Flexible Configuration of High-Performance Object-Oriented Distributed Communication Systems

Position Paper for OOPSLA '94 Workshop
on Flexibility in System Software

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

An earlier version of this paper appeared at the OOPSLA '94 workshop on Flexibility in System Software, Portland, OR, October 1994.

1 Introduction

The demand for extensible, robust, and efficient distributed communication systems is increasing. Distributed communication systems are characterized by significant amounts of network traffic. Examples of these systems include global personal communication systems, telecommunication switch management platforms, video-on-demand servers, real-time market data monitoring systems, and the underlying communication protocol stacks.

Although distributing application services among a set of autonomous hosts offers many potential benefits, developing distributed systems is more complex than developing non-distributed systems. A major source of complexity arises when mapping application services flexibly and efficiently onto host processes. Selecting an efficient mapping is difficult since service processing behavior, network/host workload, and OS/hardware platform characteristics may vary dynamically. Therefore, it is essential to develop software tools that support flexibility with respect to the following design criteria:

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

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

Various strategies and tactics for developing flexible system software have emerged in several domains. One approach (used by distributed application frameworks such as Regis [1] and Polylith [2]) represents application state attributes as abstract data types to facilitate service configuration and reconfiguration. Another approach (used by daemon management frameworks such as `inetd` and `listen` in System V Release 4 UNIX [3]) provides mechanisms for automating tedious and error-prone activities associated with configuring and reconfiguring network daemons. These mechanisms automate service initialization, reduce the consumption of OS resources by spawning service handlers "on-demand," allow daemon services to be updated without modifying existing source code, and consolidate the administration of network services via uniform configuration management operations.

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

For example, the standard version of `inetd` is written in C and its implementation is characterized by a proliferation of global variables, a lack of information hiding, and an algorithmic decomposition that precludes fine-grained reuse of its internal components. More importantly, existing frameworks do not provide automated support for (1) dynamically linking services into an application's address space at run-time and (2) executing these services in parallel via one or more lightweight processes [4] that communicate using shared memory. Instead, these frameworks configure and reconfigure application services by spawning heavyweight processes and connecting these processes via relatively heavyweight IPC mechanisms (such as pipes and sockets).

The central thesis presented in this position paper is that object-oriented frameworks for flexible, high-performance distributed communication systems must incorporate lightweight (re)configuration mechanisms based upon dynamic linking and lightweight processes. Otherwise, the performance cost of flexibility may exceed its benefits, thereby limiting widespread adoption of flexible software configuration techniques. In addition, techniques for parallelizing active objects within applications must address the primary sources of overhead associated with parallel processing. On shared memory multi-processors, for instance, these sources of overhead include context switching, synchronization, and data movement.

The remainder of this paper describes how object-oriented techniques are being applied successfully in practice to improve system software flexibility without unduly degrading performance. Section 2 outlines the structure of an object-oriented framework that simplifies the flexible configuration and reconfiguration of distributed communication systems; Section 3 discusses how this framework has been used to configure and test efficient protocol stacks on shared memory multi-processor platforms; and Section 4 presents concluding remarks.

2 An Object-Oriented Framework for Configuring Distributed Communication System Software

To help simplify the development, configuration, and reconfiguration of distributed communication systems, we have developed an object-oriented framework called the ADAPTIVE Service eXecutive (`ASX`) [5]. The `ASX` framework supports the flexible configuration of distributed communication systems that are built by specializing and composing reusable components. These application-independent components handle interprocess communication [6], event demultiplexing [7], flexible service (re)configuration [5], and concurrency [8]. Application-specific services may inherit and instantiate these framework components in order to compose a particular communication system.

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

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

- *The type and number of services associated with each process* – The `ASX` framework supports the configuration of applications that contain multiple services. These services may be instances of the same service or instances of different services.
- *The point of time at which service(s) are configured into an application* – The `ASX` framework encapsulates dynamic linking mechanisms provided by an operating system. Dynamic linking is a lightweight mechanism for configuring services into an application either statically (at compile-time or link-time) or dynamically (when an application first begins executing or while it is running). The `ASX` framework allows the choice between static and dynamic configuration to be deferred until installation-time [5].
- *The order in which hierarchically-related services are composed into an application* – An application may be represented as a series of hierarchically-related services that communicate by passing typed messages. These services may be flexibly configured in order to satisfy application requirements and enhance component reuse.

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

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

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

- *The type and number of concurrent processes and/or threads used to perform services at run-time* – The `ASX` framework encapsulates different flavors of multi-threading mechanisms (such as POSIX threads, MACH cthreads, Solaris threads, and Windows NT threads) to facilitate portability and greater functionality for multi-processing capabilities available on an OS platform. On SunOS 5.x UNIX [4], for instance, developers may select between user-level and kernel-level threads at run-time.
- *Flexible selection of process architectures* – The `ASX` framework enables the flexible configuration of several message-based and task-based process architectures [9]. A process architecture binds units of application service processing (such as layers, functions, connections, or messages) with one or more CPUs. The choice of process architecture directly impacts sources of significant overhead for distributed applications (such as memory-to-memory copying and data manipulation, context switching, scheduling, and synchronization).

The `ASX` framework enables applications to avoid premature commitment to the structural, communication, and concurrency characteristics described above until late in the development cycle (i.e., during installation-time or run-time).

This "late binding" method of configuring services into applications enhances application portability, reusability, and extensibility. It does this by deferring design and implementation decisions until sufficient information is available to select efficient policies and mechanisms. The structure and functionality of the ASX framework is described in greater detail in [6, 8, 7, 5].

3 Configuring High-performance Distributed Communication Systems

Parallel processing is a promising technique for improving the performance of distributed communication systems [10]. In this domain, flexible software configuration techniques are useful for configuring parallelism into communication systems. This section describes how the ASX framework has been used to conduct experiments with alternative process architectures on multi-processor platforms. The ASX framework contributed to these experiments by decoupling protocol-specific functionality from the underlying parallel process architecture.

Object-oriented distributed communication systems are typically structured as a set of collaborating active objects [11]. Each active object binds a separate thread of control together with its data and methods. A parallelized distributed communication system consists of both active objects (which execute autonomously in their own threads of control), as well as passive objects (which do not maintain their own thread of control). Active objects communicate by passing typed messages to other active objects. Each active object maintains a queue of pending messages that it will process. Depending on the sophistication of the underlying operating systems and number of available CPUs on hosts, multiple active objects may execute in parallel in separate lightweight or heavyweight processes.

Many communication systems decompose naturally into a series of hierarchically-related tasks. For instance, standard layered protocol stacks (such as those specified by the Internet and the ISO OSI reference models) may be modeled and implemented as a stack of layers. Each layer performs protocol processing tasks upon messages exchanged with protocol stacks running on other hosts throughout a network. This type of layered architecture is also common in other domains such as network management [12] and configurable distributed systems [2, 1, 5].

On a multi-processor host, it is tempting to design and implement layered communication systems by interconnecting a stack of active objects. Each active object encapsulates the set of tasks in a particular layer (such as the network layer, the transport layer, and the presentation layer). In this "task-based" process architecture, tasks are implemented as active objects, whereas messages processed by the tasks are treated as passive objects (shown in Figure 1 (1)). Parallelism is achieved by executing task objects in separate CPUs and passing messages between the tasks/CPUs using some form of IPC (such as pipes or UNIX-domain sockets).

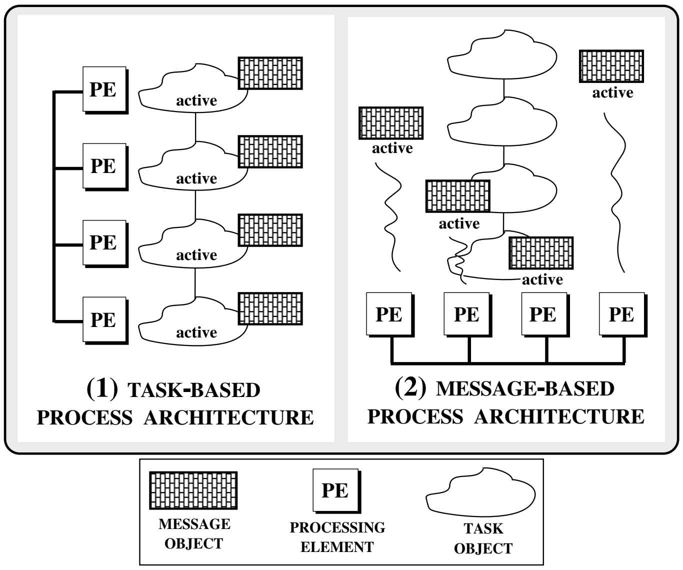

Figure 1: Process Architecture Components and Interrelationships

Task-based process architectures are an intuitive, widely-used [1, 2, 10] means to structure parallelism. They map directly onto layered communication models using a "producer/consumer" architecture, which is relatively simple to design and implement. In addition, it is straightforward to reconfigure task-based process architectures at run-time since protocol tasks are tightly coupled with processes. Therefore, replacing a task simply involves terminating an existing process and restarting another process containing the new task.

An alternative approach to modeling and implementing protocol stack parallelism involves binding the CPUs to the messages that flow through a series of hierarchically-related tasks. In this "message-based" process architecture, messages are the active objects, whereas tasks are the passive objects (shown in Figure 1 (2)). Parallelism is achieved by escorting multiple messages on separate CPUs simultaneously through a stack of interconnected processing tasks.

The primary advantage of message-based process architectures is that each message is associated with a single CPU throughout most of its task processing. This arrangement substantially reduces context switching and data movement overhead. Note, however, that dynamic linking is required to enable lightweight reconfiguration of protocol stacks implemented using message-based process architectures. This requirement occurs since protocol tasks are decoupled from processes.

Protocol stacks (such as the Internet and ISO OSI reference models) may be implemented using either task-based or message-based process architectures. However, different process architectures exhibit significantly different performance characteristics. These characteristics depend on the underlying operating system and hardware platform. For instance, in a non-shared memory transputer multi-processor environment, message-based process architectures result in high levels of synchronization overhead [13]. Likewise, on shared memory multi-processor platforms, task-based process architectures result in high data movement and context switching overhead.

We performed experiments on TCP/IP-based protocol stacks implemented using the ASX framework. These experiments indicate that message-based process architectures significantly outperform task-based process architectures on a shared memory multi-processor platform [8]. However, different results would occur on platforms that possess different relative costs for context switching, synchronization, and data movement. Therefore, an object-oriented framework that allows developers to flexibly configure process architectures at installation-time or run-time is essential to facilitate extensibility and performance of system software.

4 Concluding Remarks

The choice of techniques for configuring components into distributed communication systems have a significant impact on extensibility and performance. Conventional techniques for configuring and reconfiguring application services utilize heavyweight processes that communicate via heavyweight IPC mechanisms. However, the overhead of these techniques discourages developers of high-performance distributed communication systems from reaping the benefits of flexibly configurable software.

Achieving the wide-spread adoption of flexible system software requires lightweight (re)configuration mechanisms based upon dynamic linking and lightweight processes. Dynamic linking helps to improve extensibility by deferring design and implementation decisions until enough information is available to make effective application configuration choices. When dynamic linking is combined with lightweight processes, it becomes possible to flexibly configure efficient parallel process architectures that are tailored to their run-time environments.

The ASX framework enhances component reuse, automates configuration and reconfiguration, and helps to improve the performance of distributed communication systems that execute their services in parallel on multi-processor platforms. To improve extensibility and performance, the ASX framework decouples application-specific service functionality from both the temporal and structural bindings of these services onto heavyweight and lightweight processes. Thus, distributed applications may be updated and extended without modifying, recompiling, relinking, or restarting systems at run-time.

The ASX framework is available via anonymous ftp from `ics.uci.edu` in the file `gnu/C++_wrappers.tar.Z`. This distribution contains complete source code, documentation, and example test drivers for the C++ components. Components in the ASX framework have been ported to both UNIX and Windows NT and are currently being used in a number of commercial products including the AT&T Q.port ATM signaling software product, the Ericsson EOS family of telecommunication switch monitoring applications, and the network management portion of the Motorola Iridium global personal communication system.

References

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

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

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

[4] 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.

[5] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, to appear 1995.

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

[7] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching," in *Proceedings of the 1^{st} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), August 1994.

[8] 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.

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

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

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

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

[13] 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.