ASX: An Object-Oriented Framework for Developing Distributed Applications

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

An earlier version of this paper appeared in the Proceedings of the 6^{th} USENIX C++ Conference, Cambridge, MA, April 11-14, 1994.

Abstract

*The ADAPTIVE Service eXecutive (*ASX*) is a highly modular and extensible object-oriented framework that simplifies the development and configuration of distributed applications on shared memory multi-processor platforms. This paper describes the structure and functionality of the* ASX *framework's object-oriented architecture. In addition, the paper presents the results of performance experiments conducted using* ASX*-based implementations of connection-oriented and connectionless protocols from the TCP/IP protocol family. These experiments measure the performance impact of alternative methods for parallelizing communication protocol stacks. Throughout the paper, examples are presented to indicate how the use of object-oriented techniques facilitate application extensibility, component reuse, and performance enhancement.*

1 Introduction

Distributed computing is a promising technology for improving collaboration through connectivity and interworking; performance through parallel processing; reliability and availability through replication; scalability and portability through modularity; extensibility through dynamic configuration and reconfiguration; and cost effectiveness through resource sharing and open systems. Despite these benefits, distributed applications (such as on-line transaction processing systems, global mobile communication systems, distributed object managers, video-on-demand servers, and communication subsystem protocol stacks) are often significantly more complex to develop and configure than non-distributed applications.

A significant portion of this complexity arises from limitations with conventional tools and techniques used to develop distributed application software. Conventional application development environments (such as UNIX, Windows NT, and OS/2) lack type-safe, portable, re-entrant, and extensible system call interfaces and component libraries. For instance, endpoints of communication in the widely used socket network programming interface are identified via weakly-typed I/O descriptors that increase the potential for subtle run-time errors [1]. Another major source of complexity arises from the widespread use of development techniques based upon algorithmic decomposition [2], which limit the extensibility, reusability, and portability of distributed applications.

Object-oriented techniques offer a variety of principles, methods, and tools that help to alleviate much of the complexity associated with developing distributed applications. To illustrate how these techniques are being successfully applied in several research and commercial settings, this paper describes the structure and functionality of the ADAPTIVE Service eXecutive (ASX). ASX is an object-oriented framework containing automated tools and reusable components that collaborate to simplify the development, configuration, and reconfiguration of distributed applications on shared memory multi-processor platforms.

Components in the ASX framework are designed to decouple (1) application-independent components provided by the framework that handle interprocess communication, event demultiplexing, explicit dynamic linking, concurrency, and service configuration from (2) application-specific components inherited or instantiated from the framework that perform the services in a particular distributed application. The primary unit of configuration in the ASX framework is the *service*. A service is a portion of a distributed application that offers a single processing capability to communicating entities. Services may be simple (such as returning the current time-of-day) or highly complex (such as a real-time distributed PBX event traffic monitor [3]). By employing object-oriented techniques to decouple the application-specific service functionality from the reusable application-independent framework mechanisms, ASX facilitates the development of applications that are significantly more extensible and portable than those based on conventional algorithmic decomposition techniques. For example, it is possible to dynamic reconfigure one or more services in an ASX-based application without requiring the modification, recompilation, relinking, or restarting of a running system [4].

In addition to describing the object-oriented architecture of the ASX framework, this paper examines results obtained by using the framework to conduct experiments on protocol stack performance in multi-processor-based communication subsystems. In the experiments, the ASX components help control for several relevant confounding factors (such as protocol functionality, concurrency control schemes, and application traffic characteristics) in order to precisely measure the performance impact of different methods for parallelizing communication protocol stacks. For example, in the experiments described in Section 3, connectionless and connection-oriented protocol stacks were developed by specializing existing components in the ASX framework via techniques involving inheritance and parameterized types. These techniques hold the protocol functionality constant while allowing the parallel processing structure of the protocol stacks to be altered systematically in a controlled manner.

This paper is organized as follows: Section 2 outlines the primary features of the ASX framework and describes its object-oriented architecture, Section 3 examines empirical results from experiments conducted using the framework to parallelize communication protocol stacks; and Section 4 presents concluding remarks.

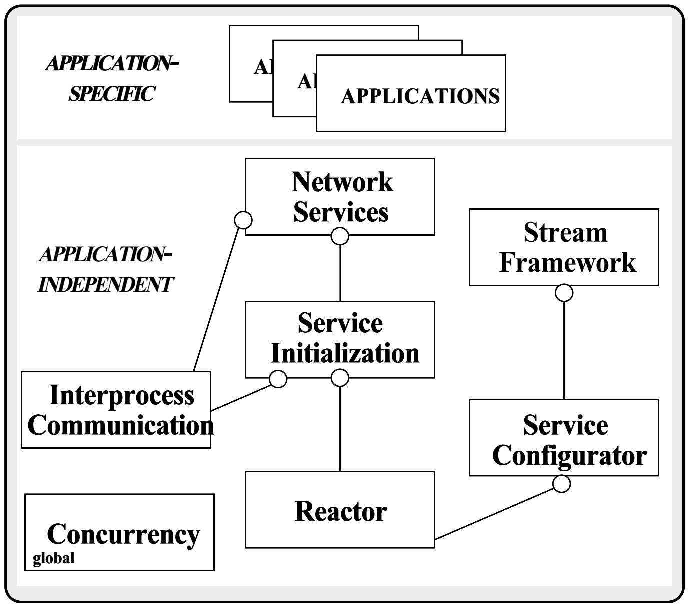

Figure 1: Class Categories in the ASX Framework

2 The ADAPTIVE Service eXecutive Framework

2.1 Overview

The ADAPTIVE Server eXecutive (ASX) is an object-oriented framework that is specifically targeted for the domain of distributed applications. The framework simplifies the construction of distributed applications by improving the modularity, extensibility, reusability, and portability of both the application-specific network services and the application-independent OS interprocess communication (IPC), demultiplexing, explicit dynamic linking, and concurrency mechanisms that these services utilize.

A framework is an integrated collection of components that collaborate to produce a reusable architecture for a family of related applications [5]. Object-oriented frameworks are becoming increasingly popular as a means to simplify and automate the development and configuration process associated with complex application domains such as graphical user interfaces [6], databases [7], operating system kernels [8], and communication subsystems [9]. The components in a framework typically include *classes* (such as message managers, timer-based event managers, demultiplexers [10], and assorted protocol functions and mechanisms [11]), *class hierarchies* (such as an inheritance lattice of mechanisms for local and remote interprocess communication [1]), *class categories* (such as event demultiplexers [12]), and *objects* (such as a service dispatch table). By emphasizing the integration and collaboration of application-specific and application-independent components, frameworks enable larger-scale reuse of software compared with simply reusing individual classes or stand-alone functions.

The ASX framework incorporates concepts from several other modular communication frameworks including System V STREAMS [13], the *x*-kernel [14], and the Conduit [9] (a survey of these and other communication frameworks appears in [15]). These frameworks all contain features that support the flexible configuration of communication subsystems by inter-connecting building-block protocol and service components. In general, these frameworks encourage the development of standard reusable communication-related components by decoupling application-specific processing functionality from the surrounding framework infrastructure. As described below, the ASX framework also contains additional features that help to further decouple application-specific service functionality from (1) the type of locking mechanisms used to synchronize access to shared objects, (2) the use of message-based vs. task-based parallel processing techniques, and (3) the use of kernel-level vs. user-level execution agents.

2.2 The Object-Oriented Architecture of ASX

The architecture of the ASX framework was developed incrementally by generalizing from extensive design and implementation experience with a range of distributed applications including on-line transaction processing systems [16], telecommunication switch performance monitoring systems [4], and multi-processor-based communication subsystems [17]. After building several prototypes and iterating through a number of alternative designs, the class categories illustrated in Figure 1 were identified and implemented. A class category is a collection of components that collaborate to provide a set of related services [2] such as communication subsystem services used to implement protocol stacks. A complete distributed application may be formed by combining components in each of the following class categories via C++ language features such as inheritance, aggregation, and template instantiation:

- The `Stream` class category – These components are responsible for coordinating the *configuration* and run-time *execution* of a Stream, which is an object containing a set of hierarchically-related services (such as the layers in a communication protocol stack) defined by an application
- The `Reactor` class category – These components are responsible for *demultiplexing* temporal events generated by a timer-driven callout queue, I/O events received on communication ports, and signal-based events and *dispatching* the appropriate pre-registered handler(s) to process these events
- The `Service Configurator` class category – These components are responsible for *dynamically linking* or *dynamically unlinking* services into or out of the address space of an application at run-time
- The `Concurrency` class category – These components are responsible for *spawning, executing, synchronizing,* and *gracefully terminating* services at run-time via one or more threads of control within one or more processes
- The `IPC SAP` class category – These components encapsulate standard OS local and remote IPC mechanisms (such as sockets and TLI) within a type-safe and portable object-oriented interface

Lines connecting the class categories in Figure 1 indicate dependency relationships. For example, components that implement the application-specific services in a particular distributed application depend on the `Stream` components, which in turn depend on the `Service Configurator` components. Since components in the `Concurrency` class category are used throughout the application-specific and application-independent portions of the ASX framework they are marked with the **global** adornment. Note that the "namespaces" feature accepted recently by the ANSI C++ committee provides explicit C++ language support for these types of class category relationships.

This section examines the main components in each class category. Relationships between components in the ASX framework are illustrated throughout the paper via Booch notation [2]. Solid rectangles indicate class categories, which combine a number of related classes into a common name space. Solid clouds indicate objects; nesting indicates composition relationships between objects; and undirected edges indicate some type of link exists between two objects. Dashed clouds indicate classes; directed edges indicate inheritance relationships between classes; and an undirected edge with a small circle at one end indicates either a composition or uses relation between two classes.

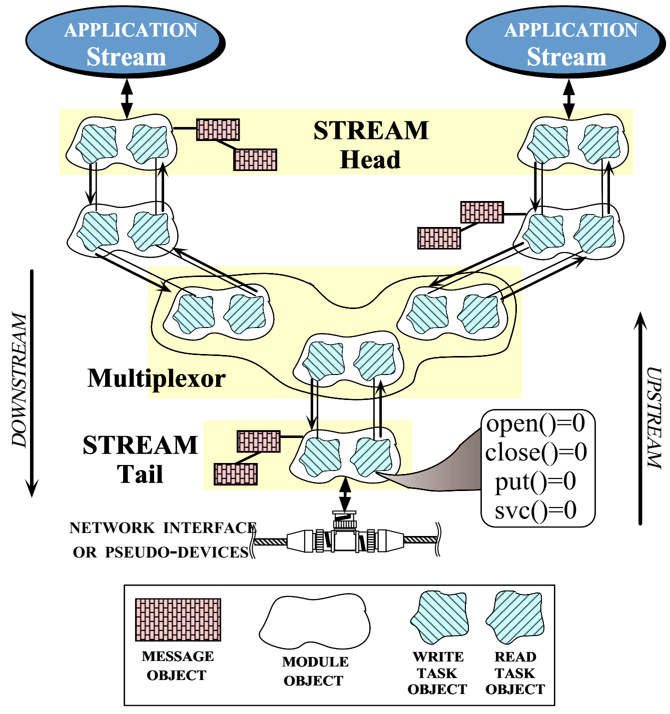

Figure 2: Components in the `Stream` Class Category

2.3 The Stream Class Category

Components in the `Stream` class category are responsible for coordinating one or more Streams. A Stream is an object used to configure and execute application-specific services into the ASX framework. As illustrated in Figure 2, a Stream uses both inheritance and object composition to link together a series of service-specific `Modules`. `Modules` are objects that developers use to decompose the architecture of a distributed application into a series of inter-connected, functionally distinct layers. Each layer implements a cluster of related service-specific functions (such as an end-to-end transport service, a presentation layer formatting service, or a real-time PBX signal routing service).

A layer that performs multiplexing and demultiplexing of message objects between one or more related Streams may be developed using a `Multiplexor` object. A `Multiplexor` is a C++ template container class that provides mechanisms to route messages between one or more `Modules` in a collection of related Streams. `Module` and `Multiplexor` objects may be configured into a Stream by developers at installation-time or by applications at run-time.

Every `Module` contains a pair of `Task` objects that partition a layer into its constituent read-side and write-side service-specific processing functionality. A `Task` provides an abstract domain class that may be specialized to target a particular application-specific domain (such as the domain of communication protocol stacks [18] or the domain of network management applications [19]). Likewise, a `Module` provides a flexible composition mechanism that allows instances of the application-specific Task domain classes to be configured dynamically into a Stream. These mechanisms increase the extensibility and applicability of the ASX framework by not limiting it to a particular domain.

A complete Stream is represented as an inter-connected series of independent Module and/or Multiplexor objects that communicate by exchanging messages with adjacent objects. Modules and Multiplexors may be joined together in essentially arbitrary configurations in order to satisfy application requirements and enhance component reuse. Service-specific functions in adjacent inter-connected Modules collaborate by exchanging typed messages via a well-defined message passing interface.

The ASX framework employs a number of object-oriented design techniques (such as design patterns [20] and hierarchical decomposition) and C++ language features (such as inheritance, dynamic binding, and parameterized types [21]). These techniques and language features enable developers to incorporate service-specific functionality into a Stream without modifying the application-independent framework components. For example, incorporating a new layer of service functionality into a Stream involves the following steps:

1. Inheriting from the Task interface and selectively overriding several methods (described below) in the Task subclass to implement service-specific functionality
2. Allocating a new Module that contains two instances (one for the read-side and one for the write-side) of the service-specific Task subclass
3. Inserting the Module into a Stream object

To avoid reinventing familiar terminology, many C++ class names in the Stream class category correspond to similar componentry available in the System V STREAMS framework. However, the techniques used to support extensibility and concurrency in the two frameworks are significantly different. For example, adding service-specific functionality to the ASX Stream classes is performed by inheriting from several interfaces and implementations defined by existing ASX framework components. Using inheritance to add service-specific functionality provides greater type-safety than the pointer-to-function idiom used in System V STREAMS. As described in Section 2.6.1 below, the ASX Stream classes also completely redesign and reimplement the co-routine-based, "weightless"[1] service processing mechanisms used in System V STREAMS. These ASX changes enable more effective use of multiple PEs on shared memory multi-processing platforms by reducing the likelyhood of deadlock and simplifying flow control between Tasks in a Stream.

The remainder of this section discusses the primary components of the ASX Stream class category (*i.e.*, Stream class, the Module class, the Task class, and the Multiplexor class) in detail.

[1] A weightless process executes on a run-time stack that is also used by other processes. This greatly complicates programming and increases the potential for deadlock. For example, a weightless process may not suspend execution to wait for resources to become available or events to occur [22].

2.3.1 The Stream Class

The Stream class defines the application interface to a Stream. A Stream object provides a bi-directional get/put-style interface that allows applications to access a stack of one or more hierarchically-related service Modules. Applications send and receive data and control messages through the inter-connected Modules that comprise a particular Stream object. The Stream class also implements a push/pop-style interface that allows applications to configure a Stream at run-time by inserting and removing objects of the Module class described below.

2.3.2 The Module Class

The Module class defines a distinct layer of service-specific functionality. A Stream is formed by inter-connecting a series of Module objects. Module objects in a Stream are loosely coupled, and collaborate with adjacent Module objects by passing typed messages. Each Module object contains a pair of pointers to objects that are service-specific subclasses of the Task class described shortly below.

As shown in Figure 2, two default Module objects (Stream Head and Stream Tail) are installed automatically when a Stream is opened. These two Modules interpret pre-defined ASX framework control messages and data messages that circulate through a Stream at run-time. The Stream Head class provides a message buffering interface between an application and a Stream. The Stream Tail class typically transforms incoming messages from a network or from a pseudo-device into a canonical internal message format that may be processed by higher-level components in a Stream. Likewise, for outgoing messages it transforms messages from their internal format into network messages.

2.3.3 The Task Abstract Class

The Task abstract class[2] defines an interface that is inherited and implemented by derived classes to provide service-specific functionality for read-side and write-side processing. One Task subclass handles read-side processing for messages sent upstream to its Module layer and the other handles write-side processing messages send downstream to its Module layer.

The Task class is an abstract class since its interface defines four pure virtual methods (open, close, put, and svc) that are described below. Defining Task as an abstract class enhances reuse by decoupling the application-independent components provided by the Stream class category from the service-specific subclasses that inherit from and use these components. Likewise, the use of pure virtual methods allows the C++ compiler to ensure that a subclass of

[2] An abstract class in C++ provides an interface that contains at least one *pure virtual method* [23]. A pure virtual method provides only an interface declaration, usually without any accompanying definition. Subclasses of an abstract class must provide definitions for all its pure virtual methods before any objects of the class may be instantiated.

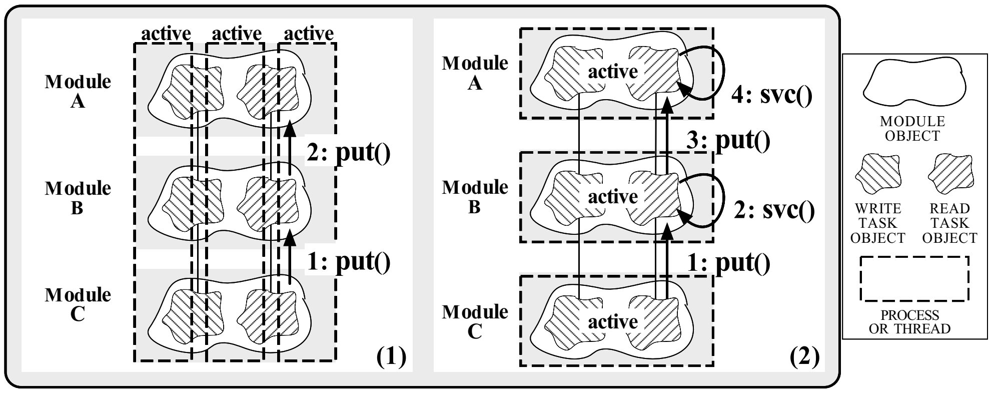

Figure 3: Alternative Methods for Invoking `put` and `svc` Methods

`Task` honors its obligation to provide the following service-specific functionality:

• **Initialization and Termination Methods:** Subclasses derived from `Task` must implement `open` and `close` methods that perform service-specific `Task` initialization and termination activities. These activities typically allocate and free resources such as connection control blocks, I/O descriptors, and synchronization locks. The `open` and `close` methods of a `Module`'s write-side and read-side `Task` subclasses are invoked automatically by the `ASX` framework when the `Module` is inserted or removed from a Stream, respectively.

• **Service-Specific Processing Methods:** Subclasses of `Task` must also define the `put` and `svc` methods, which perform service-specific processing functionality on messages that arrive at a `Module` layer in a Stream. When messages arrive at the head or the tail of a Stream, they are escorted through a series of inter-connected `Tasks` as a result of invoking the `put` and/or `svc` method of each `Task` in the Stream.

A `put` method is invoked when a `Task` at one layer in a Stream passes a message to an adjacent `Task` in another layer. The `put` method runs *synchronously* with respect to its caller, *i.e.,* it borrows the thread of control from the `Task` that originally invoked its `put` method. This thread of control typically originates either "upstream" from an application process, "downstream" from a pool of threads that handle I/O device interrupts [14], or internal to the Stream from an event dispatching mechanism (such as a timer-driven callout queue used to trigger retransmissions in a connection-oriented transport protocol `Module`).

The `svc` method is used to perform service-specific processing *asynchronously* with respect to other `Task`s in its Stream. Unlike `put`, the `svc` method is not directly invoked from an adjacent `Task`. Instead, it is invoked by a separate thread associated with its `Task`. This thread provides an execution context and thread of control for the `Task`'s `svc` method. This method runs an event loop that continuously waits for messages to arrive on the `Task`'s `Message Queue`. A `Message Queue` is a standard component in a `Task` that is used to buffer a sequence of data messages and control messages for subsequent processing in the `svc` method. When messages arrive, the `svc` method dequeues the messages and performs the `Task` subclass's service-specific processing tasks.

Within the implementation of a `put` or `svc` method, a message may be forwarded to an adjacent `Task` in the Stream via the `put_next Task` utility method. The `put_next` method calls the `put` method of the next `Task` residing in an adjacent layer. This invocation of `put` may borrow the thread of control from the caller and handle the message immediately (*i.e.,* the synchronous processing approach illustrated in Figure 3 (1)). Conversely, the `put` method may enqueue the message and defer handling to its `svc` method that is executing in a separate thread of control (*i.e.,* the asynchronous processing approach illustrated in Figure 3 (2)). As discussed in Section 3, the particular processing approach that is selected often has a significant impact on performance and ease of programming.

In addition to the `open`, `close`, `put`, and `svc` pure virtual method interfaces, each `Task` also contains a number of reusable utility methods (such as `put_next`, `getq`, and `putq`) that may be used by service-specific subclasses to query and/or modify the internal state of a `Task` object. This internal state includes a pointer to the adjacent `Task` on a Stream, a back-pointer to a `Task`'s enclosing `Module` (which enables it to locate its sibling), a `Message Queue`, and a pair of high and low water mark variables that are used to implement layer-to-layer flow control between adjacent `Modules` in a Stream. The high water mark indicates the amount of bytes of messages the `Message Queue` is willing to buffer before it becomes flow controlled. The low water mark indicates the level at which a previously flow controlled `Task` is no longer considered to be full.

Two types of messages may appear on a `Message Queue`: simple and composite. A simple message contains a single `Message Block` and a composite message contains multiple `Message Blocks` linked together. Composite messages generally consist of a *control* block followed by one or more *data* blocks. A control block contains bookkeep-

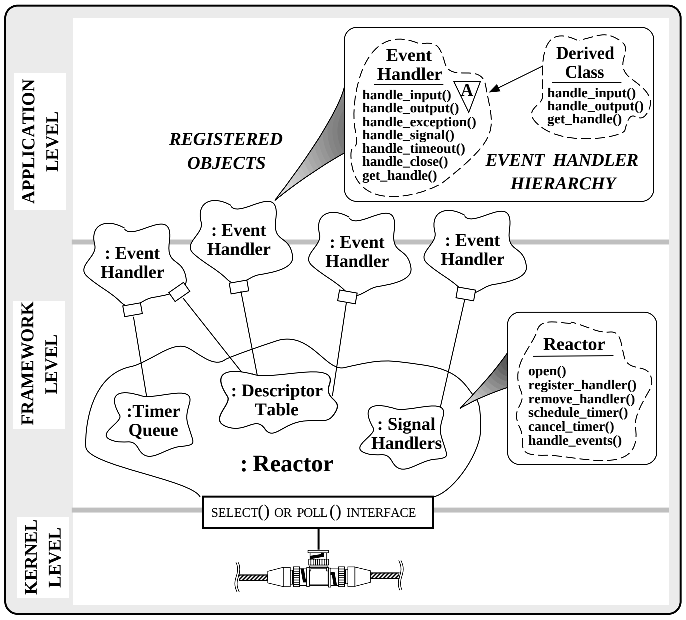

Figure 4: Components in the `Reactor` Class Category

ing information (such as destination addresses and length fields), whereas data blocks contain the actual contents of a message. The overhead of passing `Message Block`s between `Task`s is minimized by passing pointers to messages rather than copying data.

2.3.4 The Multiplexor Class

A `Multiplexor` defines mechanisms that demultiplex messages destined for different `Modules` that comprise a set of inter-related Streams. `Multiplexor`s are used to route `Message Block`s between inter-related streams (such as those used to implement complex protocol families in the Internet and the ISO OSI reference models). A `Multiplexor` is implemented via a C++ template class called `Map Manager`. `Map Manager` is parameterized by an *external identifier* (such as a network address, port number, or type-of-service field) and an *internal identifier* (such as a pointer to a `Module`). These template parameters are instantiated in protocol-specific `Stream` classes to produce specialized `Map Manager` objects that perform efficient intra-Stream message routing. Each `Map Manager` object contains a set of `Module` objects that may be linked above and below a `Multiplexor` in essentially arbitrary configurations.

2.4 The Reactor Class Category

Components in the `Reactor` class category are responsible for demultiplexing (1) temporal events generated by a timer-driven callout queue, (2) I/O events received on communication ports, and (3) signal events and dispatching the appropriate pre-registered handler(s) to process these events. The `Reactor` encapsulates the functionality of the `select` and `poll` I/O demultiplexing mechanisms within a portable and extensible C++ wrapper [12]. `Select` and `poll` are UNIX system calls that detect the occurrence of different types of input and output events on one or more I/O descriptors simultaneously. To improve portability, the `Reactor` provides the same interface regardless of whether `select` or `poll` is used as the underlying I/O demultiplexor. In addition, the `Reactor` contains mutual exclusion mechanisms designed to perform callback-style programming correctly and efficiently in a multi-threaded event processing environment.

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

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

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

2.5 The Service Configurator Class Category

Components in the `Service Configurator` class category are responsible for explicitly linking or unlinking ser-

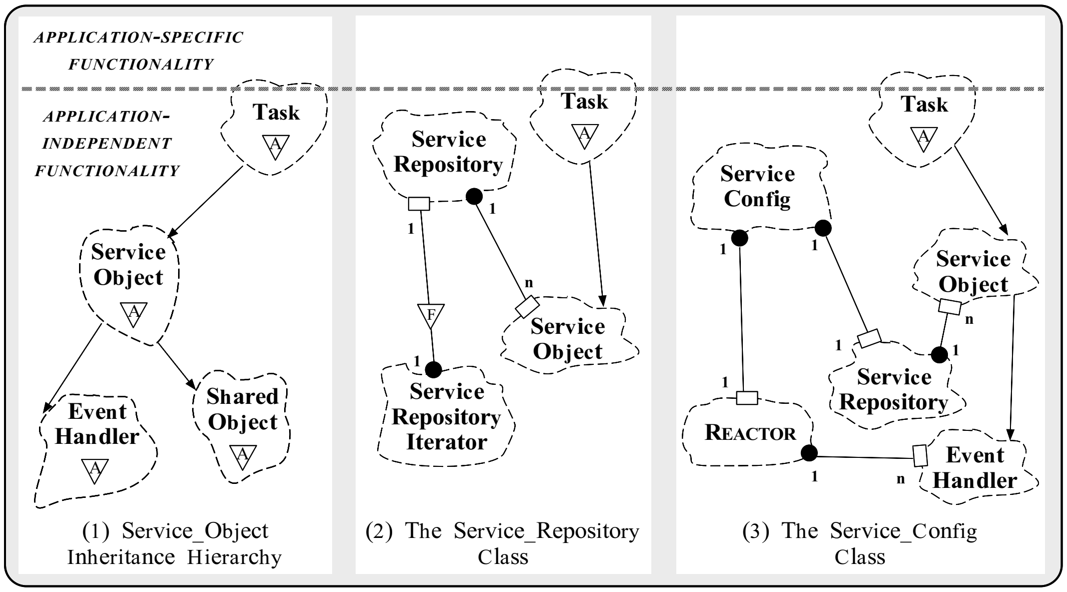
APPLICATION-SPECIFIC FUNCTIONALITY
APPLICATION-INDEPENDENT FUNCTIONALITY
Task
Service Object
Event Handler
Shared Object
(1) Service_Object Inheritance Hierarchy
Service Repository
Service Object
Service Repository Iterator
(2) The Service_Repository Class
Service Config
Reactor
Event Handler
(3) The Service_Config Class

Figure 5: Components in the `Service Configurator` Class Category

vices dynamically into or out of the address space of an application at run-time. Explicit dynamic linking enables the configuration and reconfiguration of application-specific services without requiring the modification, recompilation, relinking, or restarting of an executing application [4]. The `Service Configurator` components discussed below include the the `Service Object` inheritance hierarchy (Figure 5 (1)), the `Service Repository` class (Figure 5 (2)), and the `Service Config` class (Figure 5 (3)).

2.5.1 The Service Object Inheritance Hierarchy

The `Service Object` class is the focal point of a multi-level hierarchy of types related by inheritance. The interfaces provided by the abstract classes in this type hierarchy may be selectively implemented by service-specific subclasses in order to access `Service Configurator` features. These features provide transparent dynamic linking, service handler registration, event demultiplexing, service dispatching, and run-time control of services (such as suspending and resuming a service temporarily). By decoupling the service-specific portions of a handler object from the underlying `Service Configurator` mechanisms, the effort necessary to insert and remove services from an application at run-time is significantly reduced.

The `Service Object` inheritance hierarchy consists of the `Event Handler` and `Shared Object` abstract base classes, as well as the `Service Object` abstract derived class. The `Event Handler` class was described above in the `Reactor` Section 2.4. The behavior of the other classes in the `Service Configurator` class category is outlined below:

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

The `init` method serves as the entry-point to an object during run-time initialization. This method is responsible for performing application-specific initialization when an object derived from `Shared Object` is dynamically linked. The `info` method returns a humanly-readable string that concisely reports service addressing information and documents service functionality. Clients may query an application to retrieve this information and use it to contact a particular service running in the application. The `fini` method is called automatically by the `Service Configurator` class category when an object is unlinked and removed from an application at run-time. This method typically performs termination operations that release dynamically allocated resources (such as memory or synchronization locks).

The `Shared Object` base class is defined independently from the `Event Handler` class to clearly separate their two orthogonal sets of concerns. For example, certain applications (such as a compiler or text editor) might benefit from dynamic linking, though they might not require timer-based, signal-based, or I/O descriptor-based event demultiplexing. Conversely, other applications (such as an `ftp` server) require event demultiplexing, but might not require dynamic linking.

• **The Service Object Abstract Derived Class:** Support for dynamic linking, event demultiplexing, and service dispatching is typically necessary to automate the dynamic configuration and reconfiguration of application-specific services in a distributed system. Therefore, the `Service Configurator` class category defines the `Service Object` class, which is a composite class that combines the interfaces inherited from both the `Event Handler` and the `Shared Object` abstract base classes. During development, application-specific subclasses of `Service Object` may implement the `suspend` and `resume` virtual methods in this class. The `suspend` and `resume` methods are invoked automatically by the `Service Configurator` class category in response to certain external events (such as those triggered by receipt of the UNIX SIGHUP signal). An application developer may define these methods to perform actions necessary to suspend a service object without unlinking it completely, as well as to resume a previously suspended service object. In addition, application-specific subclasses must implement the four pure virtual methods (`init`, `fini`, `info`, and `get_handle`) that are inherited (but not defined) by the `Service Object` subclass.

To provide a consistent environment for defining, configuring, and using Streams, the `Task` class in the `Stream` class category is derived from the `Service Object` inheritance hierarchy (illustrated in Figure 5 (1)). This enables hierarchically-related, application-specific services to be linked and unlinked into and out of a Stream at run-time.

2.5.2 The Service Repository Class

The `ASX` framework supports the configuration of applications that contain one or more Streams, each of which may have one or more inter-connected service-specific `Modules`. Therefore, to simplify run-time administration, it may be necessary to individually and/or collectively control and coordinate the `Service Object`s that comprise an application's currently active services. The `Service Repository` is an object manager that coordinates local and remote queries and updates involving the services offered by an application. A search structure within the object manager binds service names (represented as ASCII strings) with instances of composite `Service Object`s (represented as C++ object code). A service name uniquely identifies an instance of a `Service Object` stored in the repository.

Each entry in the `Service Repository` contains a pointer to the `Service Object` portion of an service-specific C++ derived class (shown in Figure 5 (2)). This enables the `Service Configurator` classes to automatically load, enable, suspend, resume, or unload `Service Object`s from a Stream dynamically. The repository also maintains a handle to the underlying shared object file for each dynamically linked `Service Object`. This handle is used to unlink and unload a `Service Object` from a running application when its service is no longer required. An iterator class is also supplied along with the `Service Repository`. This class may be used to visit every `Service Object` in the repository without compromising data encapsulation.

2.5.3 The Service Config Class

As illustrated in Figure 5 (3), the `Service Config` class integrates several other `ASX` framework components (such as the `Service Repository`, the `Service Object` inheritance hierarchy, and the `Reactor`). The resulting composite `Service Config` component is used to automate the static and/or dynamic configuration of concurrent applications that contain one or more Streams. The `Service Config` class uses a configuration file to guide its configuration and reconfiguration activities. Each application may be associated with a distinct configuration file. This file characterizes the essential attributes of the service(s) offered by an application. These attributes include the location of the shared object file for each dynamically linked service, as well as the parameters required to initialize a service at run-time. By consolidating service attributes and installation parameters into a single configuration file, the administration of Streams within an application is simplified. Application development is also simplified by decoupling the configuration and reconfiguration mechanisms provided by the framework from the application-specific attributes and parameters specified in a configuration file. Further information on the configuration format utilized by the `Service Config` class is presented in [4].

2.6 The Concurrency Class Category

Components in the `Concurrency` class category are responsible for spawning, executing, synchronizing, and gracefully terminating services at run-time via one or more threads of control within one or more processes. The following section discusses the two main groups of classes (`Synch` and `Thread Manager`) in the `Concurrency` class category.

2.6.1 The Synch Classes

Components in the `Stream`, `Reactor`, and `Service Configurator` class categories described above contain a minimal amount of internal locking mechanisms to avoid over-constraining the granularity of the synchronization strategies used by an application [22]. In particular, only components in the `ASX` framework that would not function correctly in a multi-threaded environment (such as enqueueing `Message Blocks` onto a `Message Queue`, demultiplexing `Message Blocks` onto internal `Module` addresses stored in a `Multiplexor` object, or registering an `Event Handler` object with the `Reactor`) are protected by synchronization mechanisms provided by the

Synch classes. The Synch classes provide type-safe C++ interfaces for two basic types of synchronization mechanisms: Mutex and Condition objects [24]. A Mutex object is used to ensure the integrity of a shared resource that may be accessed concurrently by multiple threads of control. A Condition object allows one or more cooperating threads to suspend their execution until a condition expression involving shared data attains a particular state. The ASX framework also provides a collection of more sophisticated concurrency control mechanisms (such as Monitors, Readers Writer locks, and recursive Mutex objects) that build upon the two basic synchronization mechanisms described below.

A Mutex object may be used to serialize the execution of multiple threads by defining a critical section where only one thread executes its code at a time. To enter a critical section, a thread invokes the Mutex::acquire method. To leave a critical section, a thread invokes the Mutex::release method. These two methods are implemented via adaptive spin-locks that ensure mutual exclusion by using an atomic hardware instruction. An adaptive spin-lock operates by polling a designated memory location using the hardware instruction until (1) the value at this location is changed by the thread that currently owns the lock (signifying that the lock has been released and may now be acquired) or (2) the thread that is holding the lock goes to sleep (at which point the thread that is spinning also goes to sleep to avoid needless polling) [25]. On a shared memory multi-processor, the overhead incurred by a spin-lock is relatively minor since polling affects only the local instruction and data cache of the CPU where the thread is spinning. A spin-lock is a simple and efficient synchronization mechanism for certain types of short-lived resource contention. For example, in the ASX framework, each Message Queue in a Task object contains a Mutex object that prevents race conditions from occurring when Message Blocks are enqueued and dequeued concurrently by multiple threads of control running in adjacent Tasks.

A Condition object is a somewhat different synchronization mechanism that enables a thread to suspend itself indefinitely (via the Condition::wait method) until a condition expression involving shared data attains a particular state. When another cooperating thread indicates that the state of the shared data has changed (by invoking the Condition::signal method), the associated Condition object wakes up the suspended thread. The newly awakened thread then re-evaluates the condition expression and potentially resumes processing if the shared data is now in an appropriate state. For example, each Message Queue in the ASX framework contains a pair of Condition objects (named notfull and notempty), in addition to a Mutex object. These Condition objects implement flow control between adjacent Tasks. When one Task attempts to insert a Message Block into a neighboring Task that has reached its high water mark, the Message Queue::enTask method performs a wait operation on the notfull condition object. This operation

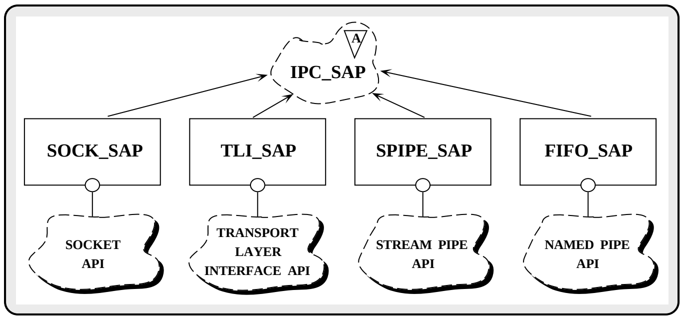

Figure 6: Components in the IPC SAP Class Category

atomically relinquishes the PE and puts the calling thread to sleep awaiting notification when flow control conditions abate. Subsequently, when the number of bytes in the flow controlled Task's Message Queue fall below its low water mark, the thread running the blocked Task is automatically awakened to finish inserting the message and resume its processing tasks.

Unlike Mutex objects, Condition object synchronization is not implemented with a spin-lock since there is generally no indication of how long a thread must wait for a particular condition to be signaled. Therefore, Condition objects are implemented via sleep-locks that trigger a context switch to allow other threads to execute. Section 3 discusses the consequences of spin-locks vs. sleep-locks on application performance.

2.6.2 The Thread Manager Class

The Thread Manager class contains a set of mechanisms that manage groups of threads that collaborate to implement collective actions (such as a pool of threads that render different portions of a large image in parallel). The The Manager class shields applications from many incompatibilities between different flavors of multi-threading mechanisms (such as POSIX threads, MACH cthreads, and Solaris threads).

In addition, the Thread Manager class provides a number of mechanisms (such as suspend_all and resume_all) that suspend and resume a set of collaborating threads atomically. This feature is useful for distributed applications that execute one or more services concurrently. For example, when initializing a Stream composed of Modules that execute in separate threads of control and collaborate by passing messages between threads, it is important to ensure that all Tasks in the Stream are completely inter-connected before allowing messages to flow through the Stream. The mechanisms in the Thread Manager class allow these initialization activities to occur atomically.

2.7 The IPC SAP Class Category

Components in the `IPC SAP` class category encapsulate standard OS local and remote IPC mechanisms (such as sockets and TLI) within a type-safe and portable object-oriented interface. `IPC SAP` stands for "InterProcess Communication Service Access Point." As shown in Figure 6, a forest of class categories are rooted at the `IPC SAP` base class. These class categories includes `SOCK SAP` (which encapsulates the socket API), `TLI SAP` (which encapsulates the TLI API), `SPIPE SAP` (which encapsulates the UNIX SVR4 STREAM pipe API), and `FIFO SAP` (which encapsulates the UNIX named pipe API).

Each class category in `IPC SAP` is itself organized as an inheritance hierarchy where every subclass provides a well-defined subset of local or remote communication mechanisms. Together, the subclasses within a hierarchy comprise the overall functionality of a particular communication abstraction (such as the Internet-domain or UNIX-domain protocol families). Inheritance-based hierarchical decomposition facilitates the reuse of code that is common among the various `IPC SAP` class categories. For example, the C++ interface to the lower-level UNIX OS device control system calls like `fcntl` and `ioctl` are inherited and shared by all the other components in the `IPC SAP` class category.

3 Performance Experiments on the Communication Subsystem

To illustrate how the components of the `ASX` framework are used in practice, this section describes results from performance experiments that measure the impact of alternative methods for parallelizing communication subsystems. A communication subsystem is a distributed system that consists of *protocol functions* (such as routing, segmentation/reassembly, connection management, end-to-end flow control, remote context management, demultiplexing, message buffering, error protection, session control, and presentation conversions) and *operating system mechanisms* (such as process management, asynchronous event invocation, message buffering, and layer-to-layer flow control) that support the implementation and execution of protocol stacks that contain hierarchically-related protocol functions [15].

Advances in VLSI and fiber optic technology are shifting performance bottlenecks from the underlying networks to the communication subsystem [26]. Designing and implementing multi-processor-based communication subsystems that execute protocol functions and OS mechanisms in parallel is a promising technique for increasing protocol processing rates and reducing latency. To significantly increase communication subsystem performance, however, the speed-up obtained from parallel processing must outweight the context switching and synchronization overhead associated with parallel processing.

A context switch is generally triggered when an executing process either voluntarily or involuntarily relinquishes the processing element (PE) it is executing upon. Depending on the underlying OS and hardware platform, performing a context switch may involve dozens to hundreds of instructions due to the flushing of register windows, instruction and data caches, instruction pipelines, and translation look-aside buffers [27]. Synchronization mechanisms are necessary to serialize access to shared objects (such as messages, message queues, protocol context records, and demultiplexing tables) related to protocol processing. Certain methods of parallelizing protocol stacks incur significant synchronization overhead from managing locks associated with processing these shared objects [28].

A number of *process architectures* have been proposed as the basis for parallelizing communication subsystems [26, 29, 28]. A process architecture binds one or more processing elements (PEs) together with the protocol tasks and messages that implement protocol stacks in a communication subsystem. Figure 7 (1) illustrates the three basic elements that form the foundation of a process architecture:

1. *Control messages and data messages* – which are sent and received from one or more applications and network devices
2. *Protocol processing tasks* – which are the units of protocol functionality that process the control messages and data messages
3. *Processing elements* (PEs) – which execute protocol tasks. There are two fundamental types of process architectures (*task-based* and *message-based*) that structure these three basic elements differently.

Two fundamental types of process architectures (*task-based* and *message-based*) may be created by structuring the three basic process architecture elements shown in Figure 7 (1) in different ways. Task-based process architectures are formed by binding one or more PEs to different units of protocol functionality (shown in Figure 7 (2)). In this architecture, tasks are the active objects, whereas messages processed by the tasks are the passive objects. Parallelism is achieved by executing protocol tasks in separate PEs and passing data messages and control messages between the tasks/PEs. In contrast, message-based process architectures are formed by binding the PEs to the protocol control messages and data messages received from applications and network interfaces (as shown in Figure 7 (3)). In this architecture, messages are the active objects, whereas tasks are the passive objects. Parallelism is achieved by escorting multiple data messages and control messages on separate PEs simultaneously through a stack of protocol tasks. Section 3 examines how the choice of process architecture significantly affects context switch and synchronization overhead. A survey of alternative process architectures appears in [15].

Selecting an effective process architecture is an important design decision in application domains other than communication subsystems. For example, real-time PBX monitoring systems [3] and video-on-demand servers also perform non-communication-related tasks (such as database query processing) that benefit from a carefully structured approach

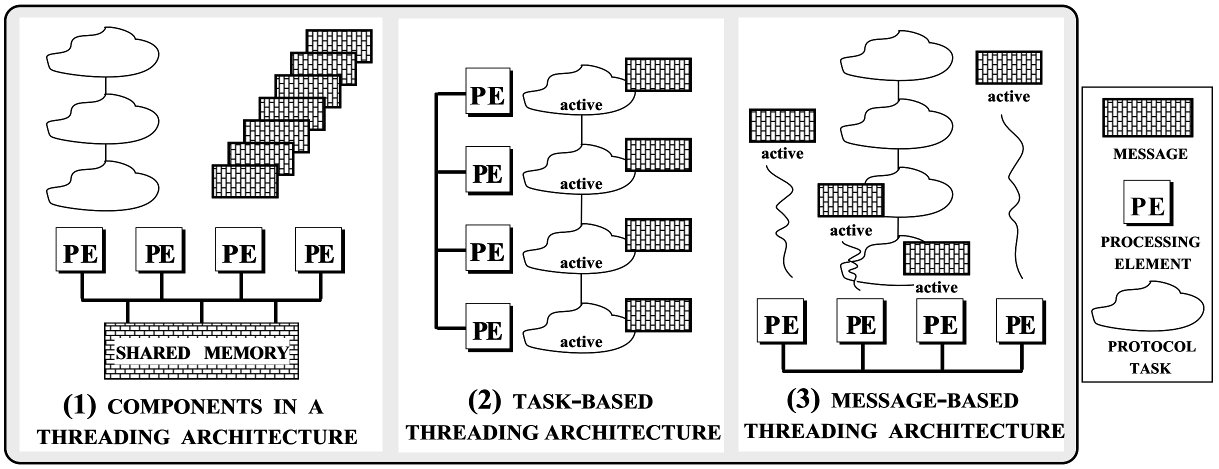
SHARED MEMORY
PE
(1) COMPONENTS IN A THREADING ARCHITECTURE
active
(2) TASK-BASED THREADING ARCHITECTURE
(3) MESSAGE-BASED THREADING ARCHITECTURE
MESSAGE
PROCESSING ELEMENT
PROTOCOL TASK

Figure 7: Process Architecture Components and Interrelationships

to parallelism. This section focuses primarily upon the impact of process architectures on communication subsystem performance since network protocol behavior and functionality is well-understood and the terminology is relatively well-defined. Moreover, a large body of literature exists with which to compare performance results presented in Section 3. The remainder of this section describes relevant aspects of performance experiments that measure the impact of different process architectures on connectionless and connection-oriented protocol stacks.

3.1 Multi-processor Platform

All experiments were conducted on an otherwise idle Sun 690MP SPARCserver, which contains 4 SPARC 40 MHz processing elements (PEs), each capable of performing at 28 MIPs. The operating system used for the experiments is release 5.3 of SunOS, which provides a multi-threaded kernel that allows multiple system calls and device interrupts to execute in parallel [25]. All the process architectures in these experiments execute protocol tasks in separate *unbound* threads multiplexed over 1, 2, 3, or 4 SunOS *lightweight processes* (LWPs) within a process. SunOS 5.3 maps each LWP directly onto a separate kernel thread. Since kernel threads are the units of PE scheduling and execution in SunOS, this mapping enables multiple LWPs (each executing protocol processing tasks in an unbound thread) to run in parallel on the SPARCserver's PEs.

Rescheduling and synchronizing a SunOS LWP involves a kernel-level context switch. The time required to perform a context switch between two LWPs was measured to be approximately 30 *u*secs. During this time, the OS performs system-related overhead (such as flushing register windows, instruction and data caches, instruction pipelines, and translation lookaside buffers) on the PE and therefore does not process protocol tasks. Measurements also revealed that it requires approximately 3 micro-seconds to acquire or release a `Mutex` object implemented with a SunOS adaptive spin-lock. Likewise, measurements indicated that approximately 90 micro-seconds are required to synchronize two LWPs using `Condition` objects implemented using SunOS sleep-locks. The larger amount of overhead for the `Condition` operations compared with the `Mutex` operations occurs from the more complex locking algorithms involved, as well as the additional context switching incurred by the SunOS sleep-locks that implement the `Condition` objects.

3.2 Communication Protocols

Two types of protocol stacks are used in the experiments, one based on the connectionless UDP transport protocol and the other based on the connection-oriented TCP transport protocol. The protocol stacks contain the data-link, transport, and presentation layers.[3] The presentation layer is included in the experiments since it represents a major bottleneck in high-performance communication systems due primarily to the large amount of data movement overhead it incurs [30, 29].

Both the connectionless and connection-oriented protocol stacks were developed by specializing existing components in the `ASX` framework via techniques involving inheritance and parameterized types. These techniques are used to hold the protocol stack functionality constant while systematically varying the process architecture. For example, each protocol layer is implemented as a `Module` whose read-side and write-side inherit standard interfaces and implementations from the `Task` class. Likewise, synchronization and demultiplexing mechanisms required by a protocol layer or protocol stack are parameterized using template arguments

[3] Preliminary tests indicated that the PE, bus, and memory performance of the SunOS multi-processor platform was capable of processing messages through the protocol stack at a much faster rate than the platform's 10 Mbps Ethernet network interface was capable of handling. Therefore, for the process architecture experiments, the network interface was simulated with a single-copy pseudo-device driver operating in loop-back mode. For this reason, the routing and segmentation/reassembly functions of the network layer processing were omitted from these experiments since both the sender and receiver portions of the test programs reside on the same host machine.

that are instantiated based on the type of process architecture being tested.

Data-link layer processing in each protocol stack is performed by the `DLP Module`. This `Module` transforms network packets received from a network interface or loop-back device into a canonical message format used internally by the Stream components. The transport layer component of the protocol stacks are based on the UDP and the TCP implementation in the BSD 4.3 Reno release. The 4.3 Reno TCP implementation contains the TCP header prediction enhancements, as well as the slow start algorithm and congestion avoidance features. The UDP and TCP transport protocols are configured into the `ASX` framework via the `UDP` and `TCP Modules`, respectively.

Presentation layer functionality is implemented in the `XDR Module` using marshalling routines produced by the ONC eXternal Data Representation (XDR) stub generator (`rpcgen`). The ONC XDR stub generator automatically translates a set of type specifications into marshalling routines that encode/decode implicitly-typed messages before/after they are exchanged among hosts that may possess heterogeneous processor byte-orders. The ONC presentation layer conversion mechanisms consist of a type specification language (XDR) and a set of library routines that implement the appropriate encoding and decoding rules for built-in integral types (*e.g.*, char, short, int, and long) and real types (*e.g.*, float and double). In addition, these library routines may be combined to produce marshalling routines for arbitrary user-defined composite types (such as record/structures, unions, arrays, and pointers). Messages exchanged via XDR are implicitly-typed, which improves marshalling performance at the expense of flexibility. The `XDR` functions selected for both the connectionless and connection-oriented protocol stacks convert incoming and outgoing messages into and from variable-sized arrays of structures containing both integral and real values. This conversion processing involves byte-order conversions, as well as dynamic memory allocation and deallocation.

3.3 Process Architectures

3.3.1 Design of the Task-based Process Architecture

Figure 8 illustrates the `ASX` framework components that implement a task-based process architecture for the TCP-based connection-oriented and UDP-based connectionless protocol stacks. Protocol-specific processing for the data-link and transport layer are performed in two `Modules` clustered together into one thread. Likewise, presentation layer and application interface processing is performed in two `Modules` clustered into a separate thread. These threads cooperate in a producer/consumer manner, operating in parallel on the header and data fields of multiple incoming and outgoing messages.

The `LP DLP::svc` and `LP XDR::svc` methods perform service-specific processing in parallel within a Stream of `Modules`. When messages are inserted into a `Task`'s

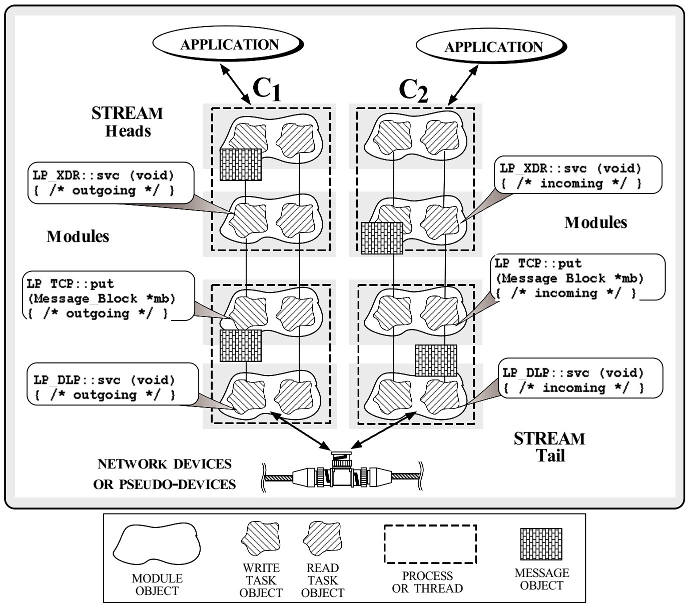
APPLICATION
APPLICATION
C1
C2
STREAM Heads
LP XDR::svc (void) { /* outgoing */ }
Modules
LP TCP::put (Message Block *mb) { /* outgoing */ }
LP DLP::svc (void) { /* outgoing */ }
LP XDR::svc (void) { /* incoming */ }
Modules
LP TCP::put (Message Block *mb) { /* incoming */ }
LP DLP::svc (void) { /* incoming */ }
STREAM Tail
NETWORK DEVICES OR PSEUDO-DEVICES
MODULE OBJECT
WRITE TASK OBJECT
READ TASK OBJECT
PROCESS OR THREAD
MESSAGE OBJECT

Figure 8: A Task-based Process Architecture

`Message Queue`, the `svc` method dequeues the messages and performs the `Task` subclass's service-specific processing tasks (such as data-link layer processing or presentation layer processing). Depending on the "direction" of a message (*i.e.*, incoming or outgoing), each cluster of `Modules` performs its associated protocol functions before passing the message to an adjacent `Module` running asynchronously in a separate thread. Messages are not copied when passed between adjacent `Tasks` since threads all share a common address space. However, moving messages between threads typically invalidates per-PE data caches.

The connectionless and connection-oriented task-based process architecture protocol stacks are designed in a similar manner. The primary difference is that the objects in the connectionless transport layer `Module` implement the simpler UDP functionality that does not generate acknowledgements, keep track of round-trip time estimates, or manage congestion windows. The design of the task-based process architecture test driver always uses PEs in multiples of two: one for the cluster of data-link and transport layer processing `Modules` and the other for the cluster of presentation layer and application interface processing `Modules`.

3.3.2 Design of the Message-based Process Architecture

Figure 9 illustrates a message-based process architecture for the connection-oriented protocol stack. When an incoming message arrives, it is handled by the `MP DLP::svc` method, which manages a pool of pre-spawned threads. Each message is associated with a separate thread that escorts the message synchronously through a series of interconnected `Tasks` in a Stream. Each layer of the protocol stack performs its protocol functions and then makes an upcall [31] to the next adjacent layer in the protocol stack by

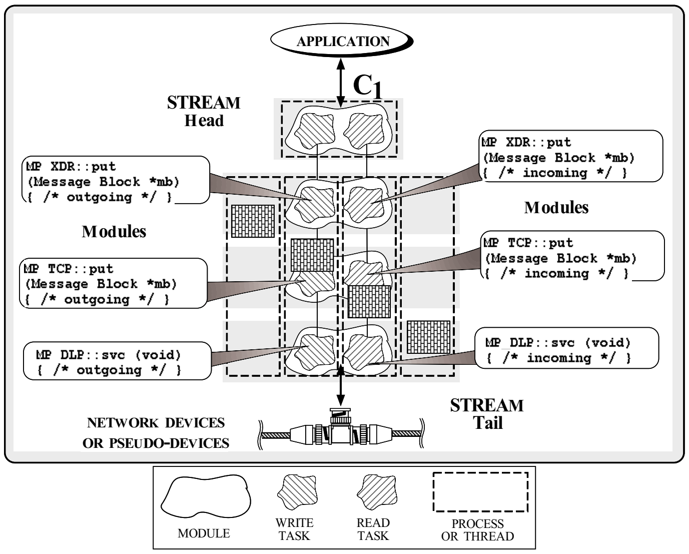
APPLICATION
C1
STREAM Head
MP XDR::put (Message Block *mb) { /* outgoing */ }
MP XDR::put (Message Block *mb) { /* incoming */ }
Modules
Modules
MP TCP::put (Message Block *mb) { /* outgoing */ }
MP TCP::put (Message Block *mb) { /* incoming */ }
MP DLP::svc (void) { /* outgoing */ }
MP DLP::svc (void) { /* incoming */ }
STREAM Tail
NETWORK DEVICES OR PSEUDO-DEVICES
MODULE
WRITE TASK
READ TASK
PROCESS OR THREAD

Figure 9: A Message-based Process Architecture

invoking the `Task::put` method in that layer. The `put` method executes the protocol tasks associated with its layer. For instance, the `MP TCP::put` method utilizes `Mutex` objects that serialize access to per-connection control blocks as separate messages from the same connection ascend the protocol stack in parallel.

The connectionless message-based protocol stack is structured in a similar manner. However, the connectionless protocol stack performs the simpler set of UDP functionality. Unlike the `MP TCP::put` method, the `MP UDP::put` method handles each message concurrently and independently, without explicitly preserving inter-message ordering. This reduces the amount of synchronization operations required to locate and update shared resources.

3.4 C++ Features Used to Simplify Process Architecture Implementation

Many of the protocol functions, process architecture synchronization mechanisms, and `ASX` framework support components (such as demultiplexing and message buffering classes) are reused throughout the process architecture test programs described above. For example, process architecture-specific synchronization strategies may be instantiated by selectively instrumenting protocol functions with different types of mutual exclusion mechanisms. When combined with C++ language features such as inheritance and parameterized types, these objects help to decouple protocol processing functionality from the concurrency control scheme used by a particular process architecture.

For example, objects of class `Multiplexor` use a `Map Manager` component to demultiplex incoming messages to `Modules`. `Map Manager` is a search structure container class that is parameterized by an external ID, internal ID, and a mutual exclusion mechanism, as follows:

```
template <class EX_ID, class IN_ID, class MUTEX>
class Map_Manager {
public:
  bool bind (EX_ID, IN_ID *);
  bool unbind (EX_ID);
  bool find (EX_ID ex_id, IN_ID &in_id);

private:
  MUTEX lock;
  // ...
```

The type of `MUTEX` that this template class is instantiated with depends upon the particular choice of process architecture. For instance, the `Map Manager` used in the message-based implementation of the TCP protocol stack described in Section 3.3.2 is instantiated with the following class parameters:

```
typedef Map_Manager <TCP_Addr, TCB, Mutex>
        MP_Map_Manager;
```

This particular instantiation of `Map Manager` locates the transport control block (`TCB`) associated with the `TCP` address of an incoming message. The `Map Manager` class uses the `Mutex` class described in Section 2.6.1 to ensure that its `find` method executes as a critical section. This prevents race conditions with other threads that are inspecting or inserting entries into the connection map in parallel.

In contrast, the task-based process architecture implementation of the TCP protocol stack described in Section 3.3.1 does not require the same type of concurrency control within a connection. In this case, demultiplexing is performed within the `svc` method in the `LP DLP` read `Task` of the data-link layer `Module`, which runs in its own separate thread of control. Therefore, the `Map Manager` used for the connection-oriented task-based process architecture is instantiated with a different `MUTEX` class, as follows:

```
typedef Map_Manager <TCP_Addr, TCB, Null_Mutex>
        LP_Map_Manager;
```

The implementation of the `acquire` and `release` methods in the `Null Mutex` class are essentially "no-op" inline functions that may be removed completely by the compiler optimizer.

The `ASX` framework employs a C++ idiom that involves using a class constructor and destructor to acquire and release locks on synchronization objects, respectively [32]. The `Mutex Block` class illustrated below defines a "block" of code over which a `Mutex` object is acquired and then automatically released when the block of code is exited and the object goes out of scope:

```
template <class MUTEX>
class Mutex_Block
{
public:
  Mutex_Block (MUTEX &m): mutex (m) {
    this->mutex.acquire ();
  }
  ~Mutex_Block (void) {
    this->mutex.release ();
  }
private:
        MUTEX &mutex;
}
```

This C++ idiom is used in the implementation of the `Map Manager::find` method, as follows:

```
template <class EX_ID, class IN_ID, class MUTEX> int
Map_Manager<EX_ID, IN_ID, MUTEX>::find
           (EX_ID ex_id, IN_ID &in_id)
{
  Mutex_Block<MUTEX> monitor (this->lock);

  if (/* ex_id is successfully located */)
    return 0;
  else
    return -1;
}
```

When the `find` method returns, the destructor for the `Mutex Block` object automatically releases the `Mutex` lock. Note that the `Mutex` lock is released regardless of which arm in the `if/else` statement returns from the `find` method. In addition, this C++ idiom also properly releases the lock if an exception is raised during processing in the body of the `find` method.

3.5 Process Architecture Experiment Results

This section presents measurement results obtained from the data reception portion of the connection-oriented and connectionless protocol stacks implemented using the task-based and message-based process architectures described above. Three types of measurements were obtained for each combination of process architecture and protocol stack: *total throughput*, *context switching overhead*, and *synchronization overhead*.

Total throughput was measured by holding the protocol functionality, application traffic patterns, and network interfaces constant and systematically varying the process architecture to determine the resulting performance impact. Each benchmarking session consisted of transmitting 10,000 4 Kbyte messages through an extended version of the widely available `ttcp` protocol benchmarking tool. The original `ttcp` tool measures the processing resources and overall user and system time required to transfer data between a transmitter process and a receiver process communicating via TCP or UDP. The flow of data is uni-directional, with the transmitter flooding the receiver with a user-specified number of data buffers. Various sender and receiver parameters (such as the number of data buffers transmitted and the size of application messages and protocol windows) may be selected at run-time.

The version of `ttcp` used in our experiments was enhanced to allow a user-specified number of communicating applications to be measured simultaneously. This feature measured the impact of multiple connections on process architecture performance (two connections were used to test the connection-oriented protocols). The `ttcp` tool was also modified to use the ASX-based protocol stacks configured via the process architectures described in Section 3.5. To measure the impact of parallelism on throughput, each test was run using 1, 2, 3, and 4 PEs successively, using 1, 2, 3, or 4 LWPs, respectively. Furthermore, each test was performed multiple times to detect the amount of spurious interference incurred from other internal OS tasks (the variance between test runs proved to be insignificant).

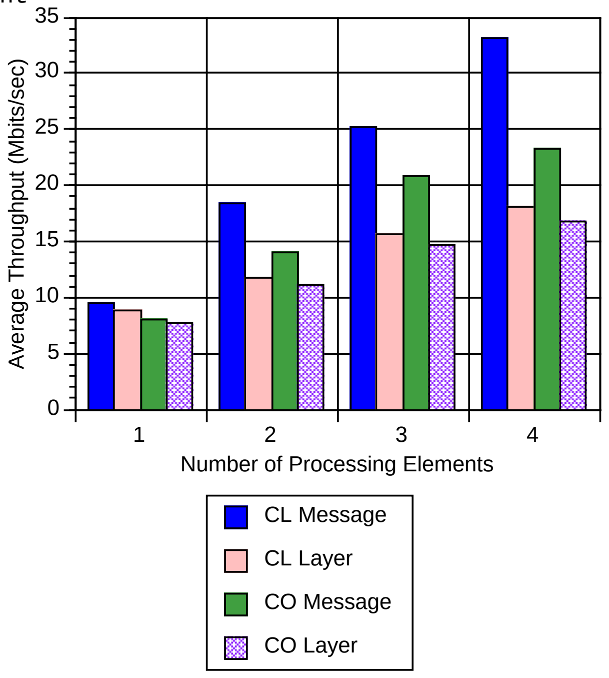

Figure 10: Process Architecture Throughput

Context switching and synchronization measurements were obtained to help explain differences in the throughput results. These metrics were obtained from the SunOS 5.3 `/proc` file system, which records the number of voluntary and involuntary context switches incurred by threads in a process, as well as the amount of time spent waiting to obtain and release locks on mutex and condition objects.

Figure 10 illustrates throughput (measured in Mbits/sec) as a function of the number of PEs for the task-based and message-based process architectures used to implement the connection-oriented (CO) and connectionless (CL) protocol stacks. The results in this figure indicate that parallelization definitely improves performance. Each 4 Kbyte message effectively required an average of between 3.2 and 3.9 milliseconds to process when 1 PE was used, but only .9 to 1.9 milliseconds to process when 4 PEs were used. However, the message-based process architectures significantly outperformed their task-based counterparts as the number of PEs increased from 1 to 4. For example, the performance of the connection-oriented task-based process architecture was only slightly better using 4 PEs (approximately 16 Mbits/sec, or 1.92 milliseconds per-message processing time) than the message-based process architecture was using 2 PEs (14 Mbits/sec, or 2.3 milliseconds per-message processing time). Moreover, if a larger number of PEs had been available, it appears likely that the performance improvement gained from parallel processing in the task-based process architectures would have leveled off sooner than the message-based tests due to the higher rate of growth for context switching and synchronization shown in Figure 11 and Figure 12.

Figure 11 illustrates the number of *involuntary* and *volun-*

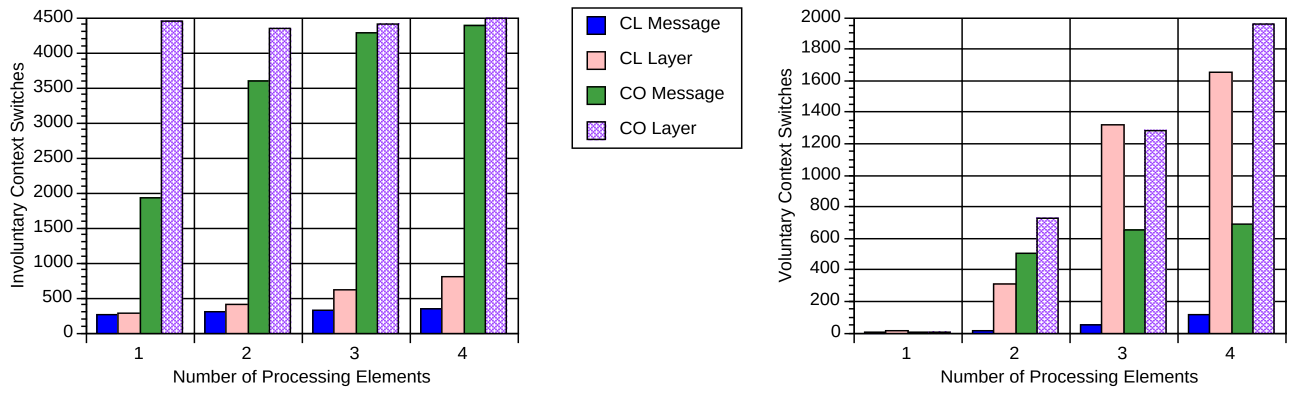

Figure 11: Process Architecture Context Switching Overhead

tary context switches incurred by the process architectures measured in this study. An involuntary context switch occurs when the OS kernel preempts a running thread. For example, the OS preempts running threads periodically when their LWP time-slice expires in order to schedule other threads to execute. A voluntary context switch is triggered when a thread puts itself to sleep until certain resources (such as I/O devices or synchronization locks) become available. For example, when a protocol task attempts to acquire a resource that may not become available immediately (such as obtaining a message from an empty list of messages in a `Task`), the protocol task puts itself to sleep by invoking the `wait` method of a condition object. This action causes the OS kernel to preempt the current thread and perform a context switch to another thread that is capable of executing protocol tasks immediately.

As shown in Figure 11, The task-based process architectures exhibited slightly higher levels of involuntary context switching than the message-based process architectures. This is due mostly to the fact that the task-based tests required more time to process the 10,000 messages and were therefore pre-empted a greater number of times. Furthermore, the task-based process architectures also incurred significantly more voluntary context switches, which accounts for the substantial improvement in overall throughput exhibited by the message-based process architectures. The primary reason for the increased context switching is that the locking mechanisms used by the message-based process architectures utilize adaptive spin-locks (which rarely trigger a context switch), rather than the sleep-locks used by task-based process architectures (which *do* trigger a context switch).

Figure 12 indicates the amount of execution time `/proc` reported as being devoted to waiting to acquire and release locks in the connectionless and connection-oriented benchmark programs. As with context switching benchmarks, the message-oriented process architectures incurred considerably less synchronization overhead, particularly when 4 PEs were used. As before, the spin-locks used by message-based process architecture reduce the amount of time spent synchronizing, in comparison with the sleep-locks used by the task-based process architectures.

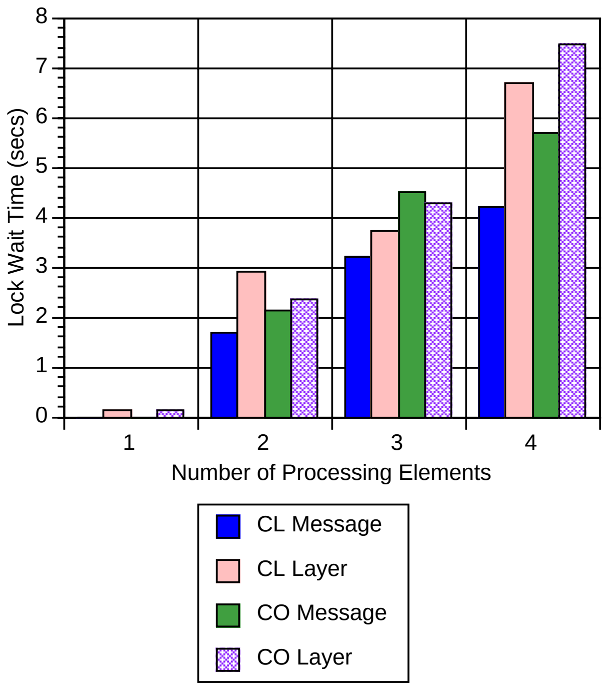

Figure 12: Process Architecture Locking Overhead

4 Concluding Remarks

Despite an increase in the availability of operating system and hardware platforms that support networking and parallel processing [25, 33, 22, 34], developing distributed applications that effectively utilize parallel processing remains a complex and challenging task. The ADAPTIVE Service eXecutive (ASX) provides an extensible object-oriented framework that simplifies the development of distributed applications on shared memory multi-processor platforms. The ASX framework employs a variety of advanced OS mechanisms (such as multi-threading and explicit dynamic linking), object-oriented design techniques (such as encapsulation, hierarchical classification, and deferred composition) and C++ language features (such as parameterized types, inheritance, and dynamic binding) to enhance software quality factors (such as robustness, ease of use, portability, reusability, and extensibility) without degrading application performance. In general, the object-oriented techniques and C++ features enhance the software quality factors, whereas the advanced OS mechanisms improve application functionality and performance.

A key aspect of concurrent distributed application performance involves the type of process architecture selected to structure parallel processing of tasks in an application. Empirical benchmark results reported in this paper indicate that the task-based process architectures incur relatively high-levels of context switching and synchronization overhead, which significantly reduces their performance. Conversely, the message-based process architectures incur much less context switching and synchronization, and therefore exhibit higher performance. The `ASX` framework helped to contributed to these performance experiments by providing a set of object-oriented components that decouple the protocol-specific functionality from the underlying of process architecture, thereby simplifying experimentation.

The `ASX` framework components described in this paper are freely 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 ADAPTIVE project [35] 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 AT&T Q.port ATM signaling software product, the Ericsson EOS family of telecommunication switch monitoring applications, and the network management portion of the Motorola Iridium mobile communications system.

Acknowledgements

Thanks to Paul Stephenson of Ericsson for countless hours of discussion on devising object-oriented techniques for developing distributed application frameworks.

References

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

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

[3] D. C. Schmidt and P. Stephenson, “An Object-Oriented Framework for Developing Network Server Daemons,” in *Proceedings of the 2nd C++ World Conference*, (Dallas, Texas), SIGS, Oct. 1993.

[4] D. C. Schmidt and T. Suda, “An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems,” *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

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

[6] M. A. Linton and P. R. Calder, “The Design and Implementation of InterViews,” in *Proceedings of the USENIX C++ Workshop*, November 1987.

[7] 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, pp. 355–398, Oct. 1992.

[8] R. Campbell, V. Russo, and G. Johnson, “The Design of a Multiprocessor Operating System,” in *Proceedings of the USENIX C++ Workshop*, pp. 109–126, USENIX Association, November 1987.

[9] J. M. Zweig, “The Conduit: a Communication Abstraction in C++,” in *Proceedings of the 2nd USENIX C++ Conference*, pp. 191–203, USENIX Association, April 1990.

[10] N. C. Hutchinson, S. Mishra, L. L. Peterson, and V. T. Thomas, “Tools for Implementing Network Protocols,” *Software Practice and Experience*, vol. 19, pp. 895–916, September 1989.

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

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

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

[14] 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.

[15] 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.

[16] D. C. Schmidt, “Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services,” in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP ’95*, (Aarhus, Denmark), August 1995.

[17] D. C. Schmidt and T. Suda, “Measuring the Performance of Parallel Message-based Process Architectures,” in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (Boston, MA), pp. 624–633, IEEE, April 1995.

[18] D. C. Schmidt and T. Suda, “Measuring the Impact of Alternative Parallel Process Architectures on Communication Subsystem Performance,” in *Proceedings of the 4th International Workshop on Protocols for High-Speed Networks*, (Vancouver, British Columbia), pp. 103–118, IFIP/IEEE, August 1994.

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

[20] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching,” in *Proceedings of the 1st Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–10, August 1994.

[21] Bjarne Stroustrup, *The C++ Programming Language, 2nd Edition*. Addison-Wesley, 1991.

[22] 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.

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

[24] A. D. Birrell, "An Introduction to Programming with Threads," Tech. Rep. SRC-035, Digital Equipment Corporation, January 1989.

[25] 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.

[26] M. Zitterbart, B. Stiller, and A. Tantawy, "A Model for High-Performance Communication Subsystems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 507–519, May 1993.

[27] J. C. Mogul and A. Borg, "The Effects of Context Switches on Cache Performance," in *Proceedings of the* 4^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Santa Clara, CA), ACM, Apr. 1991.

[28] Mats Bjorkman and Per Gunningberg, "Locking Strategies in Multiprocessor Implementations of Protocols," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (San Francisco, California), ACM, 1993.

[29] 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.

[30] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 200–208, ACM, Sept. 1990.

[31] D. D. Clark, "The Structuring of Systems Using Upcalls," in *Proceedings of the* 10^{th} *Symposium on Operating System Principles*, (Shark Is., WA), 1985.

[32] G. Booch and M. Vilot, "Simplifying the Booch Components," *C++ Report*, vol. 5, June 1993.

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

[34] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young, "Mach Threads and the Unix Kernel: The Battle for Control," in *Proceedings of the USENIX Summer Conference*, USENIX Association, August 1987.

[35] 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.