framework: An Extensible Architecture for Dynamically Configuring Concurrent, Multi-Service Network Daemons," in *Proceedings of the Second International Workshop on Configurable Distributed Systems*, (Pittsburgh, PA), pp. 190–201, IEEE, Mar. 1994.

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

[3] 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.

[4] J. M. Zweig, "The Conduit: a Communication Abstraction in C++," in *Proceedings of the* 2^{nd} *USENIX C++ Conference*, pp. 191–203, USENIX Association, April 1990.

[5] D. C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communications Systems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 489–506, May 1993.

[6] 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.

[7] J.Magee, J.Kramer, and M.Slomann, "Constructing Distributed Systems in Conic," *IEEE Transactions on Software Engineering*, vol. SE-15, June 1989.

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

[9] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation." *Software Practice & Experience*, to appear. An earlier version is available as Computer Science Division (EECS), University of California, Berkeley, Technical Report UCB/CSD 89/540, November 1989, 1991.

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

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

[12] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, June/July 1988.

[13] D. C. Schmidt, "The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications," in *Proceedings of the* 11^{th} *Annual Sun Users Group Conference*, (San Jose, CA), pp. 214–225, SUG, Dec. 1993.

[14] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

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

[16] 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.

[17] 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.

[18] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan, "Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 85–106, Jan. 1993.

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

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

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

[22] D. C. Schmidt and T. Suda, "A Framework for Developing and Experimenting with Parallel Process Architectures to Support High-Performance Transport Systems," in *Proceedings of the* 2^{nd} *Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, (Williamsburg, Virgina), IEEE, September 1993.

[23] M. Goldberg, G. Neufeld, and M. Ito, "A Parallel Approach to OSI Connection-Oriented Protocols," in *Proceedings of the* 3^{rd} *IFIP Workshop on Protocols for High-Speed Networks*, (Stockholm, Sweden), May 1992.

[24] A. Garg, "Parallel STREAMS: a Multi-Process Implementation," in *Proceedings of the Winter USENIX Conference*, (Washington, D.C.), Jan. 1990.

[25] 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.

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