An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Systems

Douglas C. Schmidt

schmidt@cs.wustl.edu
Dept. of Comp. Sci.
Washington University
St. Louis, MO 63130
314 935-6160

Tatsuya Suda

suda@ics.uci.edu
Info. and Comp. Sci. Dept.
University of California, Irvine
Irvine, California, 92717
(714) 856-4105[1]

This paper will appear in the BCS/IEE Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems), 1995.

Abstract

Developing extensible, robust, and efficient distributed systems is a complex task. To help alleviate this complexity, we have developed the ADAPTIVE Service eXecutive (ASX) framework. ASX is an object-oriented framework composed of automated tools and reusable components. These tools and components help to simplify the development, configuration, and reconfiguration of applications in a distributed environment. Using the ASX framework, the services in the applications may be updated and extended without modifying, recompiling, relinking, or restarting the applications at run-time. This paper describes the features and object-oriented architecture of the ASX framework. It also describes how the ASX framework has been used to develop a highly modular, reusable, and dynamically reconfigurable family of distributed system management applications.

1 Introduction

Mapping distributed application services flexibly and efficiently onto host processes and threads is a challenging task. Selecting an efficient mapping is difficult since service functionality, network/host workload, and OS/hardware platform characteristics may vary dynamically. Therefore, it is essential to develop software tools that support flexibility with respect to the following design criteria:

- which services to map onto which hosts in a distributed system;
- how to effectively use the parallelism available on multiprocessor platforms.

Ideally, software development tools should postpone these decisions until very late in the development cycle (*i.e.,* at installation-time or run-time). By deferring these decisions, it becomes possible to select mappings that are tailored to an application's run-time environment.

Object-oriented frameworks are a promising technique for alleviating the complexity of developing, configuring, and reconfiguring services in distributed applications. A framework is an integrated collection of software components that collaborate to produce a reusable architecture for a family of related applications [1]. Object-oriented frameworks enable fine-grain component reuse and eliminate unnecessary performance overhead introduced by many existing tools for configuring applications.

The components in a framework typically include *classes* (such as message managers, timer-based event dispatchers, and connection maps [2]), *class hierarchies* (such as an inheritance lattice of local and remote interprocess communication mechanisms [3]), *class categories* (such as a family of concurrency control mechanisms [4]), and *objects* (such as event demultiplexers [5]). Frameworks emphasize the integration and collaboration of application-specific and application-independent components. This enables larger-scale reuse of software, compared with reusing individual classes and stand-alone functions.

To illustrate how object-oriented frameworks are being applied successfully in practice, this paper examines the ADAPTIVE Service eXecutive (ASX) framework. The ASX framework provides an integrated collection of automated tools and reusable components. These tools and components are specifically targeted for developing applications in distributed systems. Examples of these systems include global personal communication systems, telecommunication switch management platforms, and real-time market data analysis applications. In these types of systems, it is often necessary to phase new versions of services into applications without disrupting current execution [6].

In addition to describing the features and object-oriented architecture of the ASX framework, this paper describes how ASX is being used to develop a family of commercial distributed system management applications. These applications monitor and control private-branch exchange (PBXs) switches and public central-office telecommunication switches running on heterogeneous hardware and software platforms.

[1]This research is supported in part by grants from the University of California MICRO program, Hughes Aircraft, Nippon Steel Information and Communication Systems Inc. (ENICOM), Hitachi Ltd., Hitachi America, Tokyo Electric Power Company, and Hewlett Packard (HP).

This paper is organized in the following manner: Section 2 outlines the primary features in the ASX framework; Section 3 examines a commercial Call Center Management distributed system built using ASX; Section 4 describes the object-oriented architecture of ASX; Section 5 compares the ASX framework with related research; and Section 6 presents concluding remarks.

2 Overview of the ADAPTIVE Service eXecutive (ASX) Framework

The ASX framework enhances the modularity, extensibility, and portability of OS mechanisms that provide *interprocess communication*, *event demultiplexing*, *dynamic linking*, and *concurrency*. These OS mechanisms are orchestrated by the ASX framework to automate the configuration and reconfiguration of services into distributed applications.

Applications developed using the ASX framework may be flexibly configured either statically and/or dynamically. Static configuration is performed by developers at compile-time. Dynamic configuration is performed by administrators or by management applications at installation-time and/or at run-time. This flexibility allows applications executing within the ASX run-time environment to update their functionality dynamically, often without being shutdown completely and restarted. The primary features of the ASX framework are outlined in Sections 2.1 and 2.2 below.

2.1 Reusable, Application-Independent Components

The ASX framework provides the following reusable components that form the basis for developing distributed applications:

- port monitoring
- buffer management and message queueing
- event demultiplexing and event handler dispatching
- local/remote interprocess communication
- concurrency control
- configuration, installation, and run-time management of application services

To implement these components efficiently and extensibly, the ASX framework employs multi-threading and dynamic linking mechanisms available in operating systems such as UNIX and Windows NT. These mechanisms facilitate the development of distributed systems that may be updated and extended without modifying, recompiling, relinking, or restarting applications at run-time. To promote reuse, the ASX components are designed using object-oriented techniques (such as design patterns [5] and hierarchical software decomposition [7]) and object-oriented language features (such as abstract base classes, inheritance, and dynamic binding, and parameterized types [8]).

2.2 Support for Highly-Decoupled System Architectures

To enhance the flexibility and extensibility of distributed systems, the ASX framework decouples the functionality of application-specific services from the characteristics of distributed systems described below:

• **Structural Characteristics:** The ASX framework decouples service functionality from the following structural characteristics of applications:

- *The type and number of services associated with each process* – The ASX framework supports the configuration of applications that contain multiple services. These services may be instances of the same service or instances of different services.
- *The point of time at which service(s) are configured into an application* – The ASX framework encapsulates dynamic linking mechanisms provided by an operating system. Dynamic linking is a lightweight mechanism for configuring services into an application either statically (at compile-time or link-time) or dynamically (when an application first begins executing or while it is running). The ASX framework allows the choice between static and dynamic configuration to be deferred until installation-time [9].
- *The order in which hierarchically-related services are composed into an application* – An application may be represented as a series of hierarchically-related services that communicate by passing typed messages. These services may be flexibly configured in order to satisfy application requirements and enhance component reuse.

• **Communication Mechanisms:** The ASX framework decouples service functionality from the following communication-related mechanisms:

- *The underlying interprocess communication (IPC) protocols and interfaces used to communicate with peers* – The ASX framework encapsulates IPC mechanisms (such as sockets, TLI, STREAM pipes, and named pipes) with a uniform, portable, and type-safe object-oriented interface.
- *The I/O-based and timer-based event demultiplexing mechanisms* – The ASX framework encapsulates several event demultiplexing mechanisms (such as the UNIX `select` and `poll` system calls and the Windows NT `WaitForMultipleObjects` function) with a uniform, extensible object-oriented interface. These demultiplexing mechanisms are used to dispatch external events (such as connection requests and application data) onto pre-registered application-specified event handlers.

• **Concurrency Strategies:** The ASX framework decouples service functionality from the following concurrency strategies:

- *The type and number of concurrent processes and/or threads used to perform services at run-time* – The ASX framework encapsulates different flavors of multi-threading mechanisms (such as POSIX threads, MACH cthreads, Solaris threads, and Windows NT threads). This encapsulation facilitates portability and greater functionality for multi-processing capabilities available on an OS platform. On SunOS 5.x UNIX [10], for instance, developers may select between user-level and kernel-level threads at run-time.

- *Flexible selection of process architectures* – The ASX framework enables the flexible configuration of several message-based and task-based process architectures [4]. A process architecture binds CPUs with the tasks and the messages associated with applications in a host computer. The choice of process architecture directly impacts sources of distributed application overhead (such as memory-to-memory copying and data manipulation, context switching, scheduling, and synchronization).

The ASX framework enables applications to avoid premature commitment to the structural characteristics, communication mechanisms, and concurrency strategies described above until late in the development cycle (*i.e.,* during installation-time or run-time). This "late binding" method of configuring services into applications enhances application portability, reusability, and extensibility. These enhancements occur since design and implementation decisions are deferred until sufficient information is available to select efficient policies and mechanisms.

3 Developing a Distributed Call Center Management System with the ASX Framework

This section motivates and illustrates the use of the ASX framework to develop a family of distributed system management applications. These applications monitor and manage PBX and central-office telecommunication switches in a *Call Center*. Call Centers contain groups of operators that interact with customers to setup airline reservations, process insurance claims, accept product orders, etc. A Call Center Management (CCM) system provides services that allow the staff of a Call Center to assess the performance of the Call Center and the quality of service provided to customers. The behaviour of the CCM system and the ASX framework components that are used to implement the CCM software are described below.

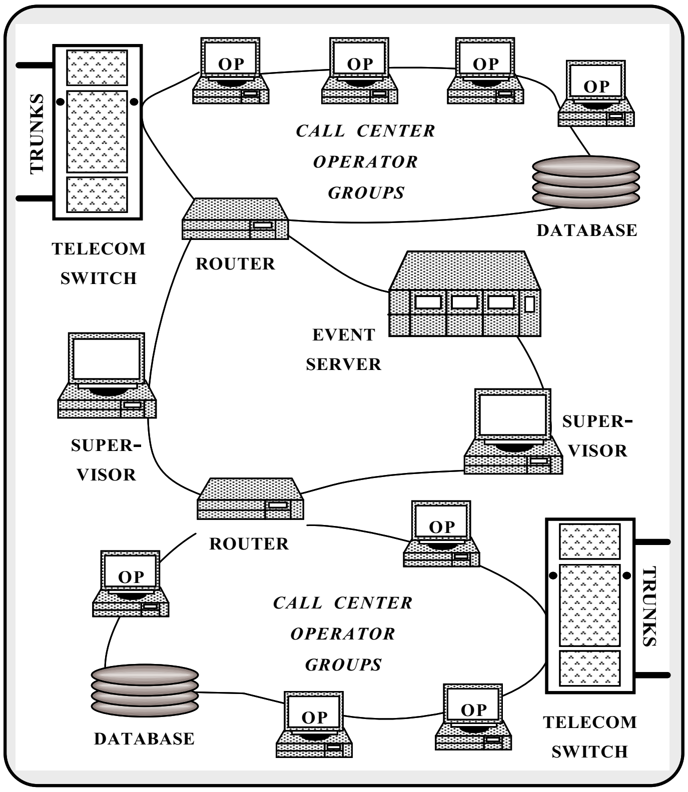

Figure 1: Distributed Components in the Call Center Management System

3.1 Overview of the Call Center Management System

The ASX-based Call Center Management (CCM) system processes real-time information generated by system operators, telecommunication switches, and networks. Supervisors use this information to monitor and optimize CCM system performance interactively. In addition, this information may be used to forecast the allocation of resources (such as operator and switch capacity planning) to meet anticipated customer demand.

Figure 1 illustrates the distributed architecture of the CCM system. In this system, telecommunication switches receive customer calls on trunk lines and forward these calls to available operators. Operators interact with customers using operator workstations (labeled "OP" in Figure 1). To expedite interactions with customers, these workstations access customer account records stored in a database attached to the network.

The switches and operator workstations in the CCM system continuously generate messages known as "activity events." These events provide a real-time status report on operator activities and telecommunication switch performance. In the CCM system, these activity events are sent through routers to one or more *event servers*. An event server runs on a separate network that is connected via LANs or WANs to the operator group networks. These servers act as mediators that analyze, filter, and forward a subset of the activity events they receive to supervisor hosts. These hosts summarize the events and display them to supervisors in a graphical format.

The event processing in the CCM system is representative of a growing class of applications known as "dynamic multi-point applications." Examples of these applications include satellite telemetry processing systems, real-time market data analysis systems, large-scale network management systems, and distributed interactive simulation systems. Dynamic multi-point applications have the following characteristics:

- a high volume of events are generated continuously in real-time by one or more suppliers;
- the data formats of the events are potentially complex;
- zero or more consumers may subscribe to a subset of the total events generated by the supplier(s);
- consumers typically reside on different hosts than suppliers;
- consumers must be able to process the events they receive in real-time;
- consumers may add, delete, or update their subscriptions dynamically.

In a dynamic multi-point application, each generated event is compared with the set of active consumer subscriptions. These applications are characterized as *multi-point* since any event that matches a subscription is delivered to the corresponding consumer(s). Likewise, the applications are characterized as *dynamic* since consumers may modify their subscriptions at run-time and each event may potentially be sent to a different subset of consumers.

3.2 Design Goals of the CCM System

The object-oriented design and implementation of the CCM system is strongly influenced by requirements for platform independence and configuration flexibility. Platform independence is necessary since the CCM system is targeted for various configurations of the following:

- *Telecommunication switches* – such as PBX and central-office switches
- *Host platforms* – such as Windows NT, Windows 3.1, OS/2, and UNIX platforms
- *Wide-area and local-area networks* – such as X.25, TCP/IP, and Novell IPX/SPX networks

Configuration flexibility is necessary since not all Call Center installations require every feature provided by the CCM system. It would be possible (although highly undesirable) to manually construct and deliver CCM systems that are customized for the platforms and the subsets of features required by a particular installation. However, this static configuration process would require the selection of services, and the division of labor between different hosts in a CCM system, to be completely fixed during initial system deployment. Our experience with earlier-generation CCM applications indicated that even if configuration information was available at the time of system deployment, it was likely to change in the future, often upon short notice.

The ASX framework facilitates platform independence to improve software component reuse across platforms and to reduce development effort. Object-oriented language features (such as abstract base classes, inheritance, and parameterized types) are used extensively throughout the CCM system. These features help to localize and minimize platform dependencies.

In addition, the ASX framework is used to defer the point of time at which a particular set of services is configured to form a CCM system. By combining advanced OS features (such as multi-threading and dynamic linking) and object-oriented language features, the ASX framework enables services offered by a CCM system to be extended without modifying, recompiling, relinking, or even restarting the system at run-time [9].

3.3 Mapping CCM Functionality onto ASX Components

Figure 2 illustrates the ASX framework components used to implement the event server portion of the CCM system shown in Figure 1.[2] These components include

[2] A comprehensive description of other components (such as routers, operator stations, switches, and supervisor hosts) in the CCM system is beyond the scope of this paper.

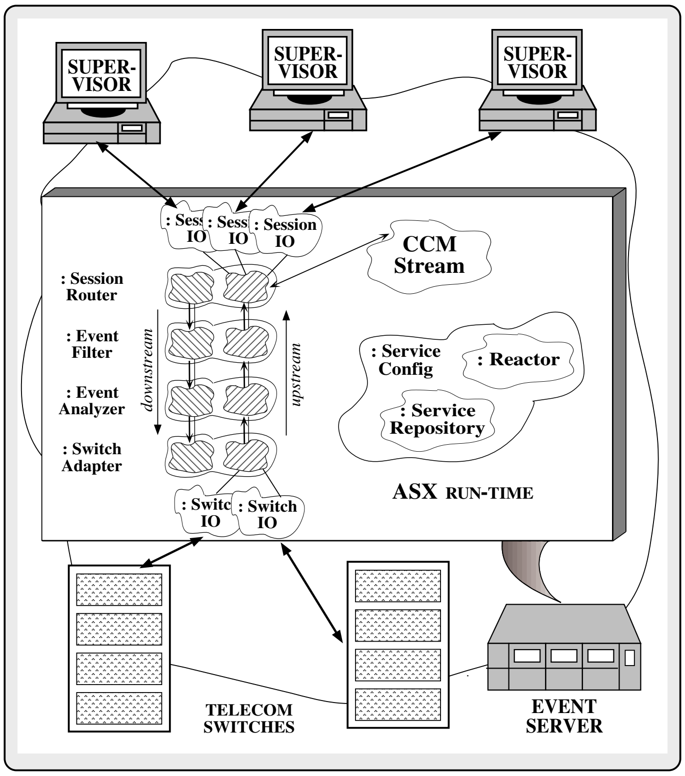

Figure 2: ASX Components in the Call Center Manager Event Server

the `Reactor`, `Service Configurator`, and `Stream` class categories. A more detailed discussion of these ASX framework components is provided in Section 4.

A CCM event server performs the following tasks:

- it receives activity events that are generated continuously by operator workstations and telecommunication switches;
- it analyzes and correlates the activity events it receives to produce a new set of derived messages known as "status events";
- it filters out status events that do not match any supervisor subscriptions;
- it forwards the status events that did match subscriptions across a network to supervisor hosts that have previously registered to receive these events.

The CCM event server contains a set of hierarchically-related processing `Modules`. A `Module` is a reusable ASX component used to decompose the CCM event server into a series of bi-directional, interconnected, functionally distinct layers. `Modules` may be configured statically and/or dynamically into the event server using mechanisms provided by the ASX framework. The behaviour of the ASX `Modules` (the `Switch Adapter`, `Event Analyzer`, `Event Filter`, and `Session Router`) that comprise the CCM event server is described below. The other ASX components in Figure 2 (such as the `Reactor`, `Service Config`, and `Service Repository`) will be explained shortly.

• **The Switch Adapter Module:** The `Switch Adapter Module` object interacts with the telecommunication switches monitored by the CCM event server. This `Module` shields the upstream layers of the event server architecture from switch-specific communication characteristics (such as activity event frame formats). The `Switch Adapter Module` maintains a collection of `Switch IO` objects that are responsible for transforming incoming activity events. This transformation encapsulates the activity events into a canonical switch-independent message format. This message format is built atop a flexible dynamic memory management class described in [4]. After being allocated and initialized, the canonical message objects are passed upstream to the `Event Analyzer Module`.

• **The Event Analyzer Module:** The `Event Analyzer Module` transforms switch-specific activity events into a switch-independent event format known as "status events." During the event analysis process, the `Event Analyzer` also synthesizes groups of activity events into a new set of derived status events. These synthesized status events are triggered in response to the occurrence of one or more activity events. After the `Event Analyzer` has transformed and/or synthesized the incoming activity events, it forwards the resulting status events to the `Event Filter Module`.

• **The Event Filter Module:** The `Event Filter Module` performs event reduction processing to minimize unnecessary network traffic and unnecessary processing at consumer endsystems. This `Module` forwards only those status events subscribed to by at least one supervisor. The `Event Filter Module` contains a collection of `Event Forwarding Discriminator` (`EFD`) objects. Each `EFD` object contains a predicate that indicates the characteristics of status event(s) a supervisor has registered to receive. `EFD` predicates may be used to selectively filter out status events based on characteristics such as event type, event value, event generation time, event frequency, and event state changes. An `EFD` predicate may contain relational operators that enable the composition of arbitrarily complex filter expressions.

Supervisors may register `EFD` objects with the event server during initial system configuration and during run-time. When status events are passed to the `Event Filter Module`, it inspects the registered `EFDs` to determine the set of supervisors that should receive the status event. If a status event matches a supervisor's `EFD` predicate, the supervisor's address is added to a data structure (called the `Session Addr Set`) that accompanies the status event message to the `Session Router Module` (described below). If a `Session Addr Set` contains at least one address after all `EFDs` are inspected by the `Event Filter Module`, the status event and the `Session Addr Set` are passed upstream to the `Session Router Module`.

• **The Session Router Module:** The Session Router Module interacts with supervisors on remote hosts. It shields the downstream layers of the CCM event server architecture from non-portable details of the communication protocols used to communicate with supervisors. Supervisors connect to the event server by establishing a session with the Session Router Module. A unique Session IO object is created to manage each supervisor session. This Session IO object handles all the data transfer and control operations between the event server and a supervisor. After connecting to the event server, a supervisor subscribes to a set of status event(s). Subsequently, when the Session Router receives a message from the Event Filter Module, it automatically forwards the status event message to all supervisors addressed by the message's Session Addr Set.

In general, the use of ASX Modules encapsulates non-portable system mechanisms (such as communication protocols and the frame formats of activity and status events) behind abstract interfaces. This helps to improve the platform independence of the CCM system. For example, when the original PBX-based version of the CCM system was ported to a different central-office switch architecture, only the Event Analyzer and Switch Adapter portions of the event server were affected significantly.

Figure 2 illustrates several other ASX-based components. The Service Config object is a reusable component from the Service Configurator class category described in Section 4.3. The CCM event server uses the Service Config object to control the initial configuration, subsequent reconfiguration(s), and removal of Modules from the CCM Stream. The Reactor object is an event demultiplexer (described in Section 4.2). It dispatches incoming messages from supervisors, and activity events from switches, to the appropriate Session IO and Switch IO objects, respectively. These IO objects are implemented by inheriting from the Event Handler abstract base class (described in Section 4.2). The IO objects communicate with their peers across the network via the IPC SAP communication objects (described in Section 4.5).

Messages sent from supervisors to the event server are received by a Session IO object, passed downstream through the CCM Stream object, and handled by the appropriate Module (*e.g.,* EFD registrations are handled by the Event Filter Module). Likewise, incoming messages from switches are received by a Switch IO object and sent upstream starting at the Switch Adapter Module. As described in the following section, the Modules that comprise the CCM Stream object may be configured into the event server at installation-time by developers, as well as at run-time by system administrators or by system management tools.

3.4 CCM Event Server Configuration

The ASX framework supports flexible configuration of processing Modules into the CCM event server. To accomplish this, the ASX framework provides a configuration scripting language based upon dynamic linking mechanisms (the scripting language is described further in Section 4.3). For example, at installation-time the Service Config object uses the following configuration script to determine which services to dynamically link into the address space of the CCM event server:

```
# Configure a stream containing 4 Modules
stream CCM_Stream dynamic STREAM *
  /svcs/CCM_Stream.so:alloc()
{
  dynamic Switch_Adapter Module *
    /svcs/SA.so:alloc() "-p 2001"
  dynamic Event_Analyzer Module *
    /svcs/EA.so:alloc()
  dynamic Event_Filter Module *
    /svcs/EF-svr.so:alloc()
  dynamic Session_Router Module *
    /svcs/SR.so:alloc() "-p 2010"
}
```

This configuration script indicates the order in which the four event server Modules (Switch Adapter, Event Analyzer, Event Filter, and Session Router, respectively) are dynamically linked and pushed into the CCM Stream object.

During the installation of the event server, the Service Config object interprets this configuration script and carries out the directives described by each entry, as follows:

- The Service Config interprets the dynamic directive as an instruction to dynamically link the shared object file (specified by a pathname ending in .so) into the address space of the CCM event server.
- The framework then extracts the alloc function from the newly linked shared object file and invokes it. This function allocates a pointer to an instance of the Module object contained within the shared object file.
- The framework then passes in any initialization parameters (appearing as string literals at the end of the line) to the initialization method of the Module. This method performs Module-specific initialization activities (such as allocating data structures, establishing connections, or registering EFDs).
- Once the Module is initialized, the framework pushes it on to the stack of interconnected Modules that are being configured to form the CCM Stream.

Once all the Modules have been dynamically linked, initialized, and connected together, the application enters an event loop managed by the Reactor object. When events arrive, this event loop dispatches the appropriate methods of the Switch IO and Session IO objects to initiate the event service processing described in Section 3.3 above.

3.5 CCM Event Server Reconfiguration

This section motivates and illustrates how the dynamic reconfiguration mechanisms provided by the ASX framework are applied in the CCM system. This flexibility proved to be quite useful for the CCM project, where different OS/hardware platforms and different network characteristics necessitate different configurations of event servers and supervisor hosts. Two types of configurations occur most commonly:

- *Centralized Event Filter Processing* – In certain environments, it is beneficial to centralize all the event filtering at the event server. This configuration is appropriate when the following conditions occur:
  - an event server is installed on a high-performance platform (such as a multi-processor);
  - the supervisor hosts are run on less powerful platforms (such as inexpensive PCs);
  - a relatively low-bandwidth (or highly congested) network connects the event server to the supervisor hosts;
  - few of the supervisors subscribe to most of the events
  - the complexity and number of Event Forwarding Discriminators (EFDs) installed by supervisors does not cause a major performance bottleneck.

  When these conditions occur, the network and the supervisor hosts at the edges of the CCM system are typically the processing bottleneck, rather than the event server. Therefore, a centralized event filtering architecture helps to off-load work from the edges of the CCM system.

- *Decentralized Event Filter Processing* – In other environments, it is beneficial to decentralize event filtering by installing it on the supervisor hosts. This configuration is appropriate when the following conditions occur:
  - the supervisor hosts are powerful workstation platforms;
  - the event server does not run on a considerably more powerful platform;
  - a dedicated, high-speed network is available to connect the event server to the supervisor hosts;
  - many of the supervisors subscribe to most of the events
  - a large number of supervisors configure complex EFD objects into an event server.

  When these conditions occur, the event server becomes a bottleneck and overall system performance is degraded, even though surplus processing capacity is available in the network and in the supervisor hosts.

Other more complex scenarios are also possible. For example, the network topology that interconnects operator groups, event servers, and supervisors may involve both local-area and wide-area networks.

The CCM Stream architecture depicted in Figure 2 is an example of a centralized event filter processing configuration. It performs all event analysis and event filter processing in the event server. However, this configuration may not be appropriate for certain CCM environments, for the reasons outlined above. To address this situation, Figure 3 illustrates how the stream configuration shown in Figure 2 may be modified to operate efficiently when event server processing constitutes the primary performance bottleneck. In this configuration, event filtering has been reconfigured into the supervisor hosts.

The following script dynamically reconfigures the Modules in the CCM system:

```
# Suspend execution of the stream
suspend CCM_Stream

# Remove Event_Filter Module from stream
stream CCM_Stream {
  remove Event_Filter
}

# Instruct all the remote Supervisors to
# dynamically configure an instance of the
# Event Filter Module into their streams

remote "-h all -p 911" {
  stream CCM_Stream {
    dynamic Event_Filter Module *
      /svcs/EF-cli.so:alloc()
  }
}

# Continue CCM stream processing
resume CCM_Stream
```

This new script transfers the event filter processing functionality from the event server to the supervisor hosts using the following steps:

- The event server's CCM Stream object is suspended. This prevents it from processing events until the reconfiguration procedure is complete. Note that this scheme assumes the switch interface either buffers incoming events or exerts flow control backpressure on the switch during the reconfiguration period.
- The Event Filter Module is removed from the CCM Stream. This invokes a finalization method on the Module and dynamically unlinks the associated shared object file from the event server's address space.
- The Event Filter Module is then dynamically linked into streams installed on all the supervisor hosts. The "-h all" argument to the remote directive informs the Service Config object to execute the configuration directive on all the remote supervisor hosts.[3]
- The execution of the event server's CCM Stream is resumed.

[3]Note that this script assumes that the /svcs/EF-cli.so shared object file is accessible to all connected hosts via a network file system.

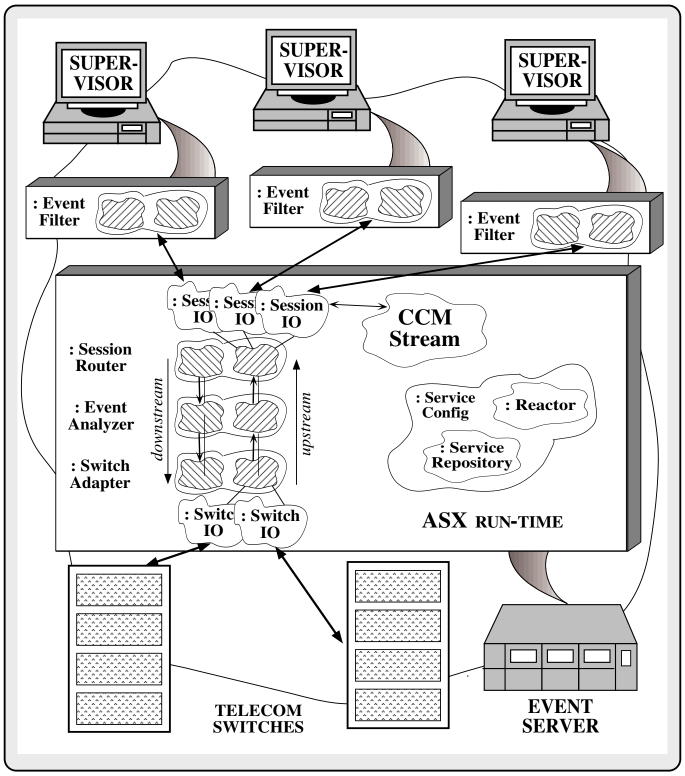

Figure 3: Reconfiguration of the Call Center Manager Event Server

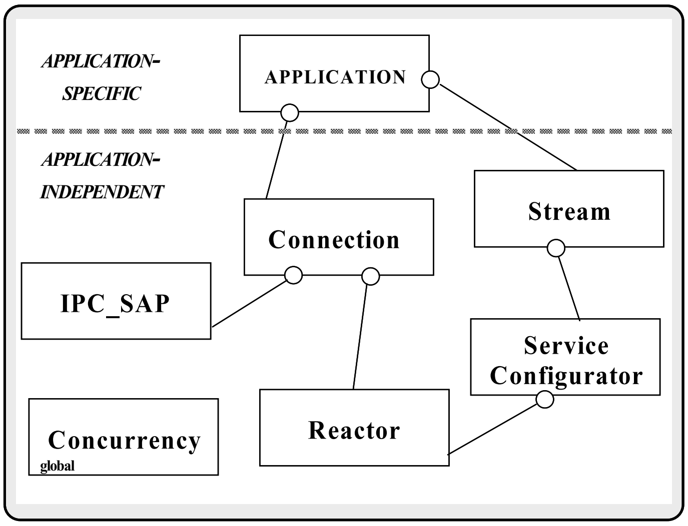

Figure 4: Class Categories in the `ASX` Framework

The result of this reconfiguration is to distribute the event filter processing among all the supervisor hosts, rather than centralizing it at the event server. Moreover, when the `Event Filter` object in the `EF-cli.so` shared object file is dynamically linked into the supervisor hosts, its initialization method is automatically invoked by `ASX`. This method may be programmed to only configure `EFDs` that are relevant to the current supervisor host. In addition, by using the dynamic linking mechanisms supplied by the `ASX` framework, the CCM server need not be completely shutdown and restarted at run-time during this filter migration process.

4 The Object-Oriented Architecture of the `ASX` Framework

The architecture of the `ASX` framework has been developed incrementally by generalizing from extensive design and implementation experience with a range of distributed systems. These systems include on-line transaction processing [5], telecommunication switch monitoring [11], and multi-processor-based communication subsystems [4]. After building several prototypes and iterating through a number of alternative designs, the class categories illustrated in Figure 4 were identified and implemented. A class category is a collection of components that collaborate to provide a set of related services [8]. An application may be configured by using object-oriented language features that specialize and compose the following `ASX` components:

- The `Stream` class category – components in this class category 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 an event server in the Call Center Management system) defined by an application.

- The `Reactor` class category – components in this class category are responsible for *demultiplexing* and dispatching *event handlers* that are triggered concurrently by various types of events.
- The `Service Configurator` class category – components in this class category 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 – components in this 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 `IPC SAP` class category – components in this class category encapsulate standard OS local and remote *interprocess communication* (IPC) mechanisms (such as sockets and TLI) within a type-safe and portable object-oriented interface.

Lines connecting the class categories in Figure 4 indicate dependency relationships. For example, components that implement the application-specific services in an application depend on the `Stream` components, which in turn depend on the `Service Configurator` components. Components in the `Concurrency` class category are used throughout the application-specific and application-independent portions of the `ASX` framework. Therefore, they are marked with the **global** adornment.

This section examines the main components in each class category. Relationships between components in the `ASX` framework are illustrated throughout the paper using Booch notation [8]. 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.

4.1 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` run-time environment. For example, Figure 5 illustrates the Stream of `Modules` that comprise the CCM event server presented in Section 3.

Inheritance and object composition are used to link a series of application-specific `Modules` together to form a Stream. `Modules` are objects that developers use to decompose the architecture of an application into functionally distinct layers. For instance, the CCM event server is decomposed into four layers (the `Switch Adapter`, `Event Analyzer`, `Event Filter`, and `Session Router Modules`). Each `Module` in a Stream implements a cluster of related application-specific functions (such as an end-to-end transport service, a presentation layer formatting service, or an event filtering service for dynamic multi-point applications).

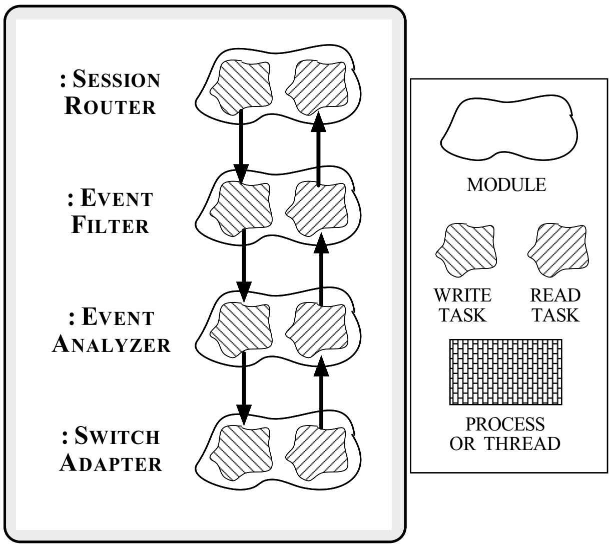

Figure 5: Components in the `Stream` Class Category

`Module` objects communicate by exchanging typed messages with adjacent `Module` objects in a Stream. Message passing overhead is minimized by passing a pointer to a message between two `Modules` rather than by copying the data. `Modules` may be joined together in essentially arbitrary configurations in order to satisfy application requirements and enhance component reuse. Moreover, `Module` objects may be configured into a Stream by developers at installation-time or by applications or administrators at run-time.

Every `Module` contains a pair of `Task` objects that partition a layer into its application-specific read-side and write-side 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 subsystems [4] or the domain of distributed system management applications [11]). 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.

The `ASX` framework enables developers to incorporate application-specific functionality into a Stream without modifying the existing application-independent framework components. Incorporating a new layer of service functionality into a Stream involves the following steps:

- Inheriting from the `Task` interface and selectively overriding several methods (described below) in the `Task` subclass to implement application-specific functionality;
- Allocating a new `Module` that contains two instances

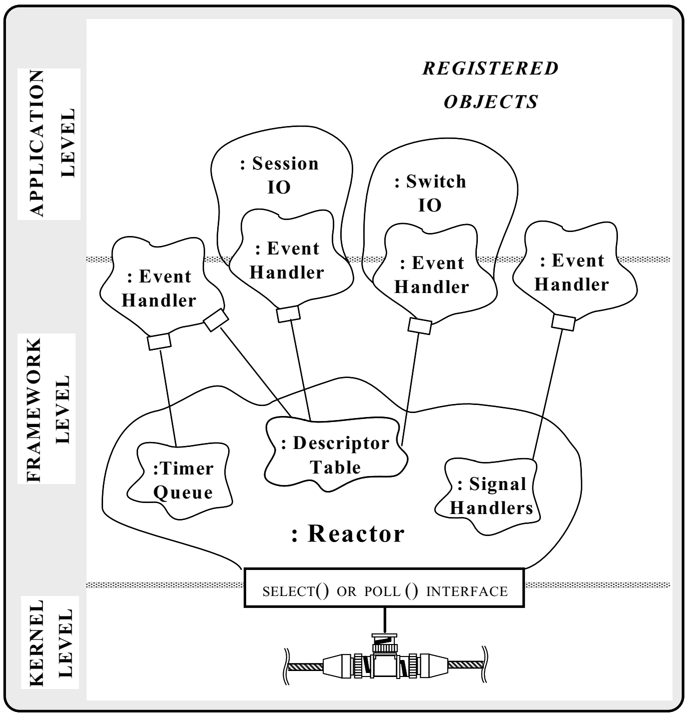

Figure 6: Components in the `Reactor` Class Category

(one for the read-side and one for the write-side) of the application-specific `Task` subclass;

- Inserting the `Module` into a Stream.

The `Task` abstract class[4] defines an interface that may be inherited and implemented by derived classes to provide application-specific functionality for read-side and write-side processing in a `Module`. One `Task` subclass handles read-side processing for messages sent upstream to its `Module` layer; the other handles write-side processing messages sent downstream to its `Module` layer.

4.2 The Reactor Class Category

Components in the `Reactor` class category [5] are responsible for *demultiplexing* and dispatching *event handlers* that are triggered concurrently by various types of events. These events may be I/O-based events received on communication ports, time-based events generated by a timer-driven callout queue, or signal-based events. When these events occur at run-time, the `Reactor` dispatches the appropriate pre-registered handler(s) to process the events. The `Reactor` encapsulates and enhances the functionality of OS event demultiplexing mechanisms (such as the Windows NT `WaitForMultipleObjects` and the UNIX `select` and `poll` system calls). These demultiplexing mechanisms detect the occurrence of different types of input and output events on one or more I/O descriptors simultaneously.

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

The `Reactor` is a container class whose methods provide a uniform interface to manage objects that implement various types of application-specific event 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 6, these handler objects derive from the `Event Handler` abstract base class. This class specifies an interface for event registration and event handler dispatching. 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.

The CCM event server described in Section 3 uses the `Reactor` to monitor communication ports connected to switches and supervisor hosts. Each communication port is associated with an event handler. The `Reactor` typically waits in an event loop for control messages to arrive from supervisors or for activity events to arrive from switches and/or operator hosts. When events arrive, the `Reactor` automatically dispatches a method on the corresponding `Session IO` or `Switch IO` event handler. This handler then performs the appropriate application-specific processing in response to the event.

4.3 The Service Configurator Class Category

Components in the `Service Configurator` class category are responsible for linking and/or unlinking services dynamically into or out of the address space of an application at run-time [9]. Dynamic linking is a lightweight mechanisms for configuring and reconfiguring application services. By using dynamic linking, it is often possible to reconfigure services without modifying, recompiling, relinking, and restarting an executing application.

The `Service Configurator` is used in the CCM event server to dynamically link and unlink `Modules` to and from a Stream, respectively. It also performs run-time control of `Modules` and Streams (such as temporarily suspending and resuming the `CCM Stream` during reconfiguration). The components of the `Service Configurator` discussed below include the the `Service Object` inheritance hierarchy (Figure 7 (1)), the `Service Repository` class (Figure 7 (2)), and the `Service Config` class (Figure 7 (3)).

- **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 application-specific subclasses. By decoupling the application-specific portions of a handler object from the underlying `Service Configurator`

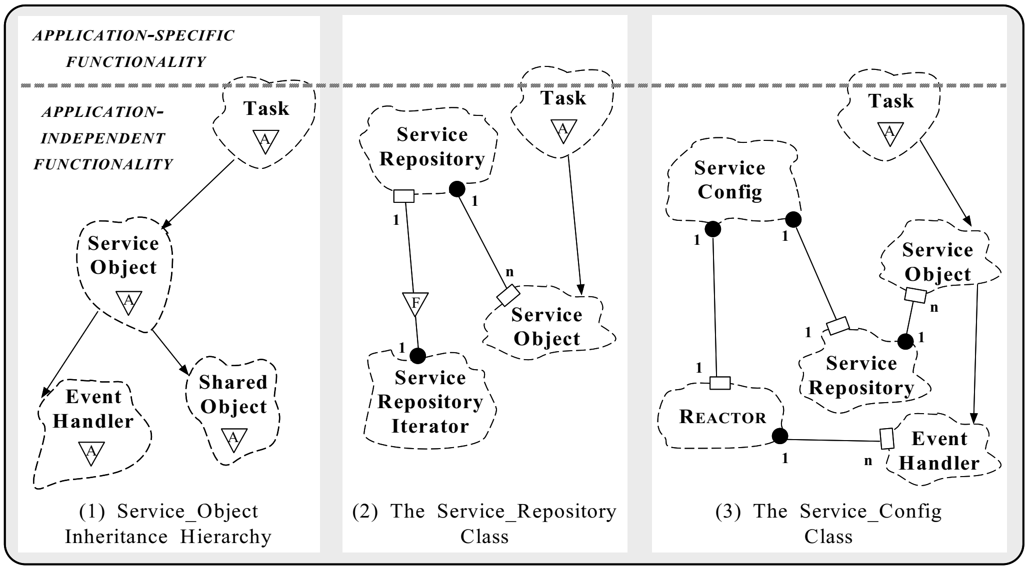

Figure 7: Components in the `Service Configurator` Class Category

mechanisms, the effort necessary to insert and remove services from an application at run-time is reduced significantly.

The `Service Object` inheritance hierarchy consists of the `Event Handler` and `Shared Object` abstract base classes, as well as the `Service Object` abstract derived class. The `Event Handler` class was described above in the `Reactor` Section 4.2. The `Shared Object` abstract base class specifies an interface for dynamically linking and unlinking objects into and out of an application's address space. This abstract base class exports pure virtual methods.

Pure virtual methods impose a contract between the reusable, application-independent components provided by the `Service Configurator` and application-specific objects that utilize these components. By using pure virtual methods, the `Service Configurator` ensures that a event handler implementation honors its obligation to provide certain configuration-related information. For example, in the CCM event server, this information is used by the `Service Configurator` to automatically link, initialize, identify, and unlink a service at run-time.

The `Shared Object` abstract 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 I/O descriptor-based, timer-based, signal-based event demultiplexing. Conversely, other applications (such as an `ftp` server) require event demultiplexing, but might not require dynamic linking.

Complex applications often require support for dynamic linking *and* event demultiplexing. For example, both these features are used to automate the dynamic configuration and reconfiguration of application-specific services in the CCM event server described in Section 3. Therefore, the `Service Configurator` class category provides the `Service Object` abstract derived class, which combines interfaces inherited from both the `Event Handler` and the `Shared Object` abstract base classes.

Note that the `Task` class in the `Stream` class category (described in Section 4.1) is derived from the `Service Object` abstract derived class (illustrated in Figure 7 (1)). This enables hierarchically-related, application-specific `Tasks` (which are grouped together to form the `Modules` comprising an application Stream) to be linked and unlinked into and out of an application at run-time. The CCM event server uses these dynamic linking and unlinking mechanisms to interpret the (re)configuration script language described further below.

• **The Service Repository Class:** The `ASX` framework supports the configuration of applications that contain one or more Streams. Each Stream may have one or more interconnected application-specific `Modules`. To simplify run-time administration, the `Service Repository` class is used to individually and/or collectively control the `Modules` that comprise an application's currently active Streams.

The `Service Repository` is an object manager used by the `ASX` framework to coordinate local/remote operations on services offered by Streams in an application. A search structure within the object manager binds service names (represented as a string) with instances of composite `Service Objects` (represented as C++ object code). A service name uniquely identifies an instance of a `Service Object` stored in the repository.

Each entry in the `Service Repository` contains a pointer to the `Service Object` portion of an application-specific subclass (shown in Figure 7 (2)). This enables the `Service Configurator` classes to automatically load, enable, suspend, resume, or unload `Service Objects` 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.

• **The Service Config Class:** the Service Config class illustrated in Figure 7 (3) is the focal point of the Service Configurator class category. It integrates the other ASX framework components (such as the Service Repository, the Service Object inheritance hierarchy, and the Reactor). Applications use the Service Config class to automate the static and/or dynamic configuration of hierarchically-related Modules to form one or more Streams. A configuration file (known as svc.conf) is used to guide the configuration and reconfiguration activities of a Service Config object. This configuration file is specified using a scripting language. Several examples of the svc.conf files used to configure and reconfigure services in the CCM event server are presented in Sections 3.4 and 3.5.

The svc.conf file may be used to configure one or more Streams into an application. Each entry in the file begins with a directive (such as dynamic, remove, suspend, or resume) that specifies which configuration activity to perform. Each entry also contains attributes that identify the location of the shared object file for each dynamically linked service, as well as any parameters required to initialize a service when it is linked at run-time.

By consolidating service attributes and initialization parameters into a single configuration file, the installation and administration of the services in an application is simplified. In addition, the svc.conf file helps to decouple the structure of an application from the behaviour of its services. This decoupling separates the application-independent configuration and reconfiguration of mechanisms provided by the framework from the application-specific attributes and parameters specified in the svc.conf file.

Figure 8 illustrates a state transition diagram depicting the Service Config and Reactor methods that are invoked in response to events occurring during service configuration, execution, and reconfiguration. For example, when the CONFIGURE and RECONFIGURE events occur, the process directives method of the Service Config class is called to interpret the svc.conf file associated with the application. The contents of a svc.conf file dictate the content of Streams that are configured into an application. This file is initially interpreted when a new instance of a daemon is configured. The file is interpreted again whenever a daemon reconfiguration is triggered upon receipt of a pre-designated external event (such as the UNIX SIGHUP signal).

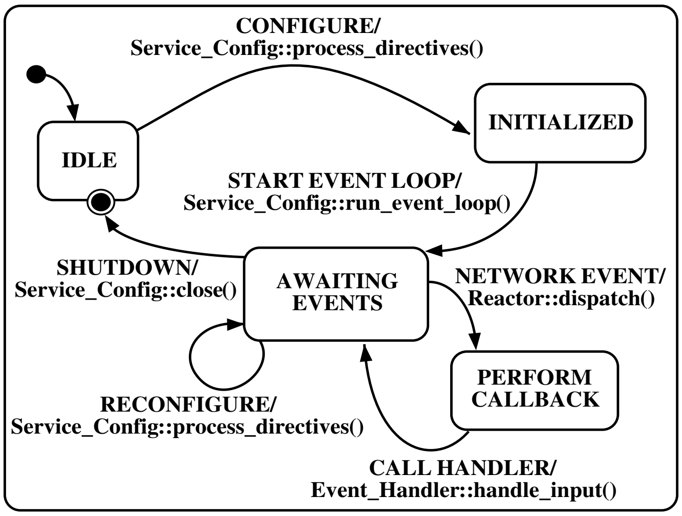

Figure 8: State Transition Diagram for Service Configuration, Execution, and Reconfiguration

4.4 The Concurrency Class Category

Components in the ASX 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 [4]. The following section discusses the Synch and Thread Manager classes in the Concurrency class category.

• **The Synch Classes:** The Synch classes provide type-safe object-oriented interfaces for two types of synchronization mechanisms: Mutex and Condition objects [10]. Applications use Mutex objects to ensure the integrity of shared resources 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 Mutex and Condition synchronization mechanisms.

• **The Thread Manager Class:** The Thread Manager class contains a set of mechanisms that manipulate multiple threads of control atomically. Typically, these threads of control collaborate to implement collective actions (such as rendering different portions of a large image in parallel). The Thread Manager class also shields applications from non-portable incompatibilities between different flavors of multi-threading mechanisms (such as POSIX threads, MACH cthreads, Solaris threads, and Windows NT threads).

The Thread Manager class provides methods that suspend and resume a set of collaborating threads of control atomically. This feature is useful for applications that execute multiple tasks or process multiple messages in parallel.

For example, a CCM event server Stream is 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 interconnected before allowing messages to flow through the Stream. The mechanisms in the `Thread Manager` class allow these initialization activities to execute atomically.

4.5 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 [3]. The `IPC SAP` class category is organized as a forest of class categories that are rooted at the `IPC SAP` base class. These class categories includes `SOCK SAP` (which encapsulates the socket interface), `TLI SAP` (which encapsulates the System V UNIX Transport Layer Interface (TLI)), `SPIPE SAP` (which encapsulates the UNIX SVR4 STREAM pipe interface), and `FIFO SAP` (which encapsulates the UNIX named pipe interface).

Each class category in `IPC SAP` is 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 enhances the reuse of code that is common among the various `IPC SAP` class categories.

5 Related Work

Various strategies and tactics for developing flexible distributed system software have emerged in several domains. One approach (used by distributed application frameworks such as Regis [12] and Polylith [6]) represents application state attributes as abstract data types to facilitate service configuration and reconfiguration.

Another approach (used by daemon management frameworks such as `inetd` and `listen` in System V Release 4 UNIX [13]) provides mechanisms for automating tedious and error-prone activities associated with configuring and reconfiguring network daemons. These mechanisms automate service initialization, reduce the consumption of OS resources by spawning event handlers "on-demand," allow daemon services to be updated without modifying existing source code, and consolidate the administration of network services via uniform configuration management operations.

Yet another approach (used by the *x*-kernel [2] and System V STREAMS [14]) configures network software flexibly by interconnecting building-block protocol and service components. These frameworks encourage the development of standard communication-related components by decoupling application-specific processing functionality from the application-independent communication framework infrastructure.

The existing frameworks outlined above have proven to be useful in practice. However, these frameworks were developed without adequate consideration of object-oriented techniques (such as class-based encapsulation, inheritance, dynamic binding, and parameterized types) and OS mechanisms (such as dynamic linking and lightweight processes) that are provided by operating systems (such as SVR4 UNIX, Windows NT, and OS/2). This deters fine-grain component reuse and introduces unnecessary performance overhead that discourages developers from fully exploiting the benefits of flexible software configuration techniques.

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 precludes fine-grained reuse of its internal components. More importantly, existing frameworks do not provide automated support for (1) dynamically linking services into an application's address space at run-time and (2) executing these services in parallel via one or more lightweight processes [10]. Instead, these frameworks typically configure and reconfigure application services by spawning heavyweight processes and connecting these processes via heavyweight IPC mechanisms (such as pipes and sockets).

The `ASX` framework extends related work by providing object-oriented components that decouple application-specific service functionality from the following structural and behavioural characteristics of parallel and distributed software systems:

- The use of dynamic linking, which facilitates the configuration of fine-grained objects into applications with minimal run-time overhead [9];
- The type of locking mechanisms used to synchronize access to shared objects;
- The use of kernel-level and user-level execution agents;
- The use of message-based and task-based process architectures [4].

6 Concluding Remarks

The `ASX` framework provides an integrated collection of object-oriented components. These components support the static and dynamic configuration of modular services into distributed applications. Component reuse is enhanced in the `ASX` framework by separating the higher-level application-specific policies (such as the CCM event analyzer and event filtering mechanisms) from the lower-level application-independent mechanisms (such as the choice of mechanisms for network communication, event demultiplexing and event handler dispatching, and process or thread concurrency control strategies). In addition, `ASX` uses dynamic binding and dynamic linking to improve flexibility and extensibility. These mechanisms facilitate the development of applications that may be extended without modifying, recompiling, relinking, or restarting an existing application at run-time [9].

We are currently evaluating the CCM distributed architecture described in Section 3 to determine techniques for improving event server performance. One technique involves optimizing event filtering by merging multiple filter expressions together using a trie data structure. This technique is similar to optimizing packet filters that monitor network traffic on an Ethernet [15]. The use of a trie reduces the amount of time required to process M messages of size L through N event filters from $O(M \times L \times N)$ to $O(M \times L)$.

Another strategy for improving performance involves parallelizing the CCM event server. The goal is to minimize the overhead of parallel processing (such as data movement, context switching, scheduling, and synchronization). The `Stream` and `Concurrency` class categories (described in Sections 4.1 and 4.4) facilitate flexible binding of threads onto either `Modules` or events in order to parallelize performance [4].

We are also investigating service migration policies to formulate guidelines that ensure the dynamic reconfiguration of the event server does not corrupt or disrupt active services. For instance, the `EFDs` in the CCM event server described in Section 3 do not maintain much context independent information. Therefore, the effort necessary to migrate them is minimized since extensive state manipulations are not required. To reconfigure other types of services (such as the `Event Analyzer Module`), however, requires more costly state preservation and restoration procedures. A more ambitious long-term project involves using the `ASX` mechanisms to experiment with service migration policies that relocate services dynamically to reduce overall system workload at run-time.

Components in the `ASX` framework are currently being used in a number of large-scale distributed systems including the AT&T Q.port ATM signaling software product, the network management portion of the Motorola IRIDIUM global personal communications system, and a family of telecommunication switch management systems developed at Ericsson/GE Mobile Communications. In addition, the `ASX` framework has being used in the ADAPTIVE Communication Environment (ACE) [16]. ACE facilitates experimentation with various aspects of communication subsystems (such as flexible process architectures for multi-processor-based communication protocol stacks [4] and adaptive protocol reconfiguration techniques [16]) and distributed applications (such as extensible frameworks for concurrent event demultiplexing [5]). A freely available subset of the `ASX` framework described in this paper may be obtained via anonymous ftp from `ics.uci.edu` in the files `gnu/C++_wrappers.tar.Z` and `gnu/C++_wrappers_doc.tar.Z`.

Acknowledgements

We would like to thank to Paul Stephenson of Ericsson/GE Mobile Communications for helping to refine many of the object-oriented components described in this paper. We would also like to thank the anonymous reviewers for their suggestions that improved the structure and content of this paper.

References

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

[2] 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.

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

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

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

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

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

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 259–270, Jan. 1993.

[16] 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.