The ADAPTIVE Communication Environment
An Object-Oriented Network Programming Toolkit for Developing Communication Software

Douglas C. Schmidt
schmidt@cs.wustl.edu
http://www.cs.wustl.edu/~schmidt/
Department of Computer Science
Washington University
St. Louis, MO 63130, (314) 935-7538

Earlier versions of this paper appeared in the 11^{th} and 12^{th} Sun User Group conferences in San Jose, California, Dec. 7–9, 1993 and San Francisco, California, June 14–17, 1993.

Abstract

The ADAPTIVE Communication Environment (ACE) is an object-oriented (OO) toolkit that implements fundamental design patterns for communication software. ACE is targeted for developers of high-performance communication services and applications on UNIX and Win32 platforms. ACE simplifies the development of OO network applications and services that utilize interprocess communication, event demultiplexing, explicit dynamic linking, and concurrency. ACE automates system configuration and reconfiguration by dynamically linking services into applications at run-time and executing these services in one or more processes or threads.

This paper describes the structure and functionality of ACE and illustrates core ACE features using examples from domains like telecommunications, enterprise medical imaging, and WWW services. ACE is freely available and is being used for many commercial projects (such as Ericsson, Bellcore, Siemens, Motorola, Kodak, and McDonnell Douglas), as well as many academic and industrial research projects. ACE has been ported to a variety of OS platforms including Win32 and most UNIX/POSIX implementations. In addition, both C++ and Java versions of ACE are available.

1 Introduction

1.1 Problem: the Distributed Software Crisis

The demand for robust and high-performance distributed computing systems is steadily increasing. Examples of these types of systems include global personal communication systems, network management platforms, enterprise medical imaging systems, online financial analysis systems, and real-time avionics systems. Distributed computing is a promising technology for improving collaboration through connectivity and interworking; performance through parallel processing; reliability and availability through replication; scalability and portability through modularity; extensibility through dynamic configuration and reconfiguration; and cost effectiveness through resource sharing and open systems.

Although distributed computing offers many potentially benefits, developing communication software is expensive and error-prone. Object-oriented (OO) programming languages, components, and frameworks are widely touted technologies for reducing software cost and improving software quality. When stripped of the hype, the primary benefits of OO stem from the emphasis on modularity and extensibility, which encapsulate volatile implementation details behind stable interfaces and enhance software reuse.

Developers in certain well-traveled domains have successfully applied OO techniques and tools for years. For instance, the Microsoft MFC GUI framework and OCX components are *de facto* industry standards for creating graphical business applications on PC platforms. Although these tools have their limitations, they demonstrate the productivity benefits of reusing common frameworks and components.

Software developers in more complex domains like telecommunications, medical imaging, avionics, and online transaction processing have traditionally lacked standard off-the-shelf middleware components. As a result, developers largely build, validate, and maintain software systems from scratch. In an era of deregulation and stiff global competition, this in-house development process is becoming prohibitively costly and time consuming. Across the industry, this situation has produced a “distributed software crisis,” where computing hardware and networks get smaller, faster, and cheaper; yet distributed software gets larger, slower, and more expensive to develop and maintain.

The challenges of building distributed software stem from *inherent* and *accidental* complexities [1] associated with distributed systems. Inherent complexities stem from fundamental challenges of developing distributed software. Chief among these is detecting and recovering from network and host failures, minimizing the impact of communication latency, and determining an optimal partitioning of service components and workload onto processing elements throughout a network.

Accidental complexities stem from limitations with tools and techniques used to develop telecom software. For instance, many standard networking mechanisms (such as sock-

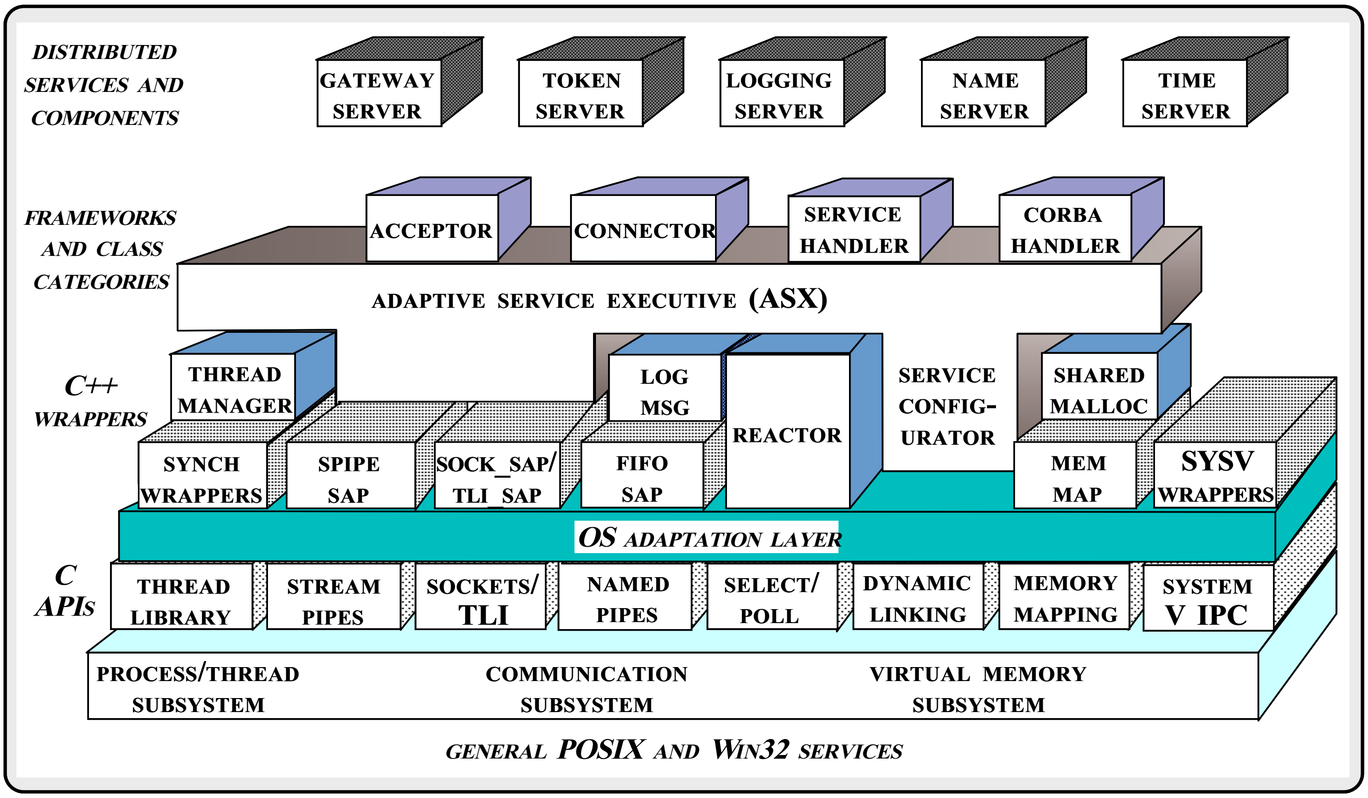

Figure 1: Components in the ADAPTIVE Communication Environment

ets [2] and TLI [3]) and reusable component libraries (such as X windows and Sun RPC) lack type-safe, portable, re-entrant, and extensible *application programming interfaces* (APIs). Likewise, common network programming interfaces like sockets and TLI use weakly-typed integer handles that can lead to subtle run-time errors [4].

Another source of complexity arises from the widespread use of algorithmic decomposition [5], which results in non-extensible and non-reusable software systems [6]. Although graphical user-interfaces (GUIs) are commonly built using object-oriented (OO) techniques, distributed software is typically developed using algorithmic decomposition. This problem is exacerbated by the fact that examples in popular network programming textbooks [7, 8, 3] are based on algorithmically-oriented design and implementation techniques.

The lack of extensibility and reuse in-the-large is particularly problematic for complex distributed software. Extensibility is essential to ensure timely modification and enhancement of services and features. Reuse is essential to leverage the domain knowledge of expert developers to avoid re-developing and re-validating common solutions to recurring requirements and software challenges.

1.2 Solution: Object-oriented Design Patterns and Frameworks

Object-oriented design patterns and frameworks are well-regarded for their ability to help alleviate costly rediscovery and reinvention of core distributed software concepts and abstractions. Patterns provide a way to encapsulate design knowledge that offers solutions to standard distributed software development problems [9]. For instance, patterns are useful for describing recurring *micro-architectures* (such as Reactor [10] and Active Object [11]), which are abstractions of common object-structures that have proven useful to build distributed communication software. However, abstractions documented as patterns do not directly yield reusable code. Therefore, it is essential to augment the study of patterns with the creation and use of *frameworks*.

Frameworks provide reusable software components for applications by integrating sets of abstract classes and defining standard ways that instances of these classes collaborate [12]. Frameworks instantiate families of design patterns to help developers avoid costly reinvention of common distributed software components. The results are "semi-complete" application skeletons that can be customized by inheriting and instantiating from reuseable building blocks components in the frameworks. Since frameworks are tightly integrated with key distributed programming tasks (such as service initialization, error handling, flow control, event demultiplexing, concurrency control), the scope of reuse can be significantly larger than by using traditional function libraries, or even conventional OO class libraries.

This paper is organized as follows: Section 2 presents an overview of the structure and functionality of the ACE toolkit; Section 3 describes the ACE C++ wrapper components and higher-level ACE framework components and patterns in detail; Section 4 examines the implementation of several networking applications built using ACE; and Section 5 presents concluding remarks.

2 Overview of the ADAPTIVE Communication Environment (ACE)

To illustrate how OO patterns and frameworks are being successfully applied to distributed software, this paper examines the ADAPTIVE Communication Environment (ACE) [6]. ACE is a freely available OO toolkit containing a rich set of reusable wrappers, class categories, and frameworks that perform common network programming tasks across a wide range of OS platforms. The tasks provided by ACE include:

- Event demultiplexing and event handler dispatching [13, 14, 10, 15];
- *Connection establishment and service initialization* [16, 17, 18];
- *Interprocess communication* [19, 4] and *shared memory management*;
- *Dynamic configuration of distributed communication services* [20, 21];
- *Concurrency/parallelism and synchronization* [22, 23, 11, 24];
- *Components for higher-level distributed services* (such as a Name service, Event service, Logging service, Time service, and Token service).

The ACE toolkit is designed using a layered architecture. Figure 1 illustrates the vertical and horizonal relationship between ACE components. The lower layers of ACE are *OO wrappers* that encapsulate existing OS network programming mechanisms. The higher layers of ACE extend the wrappers to provide *OO frameworks and components* that cover a broader range of application-oriented networking tasks and services. The remainder of this section presents an overview of the structure and functionality of the class categories in ACE (shown in Figure 2). Section 3 provides in-depth coverage of the ACE network programming features and components.

Throughout the paper, the ACE components are illustrated via Booch notation [5]. 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

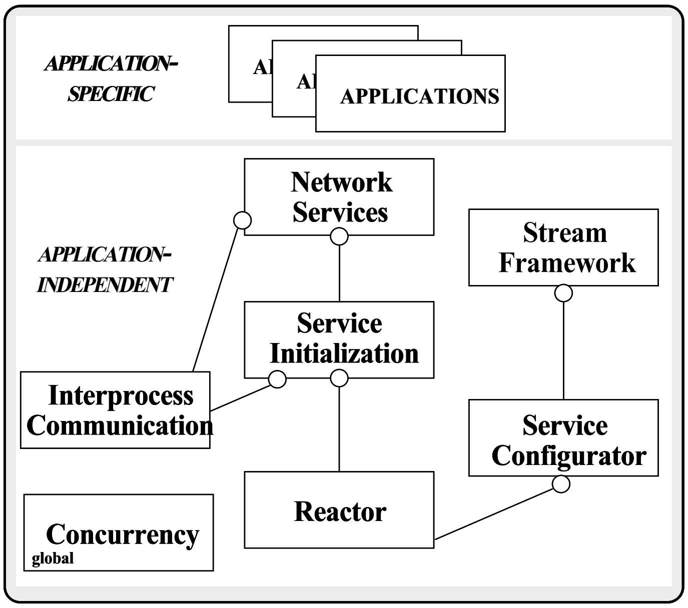

Figure 2: The Class Categories in ACE

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. The "A" inscribed within a triangle identifies a class as an *abstract class* [25]. An abstract class cannot be instantiated directly, but must be subclassed. Subclasses of an abstract class must provide definitions for all its abstract methods before any objects of the class may be instantiated.[1]

2.1 The ACE OS Adaptation Layer

The ACE source tree contains over 85,000 lines of C++. Approximately 9,000 lines of code (*i.e.,* about 10% of the total toolkit) are devoted to the *OS Adaptation Layer*. This layer shields the higher layers of ACE from platform-specific dependencies associated with the following OS mechanisms:

- Multi-threading and synchronization
- Interprocess communication
- Event demultiplexing
- Explicit dynamic linking
- Memory-mapped files and shared memory

2.2 The ACE OO Wrappers

Above the OS Adaptation Layer are OO wrappers that encapsulate and enhance the concurrency, interprocess communication (IPC), and virtual memory mechanisms (illustrated at the bottom of Figure 1) available on modern operating systems like Win32 and UNIX. Applications can combine and

[1] Abstract methods C++ are commonly called *pure virtual functions*.

compose these components by selectively inheriting, aggregating, and/or instantiating the following ACE wrapper class categories:

- `IPC SAP` – which encapsulates local and/or remote IPC Service Access Point (`IPC SAP` mechanisms such as sockets, TLI, UNIX FIFOs and STREAM pipes, and Win32 Named Pipes, [19, 4];
- **Service Initialization** – ACE provides a set of Connector and Acceptor components [18] that decouple the active and passive initialization roles, respectively, from the tasks a communication service performs once initialization is complete;
- **Concurrency mechanisms** – ACE abstracts lower-level OS multi-threading and multi-processing mechanisms (such as mutexes and semaphores [22]) to create higher-level OO concurrency abstractions (such as Active Objects [11]);
- **Memory management mechanisms** – the ACE memory management components provide a flexible and extensible abstraction for managing dynamic allocation and deallocation of shared memory and local memory;
- **CORBA integration** – ACE can be integrated with CORBA implementations [26] (such as single-threaded and multi-threaded Orbix).

The use of OO wrappers improves application robustness by encapsulating OS communication, concurrency, and virtual memory mechanisms with type-secure OO interfaces. This alleviates the need for applications to directly access the underlying OS libraries, which are written using weakly-typed C interfaces. Therefore, compilers for OO languages like C++ and Java can detect type system violations at compile-time, rather than at run-time. The C++ version of ACE uses inlining extensively to eliminate performance penalties that would otherwise be incurred from the additional type-security and abstraction provided by the wrapper layer.

2.3 The ACE Framework

ACE contains a higher layer network programming framework that integrates and enhances the lower layer OS wrappers. This framework supports the dynamic configuration of concurrent network daemons composed of application services. The framework portion of ACE contains the following class categories:

- **Reactor** – The ACE Reactor [10] provides extensible, object-oriented demultiplexer that dispatches handlers in response to various types of events (*e.g.,* I/O-based, timer-based, signal-based, and synchronization-based events);
- **Service Configurator** – The ACE Service Configurator [21] supports the construction of applications whose services may be configured dynamically at installation-time and/or run-time;

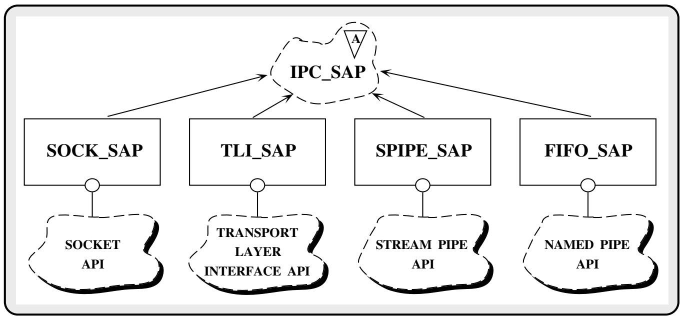

Figure 3: `IPC SAP` Class Category Relationships

- **Streams** – The ACE Streams components [6] simplify the development of concurrent communication software applications composed of one or more hierarchically-related services (such as protocol stacks);

2.4 ACE Network Service Components

In addition to the wrappers and frameworks, ACE provides a standard library of network service components. These components play two roles in ACE:

1. They illustrate how to utilize the ACE IPC wrappers, Reactor, Service Configurator, Service Initialization, Concurrency, Memory Management, and Streams components;
2. They provide reusable components for common distributed system tasks such as logging [13, 13], naming, locking, and time synchronization [21];

When combined with OO language features (such as clases, inheritance, dynamic binding, and parameterized types) and design patterns (such as Abstract Factory, Builder, and Service Configurator), the reusable ACE components facilitate the development of communication services and applications that may be updated and extended without modifying, recompiling, relinking, or even restarting running software [20].

3 Detailed Coverage of ACE Components

3.1 IPC_SAP: Local and Remote IPC Mechanisms

ACE provides a forest of class categories rooted at the `IPC SAP` ("InterProcess Communication Service Access Point") base class. `IPC SAP` encapsulates the standard I/O handle-based OS local and remote IPC mechanisms that offer connection-oriented and connectionless protocols. As shown in Figure 3, this forest of class categories includes `SOCK SAP` (which encapsulates the socket API), `TLI SAP` (which encapsulates the TLI API), SPIPE SAP (which encapsulates the UNIX SunOS 5.x STREAM pipe API), and FIFO SAP (which encapsulates the UNIX named pipe API).

Each class category is organized as an inheritance hierarchy. Every subclass provides a well-defined interface to a 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). The use of classes (as opposed to stand-alone functions) helps to simplify network programming as follows:

- *Shield applications from error-prone details* – For example, the ACE_Addr class hierarchy shown in Figure 3 supports several diverse network addressing formats via a type-secure OO interface, rather than using the awkward and error-prone C-based struct sockaddr data structures directly.
- *Combine several operations to form a single operation* – For example, the SOCK Acceptor constructor performs the various socket system calls (such as socket, bind, and listen) required to create a passive-mode server endpoint.
- *Parameterize IPC mechanisms into applications* – Classes form the basis for parameterizing an application by the type of IPC mechanism it requires. This helps to improve portability as discussed in Section 3.1.2.
- *Enhance code sharing* – Inheritance-based hierarchical decomposition increases the amount of common code that is shared amongst the various IPC mechanisms (such as the OO interface to the lower-level OS device control system calls like fcntl and ioctl).

The following sections discuss each of the class categories in IPC SAP.

3.1.1 SOCK SAP

The SOCK SAP [4] class category provides applications with an object-oriented interface to the Internet-domain and UNIX-domain protocol families [8]. Applications may access the functionality of the underlying Internet-domain or UNIX-domain socket types by inheriting or instantiating the appropriate SOCK SAP subclasses shown in Figure 4. The ACE_SOCK* subclasses encapsulate Internet-domain functionality and the ACE_LSOCK* subclasses encapsulate UNIX-domain functionality. As shown in Figure 4, the subclasses may be further decomposed into (1) the *Dgram components (which provide unreliable, connectionless, message-oriented functionality) vs. the *ACE_Stream components (which provide reliable, connection-oriented, bytestream functionality) and (2) the ACE_*_Acceptor components (which provide connection establishment functionality typically used by servers) vs. the *Stream components (which provide bi-directional bytestream data transfer functionality used by both clients and servers).

Using OO wrappers to encapsulate the socket interface helps to (1) detect many subtle application type system violations at compile-time, (2) facilitate a platform-independent transport-level interface that improves application portability, and (3) greatly reduce the amount of application code and development effort expended upon lower-level network programming details. To illustrate the latter point, the following example program implements a simple client application that uses the ACE_SOCK_Dgram_Bcast class to broadcast a message to all servers listening on a designated port number in a LAN subnet::

```
int
main (int argc, char *argv[])
{
  ACE_SOCK_Dgram_Bcast b_sap (sap_any);
  char *msg;
  unsigned short b_port;

  msg = argc > 1 ? argv[1] : "hello world\n";
  b_port = argc > 2 ? atoi (argv[2]) : 12345;

  if (b_sap.send (msg, strlen (msg),
                  b_port) == -1)
    perror ("can't send broadcast"), exit (1);
  exit (0);
}
```

It is instructive to compare this concise example with the dozens of lines of C source code required to implement broadcasting using the socket interface directly.

3.1.2 TLI SAP

The TLI SAP class category provides an OO interface to the System V Transport Layer Interface (TLI). The TLI SAP inheritance hierarchy for TLI is almost identical to the SOCK SAP wrappers for sockets. The primary difference is that TLI and TLI SAP do not define an interface to the UNIX-domain protocol family. In addition, TLI is not currently ported to Win32 platforms.

By combining C++ features (such as default parameter values and templates) together with the tirdwr (the read/write compatibility STREAMS module), it becomes relatively straight-forward to develop applications that may be parameterized at compile-time to operate correctly over either a socket-based or TLI-based transport interface. For instance, the following code illustrates how C++ templates may be applied to parameterize the IPC mechanisms used by an application. This code was extracted from the distributed logging facility described in Section 4.1. In the code below, a subclass derived from ACE_Event_Handler is parameterized by a particular type of transport interface and its corresponding protocol address class:

```
/* Logging_Handler header file */
template <class PEER_STREAM, class ADDR>
class Logging_Handler : public ACE_Event_Handler
{
public:
  Logging_Handler (void);
  virtual ~Logging_Handler (void);

  virtual int handle_input (ACE_HANDLE);
  virtual ACE_HANDLE get_handle (void) const
  {
    return this->peer_stream_.get_handle ();
```

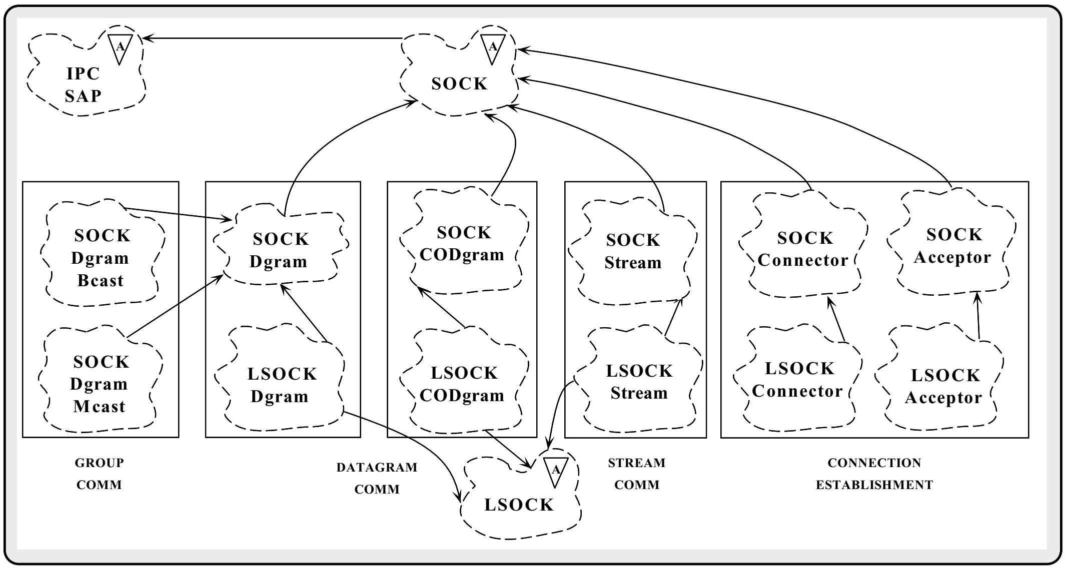

Figure 4: The `SOCK SAP` Class Categories

```
  }

protected:
  PEER_STREAM peer_stream_;
};
```

Depending on certain properties of the underlying OS platform (such as whether it is BSD-based SunOS 4.x or System V-based SunOS 5.x), the logging application may instantiate the `Client Handler` class to use either `SOCK SAP` or `TLI SAP`, as shown below:

```
/* Logging application */
class Logging_Handler
#if defined (MT_SAFE_SOCKETS)
: public Logging_Handler<ACE_SOCK_Stream, ACE_INET_Addr>
#else
: public Logging_Handler<ACE_TLI_Stream, ACE_INET_Addr>
#endif /* MT_SAFE_SOCKETS */
{
  /* ... */
};
```

The increased flexibility offered by this template-based approach is extremely useful when developing an application that must run portability across multiple OS platforms. In particular, the ability to parameterize applications according to transport interface is necessary across variants of SunOS platforms since the socket implementation in SunOS 5.2 is not thread-safe and the TLI implementation in SunOS 4.x contains a number of serious defects.

`TLI SAP` also shields applications from many peculiarities of the TLI interface. For example, the subtle application-level code required to properly handle the non-intuitive, error-prone behavior of `t_listen` and `t_accept` in a concurrent server with a `qlen` > 1 [3] is encapsulated within the `accept` method in the `TLI Acceptor` class. This method accepts incoming connection requests from clients. Through the use of C++ default parameter values, the standard method for calling the `accept` method is syntactically equivalent for both `TLI SAP`-based and `SOCK SAP`-based applications.

3.1.3 SPIPE SAP

The `SPIPE SAP` class category provides a OO wrapper interface for high-performance local IPC. On Win32 platforms, the `SPIPE SAP` class category is implemented atop Named Pipes. The Win32 Named Pipes mechanism is primarily used to transfer data efficiently among processes on the same machine. It is typically more efficient than sockets for local IPC [27].

On UNIX platforms, the `SPIPE SAP` class category is implemented with mounted STREAM pipes and `connld` [28]. SunOS 5.x provides the `fattach` system call that mounts a pipe handle at a designated location in the UNIX file system. A server application is created by pushing the `connld` STREAM module onto the mounted end of the pipe. When a client application running on the same host machine as the server subsequently opens the filename associated with the mounted pipe, the client and server each receive an I/O handle that identifies a unique, non-multiplexed, bi-directional channel of communication.

The `SPIPE SAP` inheritance hierarchy mirrors the one used for `SOCK SAP` and `TLI SAP`. It offers functionality that is similar to the `SOCK SAP ACE_LSOCK*` classes (which themselves encapsulate UNIX-domain sockets). However, on SunOS 5.x platforms `SPIPE SAP` is more flexible than the `ACE_LSOCK*` interface since it enables STREAM modules to be "pushed" and "popped" to and from `SPIPE SAP` endpoints, respectively. `SPIPE SAP` also supports bi-directional delivery of byte-stream and prioritized message-oriented data between processes and/or threads executing within the same host machine [29].

3.1.4 FIFO SAP

The `FIFO SAP` class category encapsulates the UNIX named pipe mechanism (also called FIFOs). Unlike

STREAM pipes, named pipes offer only a uni-directional data channel from one or more senders to a single receiver. Moreover, messages from different senders are all placed into the same communication channel. Therefore, some type of demultiplexing identifier must be included explicitly in each message to enable the receiver to determine which sender transmitted the message.

The STREAMS-based implementation of named pipes in SunOS 5.x provides both message-oriented and bytestream-oriented data delivery semantics. In contrast, some platforms, (such as SunOS 4.x) only provides bytestream-oriented named pipes. Therefore, unless fixed length messages are always used, each message sent via a named pipe in SunOS 4.x must be distinguished by some form of byte count or special termination symbol that allows a receiver to extract messages from the communication channel bytestream. To alleviate this limitation, the ACE `FIFO SAP` implementation contains logic that emulates the message-oriented semantics available in SunOS 5.x.

3.1.5 Other Communication Mechanisms

In addition to encapsulating handle-based I/O communication mechanisms such as sockets and TLI, ACE also provides OO wrappers for memory-mapped files and System V UNIX IPC mechanisms:

• **Memory-Mapped Files:** The `ACE_Mem_Map` class provides an OO interface to other memory-mapped file mechanisms available on Win32 and UNIX (such as the `mmap` family of system calls). These calls utilize the underlying OS virtual memory facilities [30] to map files into the address space of a process. The contents of mapped files may be accessed directly via pointers. A pointer interface is often more convenient and efficient than accessing blocks of data indirectly via the standard `read`/`write` I/O system calls. In addition, contents of memory-mapped files may be shared conveniently between two or more processes.

Existing Win32 and UNIX interfaces for memory-mapped files are somewhat baroque. For instance, developers must perform many bookkeeping details manually (such as explicitly opening a file, determining its length, performing multiple mappings, etc.). In contrast, the `ACE_Mem_Map` OO wrapper offers an interface that employs default values and multiple constructors with several type signature variants (*e.g.,* "map from an open file handle," "map from a filename," etc.) to simplify typical memory-mapped file usage patterns.

For example, the following program uses the `ACE_Mem_Map` OO wrapper to map a file specified via the command-line and print its lines in reverse:

```
static void
putline (const char *s)
{
  while (putchar (*s++) != '\n')
    continue;
}

int
main (int argc, char *argv[])
{
  char *filename = argv[1];
  char *file_p;
  Mem_Map mmap (filename);

  if (mmap (file_p) != -1)
    {
      size_t size = mmap.size () - 1;

      if (file_p[size] == '\0')
        file_p[size] = '\n';

      while (--size >= 0)
        if (file_p[size] == '\n')
          putline (file_p + size + 1);

      putline (file_p);
      return 0;
    }
  else
    return 1;
}
```

It is instructive to compare the use of this OO wrapper interface with the much more verbose C interface necessary to use I/O systems calls like `read` directly.

• **System V IPC Mechanisms:** SunOS UNIX provides a suite of shared memory, synchronization, and message passing mechanisms known colloquially as "System V IPC" [29]. Most of the functionality offered by these mechanisms has been subsumed by more recent SunOS UNIX facilities (such as `mmap`, thread synchronization [31], and STREAM pipes primitives, respectively). However, certain types of applications (such as database engines) may benefit from characteristics of System V IPC mechanisms (such as the peer-to-peer nature of Message Queues, the efficient multi-operation atomicity semantics of Semaphores, and the widespread availability of System V IPC across a range of UNIX OS platforms). However, it is somewhat challenging to understand and use System V IPC mechanisms (particularly semaphores) correctly since their interfaces are quite general and their behavior has traditionally been documented rather sparsely until recently [8, 29].

The ACE System V IPC wrapper interfaces shield developers from a myriad of unnecessary details. For example, the ACE OO wrapper version of System V IPC semaphores is more intuitive and simpler to use for applications that utilize standard `wait` and `signal` semaphore operations, as shown in the following code fragment from a typical producer/consumer example:

```
typedef ACE_SV_Semaphore_Simple SEMA;
SEMA prod (1, SEMA::CREATE, 1);
SEMA cons (2, SEMA::CREATE, 0);

void producer (void)
{
  for (;;) {
    prod.wait ();
    // produce resource...
    cons.signal ();
  }
}

void consumer (void)
{
  for (;;) {
    cons.wait ();
    // consume resource...
    prod.signal ();
  }
}
```

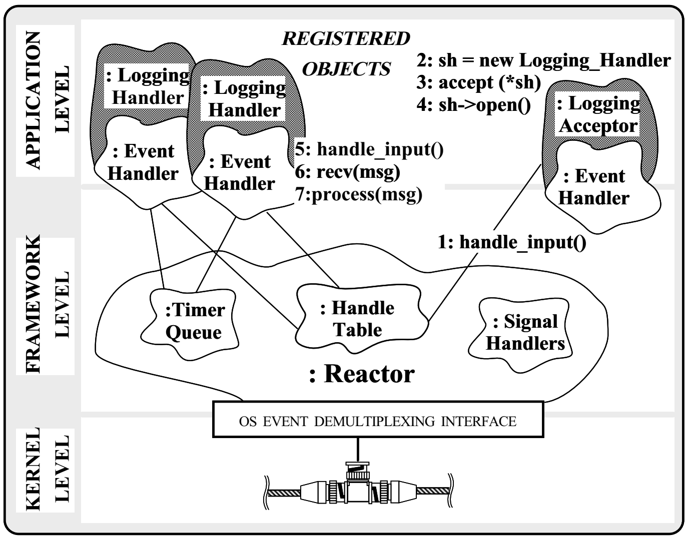

Figure 5: Software Architecture of the WWW Server

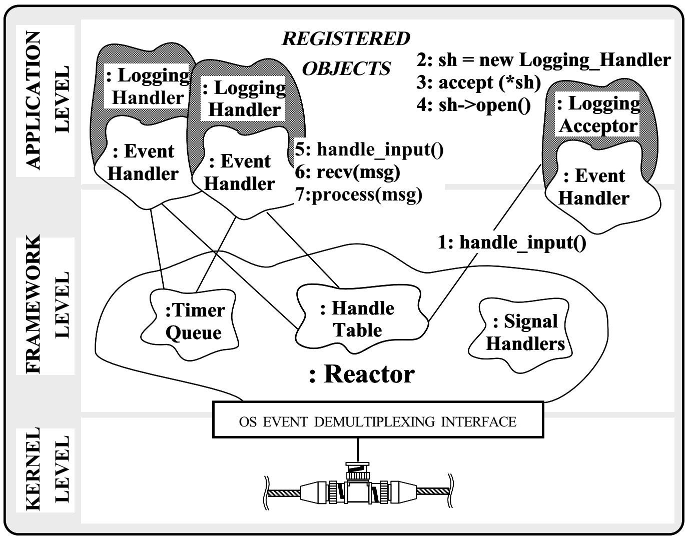

Figure 6: The `ACE_Reactor` Class Category Components

It is instructive to compare this concise OO wrapper interface with the much more verbose C interface necessary to use System V semaphores directly.

3.2 Reactor: Event Demultiplexing and Event Handler Dispatching

Communication software demultiplexes and processes many different types of events (such as timer-based, I/O-based, signal-based, and synchronization-based events). For example, a WWW server is commonly structured internally using an event loop that monitors a well-known Internet port (typically port 80). This port is associated with an application-specific handler that listens for clients to connect on port 80. When clients connect, the WWW server accepts the connection and creates an event handler to service the HTTP request. For instance, if a Netscape browser sends a `GET` request the WWW server will return the requested content to the browser.

To consolidate and automate event-driven processing activities, ACE provides an event demultiplexing and event handler dispatching framework called the `ACE_Reactor` [10]. The `Reactor` encapsulates the functionality of UNIX and Windows NT event demultiplexing mechanisms (such as `select` and `poll`) within a portable and extensible OO wrapper [10]. These OS event demultiplexing system calls detect the occurrence of different types of input and output events on one or more I/O handles simultaneously.

To facilitate application portability, the `ACE_Reactor` provides the same interface regardless of what event demultiplexing mechanism is used.[2] In addition, the `ACE_Reactor` encapsulates the mutual exclusion mechanisms necessary to perform callback-style dispatching correctly and efficiently in a multi-threaded event processing environment.

The structure of objects in the `ACE_Reactor` is illustrated in Figure 6. These objects are responsible for (1) demultiplexing of events (such as *temporal events* generated by a timer-driven callout queue, *I/O events* received on communication ports, and *signal events*) and (2) dispatching the appropriate methods of pre-registered event handler(s) to process these events. As shown in Figure 6, all the event handler objects derive from the `ACE_Event_Handler` abstract base class. This class specifies an interface that is used by the `ACE_Reactor` to dispatch certain application-specific methods in response to the arrival of certain events.

The `ACE_Reactor` uses the virtual methods declared in the `Event Handler` interface to integrate the demultiplexing of I/O handle-based, timer-based, and signal-based events. I/O handle-based events are dispatched via the `handle_input`, `handle_output`, and `handle_exceptions` methods; timer-based events are dispatched via the `handle_timeout` method; and Signal-based events are dispatched via the `handle_signal` method.

Subclasses of `ACE_Event_Handler` (such as the `Logging_Handler` described in Section 4.1) may augment the base class interface by defining additional methods and data members. In addition, virtual methods in the `ACE_Event_Handler` interface may be selectively overridden by subclasses to implement application-specific functionality. For example, application-specific subclasses in the PBX monitoring server presented in Section 4.2 define `Event Handler` objects that communicate with clients by inheriting and/or instantiating objects of the `SOCK SAP` or `TLI SAP` transport interface classes described in Section 3.1. After the virtual methods in the `ACE_Event_Handler` base class have been defined by a subclass, an application may instantiate the resulting event

[2] An extended version of the `ACE_Reactor`, called `ACE_ReactorEx`, is used on Win32 platforms to encapsulate the `WaitForMultipleObjects` event demultiplexing call. Since this functionality is not portable across OS platforms, it is not covered in this document.

handler object.

The following example implements a simple program that continuously exchanges messages back and forth between two processes using a bi-directional communication channel. This example illustrates how services inherit from the ACE_Event_Handler. It also depicts how the ACE_Reactor is used to demultiplex and dispatch I/O-based, signal-based, and timer-based events. The Ping_Pong class shown below inherits the interface from ACE_Event_Handler and implements its application-specific functionality as follows:

```
class Ping_Pong : public ACE_Event_Handler
{
public:
  Ping_Pong (char *b)
    : len (min (strlen (b) + 1, BUFSIZ)) {
    strncpy (this->buf, b, BUFSIZ);
  }
  virtual int handle_input (ACE_HANDLE handle) {
    return read (handle, this->buf, BUFSIZ);
  }
  virtual int handle_output (ACE_HANDLE handle) {
    return write (handle, this->buf, this->len);
  }
  virtual int handle_signal (int signum) {
    this->finished = 1;
  }
  virtual int handle_timeout (const Time_Value &,
                              const void *) {
    this->finished = 1;
  }
  bool done (void) {
    return this->finished == 1;
  }

private:
  sig_atomic_t finished;
  char buf[BUFSIZ];
  size_t len;
};
```

The bi-directional communication channel is created using SVR4 UNIX STREAM pipes:

```
static int
init_handles (ACE_HANDLE handles[])
{
  if (pipe (handles) == -1)
    LM_ERROR ((LOG_ERROR, "%p\n%a", "pipe", 1));

  // Enable message-oriented mode instead of
  // bytestream mode.
  int arg = RMSGN;

  if (ioctl (handles[0], I_SRDOPT, arg) == -1
      || ioctl (handles[1], I_SRDOPT, arg) == -1)
    return -1;
}
```

The main program begins by opening the appropriate communication channel. Following this, the program forks a child process, instantiates a Ping_Pong event handler object named callback in each of the two processes, registers the callback object for I/O-based, signal-based, and timer-based events with an instance of the ACE_Reactor, and then enters an event loop, as follows:

```
int main (int argc, char *argv[])
{
  ACE_HANDLE handles[2];
  ACE_Reactor reactor;

  init_handles (handles);

  pid_t pid = fork ();
```

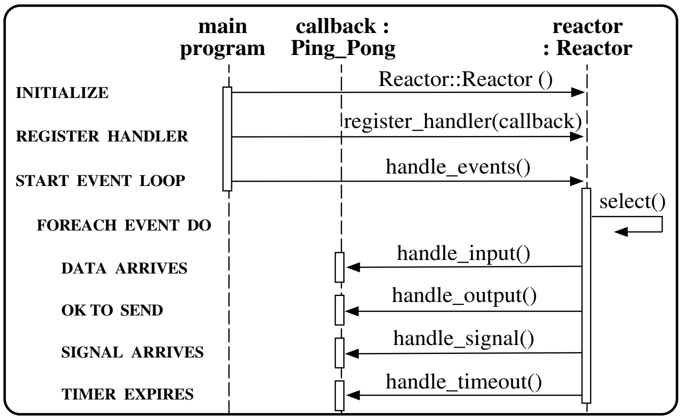

Figure 7: ACE Reactor Interaction Diagram

```
  Ping_Pong callback (argv[1]);

  // Register I/O-based event handler
  reactor.register_handler (
    handles[pid == 0],
    &callback,
    ACE_Event_Handler::READ_MASK
    | ACE_Event_Handler::WRITE_MASK);

  // Register signal-based event handler
  reactor.register_handler (SIGINT, &callback)

  // Register timer-based event handler
  reactor.schedule_timer (&callback, 0, 10)

  /* Main event loop (run in each process) */
  while (callback.done () == false)
    reactor.handle_events ();

  return 0;
}
```

The callback event handler for the timer-based and signal-based events is stored with the appropriate tables inside the ACE_Reactor. Likewise, the ACE_Reactor stores the appropriate handle in an internal table when the register_handler method is invoked to register the I/O-based event handler. When the application subsequently performs its main event loop by calling the ACE_Reactor::handle_events method, this handle is passed as an argument to the underlying OS I/O demultiplexing system call (*e.g.,* select or poll).

As input, output, signal, and timer events associated with the pre-registered event handler callback object occur at run-time, the ACE_Reactor automatically detects these events and dispatches the appropriate method(s) of the event handler object. The dispatched method of the callback object is responsible for performing application-specific functionality (such as writing a message to the communication channel, reading a message from the channel, or setting a flag that triggers termination of the program). The collaboration between these components is depicted via the object interaction diagram shown in Figure 7.

3.3 Concurrency: Multi-threading and Synchronization Mechanisms

The ACE Concurrency class category contains OO wrappers (*e.g.,* ACE_Mutex, ACE_Condition, ACE_Semaphore, and ACE_RW_Mutex) that encapsulate the corresponding Solaris [31] and POSIX Pthreads [32] multi-threading and synchronization mechanisms. These wrappers automate the initialization of synchronization objects that appear as fields in classes and also simplify typical usage patterns for the threading and synchronization mechanisms. For instance, the following code illustrates the use of the ACE wrappers for the SunOS mutex_t and cond_t synchronization mechanisms for a typical shared resource management class:

```
class Resource_Manager
{
public:
  Resource_Manager (u_int initial_resources)
    : resource_add_ (this->lock),
      resources_ (initial_resources) {}

  int acquire_resources (u_int amount_wanted)
  {
    this->lock_.acquire ();

    while (this->resources_ < amount_wanted) {
      this->waiting_++;
      // Block until resources are released.
      this->resource_add_.wait ();
    }
    this->resources_ -= amount_wanted;
    this->lock_.release ();
  }

  int release_resources (u_int amount_released)
  {
    this->lock_.acquire ();
    this->resources_ += amount_released;
    if (this->waiting_ == 1) {
      this->waiting_ = 0;
      this->resource_add_.signal ()
    }
    else if (this->waiting_ > 1) {
      this->waiting_ = 0;
      this->resource_add_.broadcast ();
    }
    this->lock_.release ();
  }
  // ...
private:
  ACE_Mutex lock_;
  ACE_Condition<ACE_Mutex> resource_add_;
  u_int resources_;
  u_int waiting_;
  // ...
};
```

Note how the constructor for the ACE_Condition object resource_add binds the ACE_Mutex object lock together with the Condition object. This simplifies the ACE_Condition::wait calling interface, in comparison with the underlying SunOS cond_t cond_wait interface.

Although the ACE_Mutex wrappers provide a relatively elegant method for synchronizing multiple threads of control, they are potentially error-prone since it is possible to forget to call the release method (either due to programmer negligence or due to the occurrence of C++ exceptions). To improve application robustness, the ACE synchronization facilities leverage off the semantics of C++ class constructors and destructors. To ensure that ACE_Mutex locks will be automatically acquired and released, ACE provides a helper class called ACE_Guard, which is defined as follows:

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

An object of the ACE_Guard class defines a block of code over which a ACE_Mutex is acquired and then released automatically when the block is exited.

Note that the ACE_Guard class is defined as a template that is parameterized by mutual exclusion mechanism. There are several different types of mutex semantics [33]. Each type of mutual exclusion shares a common interface (*i.e.,* acquire/release), but possesses different serialization and performance properties. Two types of mutual exclusion supported by ACE are *non-recursive* and *recursive* locks.

• **Non-recursive locks:** A non-recursive lock provides an efficient form of mutual exclusion that define a *critical section*, where only a single thread may execute at a time. They are non-recursive in the sense that the thread currently owning a lock may not reacquire the lock without releasing it first. Otherwise, deadlock will occur immediately. SunOS 5.x provides support for non-recursive locks via its mutex_t, rwlock_t, and sema_t types (POSIX Pthreads doesn't provide the latter two synchronization mechanisms). The ASX framework provides the Mutex, RW_Mutex, and Semaphore wrappers to encapsulate these semantics, respectively.

• **Recursive lock:** A recursive lock, on the other hand, allows acquire method invocations to nest as long as the thread that owns the lock is the one trying to re-acquire it. Recursive locks are particularly useful for callback-driven event dispatching frameworks (such as the Reactor described in Section 3.2), where the framework event-loop performs callbacks to pre-registered user-defined objects. Since the user-defined objects may subsequently re-enter the dispatching framework via its method entry points, recursive locks are necessary to prevent deadlock from occurring on locks held within the framework during callbacks.

The following C++ template class implements recursive lock semantics for the synchronization mechanisms in Solaris threads and POSIX Pthreads whose native behavior does not provide recursive locking semantics:

```
template <class MUTEX>
class ACE_Recursive_Thread_Mutex
{
public:
    // Initialize a recursive mutex.
  ACE_Recursive_Thread_Mutex (void);
    // Implicitly release a recursive mutex.
  ~ACE_Recursive_Thread_Mutex (void);
    // Acquire a recursive mutex.
  int acquire (void) const;
    // Conditionally acquire a recursive mutex.
  int tryacquire (void) const;
    // Releases a recursive mutex.
```

```
  int release (void) const;

private:
  ACE_Mutex nesting_mutex_;
  ACE_Condition<ACE_Mutex> mutex_available_;
  thread_t owner_id_;
  int nesting_level_;
};
```

Note that the interface for this class is consistent with the other locking mechanisms available in ACE [22].

The following code illustrates how the ACE_Guard and ACE_Recursive_Thread_Mutex might be used within a callback mechanism:

```
int
Callback::dispatch (const Event_Handler *eh,
                    Event *event)
{
  // Constructor acquires the lock on entry.
  ACE_Guard<ACE_Recursive_Thread_Mutex<ACE_Mutex> >
    m (this->lock_);

  eh->handle_event (event);
  // Destructor of Guard releases the lock on exit.
}
```

This code ensures that registering an Event_Handler object executes as a critical section. This example also illustrates the use of a C++ idiom [34] where the constructor of a class acquires the lock on a synchronization object automatically when the ACE_Guard object is created. Likewise, the class destruction automatically unlocks the object when the mon object goes out of scope. Moreover, the destructor for mon will be invoked automatically to release the Mutex lock regardless of which arm of the if/else statement returns from the method. In addition, the lock will also be released automatically if a C++ exception is raised during processing within the body of the register_handler method. The ACE_Recursive_Thread_Mutex is used to ensure that application-specific handle_event callbacks that are dispatched by the dispatch method do not cause deadlock if they reenter the Callback object.

ACE also provides a ACE_Thread_Manager class that contains a set of mechanisms to manage groups of threads that collaborate to implement collective actions. For instance, the ACE_Thread_Manager class provides mechanisms (such as suspend_all and resume_all) that allow any number of participating threads to be suspended and resumed atomically.

3.4 Service Configurator: Explicit Dynamic Linking Mechanisms

Static linking is a technique for composing a complete executable program by binding together all its object files at compile-time and/or static link-time. *Dynamic linking*, in contrast, enables the addition and/or deletion of object files into the address space of a process at initial program invocation or at any point later during run-time. SunOS 4.x and 5.x support both *implicit* and *explicit* dynamic linking:

- *Implicit dynamic linking* is used to implement shared object files, also known as shared libraries [35]. Shared object files reduce primary and secondary storage utilization since only one copy of shared object code exists in memory and on disk, regardless of the number of processes that are executing this code. Moreover, certain address resolution and relocation operations may be deferred until a dynamically linked function is first referenced. This "lazy evaluation" scheme minimizes link editing overhead during process start-up.
- *Explicit dynamic linking* provides interfaces that allow applications to obtain, utilize, and/or remove the run-time address bindings of symbols defined in shared object files [36]. Explicit dynamic linking mechanisms significantly enhance the functionality and flexibility of communication software since services may be inserted and/or deleted at run-time *without* terminating and/or restarting the entire application. SunOS 5.x supports explicit dynamic linking via the dlopen/dlsym/dlclose routines and Win32 supports this feature via the LoadLibrary/GetProcAddress routines.

ACE provides the Service Configurator class category to encapsulate the explicit dynamic linking mechanisms of SunOS within a set of classes and inheritance hierarchies. The Service Configurator leverages upon the other ACE components to extend the functionality of conventional daemon configuration and control frameworks [20] (such as listen [3], inetd [2], and the Windows NT Service Control Manager [37]) that provide automated support for (1) static and dynamic configuration of concurrent, multi-service communication software, (2) monitoring sets of communication ports for I/O activity, and (3) dispatching incoming messages received on monitored ports to the appropriate application-specified services. The remainder of this section discusses the primary components in the Service Configurator class category.

3.4.1 The ACE_Service_Object Inheritance Hierarchy

The primary unit of configuration in the Service Configurator is the *service*. A service may be simple (such as returning the current time-of-day) or highly complex (such as a distributed, real-time router for PBX event traffic [6, 38]). To provide a consistent environment for defining, configuring, and using communication software, all application services are derived from the ACE_Service_Object inheritance hierarchy (illustrated in Figure 8 (1)).

The ACE_Service_Object class is the focal point of a multi-level hierarchy of types related by inheritance. The standard interfaces provided by the abstract classes in this type hierarchy may be selectively implemented by application-specific subclasses in order to access certain application-independent Service Configurator mechanisms. These mechanisms provide transparent dynamic linking, event handler registration, event demultiplexing, and service dispatching. By decoupling the application-specific portions of a handler object from the underly-

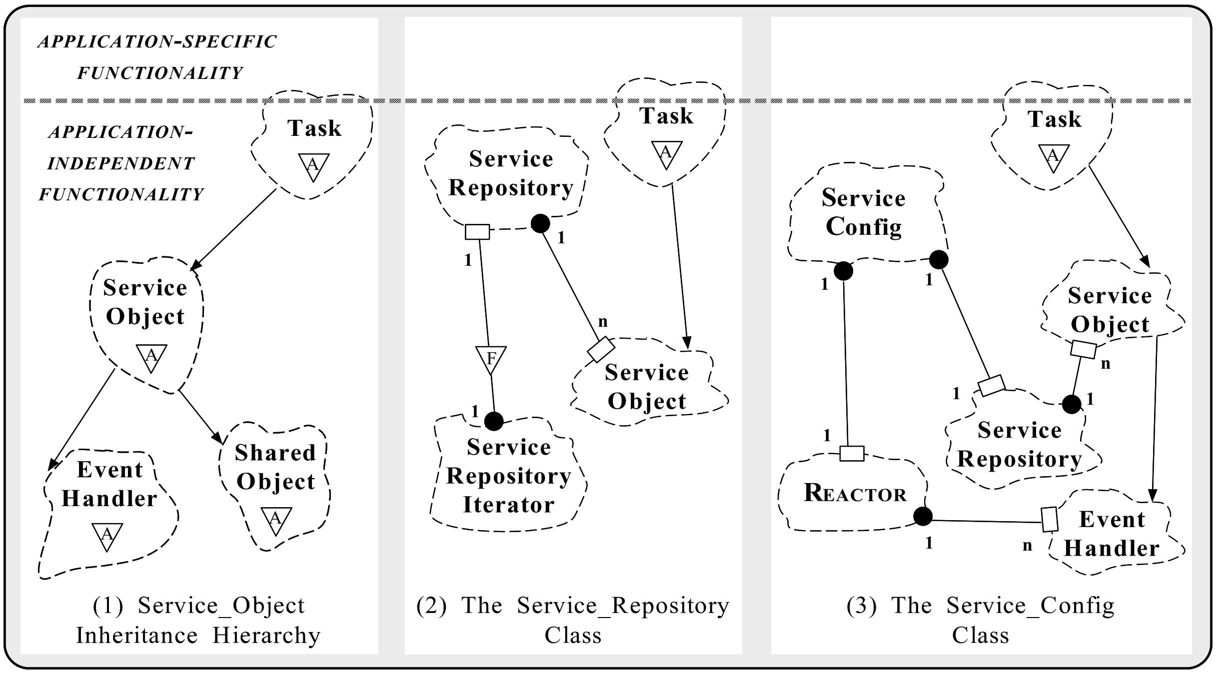

Figure 8: Component Relationships for the `Service Configurator` Class Category

ing application-independent `Service Configurator` mechanisms, the effort necessary to insert and remove services from a running application is significantly reduced.

The `ACE_Service_Object` inheritance hierarchy consists of the `ACE_Event_Handler` and `ACE_Shared_Object` abstract base classes, as well as the `ACE_Service_Object` abstract derived class. The `ACE_Event_Handler` class was described above in Section 3.2. The behavior of the other classes is outlined below.

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

The `ACE_Shared_Object` base class is defined independently from the `ACE_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 communication port event demultiplexing. Conversely, other applications (such as an `ftp` server) may require event demultiplexing, but might not require dynamic linking. By separating these interfaces into two base classes, applications are able to select a subset of `Service Configurator` mechanisms without incurring unnecessary storage costs.

• **The ACE_Service_Object Abstract Derived Class:** In general, installing and administering complex distributed systems requires support for dynamic linking, event demultiplexing, and service dispatching in order to automate the dynamic configuration and reconfiguration of application services. Therefore, the `Service Configurator` defines the `ACE_Service_Object` class, which combines the interfaces from both the `ACE_Event_Handler` and the `ACE_Shared_Object` abstract base classes. The resulting abstract derived class supplies an interface that developers use as the basis for implementing and configuring a service into the `Service Configurator`.

During development, the application-specific subclasses of `ACE_Service_Object` must implement the abstract `suspend` and `resume` methods specified by the `ACE_Service_Object` class interface. These methods are invoked automatically by the `Service Configurator` in response to external events. An application developer may control the actions the object undertakes in order to suspend a service without removing and unlinking it completely, as well as to resume a previously suspended service.

In addition, application-specific subclasses must also implement the four abstract methods (`init`, `fini`, `info`, and `get_handle`) that are inherited (but not defined) by the `ACE_Service_Object` subclass. The `init` method serves as the entry-point to a service handler during run-time initialization. This method is responsible for performing application-specific initialization when an instance of a `ACE_Service_Object` is dynamically linked. Likewise, the `fini` method is called automatically by the `Service Configurator` when a `ACE_Service_Object` is unlinked and removed from an application at run-time. This method typically performs termination operations that release dynamically allocated resources (such as memory, I/O handles, or synchronization locks). The `info` method formats a humanly-readable string that concisely reports service addressing information and documents service functionality. Clients may query an application to retrieve this information and use it to contact a particular service running in the application. Finally, the `get_handle` method is used by the `Reactor` to extract the underlying I/O handle from a service handler object. This I/O handle identifies a transport endpoint that may be used to accept connections or receive data from clients.

- **Application-Specific Concrete Derived Subclasses:** `Service Object` is an abstract class since its interface contains the abstract methods inherited from the `Event Handler` and `Shared Object` abstract base classes. Therefore, developers must supply concrete subclasses that (1) define the six abstract methods described above and (2) implement the necessary application-specific functionality. To accomplish the latter task subclasses typically define certain virtual methods exported by the `Service Object` interface. For example, the `handle_input` method is often implemented to accept connections or data that are received from clients.

The `ACE_Acceptor` class depicted in Figure 8 (1) is an example of an application-independent subclass that accepts connection requests as part of a distributed logging facility. This class is described further in the example presented in Section 4.1.

3.4.2 The ACE_Service_Repository Class

The `Service Configurator` class category supports the configuration of both single-service and multi-service communication software. Therefore, to simplify run-time administration, it is often necessary to individually and/or collectively control and coordinate the `ACE_Service_Objects` that comprise an application's currently active1 services. The `ACE_Service_Repository` is an object manager that coordinates local and remote queries involving the services offered by a `Service Configurator`-based application. A search structure within the object manager binds service names (represented as ASCII strings) with instances of `ACE_Service_Objects` (represented as object code). A service name uniquely identifies an instance of a `ACE_Service_Object` stored in the repository.

Each entry in the `ACE_Service_Repository` contains a pointer to the `ACE_Service_Object` portion of an application-specific derived class (shown in Figure 8 (2)). This enables the `Service Configurator` to load, enable, suspend, resume, or unload `ACE_Service_Objects` from an application statically or dynamically. For dynamically linked `ACE_Service_Objects`, the repository also maintains a handle to the underlying shared object file. This handle is used to unlink and unload a `ACE_Service_Object` from a running application when the service it offers is no longer required.

```
<svc-config-entries> ::=
        svc-config-entries svc-config-entry
        | NULL
<svc-config-entry> ::= <dynamic> | <static>
         | <suspend> | <resume> | <remove>
         | <stream> | <remote>
<dynamic> ::= DYNAMIC <svc-location>
                    [ <parameters-opt> ]
<static> ::= STATIC <svc-name>
                  [ <parameters-opt> ]
<suspend> ::= SUSPEND <svc-name>
<resume> ::= RESUME <svc-name>
<remove> ::= REMOVE <svc-name>
<stream> ::= STREAM <stream_ops>
             '{' <module-list> '}'
<stream_ops> ::= <dynamic> | <static>
<remote> ::= STRING '{' <svc-config-entry> '}'
<module-list> ::= <module-list> <module>
                | NULL
<module> ::= <dynamic> | <static>
           | <suspend> | <resume> | <remove>
<svc-location> ::= <svc-name> <type>
                   <svc-initializer> <status>
<type> ::= SERVICE_OBJECT '*' | MODULE '*'
         STREAM '*' | NULL
<svc-initializer> ::= <object-name>
                    | <function-name>
<object-name> ::= PATHNAME ':' IDENT
<function-name> ::= PATHNAME ':' IDENT '(' ')'
<status> ::= ACTIVE | INACTIVE | NULL
<parameters-opt> ::= STRING | NULL
```

Figure 9: EBNF Format for a Service Config Entry

Directive	Description
dynamic	Dynamically link and enable a service
static	Enable a statically linked service
remove	Completely remove a service
suspend	Suspend service without removing it
resume	Resume a previously suspended service

Table 1: Service Config Directives

An iterator class is also supplied in conjunction with the `ACE_Service_Repository`. This class is used to visit every `ACE_Service_Object` in the repository without unduly compromising data encapsulation.

3.4.3 The ACE_Service_Config Class

The `ACE_Service_Config` class is the unifying component in the `Service Configurator` framework. As illustrated in Figure 8 (3), this class integrates the other `Service Configurator` framework components (such as the `ACE_Service_Repository` and the `ACE_Reactor`) to automate the static and/or dynamic configuration of concurrent, multi-service communication software.

The `ACE_Service_Config` class uses a configuration file (known as `svc.conf`) to guide its configuration and reconfiguration activities. Each application may be associated

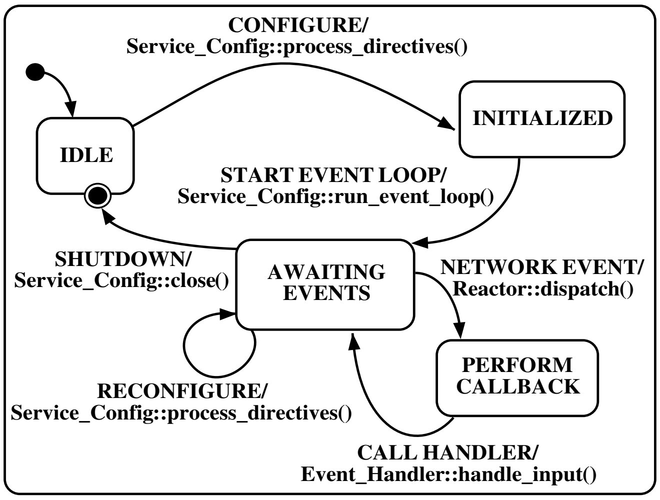

Figure 10: State Transition Diagram for Service Configuration, Execution, and Reconfiguration

with a distinct `svc.conf` configuration file. Likewise, a set of applications may be described by a single `svc.conf` file. Figure 9 describes the primary syntactical elements in a `svc.conf` file using extended-Backus/Naur Format (EBNF). Each *service config entry* in the file begins with a *service config directive* that specifies the configuration activity to perform. Table 1 summarizes the valid service config directives.

Each service config entry contains attributes that indicate the location of the shared object file for each dynamically linked service, as well as the parameters required to initialize the service 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 significantly simplified. The `svc.conf` file helps to decouple the structure of an application from the behavior of its services. This decoupling also permits the "lazy" configuration and reconfiguration of mechanisms provided by the framework, based on the application-specific attributes and parameters specified in the `svc.conf` file.

Figure 10 depicts a state transition diagram illustrating the methods in the `Service Configurator` class category 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 `ACE_Service_Config` class is called to consult the `svc.conf` file. This file is first consulted when a new instance of a application is initially configured. The file is consulted again whenever application reconfiguration is triggered upon receipt of a pre-designated external event (such as the UNIX SIGHUP signal or a notification arriving from a socket).

3.5 Stream: Layered Serivce Integration

The Stream class category is the primary focal point of the ADAPTIVE Communication Environment. This class category contains the ADAPTIVE Service eXecutive (`ASX`) framework [6] (`ASX`) framework, which integrates the lower-level OO wrapper components (like `IPC SAP`) and higher-level class categories (like the `Reactor` and the `Service Configurator`). The `ASX` framework helps to simplify the development of hierarchically-integrated communication software, particularly user-level communication protocol stacks and network servers. The `ASX` framework is designed to improve the modularity, extensibility, reusability, and portability of both the application-specific services and the underlying OS concurrency, IPC, explicit dynamic linking, and demultiplexing mechanisms upon which these services are built.

The `ASX` framework provides the following two benefits to developers of communication software:

1. *It embodies, encapsulates, and implements key design patterns* that are commonly used to develop communication software. Design patterns help to enhance software quality by addressing fundamental challenges in large-scale system development. These challenges include communication of architectural knowledge among developers; accommodating new design paradigms or architectural styles; resolving non-functional forces such as reusability, portability, and extensibility; and avoiding development traps and pitfalls that are usually learned only by experience.
2. *It strictly separates key development concerns* – ACE separates communication software development into two distinct categories: (1) *application-independent concerns*, which are common to most or all communication software (such as port monitoring; message buffering, queueing, and demultiplexing; service dispatching; local/remote interprocess communication; concurrency control; and application configuration, installation, and run-time service management) and (2) *application-specific concerns*, which depend on an individual application. By reusing the OO wrappers and frameworks provided by ACE, developers are freed from spending their time reinventing solutions to commonly recurring tasks. In turn, this enables them to concentrate on the key higher-level functional requirements and design concerns that constitute particular applications.

3.5.1 Primary `ASX` Features

The `ASX` framework increases the flexibility of communication software by decoupling application-specific processing policies from the following configuration-related development activities and mechanisms:

• **The type and number of services associated with each application process:** The `ASX` framework permits applications to consolidate one or more services into a single administrative unit. This multi-service approach to configuring communication software helps to (1) simplify development and reuse code by performing common service initialization activities automatically, (2) reduce the consumption of OS resources (such as process table slots) by spawning service handlers "on-demand," (3) allow application services to be updated without modifying existing source code or terminating an executing dispatcher process (such as the `inetd` superserver), and (4) consolidate the administration of network services via a uniform set of configuration management operations.

• **The point of time at which a service is configured into an application:** The `ASX` framework leverages off the `Service Configurator` framework (described in Section 3.4.1) to provide an extensible object-oriented interface that automates the use of OS mechanisms for explicit dynamic linking. Dynamic linking enhances the extensibility of communication software by permitting internal services to be configured when an application first begins executing or while it is running. This feature enables an application's services to be dynamic reconfigured *without* requiring the modification, recompilation, relinking, or restarting of active services. In the `ASX` framework, the choice between static or dynamic configuration may selected on a per-service basis. Furthermore, this choice may be deferred until an application begins execution.

• **The type of execution agents:** In the `ASX` framework, services may be performed at run-time via several different types of process and thread execution agents. By decoupling service functionality from the execution agent used to invoke the service, the `ASX` framework increases the range of application concurrency configuration alternatives available to developers.

An efficient application concurrency configuration often depends upon certain service requirements and platform characteristics. For example, a process-based configuration may be appropriate for implementing long-duration services (such as the Internet `ftp` and `telnet`) that base their security mechanisms on process ownership. In this case, each service (or each active instance of a service) may be mapped onto a separate process and executed in parallel on a multi-processor platform. Different configurations may be more suitable in other circumstances, however. For instance, it is often simpler and more efficient to implement cooperating services (such as those found in end-systems of distributed database engines) in separate threads since they frequently reference common data structures. In this approach, each service may be executed on a separate thread within the same process to reduce the overhead of scheduling, context switching, and synchronization [6].

• **The order in which hierarchically-related services are combined into an application:** Complex services may be composed using an interconnected series of independent service objects that communicate by passing messages. These objects may be joined together in essentially arbitrary configurations to satisfy application requirements and enhance component reuse.

• **The I/O handle-based and timer-based event demultiplexing mechanisms:** These mechanisms are used to dispatch incoming connection requests and data onto a pre-registered application-specific handler. The `ASX` framework uses the `Reactor` class category to integrate the demultiplexing of I/O handle-based, timer-based, and signal-based events via an extensible and type-safe object-oriented interface.

• **The underlying IPC mechanisms:** Application services may use the `IPC SAP` mechanisms described in Section 3.1 to exchange data with participating communication entities on local or remote end-systems. Unlike the weakly-typed, "handle-based" socket and TLI interfaces, the `IPC SAP` wrappers enable applications to access the underlying OS IPC mechanisms via a type-safe, portable interface.

The `ASX` framework incorporates concepts from several modular communication frameworks including System V STREAMS [39], the *x*-kernel [40], and the Conduit framework [41] from the Choices object-oriented operating system (a survey of these and other communication frameworks appears in [42]). These frameworks all contain features that support the flexible configuration of communication subsystems by inter-connecting "building-block" protocol and service components. In general, these frameworks encourage the development of standard communication-related components (such as message managers, timer-based event dispatchers, demultiplexors [40], and assorted protocol functions [43]) by decoupling processing functionality from the surrounding framework infrastructure. As described below, the `ASX` framework contains additional features that further decouple processing functionality from the underlying process architecture.[3]

Unlike STREAMS, application services configured into the `ASX` framework execute in user-space rather than in kernel-space. There are several advantages to developing general-purpose communication software in user-space, rather than within the OS kernel:

• **Access to general OS features:** Applications developed to run in user-space have access to the full range of OS mechanisms (such as dynamic linking, memory-mapped files, multi-threading, large virtual address spaces, interprocess communication mechanisms, file systems, and databases). In contrast, kernel-resident components are often restricted to a limited set of kernel-specific mechanisms. Although there are idioms that overcome some of these limitations (*e.g.,* maintaining a user-level daemon that performs file I/O on behalf of a kernel-resident component), these workarounds tend to be somewhat inelegant and non-portable.

[3] A process architecture represents a binding between one or more CPUs together with the application tasks and messages that implement services in a communication system [6].

• **Enhanced development environment:** Higher-level programming tools (such as symbolic debuggers) may be used to develop applications in user-space. Conversely, developing network services within a OS kernel is a complex and challenging task, due to the primitive debugging tools and subtle timing interactions in a kernel programming environment [44]. It is risky to expect application developers to program effectively in such a constrained environment.

• **Increased system robustness:** Exceptional conditions (such as dereferencing NULL pointers, dividing by 0, etc.) generated in a user-level process or thread should only affect the offending process or thread. However, exceptional conditions within a OS kernel may cause the entire operating system to panic and crash. Moreover, rebooting the OS after every crash quickly becomes tedious.

• **Portability:** Porting kernel-level drivers between different OS platforms (and even different variants of the same OS platform) typically entails many more complications than porting user-level application components between platforms. The SunOS 5.x DDI/DKI API is intended to alleviate some of the portability problems on UNIX platforms, but that does not solve the portability problem for applications running on other platforms such as OS/2, Windows NT, VMS, and Novell Netware.

The primary rationale for implementing distributed services in a OS kernel is to improve performance. For example, a kernel-resident implementation of a communication protocol stack often helps reduce scheduling, context switching, and protection-domain boundary crossing overhead and may exhibit more predictable response time due to the use of "wired" (rather than paged) memory. However, for many communication software applications, the increased flexibility, simplicity, robustness, and portability offered by user-space development are key requirements that offset potential performance degradations.

3.5.2 `Stream` Class Category Components

Components in the `Stream` class category are responsible for coordinating the configuration and run-time execution of one or more *ACE_Stream* objects. An `ACE_Stream` is an object that user applications collaborate with to configure and execute application-specific services in the `ASX` framework. As illustrated in Figure 3.5, an `ACE_Stream` contains a series of inter-connected `ACE_Module` objects that may be linked together by (1) developers at installation-time or (2) applications at run-time. `ACE_Modules` are objects used to decompose the architecture of an application into a series of inter-connected, functionally distinct layers. Each layer typically implements a cluster of related service functionality (such as an end-to-end transport service or a presentation layer formatting service). Every `ACE_Module` contains a pair of `ACE_Task` objects that partition a layer into its constituent read-side and write-side processing functionality.

OO language features (such as classes, inheritance, dynamic binding, and parameterized types) enable developers

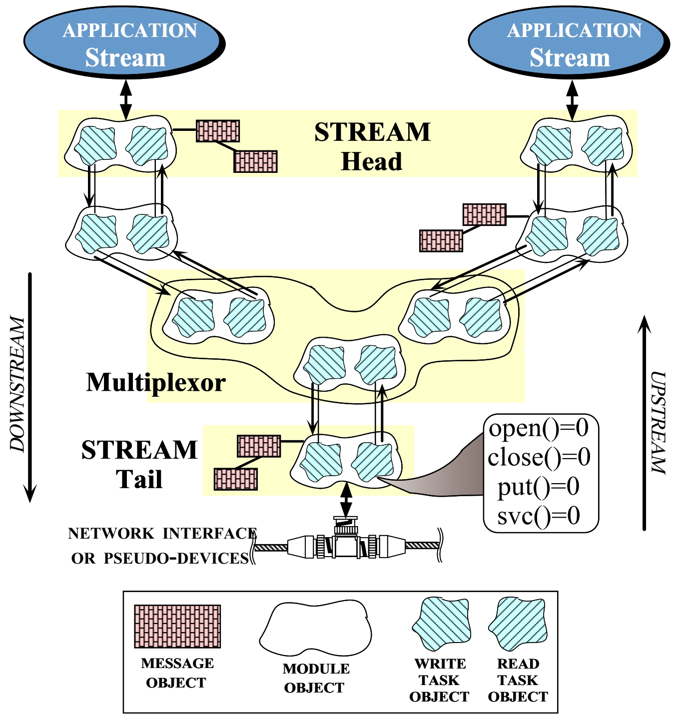

Figure 11: Components in the `ASX` Framework

to incorporate application-specific functionality into a Stream without modifying the application-independent `ASX` framework components. For example, adding a service layer into a Stream involves (1) inheriting from the default `ACE_Task` interface and selectively overriding several virtual methods in the subclass to provide application-specific functionality, (2) allocating a new `ACE_Module` that contains two instances (one for the read-side and one for the write-side) of the application-specific `ACE_Task` subclass, and (3) inserting the `ACE_Module` into a Stream. Services in adjacent inter-connected `ACE_Tasks` collaborate by exchanging typed messages via a message passing interface.

To avoid reinventing familiar terminology, many class names in the Stream class category correspond to similar componentry available in the System V STREAMS framework [39]. However, the techniques used to support extensibility and concurrency in the two frameworks are significantly different. For example, adding application-specific functionality to the `ASX` Stream classes is performed by inheriting several well-defined interfaces and implementations from existing framework components. Using inheritance to add new functionality provides greater type-safety than the pointer-to-function techniques used in System V STREAMS. The `ASX` Stream classes also employ a different concurrency control scheme to reduce the likelyhood of deadlock and to simplify flow control between `Tasks` in a Stream. The `ASX` Stream classes completely redesign and reimplement the co-routine-based, "weightless"[4] service processing mechanisms

[4] A weightless process executes on a run-time stack that is also used by other processes, which complicates programming and increases the potential for deadlock. For example, a weightless process may not suspend execution

used in System V STREAMS [45]. These ASX changes are intended to utilize multiple PEs on a shared memory multiprocessor platform more effectively.

The remainder of this section discusses the primary components of the Stream class category (*e.g.,* ACE_Stream class, the ACE_Module class, the ACE_Task class in detail:

• **The ACE_Stream Class:** The ACE_Stream class defines the application interface to a Stream. An ACE_Stream object contains a stack of one or more hierarchically-related services that provide applications with a bi-directional get/put-style interface for sending and receiving data and control messages through the inter-connected Modules that comprise a particular Stream. The ACE_Stream class also implements a push/pop-style interface that allows applications to configure a Stream at run-time by inserting and removing objects of the ACE_Module class described below.

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

As shown in Figure 3.5, two default ACE_Module objects (ACE_Stream_Head and ACE_Stream_Tail) are installed automatically when a Stream is opened. These two ACE_Modules interpret pre-defined ASX framework control messages and data messages that circulate through a Stream at run-time. The ACE_Stream_Head class provides a message buffering interface between an application and a Stream. The ACE_Stream_Tail class typically transforms incoming messages from a network or from a pseudo-device into a canonical internal message format that may be processed by higher-level components in a Stream. Likewise, for outgoing messages it transforms messages from their internal format into network messages.

• **The ACE_Task Abstract Class:** The ACE_Task class is the central mechanism in ACE for creating user-defined *active objects* [11] and *passive objects* that process application messages. An ACE Task can perform the following activities:

- Can be dynamically linked;
- Can serve as a demultiplexing endpoint for I/O operations;
- Can be associated with multiple threads of control (i.e., become a so-called "active object");
- Can store messages in a queue for subsequent processing;
- Can execute user-defined services.

to wait for resources to become available or events to occur.

The ACE_Task abstract class defines an interface that is inherited and implemented by derived classes in order to provide application-specific functionality. It is an abstract class since its interface defines the abstract methods (open, close, put, and svc) described below. Defining ACE_Task as an abstract class enhances reuse by decoupling the application-independent components provided by the ACE_Stream class category from the application-specific subclasses that inherit from and use these components. Likewise, the use of abstract methods allows the compiler to ensure that a subclass of Task honors its obligation to provide the following functionality:

- *Initialization and Termination Methods* – Subclasses derived from ACE_Task must implement open and close methods that perform application-specific ACE_Task initialization and termination activities. These activities typically allocate and free resources such as connection control blocks, I/O handles, and synchronization locks.

  ACE_Tasks can be defined and used either together with ACE_Modules or separately. When used with ACE_Modules they are stored in pairs: one ACE_Task subclass handles read-side processing for messages sent upstream to its ACE_Module layer and the other handles write-side processing messages send downstream to its Module layer. The open and close methods of a Module's write-side and read-side ACE_Task subclasses are invoked automatically by the ASX framework when the ACE_Module is inserted or removed from a Stream, respectively.

- *Application-Specific Processing Methods* – In addition to open and close, subclasses of ACE_Task must also define the put and svc methods. These methods perform application-specific processing functionality on messages. For example, when messages arrive at the head or the tail of a Stream, they are escorted through a series of inter-connected ACE_Tasks as a result of invoking the put and/or svc method of each ACE_Task in the Stream.

  A put method is invoked when a ACE_Task at one layer in a Stream passes a message to an adjacent ACE_Task in another layer. The put method runs *synchronously* with respect to its caller, *i.e.,* it borrows the thread of control from the Task that originally invoked its put method. This thread of control typically originate either "upstream" from an application process, "downstream" from a pool of threads that handle I/O device interrupts [40], or internal to the Stream from an event dispatching mechanism (such as a timer-driven callout queue used to trigger retransmissions in a connection-oriented transport protocol ACE_Module).

  If an ACE_Task executes as a *passive object* (*i.e.,* it always borrows the thread of control from the caller), then the ACE_Task::put method is the entry point into the ACE_Task and serves as the con-

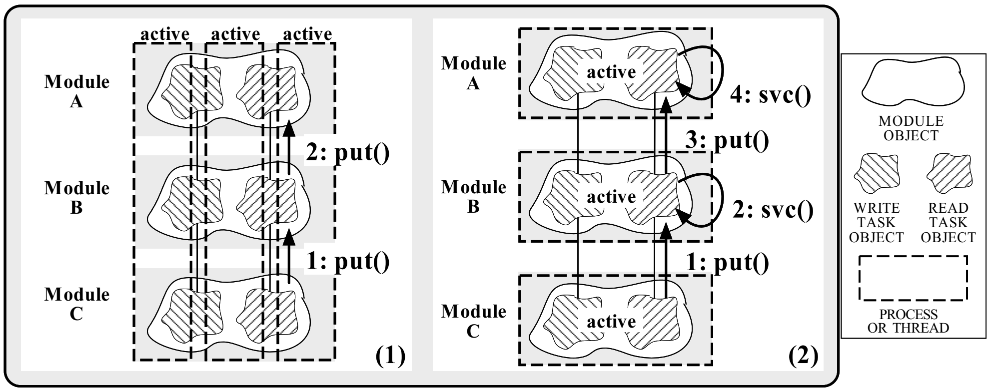

Figure 12: Alternative Methods for Invoking `put` and `svc` Methods

text in which `ACE_Task` executes its behavior. In contrast, if an ACE `ACE_Task` executes as an *active object* the `ACE_Task::svc` method is used to perform application-specific processing *asynchronously* with respect to other `ACE_Tasks`. Unlike `put`, the `svc` method is not directly invoked from an adjacent `ACE_Task`. Instead, it is invoked by a separate thread associated with its `ACE_Task`. This thread provides an execution context and thread of control for the `ACE_Task`'s `svc` method. This method runs an event loop that continuously waits for messages to arrive on the `ACE_Task`'s `ACE_Message_Queue` (see next bullet).

Within the implementation of a `put` or `svc` method, a message may be forwarded to an adjacent `ACE_Task` in the Stream via the `put_next` `ACE_Task` utility method. `Put_next` calls the `put` method of the next `ACE_Task` residing in an adjacent layer. This invocation of `put` may borrow the thread of control from the caller and handle the message immediately (*i.e.,* the synchronous processing approach illustrated in Figure 12 (1)). Conversely, the `put` method may enqueue the message and defer handling to its `svc` method that is executing in a separate thread of control (*i.e.,* the asynchronous processing approach illustrated in Figure 12 (2)). As discussed in [6], the particular processing approach that is selected has a significant impact on performance and ease of programming.

- *Message Queueing Mechanisms* – In addition to the `open`, `close`, `put`, and `svc` abstract method interfaces, each `ACE_Task` also contains an `ACE_Message_Queue`. An `ACE_Message_Queue` is a standard component in ACE that is used pass information between `ACE_Tasks`. Moreover, when a `ACE_Task` executes as an active object, its `ACE_Message_Queue` is used to buffer a sequence of data messages and control messages for subsequent processing in the `svc` method. As messages arrive, the `svc` method dequeues the messages and performs the `ACE_Task` subclass's application-specific processing tasks.

  Two types of messages may appear on a `ACE_Message_Queue`: simple and composite. A simple message contains a single `ACE_Message_Block` and a composite message contains multiple `ACE_Message_Blocks` linked together. Composite messages generally consist of a *control* block followed by one or more *data* blocks. A control block contains bookkeeping information (such as destination addresses and length fields), whereas data blocks contain the actual contents of a message. The overhead of passing `ACE_Message_Blocks` between `Tasks` is minimized by passing pointers to messages rather than copying data.

  `ACE_Message_Queues` contain a pair of high and low water mark variables that are used to implement layer-to-layer flow control between adjacent `ACE_Modules` in a Stream. The high water mark indicates the amount of bytes of messages the `ACE_Message_Queue` is willing to buffer before it becomes flow controlled. The low water mark indicates the level at which a previously flow controlled `ACE_Task` is no longer considered to be full.

4 ACE Examples

The ACE components are currently being used in several research [46] and commercial environments [6, 38, 47] to enhance the configuration flexibility and software component reuse of communication software that operate efficiently and portably across multiple hardware and software platforms. To illustrate how the `ASX` framework is used in practice, this section examines the architecture of two commercial applications currently being developed using ACE components: a distributed logging facility and a distributed monitoring system for telecommunication switch devices.

4.1 Distributed Logging Example

Debugging distributed communication software is frequently challenging since diagnostic output appears in different win-

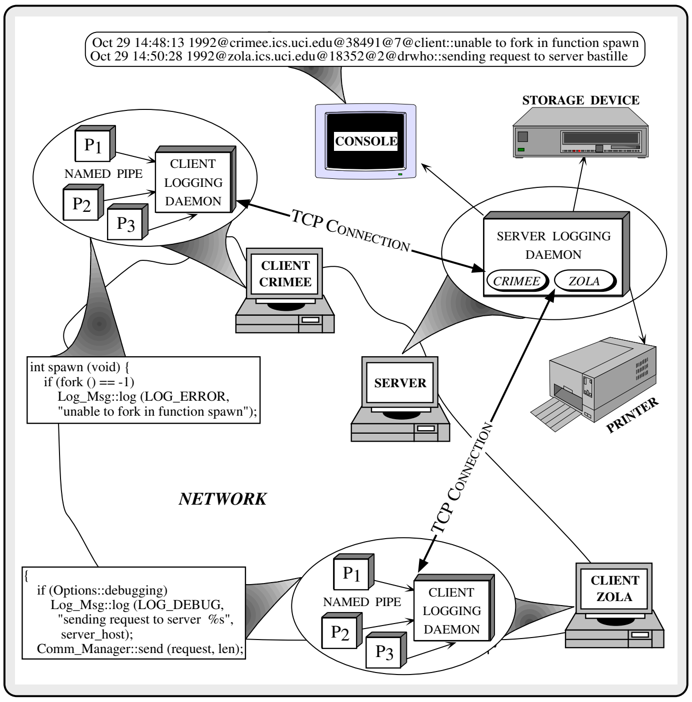

Figure 13: The Distributed Logging Facility

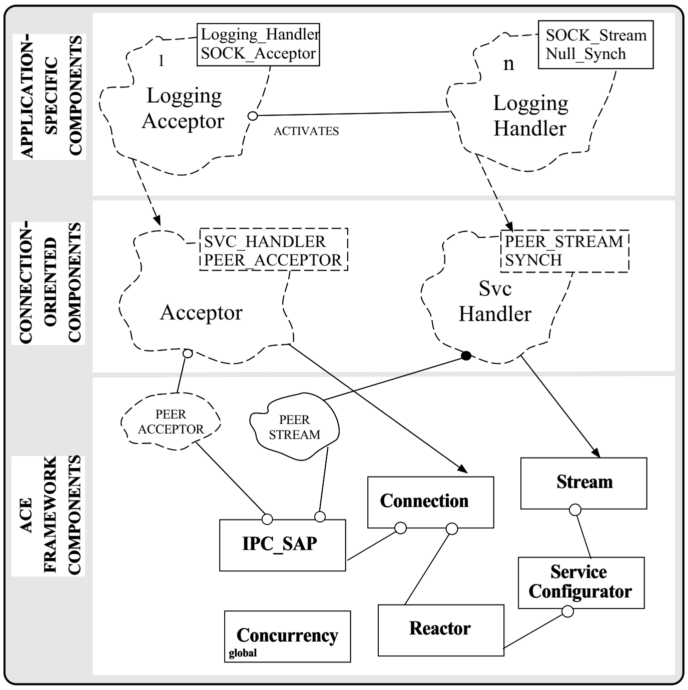

Figure 14: Class Components in the Server Logging Daemon

dows and/or on different remote host systems. Therefore, ACE provides a distributed logging facility that simplifies debugging and run-time tracing. This facility is currently used in a commercial on-line transaction processing system [14] to provide logging services for a cluster of workstations and multi-processor database servers in a high-speed network environment.

As shown in Figure 13, the distributed logging facility allows applications running on multiple client hosts to send logging records to a server logging daemon running on a designated server host. This section focuses on the architecture and configuration of the server daemon portion of the logging facility, which is based on the `Service Configurator` and `ACE_Reactor` class categories provided by the `ASX` framework. The complete design and implementation of the distributed logging facility is described in [10].

4.1.1 Server Logging Daemon Components

The server logging daemon is a concurrent, multi-service daemon that processes logging records received from one or more client hosts simultaneously. The object-oriented design of the server logging daemon is decomposed into several modular components (shown in Figure 14) that perform well-defined tasks. The application-specific components (`Logging_Acceptor` and `Logging Handler`) are responsible for processing logging records received from clients. The connection-oriented application components (`Acceptor` and `Client_Handler`) are responsible for accepting connection requests and data from clients. Finally, the application-independent ASX framework components (the `ACE_Reactor`, `Service Configurator`, and `IPC SAP` class categories) are responsible for performing IPC, explicit dynamic linking, event demultiplexing, service dispatching, and concurrency control.

The `Logging_Handler` subclass is a parameterized type that is responsible for processing logging records sent to the server logging daemon from participating client hosts. Its communication mechanisms may be instantiated with either the `SOCK SAP` or `TLI SAP` wrappers, as follows:

```
class Logging_Handler : public Client_Handler <
#if defined (MT_SAFE_SOCKETS)
                    ACE_SOCK_Stream,
#else
                    ACE_TLI_Stream,
#endif /* MT_SAFE_SOCKETS */
                    ACE_INET_Addr>
{
  /* ... */
};
```

The `Logging_Handler` class inherits from `Event Handler` (indirectly via `Client_Handler`) rather than `Service Object` since it is not dynamic linked into the server logging daemon.

When logging records arrive from the client host associated with a particular `Logging_Handler` object, the `ACE_Reactor` automatically dispatches the object's `handle_input` method. This method formats and displays the records on one or more output devices (such as the printer, persistent storage, and/or console devices illustrated in Figure 13).

The `Logging_Acceptor` subclass is also a parameterized type that is responsible for accepting connections from client hosts participating in the logging service:

```
class Logging_Acceptor :
  public Client_Acceptor<Logging_Handler,
#if defined (MT_SAFE_SOCKETS)
                         ACE_SOCK_Acceptor,
#else
                         ACE_TLI_Acceptor,
#endif /* MT_SAFE_SOCKETS */
                         ACE_INET_Addr>
{
  /* ... */
};
```

Since the Logging_Acceptor class inherits from ACE_Service_Object (indirectly via its ACE_Acceptor base class), it may be dynamically linked into the server logging daemon and manipulated at run-time via the server logging daemon's svc.conf configuration file. Likewise, since Logging_Acceptor indirectly inherits from the ACE_Event_Handler interface, its handle_input method will be invoked automatically by the ACE_Reactor when connection requests arrive from clients. When a connection request arrives, the Logging_Acceptor subclass allocates a Logging_Handler object and registers this object with the ACE_Reactor.

The modularity, reusability, and configurability of the distributed logging facility is significantly enhanced by decoupling the functionality of connection establishment and logging record reception into the two distinct class hierarchies shown in Figure 14. This decoupling allows the ACE_Acceptor class to be reused for other types of connection-oriented services. In particular, to provide completely different processing functionality, only the behavior of the ACE_Client_Handler portion of the service would need to be reimplemented. Furthermore, the use of parameterized types decouples the reliance on a particular type IPC mechanism.

4.1.2 Server Logging Daemon Configuration

The ASX framework uses the Service Configurator to enable the dynamic and static configuration of logging services into the server logging daemon. Dynamically configured services may be inserted, modified, or removed at run-time, thereby improving service flexibility and extensibility. The following svc.conf file entry is used to to dynamically configure the logging service into the server logging daemon:

```
dynamic Logger Service_Object *
         ./Logger.so:_alloc() "-p 7001"
```

The <svc-name> token Logger specifies the service name that is used at installation and run-time to identify the corresponding Service Object within the ACE_Service_Repository. Service Object * is the return type of the _alloc method that is located in the shared object file indicated by the pathname ./Logger.so. The Service Configurator framework locates and dynamically links this shared object file into the logging daemon's address space. The service location also specifies the name of the application-specific object derived from

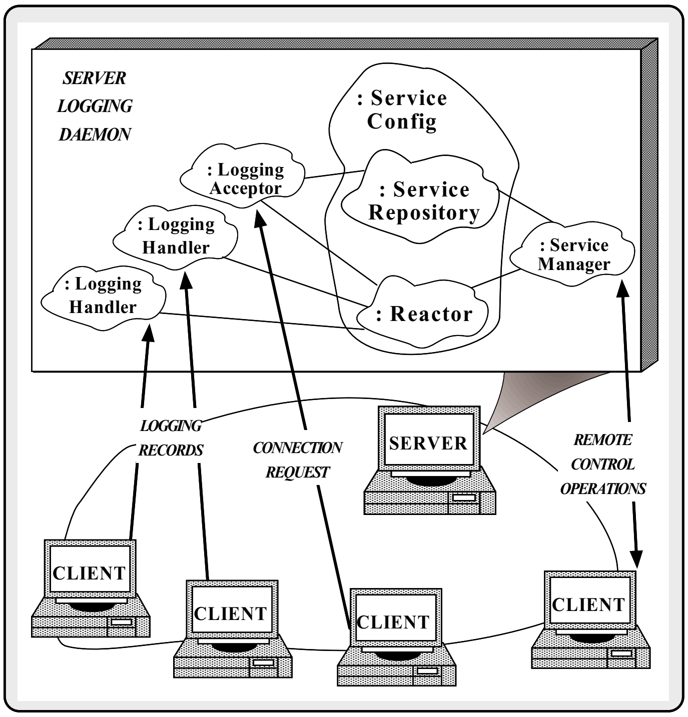

Figure 15: ACE Components in the Distributed Logging Facility

Service Object. In this case, the _alloc function is used to dynamically allocate a new Logging_Acceptor object. The remaining contents on the line ("-p 7001") represent a application-specific set of configuration parameters. These parameters are passed to the init method of the service as argc/argv-style command-line arguments. The init method for the Logging_Acceptor class interprets "-p 7001" as the port number where the server logging daemon listens for client connection requests.

Statically configured services are always available to a daemon when it first begins execution. For example, the Service Manager is a standard Service Configurator framework component that clients use to obtain a listing of active daemon services. The following entry in the svc.conf file is used to statically configure the Service Manager service into the server logging daemon during initialization:

```
static ACE_Svc_Manager "-p 911"
```

In order for the static directive to work, the object code that implements the ACE_Svc_Manager service must be statically linked together with the main daemon driver executable program. In addition, the ACE_Svc_Manager object must be inserted into the Service Repository before dynamic configuration occurs (this is done automatically by the ACE_Service_Config constructor). Due to these constraints, a statically configured service may not be reconfigured at run-time without being removed from the Service Repository first.

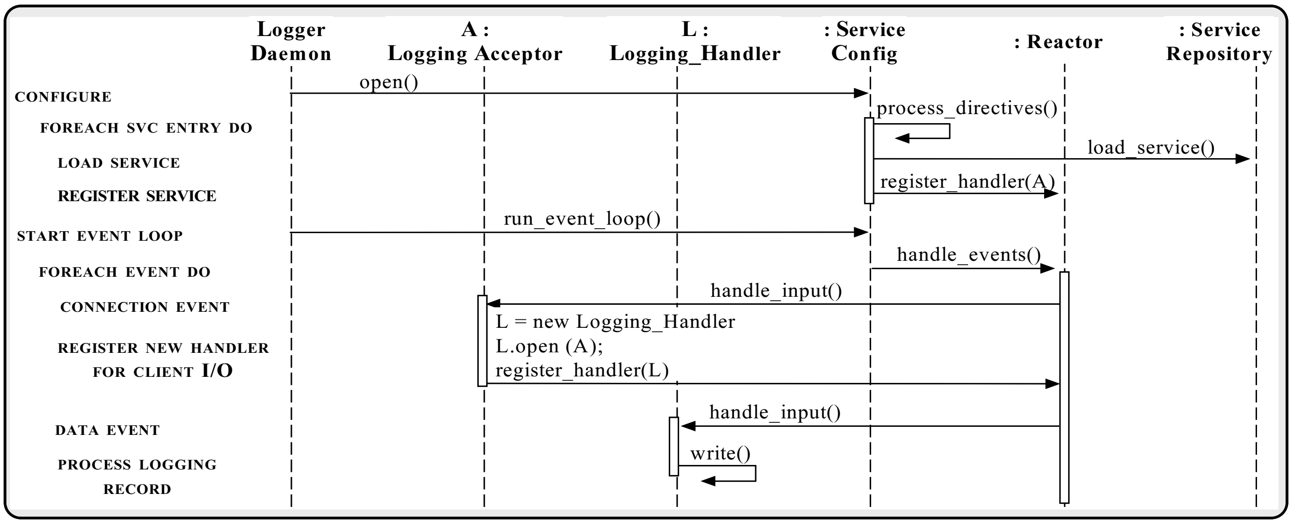

Figure 16: Interaction Diagram for the Server Logging Daemon

The main driver program for the server logger daemon is implemented by the following code:

```
int
main (int argc, char *argv[])
{
  ACE_Service_Config loggerd;

  // Configure server logging daemon.
  if (loggerd.open (argc, argv) == -1)
    return -1;

  // Perform logging service.
  loggerd.run_reactor_event_loop ();
  return 0;
}
```

Figure 16 depicts the run-time interaction between the various framework and application-specific objects that collaborate to provide the logging service. Daemon configuration is performed in the `ACE_Service_Config::open` method. This method consults the following `svc.conf` file, which specifies the services to configure into the daemon:

```
static ACE_Svc_Manager -p 911
dynamic Logger Service_Object *
          ./Logger.so:_alloc() "-p 7001"
```

Each of the service config entries in the `svc.conf` file is processed by inserting the designated `ACE_Service_Object` into the `ACE_Service_Repository` and registering the `ACE_Event_Handler` portion of the service object handler with the `ACE_Reactor`.

When all the configuration activities have been completed, the main driver program shown above invokes the `run_reactor_event_loop` method of the `ACE_Service_Config`. This method enters an event loop that continuously calls the `ACE_Reactor::handle_events` service dispatch method. As shown in Figure 10, this dispatch function blocks awaiting the occurrence of events (such as connection requests or I/O from clients). As these events occur, the `ACE_Reactor` automatically dispatches previously-registered event handlers to perform the designated application-specific services.

The `ASX` framework also responds to external events that trigger daemon reconfiguration at run-time. The dynamic configuration steps outlined above are performed whenever an executing `ASX`-based daemon receives a pre-designated external event (such as the UNIX SIGHUP signal). Depending on the updated contents of the `svc.conf` file, services may be added, suspended, resumed or removed from the daemon.

The `ASX` framework's dynamic reconfiguration mechanisms enable developers to modify server logging daemon functionality or fine-tune performance without extensive redevelopment and reinstallation effort. For example, debugging a faulty implementation of the logging service simply requires the dynamic reinstallation of a functionally equivalent service that contains additional instrumentation to help isolate the source of erroneous behavior. Note that this reinstallation process may be performed without modifying, recompiling, relinking, or restarting the currently executing server logging daemon.

4.2 Distributed PBX Monitoring System

Figure 17 illustrates the client/server architecture of a private branch exchange (PBX) telecommunication switch monitoring system implemented using `ASX` framework components [20]. In this distributed communication system, the server receives and processes status information generated by one or more PBXs attached to the server via a high-speed communication link. The server transforms and forwards this status information across a network to client end-systems that graphically display the information to end-users. End-users are typically supervisors who use the PBX status information to monitor the performance of personnel and forecast the allocation of resources to meet customer demands.

The PBX devices attached to the server are controlled by

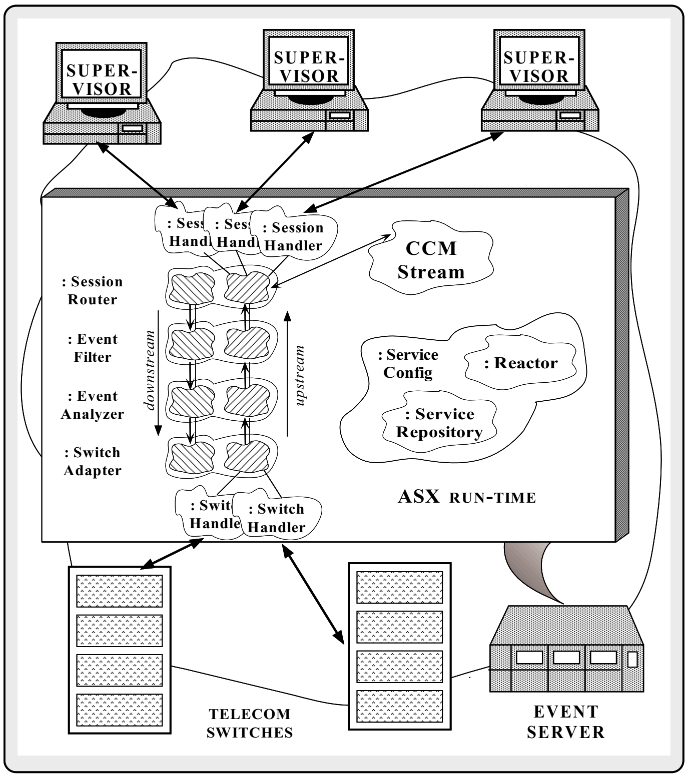

Figure 17: ASX Components for the PBX Application

the Device_Adapter ACE_Module. This ACE_Module shields the rest of the server from PBX-specific communication characteristics. The read-side of the Device_Adapter ACE_Module maintains a collection of Device_Handler objects (one per-PBX) that are responsible for parsing and transforming incoming device events into a canonical PBX-independent message object built atop a flexible message management class described in [6].

After being initialized, incoming canonical message objects are passed to the read-side of the Event_Analyzer ACE_Module. This Module implements the application-specific functionality for the server. An internal addressing table maintained within the Event Analyzer is used to determine which client(s) should receive each message object. After the Event_Analyzer determines the proper destination(s), the message object is forwarded to the read-side of the Multicast_Router ACE_Module.

The Multicast_Router ACE_Module is a reusable component that shields the rest of the application-specific server code from knowledge of the client/server interactions and from the particular choice of communication protocols. Clients subscribe to receive events published by the server by establishing a connection with the Multicast_Router ACE_Module. The write-side of the Multicast_Router ACE_Module accepts connection requests from clients and creates a separate Client_Handler object to manage each client connection. This Client_Handler object handles all subsequent data transfer and control operations between the server and its associated client. Once a client has connected with the server, it indicates the type of PBX event(s) it is interested in monitoring. From that point, when the read-side of the Multicast_Router receives a message object from the Event_Analyzer, it automatically multicasts the message to all clients that have subscribed to receive the particular type of event encapsulated in the message object.

The ACE_Service_Config object is used by the server to control the initialization and termination of Stream ACE_Module components that are configured statically at installation-time or dynamically during run-time. The ACE_Service_Config object contains an instance of the ACE_Reactor event demultiplexor that is used to dispatch incoming client messages to the appropriate Client_Handler or Device_Handler event handler. Control messages arriving from clients are sent down the write-side of the Stream, starting with the Multicast_Router and continuing through the interconnected write-side of the Stream ACE_Modules to the Device_Adapter, which sends them to the appropriate PBX device. Likewise, the Reactor detects incoming events from PBX devices and dispatches them up the Stream starting at the Device_Adapter ACE_Module.

4.3 Server Configuration

The ACE_Modules that comprise the PBX server may be configured into the server at any time. The ASX framework provides this degree of flexibility via the use of explicit dynamic linking driven by the svc.conf configuration script. The following configuration script indicates which services are to be dynamically linked into the address space of the server:

```
stream Server_Stream dynamic
  STREAM * /svcs/Server_Stream.so : _alloc()
{
  dynamic Device_Adapter
    Module * /svcs/DA.so:_alloc() "-p 2001"
  dynamic Event_Analyzer
    Module * /svcs/EA.so:_alloc()
  dynamic Multicast_Router
    Module * /svcs/MR.so:_alloc() "-p 2010"
}
```

This configuration script indicates the order in which the hierarchically-related services are dynamically linked and pushed onto the Server Stream. During application initialization, the Service Config class parses this configuration script and carries out the directives that describe each entry.

The Server Stream is composed of three ACE_Modules (the Device_Adapter, the Event_Analyzer, and the Multicast_Router) that are dynamically configured into the server. The indicated shared object file is linked dynamically into the server (as specified by the dynamic directive). An instance of Module object is then extracted from the shared object library by calling the _alloc function. As described below, these Modules may be subsequently updated and relinked if necessary (*e.g.,* to install an updated version of a ACE_Module) without completely terminating the executing PBX server.

4.4 Server Reconfiguration

There are a number of drawbacks associated with statically configuring services into a communication software application. For example, performance bottlenecks may result if too many services are configured into the server-side an application and too many active clients simultaneously access these services. Conversely, configuring too many services into the client-side may also result in bottlenecks since clients often execute on less powerful end-systems. In general, it is difficult to determine the appropriate partitioning of application services *a priori* since processing characteristics and workloads may vary over time. Therefore, a primary objective of the ASX framework was to develop object-oriented service configuration mechanisms that allow developers to defer until very late in the development cycle (i.e., installation-time or run-time) decisions regarding which services ran in the client-side and which ran in the server-side.

To facilitate flexible reconfiguration, the run-time control environment provided by the ASX framework enables developers to alter their application service configurations either *statically* at installation-time or *dynamically* during run-time. This is useful since different OS/hardware platforms and different network characteristics often require different service configurations. For example, in some configurations the server performs most of the work, whereas in others the clients do more of the work. Moreover, different end-system configurations may be appropriate under different circumstances (such as whether multi-processor server platforms or high-speed networks are available). Figure 18 illustrates how the configuration shown in Figure 17 may be altered to operate efficiently in a distributed environment where the server processing constitutes the primary bottleneck.

This reconfiguration process is accomplished via the following script:

```
suspend Server_Stream
stream Server_Stream
{
  remove Event_Analyzer
}
remote "-h all -p 911"
{
  stream Server_Stream
  {
    dynamic Event_Analyzer
      Module * /svcs/EA.so : _alloc()
  }
}
resume Server_Stream
```

This new script migrates processing functionality from the server to the clients by dynamically unlinking ACE_Modules from the server's Stream and dynamically linking them into each client's Stream [20].

The ASX framework replaced a previous architecture that used *ad hoc* techniques (such as parameter passing and shared memory) to exchange messages between related services in the server. In contrast to the previous approach, the highly

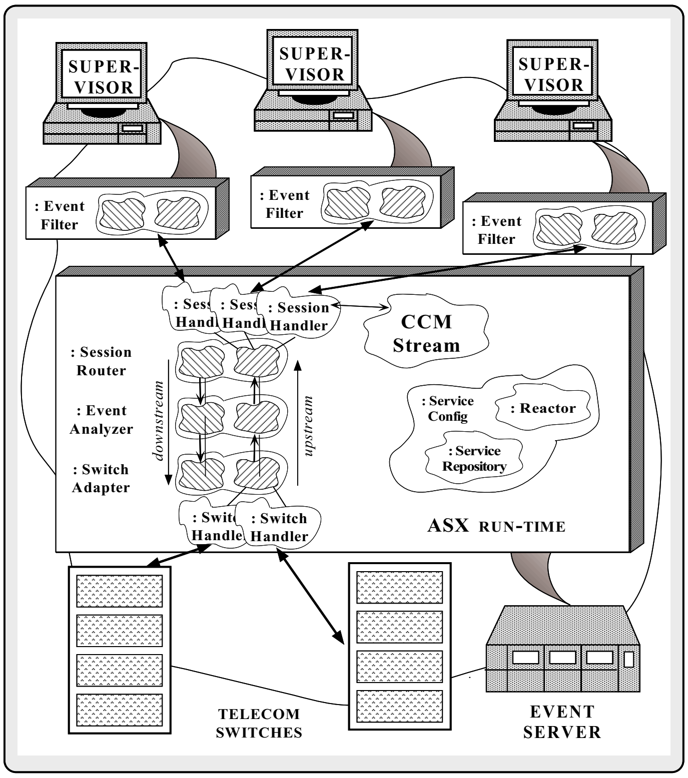

Figure 18: Reconfiguration of PBX Monitoring System

uniform ACE_Module inter-connection mechanisms provided by the ASX framework greatly simplify portability and configurability.

5 Concluding Remarks

The ADAPTIVE Communication Environment (ACE) is a toolkit containing OO components that help reduce distributed software complexity by reifying successful design patterns and software architectures. ACE consolidates common communication-related activities (such as local and remote IPC [4], event demultiplexing and service handler dispatching [14], service initialization [16, 17] configuration mechanisms for distributed applications containing monolithic and layered services [20], distributed logging [13], and intra- and inter-service concurrency [46]) into reusable OO components and frameworks.

ACE is freely available via the World Wide Web at URL www.cs.wustl.edu/~schmidt/ACE.html. This distribution contains the source code, documentation, and example test drivers developed at Washington University, St. Louis. ACE is currently being used in communication software at many companies including Bellcore, Siemens, DEC, Motorola, Ericsson, Kodak, and McDonnell Douglas. ACE has been ported to Win32 (i.e., WinNT and Win95), most versions of UNIX (e.g., SunOS 4.x and 5.x, SGI IRIX, HP-UX, OSF/1, AIX, Linux, and SCO), and POSIX systems (such as

VxWorks and MVS OpenEdition). There are both C++ [6] and Java [48] versions of ACE available.

References

[1] F. P. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.

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

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

[4] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the* 1^{st} *Conference on Object-Oriented Technologies and Systems*, (Monterey, CA), USENIX, June 1995.

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

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

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

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

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[10] 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.

[11] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

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

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

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

[15] T. H. Harrison, D. C. Schmidt, and I. Pyarali, "Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling," in *The* 3^{rd} *Annual Conference on the Pattern Languages of Programs (Washington University technical report #WUCS-97-07)*, (Monticello, Illinois), pp. 1–7, February 1997.

[16] D. C. Schmidt, "Design Patterns for Initializing Network Services: Introducing the Acceptor and Connector Patterns," *C++ Report*, vol. 7, November/December 1995.

[17] D. C. Schmidt, "Connector: a Design Pattern for Actively Initializing Network Services," *C++ Report*, vol. 8, January 1996.

[18] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *The* 1^{st} *European Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07)*, July 1997.

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

[20] 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.

[21] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration and Reconfiguration of Communication Services," in *The* 3^{rd} *Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07)*, February 1997.

[22] D. C. Schmidt, "An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit," Tech. Rep. WUCS-95-31, Washington University, St. Louis, September 1995.

[23] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[24] D. C. Schmidt and T. Harrison, "Double-Checked Locking – An Object Behavioral Pattern for Initializing and Accessing Thread-safe Objects Efficiently," in *The* 3^{rd} *Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07)*, February 1997.

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

[26] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[27] R. Davis, *Win32 Network Programming*. Reading, MA: Addison-Wesley, 1996.

[28] D. L. Presotto and D. M. Ritchie, "Interprocess Communication in the Ninth Edition UNIX System," *UNIX Research System Papers, Tenth Edition*, vol. 2, no. 8, pp. 523–530, 1990.

[29] W. R. Stevens, *Advanced Programming in the UNIX Environment*. Reading, Massachusetts: Addison Wesley, 1992.

[30] R. Gingell, J. Moran, and W. Shannon, "Virtual Memory Architecture in SunOS," in *Proceedings of the Summer 1987 USENIX Technical Conference*, (Phoenix, Arizona), 1987.

[31] 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.

[32] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

[33] D. C. Schmidt, "Transparently Parameterizing Synchronization Mechanisms into a Concurrent Distributed Application," *C++ Report*, vol. 6, July/August 1994.

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

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

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

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

[38] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in *Proceedings of the* 9^{th} *European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, August 1995.

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

[40] 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.

[41] 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.

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

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

[44] A. McRae, "Hardware Profiling of Kernels," in *USENIX Winter Conference*, (San Diego, CA), USENIX Association, Jan. 1993.

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

[46] 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.

[47] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, vol. 2, no. 1, 1996.

[48] P. Jain and D. Schmidt, "Experiences Converting a C++ Communication Software Framework to Java," *C++ Report*, vol. 9, January 1997.