Experiences with an Object-Oriented Architecture for Developing Dynamically Extensible Distributed System Management Software

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130

Tatsuya Suda
suda@ics.uci.edu
Information and Computer Science Department
University of California, Irvine, CA 92717[1]

An earlier version of this paper appeared in the proceedings of the IEEE GLOBECOM conference, November 27^{th} to December 1^{st}, 1994.

Abstract

*Developing extensible, robust, and efficient distributed applications 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 C++ components. These tools and components simplify the development, configuration, and reconfiguration of distributed applications in a heterogeneous environment. Using* ASX *applications may be configured dynamically to contain multiple network services that execute concurrently in one or more processes or threads. Components in the* ASX *framework have been ported to UNIX and Windows NT and are currently being used in a number of large-scale production distributed systems. This paper describes our experience gained using the* ASX *framework to build highly modular, reusable, and extensible software for a family of distributed system management applications.*

1 Introduction

The demand for extensible, robust, and efficient distributed systems is increasing dramatically in research and commercial environments. Although distributing application services among a set of autonomous hosts offers many potential benefits, developing distributed systems is more complex than developing non-distributed systems. Much of this complexity arises from limitations with conventional tools and design techniques used to develop distributed application software. In particular, network programming tools (such as sockets, named pipes, and RPC) available in contemporary operating systems (such as UNIX, Windows NT, and OS/2) lack type-safe, portable, re-entrant, and extensible programming interfaces. For example, both sockets and named pipes identify endpoints of communication via weakly-typed I/O descriptors. The use of these descriptors increases the potential for subtle run-time errors.

Another major source of software complexity stems from the widespread use of algorithmic design techniques to develop distributed application software [1]. Many distributed systems are developed using algorithmic design techniques that result in monolithic, non-extensible software architectures [2]. This problem is exacerbated by the fact that the source code examples in popular network programming textbooks [3, 4, 5] are based on algorithmic-oriented design and implementation techniques.

Object-oriented frameworks help to alleviate the complexity associated with developing distributed application software. A framework is an integrated collection of software components that collaborate to produce a reusable architecture for a family of related applications [6]. Object-oriented frameworks are becoming increasingly popular as a means to simplify and automate the development and configuration of complex applications in domains such as graphical user interfaces [7, 8], databases [9], operating system kernels [10], and communication subsystems [11, 12].

The components in a framework typically include *classes* (such as message managers and timer-based event managers, and connection maps [13]), *class hierarchies* (such as an inheritance lattice of mechanisms for local and remote interprocess communication [14]), *class categories* (such as a family of concurrency mechanisms [12]), and *objects* (such as an event demultiplexer [15]). By emphasizing the integration and collaboration of application-specific and application-independent components, frameworks enable larger-scale reuse of software, compared to reusing individual classes and stand-alone functions.

To illustrate how object-oriented frameworks are being applied successfully in practice, this paper examines the features, structure, and usage of the ADAPTIVE Service eXecutive (ASX). We developed the ASX framework to provide an integrated collection of reusable, object-oriented network software components. These components simplify the development of distributed applications by enhancing the modularity, extensibility, reusability, and portability of software that utilizes operating system (OS) concurrency, explicit dynamic linking, interprocess communication (IPC), and I/O demultiplexing mechanisms.

[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).

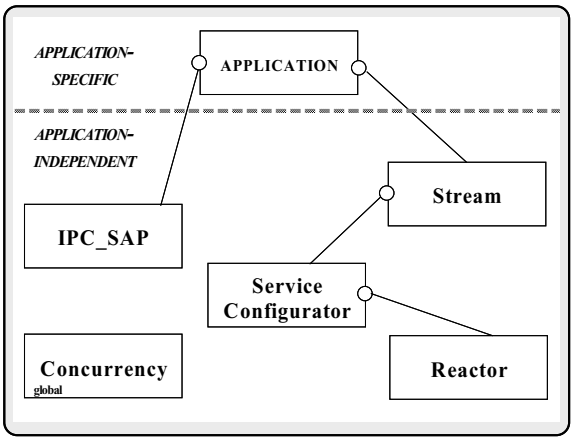

Figure 1: Class Categories in the ASX Framework

In addition to describing the object-oriented architecture of the ASX framework, this paper describes our experiences using the ASX framework to develop commercial software for a family of distributed system management applications at a major telecommunications company. These applications manage private-branch exchange (PBXs) switches and public central-office telecommunication switches across heterogeneous hardware and software platforms.

This paper is organized in the following manner: Section 2 outlines the architectural components in the ASX framework; Section 3 examines the object-oriented structure of a production distributed Call Center Management system built using ASX; and Section 4 presents concluding remarks.

2 The Object-Oriented Architecture of the ASX Framework

This section outlines the class categories in the ASX framework. A class category is a collection of software components that collaborate to provide a set of related services [1]. The architecture of the ASX framework was developed incrementally by generalizing from extensive experience gained by building a wide range of distributed systems (including online transaction processing systems [15], telecommunication switch monitoring systems [16], and parallel communication subsystems [12]). After developing several prototypes and iterating through a number of alternative designs, the class categories illustrated in Figure 1 were identified and implemented.[2]

Using ASX, a distributed application may be formed by combining and customizing components in the following class categories via object-oriented language features such as data abstraction, inheritance, dynamic binding, object composition, and template instantiation:

[2]Throughout the paper, object-oriented component relationships are illustrated via Booch notation [1]. Solid clouds indicate objects; nesting indicates composition relationships between objects; and undirected edges indicate a link exists between two objects. Solid rectangles indicate class categories, which combine a number of related classes into a common name space.

• **Stream Class Category:** These components are responsible for coordinating the configuration and run-time execution of one or more Streams [12]. A Stream is an object used to configure and execute application-specific services in the ASX framework. A Stream contains a series of inter-connected Module objects that may be linked together *statically* by developers at compile-time or *dynamically* by administrators or applications at installation-time and at run-time. Modules are used to decompose the architecture of a distributed application into functionally distinct layers. Each layer implements a cluster of related services (such as an end-to-end transport service, a presentation layer formatting service, or an event routing service for monitoring the behavior of telecom switches [17]). Every Module contains a pair of Queue objects that are used to partition a layer into its constituent read-side and write-side processing functionality.

A distributed application may be implemented as an interconnected series of Module objects that communicate with adjacent Modules by exchanging typed message objects. Modules may be joined together statically and/or dynamically in essentially arbitrary configurations to satisfy application requirements and enhance component reuse.

• **Service Configurator Class Category:** These components are responsible for inserting and removing the run-time address bindings of services implemented by Modules in shared object libraries. The Service Configurator components provide an extensible object-oriented interface and a configuration scripting language that automates the use of OS mechanisms for explicit dynamic linking [16]. This enables one or more Streams to be dynamically reconfigured *without* requiring the modification, recompilation, relinking, or restarting of executing applications.

• **Reactor Class Category:** These components are responsible for demultiplexing various types of events. Sources of events include I/O-based events received on communication ports, time-based events generated by a timer-driven callout queue, or signal-based events [15]. When these events occur at run-time, the Reactor dispatches the appropriate methods of pre-registered handler(s) to process the events.

The Reactor encapsulates the select, poll, and WaitForMultipleObjects I/O demultiplexing system calls via a portable, extensible, and type-safe object-oriented interface. 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. WaitForMultipleObjects is a Windows NT system call that provides similar demultiplexing functionality.

• **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 [12]. In the ASX framework, services may be executed at run-time using several different types of OS multitasking mechanisms such as kernel-based and/or user-based threads. By decoupling service behavior from the type of mechanisms used to invoke a service, the ASX framework

increases the range of concurrency configuration alternatives available to developers [12].

• **IPC_SAP Class Category:** These components are responsible for receiving and transmitting data with their peers residing on other processes in local or remote hosts. The `IPC_SAP` components encapsulate standard OS local and remote IPC mechanisms (such as UNIX sockets and Windows NT named pipes) within a type-safe object-oriented interface [14]. To improve service portability, the `IPC_SAP` classes may be used in conjunction with object-oriented language features (such as inheritance and parameterized types) to minimize an application's reliance on a particular type of IPC mechanism.

The lines that connect the class categories in Figure 1 indicate dependency relationships. For example, components implementing 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.

The `ASX` framework incorporates concepts from several other modular communication frameworks such as the System V STREAMS [18], the *x*-kernel [13], and the Conduit framework [11]. These frameworks contain features that support the flexible configuration of network software via the inter-connection of building-block protocol and service components. In general, these frameworks encourage the development of standard communication-related components by decoupling application-specific processing functionality from the application-independent communication framework infrastructure.

3 Implementing a Distributed System with the `ASX` Framework

The `ASX` framework is being used to develop a family of distributed system management applications that monitor and control PBX and central-office telecommunication switches in a Call Center Management (CCM) system. A CCM system provides a set of services that allow the staff of a call center (such as an airline reservation center or an insurance claims processing center) to assess the performance of the call center and the quality of service provided to customers. This section describes the behavior of the CCM system and illustrates `ASX` framework components that were used to implement the distributed CCM software.

3.1 Overview of the Call Center Management System

The `ASX`-based CCM system processes information that is generated continuously by system operators and telecommunication switches. Supervisors use this information to interactively monitor and optimize system performance, as well as to forecast future allocation of resources (such as operators and switch capacity) to meet customer demands.

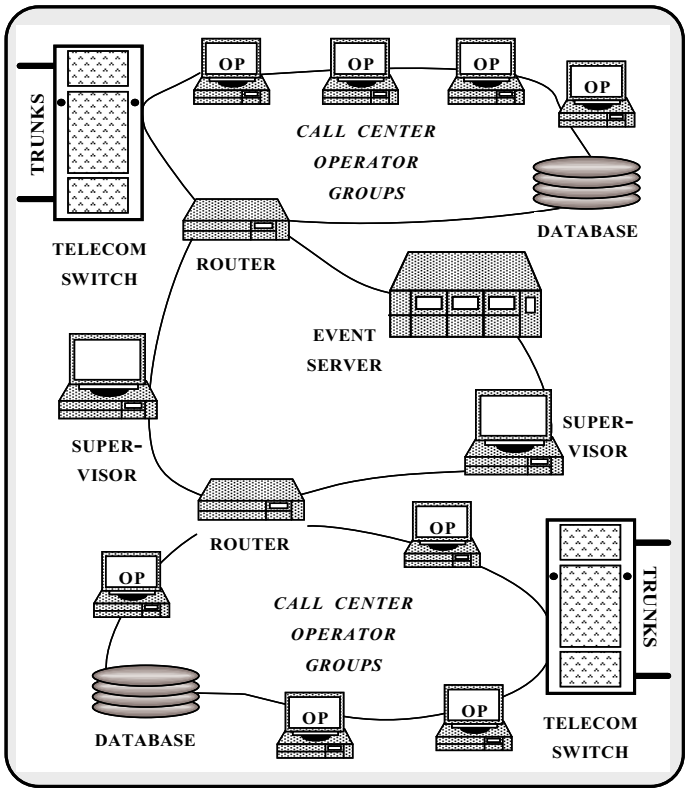

Figure 2: Distributed Components in the Call Center Management System

Figure 2 illustrates the distributed architecture of the CCM system. In this system, telecommunication switches route incoming customer calls (arriving on trunks) to an available operator (attached via a LAN). Operator interactions with customers are expedited by graphical user interfaces on hosts that access a database of customer account records. The CCM system continuously generates performance data (known as "activity events") that reports the activities of operators and switches. These activity events are sent automatically to a central event server, which runs on a separate network connected to the operator group networks. The event server is a mediator that analyzes, filters, transforms, and forwards the activity events it receives to other hosts throughout the network. CCM applications on these hosts summarize the activity events they receive, and display them to supervisors in a concise, graphical format.

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 telecommunication switches (such as PBX and central-office switches), host platforms (such as Windows NT, Windows 3.1, OS/2, and UNIX), and wide-area

and local-area networks (such as X.25, TCP/IP, and Novell IPX/SPX). Configuration flexibility is necessary since not all call center installation sites require every feature provided by the CCM system.

It would be possible (although highly undesirable) to manually construct and deliver one or more CCM systems that are customized for the platforms and the subsets of features required by a particular site. However, such a static configuration process would require the selection of services and the division of labor between different hosts in a distributed CCM system to be completely fixed during initial system deployment. Our experience with earlier-generation CCM systems indicated that even if this information was available at the time of deployment, it was likely to change in the future, often upon short notice.

The ASX framework facilitates both platform independence and configuration flexibility to improve software component reuse across platforms and to reduce development effort. For example, C++ abstract base classes, inheritance, dynamic binding, and parameterized types are used extensively throughout the CCM system to localize and minimize platform dependencies. Likewise, the ASX framework is used to defer the point of time at which a particular set of services are configured to form a CCM application. By combining advanced OS features (such as multi-threading and explicit dynamic linking) and C++ language features (such as templates, inheritance, and dynamic binding), the ASX framework enables services offered by CCM applications to be extended without modifying, recompiling, relinking, or even restarting the system at run-time [16].

3.2 Mapping CCM Functionality onto ASX Components

Figure 3 illustrates the ASX framework components used to implement the event server portion of the CCM system (the other components in the CCM system are not presented in this example). The event server performs the following tasks:

- It receives activity events generated continuously by operator hosts and telecom switches;
- It analyzes and filters the activity events it receives to determine which actions to perform, as well as which supervisors should receive which incoming activity events;
- It forwards the filtered activity events across a network to the subset of supervisor hosts that have previously subscribed to receive these events.

The CCM event server is composed of four hierarchically-related Modules that may be configured statically and/or dynamically by the run-time environment available in the ASX framework. The use of ASX Modules helps to improve the platform independence of the CCM system by encapsulating non-portable system mechanisms (such as communication protocols and activity event frame formats) behind abstract interfaces. This section outlines the

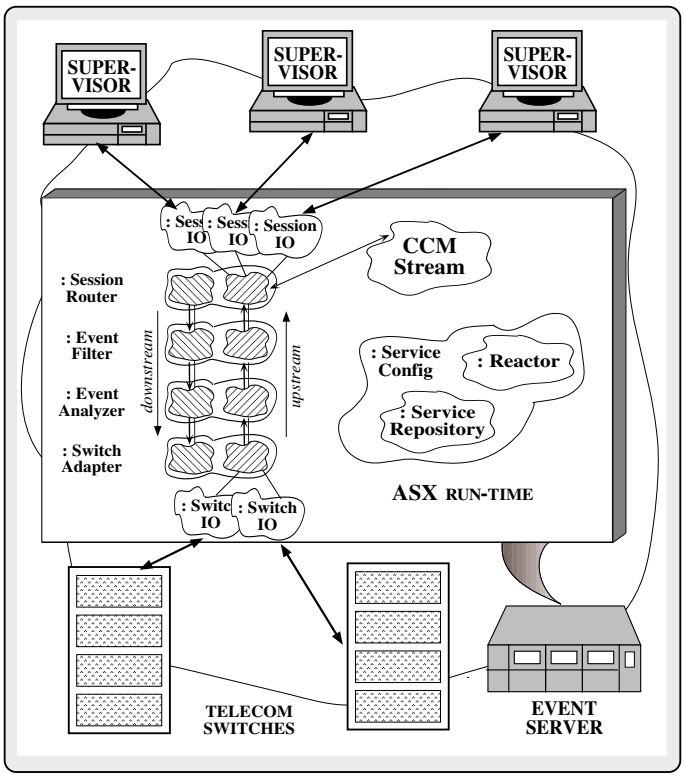

Figure 3: ASX Components in a Distributed Call Center Manager

behavior of the Switch_Adapter, Event_Analyzer, Event_Filter, and Session_Router Module objects that comprise the CCM event server.

3.2.1 The Switch_Adapter Module

The Switch_Adapter Module object coordinates communication between the CCM event server and the telecom switches monitored by the event server. This Module shields the higher 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 parsing incoming activity events from switches. These activity events are transformed and encapsulated into a canonical switch-independent message object, which is built atop a flexible message management class described in [12]. After being allocated and initialized, the incoming canonical message objects are passed upstream to the Event_Analyzer Module.

3.2.2 The Event_Analyzer Module

The Event_Analyzer Module is used to transform switch-specific activity events into a switch-independent format. This transformation process helps to improve system portability. For instance, only the Switch_Adapter and Event_Analyzer portion of the event server was af-

fected significantly when the original PBX-based version of the CCM system was ported to a different central-office switch architecture. During the event analysis process, the Event_Analyzer also synthesizes switch-independent derived events that are triggered by the occurrence of one or more switch-specific events. After the Event_Analyzer has transformed and/or synthesized incoming activity events, it forwards the new events to the Event_Filter Module.

3.2.3 The Event_Filter Module

The Event_Filter Module minimizes unnecessary network traffic by forwarding only those activity events that at least one supervisor has subscribed to receive. The Event_Filter contains a collection of Event Forwarding Discriminator (EFD) objects. An EFD object contains a predicate that indicates the type of activity event(s) a supervisor wants to receive. An EFD predicate may be used to selectively filter out activity events based on criteria such as event type, event value, event generation time, and event frequency. An EFD predicate may contain relational operators that allow the composition of arbitrarily complex filter expressions.

During system configuration, supervisors may subscribe to receive particular events by registering EFD objects with the event server. During system execution, the event server inspects the EFDs to determine the set of supervisors that should receive each incoming activity event. If an activity event matches a supervisor's EFD predicate, the supervisor's addressing information is added to the Session_Set in the message object that encapsulates the activity event. After all the EFDs are inspected by the Event_Filter Module, the message object containing the activity event and the Session_Set is passed upstream to the Session_Router Module.

3.2.4 The Session_Router Module

The Session_Router Module is a reusable ASX component. It shields the lower layers of the CCM event server 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 separate Session_IO object is created to manage each supervisor session. This Session_IO object use an IPC_SAP object to handle all the data transfer and control operations between the event server and a supervisor. After connecting to the event server, a supervisor indicates the type of activity event(s) he or she would like to monitor. Subsequently, when the Session_Router receives a message object from the Event_Filter Module, it automatically multicasts the message to all the supervisors indicated by the addressing information residing in the message object's Session_Set.

The Service_Config object illustrated in the middle of Figure 3 is a reusable component from the ASX framework's Service Configurator class category [16]. The event server uses this object to control the initial configuration, subsequent reconfiguration(s), and termination of Modules from the Stream class category. Modules may be configured statically at installation-time or reconfigured dynamically at run-time. The Service_Config object integrates other ASX framework components such as the Service_Repository and the Reactor. The Service_Repository is an object manager that simplifies the run-time configuration and administration of the Modules used to implement layered communication services in a Stream. The Reactor is an event demultiplexer that dispatches incoming messages from supervisors and activity events from switches to the appropriate Session_IO and Switch_IO event handlers, respectively. Messages arriving from supervisors are received by a Session_IO object, sent downstream through the CCM_Stream object, and handled by the appropriate Module (*e.g.,* EFD subscriptions are handled by the Event_Filter Module). Likewise, incoming events from switches are received by a Switch_IO object and sent upstream starting at the Switch_Adapter Module.

3.3 CCM Event Server Configuration

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 applications. The ASX framework provides this high degree of flexibility by combining OS explicit dynamic linking mechanisms with a configuration scripting language (described in [16]). For example, the Service Configurator class category uses following configuration script to determine which services to dynamically link into the address space of the CCM event server at installation-time:

```
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.so:alloc()
  dynamic Session_Router
    Module * /svcs/SR.so:alloc() "-p 2010"
}
```

The configuration script shown above indicates the order in which the four Modules in the CCM event server are dynamically linked and pushed onto the CCM_Stream object. During the installation of the event server, the Service_Config class parses this configuration script and carries out the directives described by each entry, as follows:

1. The dynamic directive instructs the ASX framework to dynamically link the shared object file (specified by a pathname ending in .so) into the address space of the CCM event server;

2. The framework then consults the shared object's symbol table to locate, extract, and invoke the `alloc` function, which allocates an instance of the specified `Module` object;
3. The framework then invokes the initialization method of the two `Queues` in the `Module`, passing in any initialization parameters (which appear as string literals at the end of each line);
4. At this point, the framework enters an event loop that waits for control messages to arrive from supervisors or for activity events to arrive from switches and/or operator hosts.

When events arrive at run-time, the `Reactor` automatically dispatches the appropriate callback methods of the `Switch_IO` and `Session_IO` objects to initiate Stream processing.

3.4 CCM Event Server Reconfiguration

This section motivates and illustrates the dynamic reconfiguration mechanisms provided by the `ASX` framework. A major objective of the CCM project was to allow developers to decide very late in the development cycle (*i.e.,* at installation-time or run-time) which services would run in supervisor hosts and which would run in the event server. Our experience with earlier versions of the CCM system indicated that it was difficult to determine the appropriate mapping of services onto hosts *a priori* since processing characteristics, workloads, and OS/hardware platforms vary over time.

The run-time control environment provided by the `ASX` framework supports the flexible reconfiguration requirements of the CCM system. This flexibility proved to be quite useful for the CCM project since different OS/hardware platforms and different network characteristics required different service configurations. For example, in some environments the event server performed most of the work since it ran on a multi-processor platform, whereas the supervisor hosts were inexpensive PCs attached to the event server via networks that possess low-bandwidth or are highly congested. Conversely, in other environments the supervisor hosts performed most of the work since these hosts were powerful workstations connected to a high-speed network.

The `CCM_Stream` shown in Figure 3 performs all event analysis and event filtering processing directly in the event server. As described above, however, this configuration may not be appropriate for certain CCM environments. For example, performance may be degraded if supervisors configure a large number of Event Forwarding Discriminator (`EFD`) objects into an event server. In this case, the centralized event server becomes a bottleneck and performance suffers, even if surplus processing capacity were available in the network and in the supervisor hosts. Figure 4 illustrates how the configuration shown in Figure 3 may be modified to operate efficiently in a distributed environment where event server processing constitutes the primary performance bottleneck.

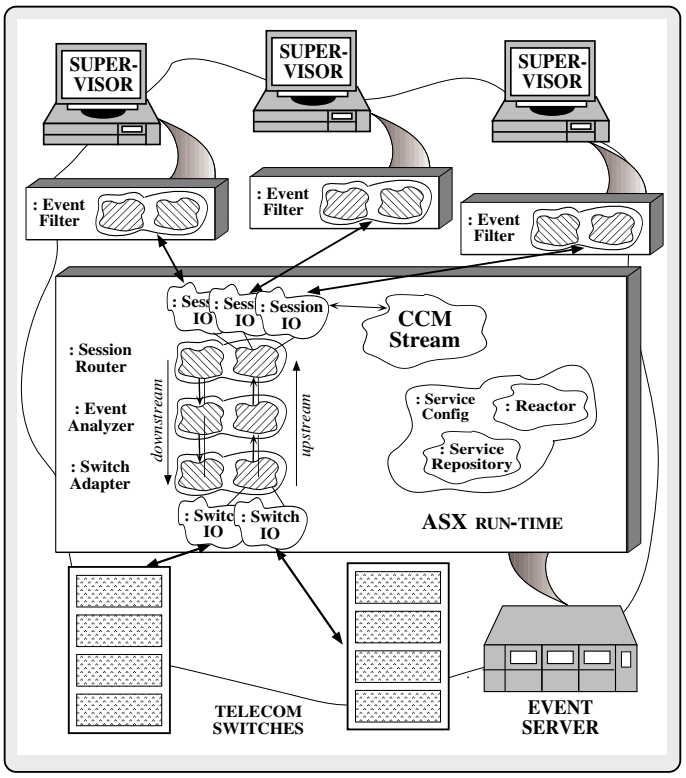

Figure 4: Reconfiguration of the Call Center Manager

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

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

This script transfers the event filtering functionality from the event server to the supervisor hosts using the following steps:

1. It suspends the event server's `CCM_Stream` object;
2. It removes the `Event_Filter Module` from the `CCM_Stream` and dynamically unlinks the associated shared object library;
3. It then dynamically links the `Event_Filter Module` into Streams on all the supervisor hosts;
4. Finally, it resumes the event server's `CCM_Stream` object.

As a result of this reconfiguration, the overhead of event filtering is distributed among all the supervisor hosts, as opposed to being centralized at the event server.

We are currently evaluating the performance of the CCM distributed architecture depicted in Figures 3 and 4 to determine how to parallelize the CCM event server more effectively. Using the ASX framework, it is straightforward to reconfigure the binding of threads onto Modules or messages in order to reduce programming effort and improve performance [12]. We are also investigating service migration policies to formulate guidelines that ensure the dynamic reconfiguration of the event server does not disrupt or corrupt active services. A more ambitious long-term project involves using the ASX reconfiguration mechanisms to experiment with service migration policies that relocate certain services dynamically to reduce overall system workload at run-time.

4 Concluding Remarks

The ASX framework provides an object-oriented infrastructure that supports static and/or dynamic configuration of network services that execute within one or more OS processes and threads. The object-oriented design principles underlying the ASX framework separate policies from mechanisms via object-oriented language features (such as abstract base classes, inheritance, dynamic binding, and parameterized types). This separation of concerns enhances the reuse of common distributed application software components. These components include C++ wrappers for local and remote IPC mechanisms [14]; frameworks for event demultiplexing and service dispatching [15]; tools for automating service configuration and reconfiguration [17]; and C++ subclasses that encapsulate and enhance various dynamic linking and concurrency mechanisms [16].

Component reuse is also facilitated in the ASX framework by decoupling the higher-level application-specific policies (such as activity event filtering) from the lower-level application-independent mechanisms (such as the choice of mechanisms for network communication, event demultiplexing and service dispatching, and process/thread execution agents). In addition, the ASX framework helps to decouple application-specific service functionality from the binding onto OS processes and threads in order to improve flexibility and performance. Explicit dynamic linking and dynamic binding are also utilized to help improve extensibility and permit fine-grained time/space tradeoffs. Together, the object-oriented design principles and OS/language features facilitate the development of network services that may be updated and extended without modifying, recompiling, relinking, or restarting existing applications at run-time [16].

The ASX framework described in this paper has been used in a production environment to simplify the configuration, installation, and administration of a family of distributed system management applications for a major telecommunications company. By using the ASX framework, developers have been able to enhance distributed application functionality and reliability, as well as fine-tune system performance, without extensive redevelopment and re-installation effort. For example, debugging a faulty service typically involves dynamically reinstalling a functionally equivalent service containing additional instrumentation that helps to isolate the source of the erroneous behavior.

Thus far, the primary obstacles encountered by using object-oriented techniques and C++ have been managerial and tool-related, rather than technical problems. For example, it is difficult to find experienced systems analysts, designers, and programmers who are intimately familiar with applying C++ and object-oriented design methods in distributed communication system environments. Furthermore, the level of maturity of many C++ compilers and language processing tools has been inadequate on UNIX, Windows NT, and OS/2 platforms. For example, many C++ debuggers do not support multi-threading correctly and many C++ compilers implement only a subset of the language.

Over time, it is likely that the tool-related concerns will become less problematic, particularly once the ISO/ANSI C++ standard is adopted. However, for the interim period, it is essential to staff complex software projects carefully to minimize development risks. For the CCM project, we found it useful to hire a small number of experienced OOD/C++ experts. These experts have worked closely with less experienced developers to shepard them through the object-oriented learning curve.

Components in the ASX framework are 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). [19] ACE facilitates experimentation with various aspects of communication subsystems (such as flexible process architectures for multi-processor-based communication protocol stacks [12] and adaptive protocol reconfiguration techniques [19]) and distributed applications (such as extensible frameworks for concurrent event demultiplexing [15]). Public domain versions of the ASX framework described in this paper is available via anonymous ftp from ics.uci.edu in the file gnu/C++_wrappers.tar.Z.

References

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

[2] D. C. Schmidt, "ASX: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6^{th} USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

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

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

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

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

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

[8] A. Weinand, E. Gamma, and R. Marty, "ET++ - an object-oriented application framework in C++," in *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, pp. 46–57, ACM, Sept. 1988.

[9] 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.

[10] 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.

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

[12] M. Jayaram, R. K. Cytron, D. C. Schmidt, and G. Varghese, "Efficient Demultiplexing of Network Packets by Automatic Parsing," in *Submitted to the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, ACM, 1994.

[13] 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.

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

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

[16] 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.

[17] 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.

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

[19] 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.