A Domain Analysis of Network Daemon Design Dimensions

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

An earlier version of this paper appeared in the C++ Report, March/April, 1994.

1 Introduction

Applications that effectively utilize multi-processing and network services are able to deliver increased system throughput, reliability, scalability, and cost effectiveness. Designing and implementing such applications is a challenging task, however. This article is part of a continuing series that describes object-oriented techniques that may be used to simplify the development of reliable, robust, and extensible distributed applications. Previous articles in this series have examined C++ wrappers that encapsulate the socket interprocess communication (IPC) interface [1] and the event demultiplexing mechanisms provided by the `select` and `poll` system calls [2, 3]. This article presents an object-oriented domain analysis of key design dimensions for network server daemons. Subsequent articles will describe a framework called the ADAPTIVE Service eXecutive (`ASX`) [4] that provides flexibility across the design dimensions identified in this article. The `ASX` framework combines C++ features (such as parameterized types, inheritance, and dynamic binding) and advanced OS mechanisms (such as multi-threading and dynamic linking) to provide a highly extensible environment for developing and configuring a wide variety of network daemons.

A daemon is an operating system (OS) process that executes services on a host machine in the "background" (*i.e.,* disassociated from any controlling terminal) [5]. A service is a portion of a daemon that offers a single processing capability to communicating entities. Server daemons commonly available in the UNIX environment provide clients with communication-related services that resolve distributed name binding queries (`named` and `rpcbind`), access network file systems (`nfsd`), manage routing tables (`gated` and `routed`), perform local system services such as logging (`syslogd`) and printing (`lpd`), and integrate multiple remote services such as terminal access (`rlogin` and `telnet`) and file transfer (`ftp`) together into a single "superserver" framework (`inetd`). In this series of articles, particular emphasis is given to network server daemons that support multiple services via one or more processes or threads.

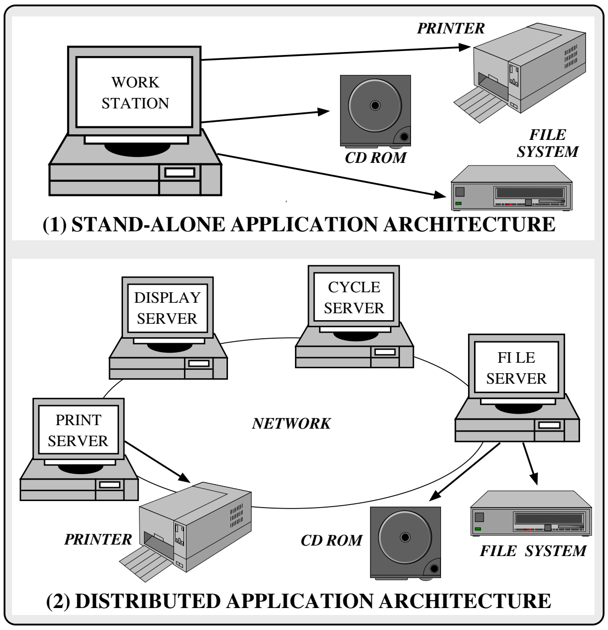

Figure 1: Stand-alone vs. Distributed Application Architectures

2 Background Material

In contrast to stand-alone applications, the components in a distributed application reside in a number of loosely coupled host machines connected together by a network. Figure 1 depicts the basic differences between the two approaches. The stand-alone application architecture illustrated Figure 1 (1) consolidates the interactive graphical user interface (GUI), instruction processing, and persistent data resources within a single machine and the peripherals directly attached to it. The flow of control in a stand-alone application is localized to the individual machine where it began executing.

The distributed application architecture portrayed by Figure 1 (2) partitions the interactive GUI, instruction processing, and persistent data resources among a number of otherwise independent machines in a network. The flow of control in a distributed application migrates between different machines during run-time. This allows certain application services (*e.g.,* database access, high-resolution graphics) to be delegated to run on hosts that possess specialized processing attributes such as high-speed disk controllers, large amounts of memory, or enhanced floating point performance. This delegation of responsibilities is beneficial since it permits the sharing of expensive peripherals (such as high-capacity file servers and high-volume printers) by a number of users and applications.

A network of diskless X-terminals is a relatively common example of a distributed architecture. In this environment a number of decentralized components collaborate to provide GUI-based computing services to end-users. The interactive GUI is typically managed by the `X server` on each X-terminal; the instruction processing capabilities are provided by the client host(s) where all or part of an application's services run; and access to persistent resources (such as databases and object managers) is mediated by one or more file servers. Interoperability is possible as long as the protocols used to inter-communicate between these separate components are compatible. Therefore, the networks, operating systems, hardware platforms, and programming languages in an X-terminal environment may be completely heterogeneous.

Although distributing services among machines in a network offers many potential advantages, distributed applications are often significantly more difficult to design, implement, debug, optimize, and monitor than their stand-alone counterparts. To handle the requirements of distributed applications, developers must address many topics (such as service partitioning and load balancing across process and host boundaries) that are either irrelevant or are less problematic for stand-alone applications. Object-oriented design and implementation techniques offer a variety of principles, methods, and tools that may help to alleviate much of the complexity related to developing and configuring distributed applications, in particular *accidental complexity* [6].

Accidental complexity is an artifact of limitations with tools and techniques used to develop software systems within an application domain. One common example of accidental complexity is the lack of type-safe, portable, and extensible system call interfaces and reusable component libraries. Debuggers are another source of accidental complexity. Many debuggers do not function properly when used on multi-threaded programs containing IPC calls. Yet another source of accidentally complexity arises from the widespread use of algorithmic decomposition. Many distributed applications are developed using algorithmic decomposition techniques that frequently result in non-extensible system architectures.

Inherent complexity, on the other hand, is a consequence of fundamental properties of a domain that complicate application development. For example, inherently complex aspects of developing reliable and efficient distributed application services involve detecting and recovering from transient network and host failures and minimizing the impact of communication latency on application performance. Likewise, determining *how* to partition an application into separate services and *where* to distribute these services throughout a network are also inherently complex development issues.

In recent years, a number of approaches have been developed to alleviate the dual problems of accidental and inherent complexity mentioned above. Two generally successful techniques that have been widely employed to reduce the complexity of network daemon implementation, configuration, and use are (1) the development of reusable component libraries and (2) the development of automated daemon configuration tools.

RPC-based toolkits and *daemon control frameworks* are notable examples of reusable component libraries and automated daemon configuration tools, respectively. RPC-based toolkits include the OSF Distributed Computing Environment (DCE), the Open Network Computing (ONC+) transport-independent RPC facility, and the Common Object Request Broker Architecture (CORBA). Daemon control frameworks include the `inetd` superserver that originated with BSD UNIX, the `listen` port monitoring facility available with System V Release 4 (SVR4) UNIX, and the `Service Control Manager` from the Windows NT operating system.

RPC-based toolkits and daemon control frameworks automate many tedious and error-prone activities associated with developing distributed applications. These activities include defining service interfaces, performing presentation layer conversions, registering services with endpoint port mappers, locating and selecting remote services, authenticating and authorizing clients, ensuring data security, demultiplexing packets, and dispatching client requests to service providers.[1]

Although they have proven to be quite useful in practice, many RPC-based toolkits and daemon control frameworks were developed without adequate consideration of object-oriented techniques and advanced OS mechanisms. This fact complicates component reuse and limits functionality. For example, the standard version of `inetd` is written in C and its implementation is characterized by a proliferation of global variables, a lack of information hiding, and an algorithmic decomposition that deters fine-grained reuse of its internal components. Likewise, `inetd`, `listen`, and the `Service Control Manager` do not provide fully automated support for (1) dynamically linking services into the daemon controller's process address space at run-time and (2) executing these services concurrently via one or more threads. Therefore, developers who want to benefit from these advanced OS mechanisms must manually program them into their applications.

3 Network Daemon Design Dimensions

The increased availability of advanced OS mechanisms (such as multi-threading and explicit dynamic linking), cou-

[1] A thorough discussion of these topics is beyond the scope of this paper; see [5] for additional details.

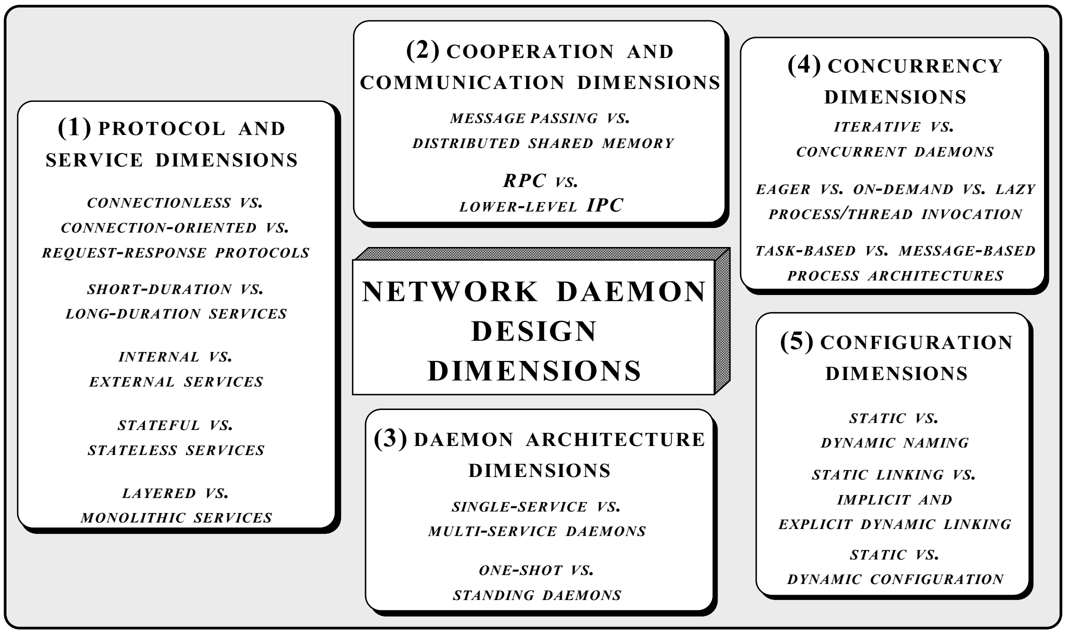

Figure 2: Network Daemon Design Dimensions

pled with the growing adoption of C++ and object-oriented methods, provides an opportunity to re-examine and revise conventional techniques for developing network daemons. To reap significant benefits from object-oriented methods and advanced OS mechanisms, it is important to gain a general understanding of the basic architectural choices that are repeatedly faced by network daemon developers. This paper provides a *domain analysis* of key dimensions that characterize the design of network daemons. This analysis has been used to guide the object-oriented design and implementation of components in the ASX framework.

A domain analysis is an incremental, feedback-driven process that examines an application domain to identify its key abstractions. Common application domains include window systems, databases, network server daemons, distributed applications, and operating system kernels. The key abstractions that are identified via analysis constitute the "vocabulary" of the application domain [7]. For example, tasks, threads, memory objects, and communication ports are several key abstractions in the domain of micro-kernel operating systems [8].

A thorough domain analysis yields several significant benefits. First, identifying and concisely defining the vocabulary of key abstractions within a domain enables developers to communicate more effectively with one another. Second, domain analysis separates development concerns into two general categories: (1) those that are common to the domain and (2) those that are specific to a particular application. By focusing on the common concerns, developers may recognize opportunities for adapting or building reusable software components. These components may be integrated to form a stable application framework that helps to reduce subsequent development effort. Ideally, the application-specific concerns also may be handled by extending well-defined portions of a framework using language features like inheritance, dynamic binding, and parameterized types.

Within the domain of network daemons, developers must carefully consider the (1) protocol and service dimensions, (2) cooperation and communication dimensions, (3) daemon architecture dimensions, (4) concurrency dimensions, and (5) configuration dimensions listed in Figure 2. These five dimensions were identified by generalizing from practical design and implementation experience with a number of distributed applications ranging from on-line transaction processing systems [3], PBX switch performance monitoring systems [9], and multi-processor-based communication subsystems [4].

Each of the five dimensions discussed below offer a set of relatively orthogonal design alternatives. These alternatives are examined to define the vocabulary of the network daemon domain, illustrate key abstractions, and clarify the boundaries of the domain. Although this discussion focuses primarily upon server-related dimensions, most topics presented below are directly applicable to the design of client applications, as well.

3.1 Protocol and Service Dimensions

A protocol is a set of rules that dictate how data and control information is exchanged between communicating entities. These entities may include one or more clients, servers, or peers interacting within a networked operating environment. A service is generally defined as either: (1) a single capability offered by a server daemon (such as the `echo` service provided by the `inetd` superserver), (2) a collection of capabilities offered by a server daemon (such as the `inetd` superserver itself), or (3) a collection of server daemons that cooperate to achieve a common task (such as a collection of rwho daemons in a LAN subnet that periodically broadcast and receive status information reporting user activities to other hosts). Unless otherwise indicated, this article uses the first definition of service, *i.e.,* a portion of a daemon that offers a single capability to communicating entities.

Services are implemented by protocols that shield applications from many low-level details of the underlying communication subsystem. For instance, the `telnet` application provides a remote login service implemented via the `telnet` protocol. The `telnet` protocol [10] precisely defines the format of messages exchanged to indicate terminal characteristics, negotiate options, and control the flow of data for users that access resources on remote hosts connected via a TCP/IP network. As shown in Figure 3 (1), a single protocol (such as TCP) may be used to implement a number of network services (such telnet, DNS, ftp, and the `X server` from the standard MIT X windows system). Likewise, a service (such as "reliable, in-sequence, non-duplicated data delivery") may be implemented by multiple protocols (such as TCP, TP4, SPX, VMTP, and XTP), as shown in Figure 3 (2).

When creating distributed applications, developers implicitly or explicitly select from among the following protocol and service dimensions:

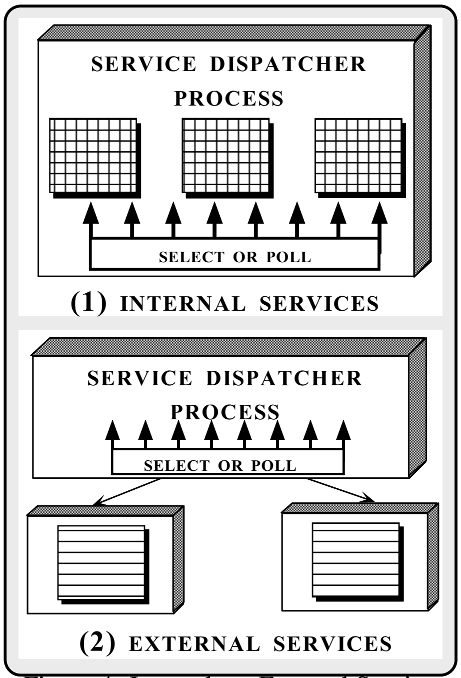

Figure 4: Internal vs. External Services

• **Connectionless vs. Connection-Oriented vs. Request-Response Protocols:** *Connectionless protocols* (such as CLNP, IP, and UDP) provide an unreliable, message-oriented service where each message may be routed independently. Since "best-effort" delivery semantics are used, there is no guarantee that a particular message will arrive at its destination. Connectionless protocols are used by applications (such as rwho daemons) that tolerate some degree of loss. They also form the foundation for higher-layer reliable protocols.

Connection-oriented protocols (such as TCP, TP4, SPX, and XTP) provide a reliable, sequenced, non-duplicated data delivery service for applications. These protocols are typically used for long-duration applications that are not loss tolerant. To enhance performance and ensure reliability, connection-oriented protocols exchange and maintain state information at the sender and/or receiver. In addition, connection-oriented protocols offer different types of data framing semantics. For example, TCP is a bytestream-oriented protocol that does not preserve application message boundaries. Therefore, if an application makes four `send` calls to transmit four distinct messages only a single TCP segment may be transmitted to a receiver. If the application requires message-oriented delivery, the receiver must perform extra processing to extract and re-frame the four messages from the single segment it received. This is a relatively simple operation if messages always possess equal lengths and network errors do not occur; otherwise, it may be a non-trivial problem. Message-oriented delivery semantics are offered by protocols such as TP4 and XTP.

Request-response protocols (such as the RPC protocols used by ONC+ or DCE) provide a reliable, transaction-oriented service that is commonly used for short-duration exchanges of messages between clients and server(s) located in a LAN environment. Request-response protocols may be implemented as a separate layer on top of connectionless or connection-oriented protocols. They also may be directly implemented via a reliable message-passing protocol such as VMTP [11].

• **Short-Duration vs. Long-Duration Services:** Services offered by network daemons may be classified loosely as *short-duration* or *long-duration.* Short-duration services execute in brief, typically fixed amounts of time. Computing the current time-of-day, resolving the Ethernet number of an IP address, and retrieving a disk block from a network file server are examples of relatively short-duration services. These services are often implemented using request-response RPC protocols and/or connectionless protocols such as UDP.

Long-duration services typically execute for extended, often variable lengths of time. Transferring a large file via `ftp` or `ftam`, accessing host resources remotely via `telnet`, or performing remote computer account backups over a network are examples of long-duration services. To improve efficiency and reliability, these services are usually implemented with connection-oriented protocols.

• **Internal vs. External Services:** An *internal service* is executed within the same address space as the daemon that receives the request (shown in Figure 4 (1)). An *external service*, on the other hand, is executed in a different address space (shown in Figure 4 (2)). In the latter case, a master service dispatcher process may be used to monitor a set of communication ports, accept connection requests and receive data indications, and then spawn a new process to perform the requested service(s) externally. Some daemon control frameworks (such as `inetd` [5] and the `ASX` framework [4]) support both internal and external services. `Inetd` executes

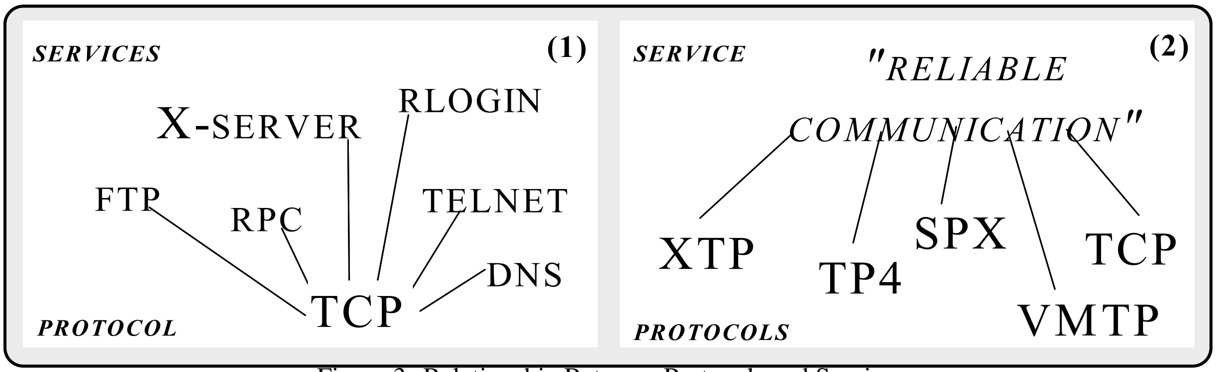

Figure 3: Relationship Between Protocols and Services

short-duration services (such as `echo` and `daytime`) internally via function-calls, whereas it executes longer-duration services (such as `ftp` and `telnet`) externally by spawning a separate process via `fork` and `exec`.

• **Stateful vs. Stateless Services:** A *stateful* service caches certain information (such as authentication keys, identification numbers, and file handles) in the daemon to reduce communication and computation overhead. For instance, the Remote File Sharing (RFS) file system [12] is a stateful service that allows the selective sharing of resources across a network. A *stateless* service, on the other hand, retains no volatile per-connection state information in a daemon. For example, the Network File System (NFS) [13] provides distributed data storage and retrieval services that do not maintain volatile state information within a server daemon.

These two approaches tradeoff efficiency and reliability, with a suitable choice depending on factors such as the probability and impact of host and network failures. Stateless services are generally simpler to configure and reconfigure reliably, and are usually implemented via connectionless protocols. Many common network applications (such as `ftp` and `telnet`) do not require retention of persistent state information between consecutive service invocations.

• **Layered Services vs. Monolithic Services:** Certain network daemon services decompose naturally into a series hierarchically-related tasks. For instance, standard layered protocol suites (such as the Internet and the ISO OSI reference models) may be designed and/or implemented via *layered services* (illustrated in Figure 5 (1)). These inter-connected services communicate by exchanging control and data messages. Several communication subsystem frameworks have been developed to simplify and automate the development and configuration of layered services [14, 15, 16, 4]. In general, these frameworks decouple the protocol and service functionality from (1) the time and/or order in which services are composed together and (2) the processing agent(s) (*e.g.,* processes and/or threads) used to execute services at run-time.

Monolithic services, on the other hand, are typically implemented via clusters of functionality that are neither related hierarchically nor directly inter-connected. One example of

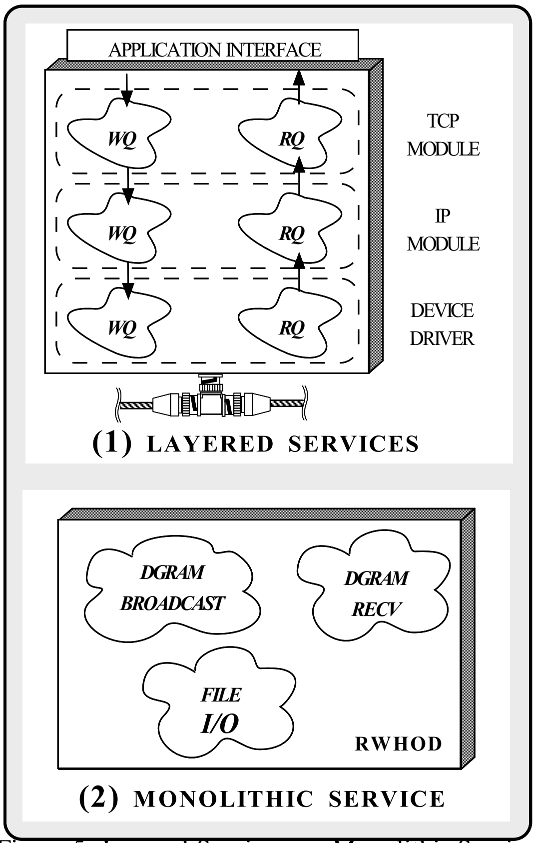

Figure 5: Layered Services vs. Monolithic Services

a monolithic service is the standard UNIX implementation of the rwho daemon [5]. As illustrated in Figure 5 (2), the datagram broadcast and receiver functions in an rwho daemon operate in relative isolation from each other to report and store user activities within hosts on a LAN subnet.

Developers typically have the opportunity to select either layered or monolithic service architectures for structuring their network daemons. This choice involves tradeoffs between modularity, extensibility, and efficiency. There are several advantages to designing server daemons in a layered manner. First, layering enhances reuse since a number of upper-layer application components may share lower-layer services. Second, representing applications as a series of inter-connected services enables transparent, incremental enhancement of daemon functionality. Third, a modular, layered approach facilitates macro-level performance improvements by allowing the selective omission of unnecessary service functionality. In general, modular designs improve the implementation, testing, and maintenance of network daemons.

There are also several disadvantages associated with using a layered approach for developing server daemons. A common criticism of layered implementations is that their modularity introduces too much overhead. For instance, layering may cause inefficiencies if buffer sizes are not matched appropriately in adjacent layers, thereby causing additional segmentation/reassembly and transmission delays [17].

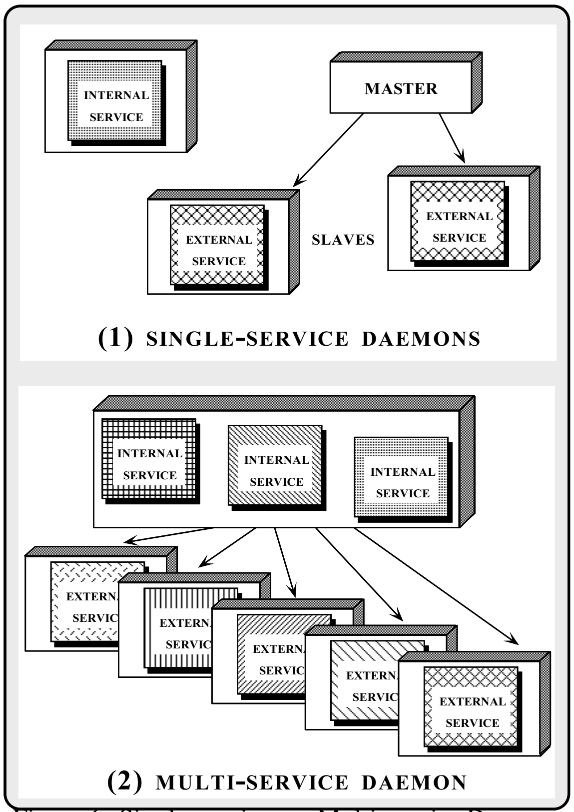

Figure 6: Single-service vs. Multi-service Daemons

3.2 Cooperation and Communication Dimensions

Separate components in a stand-alone application generally cooperate within a single address space by passing parameters via function calls and/or accessing global variables. In contrast, separate components in a distributed application must interact via more complex cooperation and communication mechanisms like the following:

• **Message Passing vs. Distributed Shared Memory:** Two widely studied cooperation models are based on *message passing* and *distributed shared memory*. Both models enable applications to interact with resources residing in separate processes executing on the same or different machines. IPC mechanisms based on message passing explicitly exchange different types of bytestream-oriented and record-oriented data. Distributed shared memory is a higher-level programming abstraction that attempts to provide applications with "network virtual memory." This article focuses on message passing communication mechanisms since distributed shared memory is primarily a research topic.

• **RPC vs. Lower-level IPC Mechanisms:** Modern operating systems provide applications with a variety of local and remote IPC mechanisms that access a wide-range of underlying communication protocol stacks such as TCP/IP, Novell IPX, SNA, and ISO OSI. RPC mechanisms available in toolkits such as DCE and ONC+ are an attractive level of abstraction for developing distributed applications. They provide developers with a programming paradigm that closely resembles the familiar procedure calling conventions used in stand-alone applications.

To meet stringent performance, functionality, and portability requirements, however, certain applications may need to directly access lower-level IPC mechanisms such as sockets or TLI. Often these applications require long-duration, bi-directional, uninterpreted byte-stream communication services that are not well-suited to the RPC "request-response" paradigm. Lower-level IPC mechanisms are often more efficient than RPC since they allow applications to omit functionality that may not be necessary (*e.g.,* presentation layer conversions for ASCII data) and enable finer-grain control over communication behavior (*e.g.,* permitting multicast transmission and signal-driven asynchronous I/O).

3.3 Daemon Architecture Dimensions

Protocols and services do not typically operate in isolation, but instead are accessed by applications within the context of a *daemon architecture*. These daemon architectures are generally structured according to the following alternatives:

• **Single-Service vs. Multi-Service Daemons:** *Single-service daemons* offer only one service. The rwho daemon (`rwhod` [5]) is an example of a single-service daemon. Early versions of UNIX ran standard network services (such as `ftp` and `telnet`) as distinct single-service daemons that were

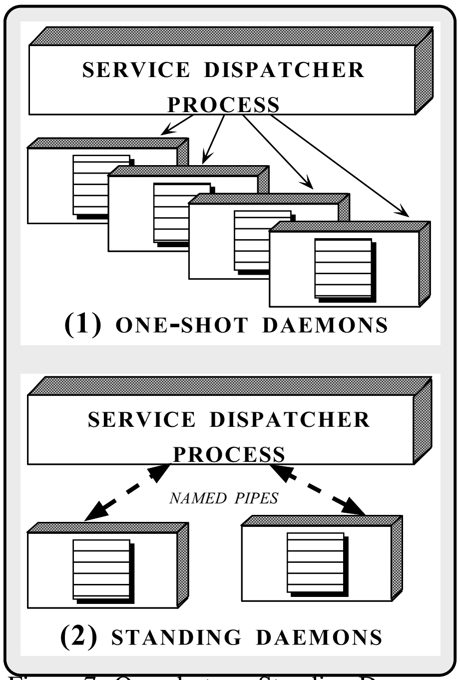

Figure 7: One-shot vs. Standing Daemons

initiated at OS boot-time [5]. Each instance of a service was executed externally in a separate process.

As the number of system daemons increased, however, this statically configured, single-service per-process approach revealed several limitations that are addressed by *multi-service daemons* such as `inetd` and `listen`. Multi-service daemons offer more than one service by integrating a collection of single-service daemons into one administrative unit. This approach helps to (1) reduce the consumption of OS resources (such as process table slots) by spawning daemons "on-demand," (2) simplify daemon development and reuse common code by automatically performing daemonization[2] operations, transport endpoint initialization, port monitoring, and service dispatching, (3) allow external services to be updated without modifying source code or terminating an executing service dispatcher process, and (4) consolidate the administration of network services via a standard set of configuration management utilities. For example, `inetd` is a multi-service daemon that coordinates and/or performs a wide variety of external (*e.g.,* `ftp` and `telnet`) and internal (*e.g.,* `daytime` and `echo`) services.

• **One-shot vs. Standing Daemons:** A *one-shot daemon* is spawned interactively by a master service dispatcher daemon such as `inetd` or `listen` to perform a service request in a separate thread or process (shown in Figure 7 (1)). One-shot daemons typically terminate once the request that triggered their creation completes. This approach may be less consumptive of system resources (such as memory and process table slots) since a one-shot daemon does not remain in system memory when it becomes idle.

A *standing daemon*, on the other hand, continues to run beyond the lifetime of the service request(s) it performs. Standing-daemons are often initiated at boot-time and receive connection and/or service requests via local interprocess communication (IPC) channels (such as `named pipes`) that are attached to a master service dispatcher daemon (shown in Figure 7 (2)). Standing-daemons improve service response time by amortizing the cost of spawning a daemon over a series of client requests. In addition, they also enable applications to reuse endpoint initialization, connection establishment, port demultiplexing, and service dispatching code.

3.4 Concurrency Dimensions

Daemons may associate one or more OS processes or threads with one or more services offered by a distributed application, leading to the following concurrency alternatives:

• **Iterative vs. Concurrent Daemons:** An *iterative daemon* handles each client request in its entirety before servicing subsequent requests. Moreover, while processing the current request, an iterative daemon either queues or ignores additional requests. This iterative structure is most suitable for (1) short-duration services that exhibit minimal variation in their execution time (such as the standard Internet `echo` and `daytime` services) and (2) infrequently run services (such as a remote file system backup service that runs nightly when systems are lightly loaded).

Iterative daemons are relatively straight-forward to design and implement since they often execute their service requests internally within a single process address space, as shown by the following pseudo-code:

```
void iterative_daemon (void)
{
   initialize listener endpoint(s)
   for (each new client request)
   {
      retrieve request from input queue
      perform requested service
      if (response required)
         send response to client
   }
}
```

As a consequence of this iterative structure, the processing of each request is serialized at a relatively coarse-grained level (*e.g.,* at the interface between the application and the transport layer). However, this coarse-grained level of concurrency potentially underutilizes the processing resources (such as multiple CPUs) and OS features (such as support for parallel DMA transfer to/from I/O devices) available on a host platform. In addition, iterative daemons may also prevent clients from making progress while they are blocked awaiting their turn. Client-side blocking tends to complicate retransmission timeout calculations. This, in turn, triggers

[2]Daemonization typically involves (1) dynamically spawning a new process, (2) closing all unnecessary file descriptors, (3) changing the current working directory to the root directory, (4) resetting the file access creation mask, (5) disassociating from the controlling process group and the controlling terminal, and (6) ignoring terminal I/O-related signals [5].

excessive network traffic and may result in duplicate requests being received by a server.

A *concurrent daemon*, on the other hand, handles multiple requests from clients simultaneously. Depending on the OS and hardware platform, a concurrent daemon either executes its services on multiple CPUs or time-slices its attention between separate services on a single CPU. If the server is a single-service daemon, multiple copies of the same service may run simultaneously. Moreover, if the server is a multi-service daemon, multiple copies of different services may also run simultaneously.

Concurrent daemons are well-suited for I/O-bound and/or long-duration services that require a variable amount of time to execute. Unlike iterative daemons, concurrent daemons allow more fine-grained synchronization techniques that serialize requests at an application-defined level (such as record-level locking in a database). This requires concurrency control mechanisms (such as semaphores or mutex locks [18]) that ensure robust cooperation and sharing of data between simultaneously active processes and threads.

Multi-threading mechanisms are rapidly becoming available on most OS platforms [18]. A thread is an independent series of instructions executed within a single process address space. This address space may be shared with other independently executing threads. Threads are often characterized as "lightweight processes" since they maintain minimal state information, require less overhead to spawn and synchronize, and inter-communicate via shared memory rather than message passing [19]. Under certain circumstances, it is advantageous to implement concurrent daemons that perform multiple service requests in separate threads rather than separate processes. For example, cooperating services that frequently reference common memory-resident data structures are often simpler and more efficient to implement via threads.

Executing all services via threads within the same address space potentially reduces daemon robustness, however. This problem occurs since separate threads within a single process are generally not protected from one another in order to minimize time and space overhead. Therefore, one faulty service may corrupt global data shared by services running in other threads in the process. This may produce incorrect results, crash an entire process, or cause a daemon to hang indefinitely. To increase robustness, many traditional concurrent daemons are implemented as external services. For example, services that base their security mechanisms on process ownership (such as the Internet `ftp` and `telnet`) are typically implemented as external services to prevent accidental or intentional access to unauthorized resources.

Concurrent daemons are often structured with a master service dispatcher that spawns a separate slave process or thread to perform each request asynchronously. Typically, the master dispatcher process continues to listen for new requests, as follows:

```
void concurrent_daemon_master (void)
{
   initialize listener endpoint(s)
   for (each new client request)
   {
      retrieve request from input queue
      if (one-shot daemon)
         spawn new slave process or thread
      pass request to the slave process
         or thread that performs the
         requested service
   }
}
```

The slave portion of a concurrent daemon generally executes within a separate process or thread using the following algorithm:

```
void concurrent_daemon_slave (void)
{
   for (each request from master daemon)
   {
      perform requested service
      if (response required)
         send response to client
      if (slave is a one-shot daemon)
         terminate
   }
}
```

In most modern operating systems, concurrent processing of multiple service requests is supported directly by the OS. The OS handles all process and thread management activities required to schedule, suspend, and resume. Moreover, if multiple CPUs are available, daemon services may execute in parallel [18].

A concurrent daemon may also be designed to handle multiple requests simultaneously within a single-threaded process. For instance, the standard `X server` from the MIT X windows system operates as a single-threaded concurrent daemon. Single-threaded concurrent daemons may be implemented by explicitly time-slicing their attention to each request via techniques such as port demultiplexing (*e.g.,* `select` or `poll`), non-blocking I/O, or signal-based user-level coroutines. The following pseudo-code illustrates the typical style of programming used for a single-threaded concurrent daemon based on `select` or `poll`:[3]

```
void single_threaded_concurrent_daemon (void)
{
   initialize listener endpoint(s)
   for (;;)
   {
      wait for client requests on
         multiple endpoints simultaneously
      for (each active client request)
      {
         perform request
         if (response is necessary)
            send response to client
      }
   }
}
```

[3]It is tempting to decompose a network daemon's internal architecture by refining these algorithmic pseudo-code examples directly. As subsequent articles demonstrate, however, structuring daemons via object-oriented techniques results in significantly more modular, extensible, and reusable server designs and implementations.

Compared with full-fledged OS support for multi-threading, single-threaded concurrency techniques possess several limitations. They are generally complicated to program since developers must manually perform process preemption by explicitly yielding the thread of control periodically and manually saving and restoring context information during service suspension and resumption. Moreover, each request must be executed for a sufficiently short duration so it appears to clients that requests are being handled concurrently rather than sequentially. Another limitation with single-threaded concurrent daemons is that their performance may be greatly reduced if an OS blocks all services in an entire process whenever one service makes a system call or incurs a page fault. Multi-threading mechanisms in many modern operating systems [18, 20] overcome these performance limitations by allowing preemptive, fully parallel execution of independent services running in separate threads.

Both iterative and concurrent daemons may offer multiple services or simply a single service. For example, `inetd` is a concurrent multi-service daemon with respect to its external services, whereas it is an iterative, multi-service daemon with respect to its internal services. Other daemon control frameworks (such as the `ASX` framework [4]) offer more flexible support for concurrent execution of both internal and external services.

• **Eager vs. On-demand vs. Lazy Process/Thread Invocation:** Several process- and thread-based invocation mechanisms help developers optimize concurrent daemon performance. The alternatives discussed below enable developers to adaptively tune daemon concurrency levels to match client demands and available OS processing resources. In general, the different approaches tradeoff decreased startup overhead for increased resource consumption.

Eager invocation pre-spawns one or more OS processes or threads at daemon creation time. These "warm-started" execution resources form a pool that helps improve response time by reducing service startup overhead. Depending on factors such as number of available CPUs, current machine load, or the length of a client request queue, this pool may be expanded or contracted dynamically. *On-demand invocation* spawns a new slave process or thread in response to the arrival of client requests. This eliminates unnecessary resource consumption at the expense of higher costs for starting services. *Lazy invocation* does not immediately spawn a process when a client request is received. Instead, a timer is set and the request is handled iteratively by the daemon. If the timer expires a new slave process is automatically spawned to continue processing the service independently from the master service dispatcher process [21].

• **Task-based vs. Message-based Process Architectures:** A process architecture represents a binding between various units of application services processing (such as layers, functions, connections, and messages) and various structural configurations of logical and/or physical CPUs [22]. The process architecture is one of several factors (along with protocol, bus, memory, and network interface characteristics) that significantly impact network daemon performance.

The three basic elements that form the foundation of a process architecture are (1) the *processing elements* (CPUs), which are the underlying execution agents for daemon code, (2) *data and control messages*, which are typically sent and received from one or more applications and network devices, and (3) *service processing tasks*, which perform services upon messages as they arrive and depart. Based upon this classification, two basic types of process architectures may be distinguished: *task-based* and *message-based*.

In general, task-based process architectures structure multiple CPUs according to units of daemon service functionality. Conversely, message-based process architectures structure the CPUs according to the control and data messages received from applications and network interfaces. The choice of process architecture affects key sources of application performance overhead (such as context switching, synchronization, scheduling, and data movements costs), as well as influencing demultiplexing strategies and protocol programming techniques [15]. An in-depth survey of alternative process architectures appears in [23].

3.5 Configuration Dimensions

A complete network server daemon is typically created by configuring together its constituent services at static link-time or during daemon execution. The steps required to configure a network daemon involve (1) naming and locating a set of relevant services and (2) linking these services into the address space of one or more daemons. These configuration steps may be performed statically and/or dynamically, as discussed below:

• **Statically Named vs. Dynamically Named Services:** *Statically named services* bind the name of a service onto object code that exists at compile-time and/or static link-time. For example, the internal services offered by `inetd` (such as `echo` and `daytime`) are statically named via "built-in" functions stored internally within `inetd`.

Dynamically named services, on the other hand, defer the binding of a service name onto the object code that implements the service. Therefore, the code need not be identified, nor even exist, until a daemon begins executing the corresponding service at run-time. A common example of dynamic naming is demonstrated by `inetd`'s external services (*e.g.,* `rlogin` and `ftp`). These services may be updated by modifying the `inetd.conf` configuration file and sending the SIGHUP signal to the `inetd` process. When `inetd` receives this signal, it re-reads its configuration file and reinitializes the services it offers.

• **Static vs. Dynamic Linking:** *Static linking* is a technique for creating a complete executable program by binding together all its object files at compile-time and/or static link-time (shown in Figure 8 (1)). *Dynamic linking*, on the other hand, inserts and/or removes object files into or from the address space of a process when a program is initially invoked or later during run-time (as shown in Figure 8 (2)). Modern

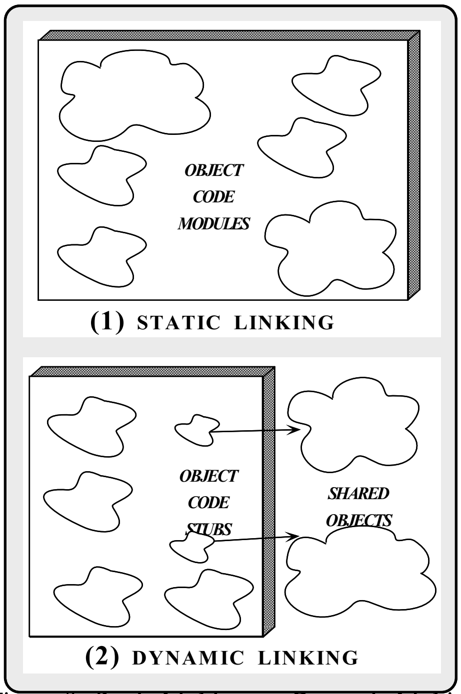

Figure 8: Static Linking vs. Dynamic Linking

operating systems generally support both *implicit* and *explicit* dynamic linking:

- Implicit dynamic linking defers most address resolution and relocation operations until a dynamically linked function is first referenced. This "lazy evaluation" technique minimizes link editing overhead during daemon initialization. Implicit dynamic linking is used to implement *shared libraries* [24], also known as *dynamic-link libraries* (DLLs) [20]. Ideally, only one copy of shared library code exists, regardless of the number of processes that are simultaneously executing the library's code. Therefore, in many circumstances shared libraries reduce the memory consumption of both a process in memory and the corresponding program image stored on disk.
- Explicit dynamic linking allows an application to obtain, utilize, and/or remove the run-time address bindings of certain symbols defined in shared object files. Widely available explicit dynamic linking facilities include the `dlopen/dlsym/dlclose` routines in SVR4 and the `LoadLibrary/GetProcAddress` routines in the WIN32 subsystem of Windows NT. Developers must carefully consider the subtle tradeoffs between flexibility and time/space efficiency when choosing between dynamic and static linking ([24] enumerates many of the tradeoffs).

• **Static vs. Dynamic Configuration:** In the context of network daemons, *static configuration* refers to the process of initializing a daemon that contains statically named internal and/or external services. In this case, a daemon's services are not extensible at run-time, which may be necessary for secure daemons that contain only "trusted" services.

Dynamic configuration, on the other hand, refers to the process of initializing a daemon that offers dynamically named internal and/or external services. When combined with dynamic linking and process/thread creation mechanisms, the services offered by dynamically configured daemons may be extended flexibly at invocation-time or during run-time. This type of flexibility is appealing since it facilitates the following configuration-related activities:

- *Functional Subsetting* – dynamic configuration simplifies the steps necessary to produce subsets of functionality for application families developed to run across a range of platforms. For example, by enabling the fine-grain addition, removal, or modification of services, explicit dynamic linking allows the same application framework to be used for space efficient ROM-based applications, as well as for larger GUI-based distributed applications.
- *Application Workload Balancing* – It is difficult to determine the relative processing characteristics of application services *a priori* since workloads often vary at run-time. Therefore, it may be necessary to experiment with alternative configurations that locate application services on different host machines throughout a network. For example, developers may have the opportunity to place certain services (such as image rendering) on either side of a client/server application. Bottlenecks may result if many services are configured into the server-side of an application and too many active clients simultaneously access these services. Conversely, configuring many services into the client-side may also result in a bottleneck since clients often execute on cheaper, less powerful host machines.
- *Dynamic Service Reconfiguration* – Highly available distributed applications (such as mission-critical systems that perform on-line transaction processing or real-time remote process control) may require flexible dynamic reconfiguration management capabilities. For example, it may be necessary to phase new versions of a service into a daemon without disrupting its currently executing services. Explicit dynamic linking mechanisms significantly enhance the functionality and flexibility of network daemons since they enable services to be inserted, deleted, or modified at run-time *without* first terminating and restarting the underlying process or thread(s) [25].

4 Concluding Remarks

Distributed computing is a promising technology for improving collaboration through connectivity and interworking; performance through multi-processing; reliability and availability through replication; scalability, extensibility, and portability through modularity; and cost effectiveness through resource sharing and open systems. Compared with standalone applications, however, implementing distributed applications is often far more challenging since developers must address additional topics and consider more design alternatives. This article presents a domain analysis of five design dimensions that developers of network daemons must address.

A domain analysis is frequently more effective when it evolves along with the design and implementation of an application framework. To illustrate how this evolution occurs in practice, upcoming articles in this series utilize the domain analysis presented in this article to motivate and characterize the structure and functionality of the ADAPTIVE Service eXecutive (`ASX`) framework.

The `ASX` framework is an object-oriented infrastructure that simplifies the development of distributed applications by improving the modularity, extensibility, reusability, portability, and correctness of their communication, concurrency, and configuration mechanisms. The tools and techniques in the `ASX` framework leverage off advanced OS mechanisms (such as dynamic linking, multi-threading, and port demultiplexing) to support the development, configuration, and use of a wide range of network daemons.

The complete source code, documentation, and example test drivers for the `ASX` framework is available via anonymous ftp from `ics.uci.edu` in the file `gnu/C++_wrappers.tar.Z`. The current release has been tested extensively on a variety of Sun workstations running SunOS 4.x and 5.x.

Acknowledgements

Special thanks to Stan Lippman, Ron Guilmette, and Eli Charne for comments and suggestions that improved the style and substance of this article.

References

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

[2] 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.

[3] 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.

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

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

[6] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, vol. 20, pp. 10–19, Apr. 1987.

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

[8] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," in *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

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

[10] J. Postel and J. Reynolds, "Telnet Protocol Specficiation," *Network Information Center RFC 854*, pp. 1–45, May 1983.

[11] D. R. Cheriton, "VMTP: Versatile Message Transaction Protocol Specification," *Network Information Center RFC 1045*, pp. 1–123, Feb. 1988.

[12] A. L. Sabsevitz, "Distributed UNIX System – Remote File Sharing," in *UNIX System Software Readings*, pp. 109–136, AT&T Unix Pacific Co. Ltd, 1988.

[13] R. Sandberg, "The Sun Network Filesystem: Design, Implementation, and Experience," in *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, 1986.

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

[15] 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.

[16] 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.

[17] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica, "Is Layering Harmful?," *IEEE Network Magazine*, January 1992.

[18] 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.

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

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

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

[22] D. C. Schmidt, 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.

[23] 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.

[24] 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.

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