Title
An object-oriented framework for high performance distributed applications

Authors
Box, Donald F.
Suda, Tatsuya

Publication Date
1993

Peer reviewed

An Object-Oriented Framework for High Performance Distributed Applications

TR-93-9

Donald F. Box and Tatsuya Suda
Department of Information and Computer Science,
University of California, Irvine,
Irvine, CA 92717, U.S.A.
(714) 725-3097 (phone)
(714) 856-4056 (fax)
dbox@ics.uci.edu *

Abstract

For a variety of reasons, distributed applications often must be implemented using existing conventional programming languages and operating systems. Creating new high performance distributed applications without high-level support from the programming language or operating system requires that the mechanism of distribution be selected early in the design stage, reducing the flexibility and/or efficiency of the design and subsequent implementation. Adding distribution to existing applications can result in an inordinate amount of reengineering due to the complexity and heterogeneity of interapplication communication mechanisms and their interfaces.

This paper describes a new architecture for high performance distributed applications and a supporting framework. This architecture applies object-oriented design and implementation techniques to build a framework for platform-independent distributed application specification and implementation using existing programming languages and operating systems. It utilizes an efficient and extensible layering architecture that allows new abstract data types, new protocols, *and* new interprocess communication mechanisms to be added as they become necessary.

*This material is based upon work supported by the National Science Foundation under Grant No. NCR-8907909. This research is also in part supported by University of California MICRO program.

1 Introduction

There has been considerable research in the areas of both distributed operating systems[1, 2, 3, 4, 5, 6], as well as programming languages for distributed application development[7, 8, 9, 10, 11]. However, distributed applications must often be implemented using existing conventional programming languages and operating systems. Conventional languages are and will be used to accommodate the "dusty deck" of existing source code, as well as to utilize the experience of programmers accustomed to working in these languages. Conventional operating systems are and will be used where there is no single distributed operating system for every entity in a particular installation (*e.g.*, a network management application that must communicate with dedicated routing hardware, a distributed database application that must communicate with hosts running a variety of operating systems). It is the current and continued presence of these environments that motivates this work.

Developing high performance distributed applications using conventional programming languages and operating systems is a non-trivial task due to the complexity and heterogeneity of interapplication communication mechanisms and their interfaces. Adding distribution to existing applications can result in an inordinate amount of reengineering due to the lack of high-level support for distribution in most traditional environments. Creating new distributed applications requires the mechanism of distribution be selected very early in the design stage to avoid this reengineering effort. This causes the distribution mechanisms to either be (a) tightly coupled with the entire application (which makes changing the mechanism very difficult), or (b) loosely coupled via abstract interfaces (requiring explicit calls to initiate and support distribution and reducing efficiency due to additional function calls and data copying).

To simplify the task of implementing high performance distributed applications, we are proposing a two-level approach to the implementation and design of distributed applications. Each of these two levels applies object-oriented design principles to provide a framework for transparent object distribution.

- *Level 1: Communication Substrate Dependent* – By creating a uniform abstract interface to distribution/communication services (*e.g.*, sockets[12], RPC[13], XTI[14], ADAPTIVE[15], SNMP[16], various local and remote IPC and shared memory mechanisms), exactly one interface to distributional services is necessary. This allows distribution mechanisms to be selected late in the design process, as well as lessening the development effort when applications must be ported to a new operating environment. Level 1 represents the communication infrastructure of the framework.

- *Level 2: Communication Substrate Independent* – By providing a family of fundamental base classes that are capable of encoding themselves in a platform-independent manner, migrating instances of these classes over diverse platforms requires no additional effort. The substrate independent level is capable not only of explicitly importing and exporting objects, but is also capable of hiding the distributed nature of an object by using techniques

referred to as *distributed instantiation* and *remote delegation.* This allows an object to appear to exist on a local host, but any accesses to it are transparently delegated to the remote host where the object actually resides. Level 2 provides the paradigm for distributed object management.

We combine these two levels of abstraction to create a framework for high performance distributed application design and implementation. We are implementing and experimenting with this framework using the C++ programming language, which was chosen as the implementation language for its efficiency, its compatibility with the large body of existing C language code, and its support for object-oriented programming.

2 Design Goals

The framework described in Section 3 is designed to satisfy the following goals:

Basic Encoding Rule Independence and Transparency: Due to variances in processor architectures, operating systems, programming languages and their compilers, data (objects) that must be transported to a foreign host must be encoded into a format that all communicating parties can recognize. Under the OSI Reference Model[17], this task is the responsibility of Layer 6, the Presentation Layer. Presentation protocols are traditionally implemented using a *data definition language* for humans to specify the data formats to be exchanged. These data definitions are then compiled according to a set of *basic encoding rules* (BER) that specify the exact binary representation of each data type. For maximum flexibility, our framework allows Presentation Layer services to be transparently selected either for compatibility with existing applications (*e.g.*, ASN.1[18] for compatibility with OSI applications) or for the best match to new applications (*e.g.*, XDR with Multimedia extensions for bandwidth/processing intensive applications). A single object can be transported using multiple encoding schemes, with the correct one for a given end-to-end association selected automatically via strong typing and function/operator overloading. This allows a communicating entity to maintain a single internal data format that is most efficient for local processing, while using ASN.1 to communicate information to network management applications and a more efficient format for more time-critical communications.

Object/Task Location Independence and Transparency: The emergence of distributed object management systems and languages[19, 8, 20, 21] has shown that using a distributed object-oriented paradigm is a powerful and expressive way to design and implement distributed applications. Our framework seeks to provide distributed object management facilities by augmenting objects with the necessary member functions to transparently or explicitly designate the location of data members and the constituent operations performed. The actual location of various

application objects can be transparently selected by the application designer to match the communication characteristics of the application. Object location can be explicitly designated either at the time of object instantiation, or during the object's lifetime via a single member function. Finer-grain control of object location and migration can be specified by designating member functions or data members for remote invocation/instantiation, allowing the application designer to distribute a single object across multiple locations.

Communication Substrate Independence and Transparency: There is presently a very large number of communication substrates available to the distributed system designer (*e.g.*, TCP[22], OSI-TP4[23], VMTP[24], NETBLT[25], XTP[26], etc.), with new protocols on the horizon (*e.g.*, Bellcore's TP++, OSI HSTP). Each of these protocols provides varying levels of performance and types of service. To take advantage of advances in network technology, it is essential that applications sufficiently insulate themselves from idiosyncrasies of a given substrate without unduly reducing efficiency. Our framework seeks to decouple object transmission/reception from the underlying communication subsystem. This allows the same code to be used portably across many different communication subsystems without regard to the selected substrate. The framework provides a minimal yet functional base interface to basic communication services, while allowing access to substrate-specific features, functions and formats in an efficient yet *isolatable* manner.

Efficient yet Robust "Higher Layer" Protocol Services: By designing the layering architecture for both transparency and efficiency, protocol layers which were previously considered bottlenecks in distributed applications can now be used in high performance systems. Studies have shown that presentation layer processing is a major bottleneck in network performance[27, 28], due to both the complexity of the processing involved and the additional data movement incurred from translating data between formats. Our framework addresses both of these issues:

1. *Complexity* – The fundamental data types used in a presentation protocol are hand-coded and inlined to yield a highly efficient translation. Additionally, every built-in data type has a hand-tuned *type conversion* operator to allow efficient processing of built-in types. As composite data types are directly composed of the fundamental or built-in types, their translations are efficient as well. However, implementors are free to experiment with and hand-tune a given composite object's encoding and/or decoding.

2. *Redundant Data Copying* – The entire "data path" of the framework is designed to allow conversion-on-copy operations, scatter-read/gather-write, memory-mapped I/O, and "pipelining" of protocol processing operations. Additionally, we are experimenting with alternatives to the traditional socket interface to further reduce the need for copying.

Streamlined Development Process: Conventional systems require the designer to maintain a data description in a language other than the language being used to develop the application. Our framework allows designers and implementors to specify objects directly in the implementation language (*e.g.*, C++), without requiring an additional preprocessor or stub compiler. The fundamental data types and formats are precisely defined within the framework specification. This allows formats to be expressed unambiguously, while allowing the development cycle to be streamlined by using rapid prototyping techniques.

Medium Independence and Transparency: The class libraries used to encode data objects can easily be combined with the exiting C++ `iostream`[1] class libraries currently being standardized by ANSI. This allows objects to be stored in a platform-independent format with no additional implementation effort. It also allows persistent objects to be "played out" over a communication channel by an application that is unaware of the underlying format simply by transmitting the contents of a file.

3 Framework Components

Figure 1 shows the layering model used in our architecture. The Data Transport and Media Convergence layers correspond to the *Communication Substrate Dependent* level, as shown in Figure 2. The following is a description of each layer. The Presentation and Distribution layers correspond to the *Communication Substrate Independent* level, as shown in Figure 3.

3.1 Data Transport Layer

The Data Transport Layer provides the basic local and remote interprocess communication channel. This layer represents both the IPC mechanisms and their constituent Application Programmatic Interfaces (APIs). It is assumed that each underlying IPC mechanism provides either (1) a basic duplex data stream with either connection-oriented or connectionless semantics *or* (2) a shared memory interface with support for mapping a memory segment into and out of an address space.

Network Subsystems: The remote interprocess communication substrate. The basic connection-oriented network service is expected to provide an error-free, in-order delivery of bytes (*i.e.*, TCP[22] or equivalent). The basic connectionless network service is expected to simply provide a best effort delivery of datagrams (*i.e.*, UDP[29] or equivalent). Additionally, more diverse classes of network services can also be supported in this model. For example, ADAPTIVE[15]

[1]The `iostream` library is the C++ analog to the C programming language's `stdio` library. However, it offers the advantages of being type-safe and extensible to encompass new data types and I/O devices

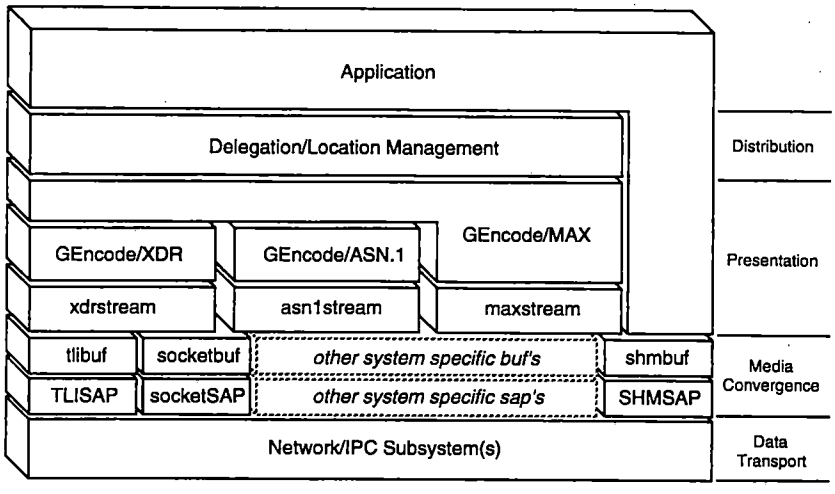

Figure 1: Layering Model

provides a multi-stream transport substrate that can be flexibly and adaptively configured to provide diverse grades of service to multimedia applications.

The Network Subsystem Layer also includes the APIs to network services, that allow user-space applications to access data transport operations in a protocol independent manner. Several network APIs supported include the BSD sockets, System V TLI, POSIX XTI, ADAPTIVE API, *x*-kernel[30], NetBIOS[31], and the WINSOCK[32] library. Each of the interfaces provides both communication operations (*e.g.*, open, close, send, recv) and addressing/naming services (*e.g.*, address formats, name resolution).

3.2 Media Convergence Layer

The Media Convergence Layer provides a consistent, buffered interface to Data Transport services. It provides a basic data source/sink interface for higher layer subsystems, and uses the underlying Data Transport services to drain or replenish its internal buffering layer.

SAP Layer: The collection of uniform Service Access Points (SAPs) that provide a consistent interface to diverse interapplication communication services. Each SAP provides an *impedance match* between the native API provided by a given communication substrate and the basic communication service abstraction required by higher layer subsystems. There are two primary classes of SAPs, those based on duplex communication channels and those based on shared memory. SAPs based on duplex channels are required to support read, write, and connection management operations. SAPs based on shared memory must support basic attachment and detachment operations. Both classes of SAP must support a *SAPAddress* that can represent a

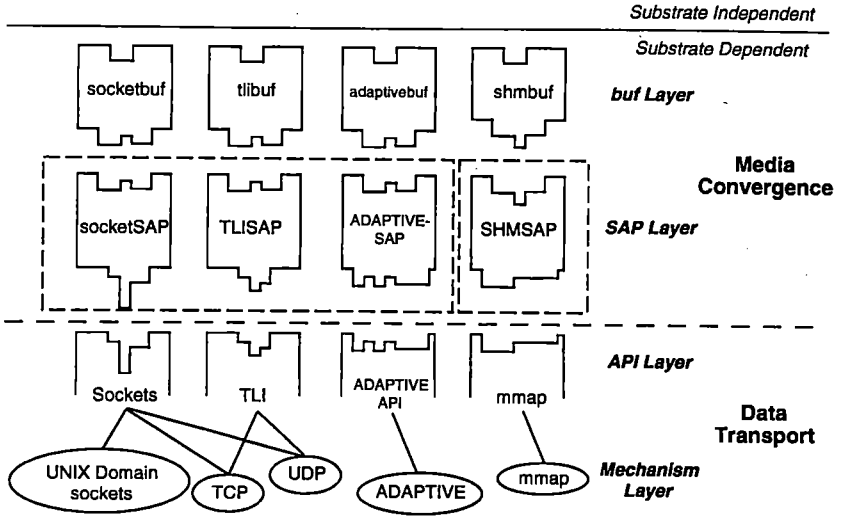

Figure 2: Communication Substrate Dependent Level

communications endpoint for a given communication *mechanism*. This separation of addressing from basic communication operations allows a given communication mechanism (*e.g.*, TCP) to utilize several possible API's (*e.g.*, sockets, XTI) and still maintain a single abstract SAPAddress format, thus decoupling the mechanism from the API. The uniformity of the SAP abstraction decouples the Communication Substrate from higher layer client subsystems. SAPs currently provided include socketSAP, TLISAP, ADAPTIVESAP, SHMSAP (Shared Memory via `mmap`). SAPAddresses include TCPAddress, UDPAddress, UNIXDSAddress (UNIX Domain Sockets), ADAPTIVEAddress, and SHMAddress.

buf Layer: The collection of transparent buffer managers that provide an efficient buffering scheme to the SAPs provided above. The buf[2] layer is necessary to reduce the number of system calls needed to send a *composite* object (*i.e.*, an object with multiple data members) and to minimize the amount of redundant data copying. The buf Layer is based on and is interoperable with the C++ iostream library[33, 34], which provides two sets of abstractions:

1. *streambuf* – the abstraction for a consumer/producer of bytes. To extend the iostream library to include a new I/O device or interface, one need only supply a streambuf interface to the device, and combine it with four specific classes via inheritance to allow existing classes to read and write to it automatically. The iostream library that accompanies the AT&T distribution of C++ provides streambuf interfaces to files and in-core memory.

[2]Identifiers containing *buf* and *stream* are in lowercase to remain consistent with the C++ iostream capitalization convention.

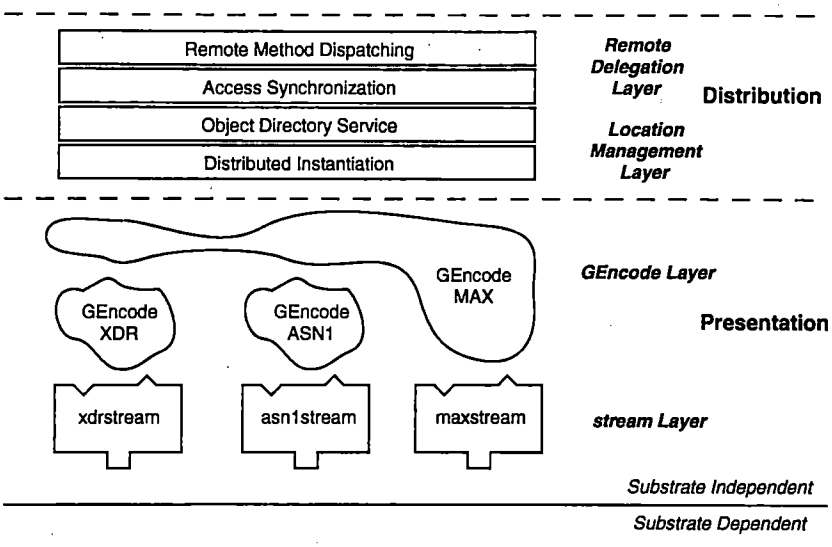

Figure 3: Communication Substrate Independent Level

2. *iostream* – the abstraction for formatted *insertion* and *extraction* of objects into/from a stream. The base classes *ostream* and *istream* each provide the insertion (output) and extraction (input) operators (<< and >> respectively) for each of the built-in data types supported by the language (*e.g.*, char, int, float). istreams and ostreams must be combined with a streambuf to provide a usable stream (*e.g.*, istream + filebuf = ifstream, a stream that *extracts* objects from a *file*). iostreams are not only extensible with respect to the devices they can support, but also with respect to the types of objects they can insert or extract. User-created data types (classes) can define their own input and output operations by *overloading* the insertion and extraction operators to support the new data type.

Each available communication subsystem has a corresponding SAP and buf that accesses its services (*e.g.*, socketbuf, tlibuf, adaptivesap, shmbuf). These bufs can then be combined with the standard *istream* and *ostream* classes to provide a *formatted* I/O channel, or with a new *stream* class (described below) to provide an *encoded* I/O channel.

3.3 Presentation Layer

The Presentation Layer is responsible for resolving differences in data representations between heterogeneous host architectures. It accomplishes this by translating local internal data formats into an external format that can interpreted by the remote entity. It typically accomplishes this via one of two means:

1. *Explicit Typing* – each data object is "tagged" with a type identifier field that specifies the data type of the object in transit. It can then be followed by a length field that indicates the

remaining number of octets (or fundamental data units). These two fields are then followed by the actual data octets. This is known as a T-L-V scheme ("Tag-Length-Value") and is used as the basis for the Basic Encoding Rules of OSI ASN.1. This approach is in contrast to:

2. *Implicit Typing* – it is assumed that the receiver of the data is aware of exactly what type of data object is coming, therefore the tag field is redundant at best. This is the approach taken by XDR, a protocol that is designed to take advantage of regular data alignment and hardware-dictated formats.

[35] contains a comparison of three well known presentation protocols (XDR, ASN.1, and Apollo NDR). The authors resolve that T-L-V encodings are more general and potentially more bandwidth efficient, yet can be more complex to process, while fixed-format encodings such as XDR are more efficient to process, only slightly less efficient with respect to bandwidth, and can provide T-L-V functionality if necessary.

stream Layer: The stream layer is used to bind the various coding schemes listed below to an I/O channel (via its corresponding buf). For a given encoding scheme *Z*, an *izstream* and *ozstream* are implemented, providing *at least* the basic insertion or extraction operators for the built-in data types. Additionally, insertion or extraction operators will be provided for the corresponding GEncode class hierarchy that defines the basic data types used by the encoding scheme. In addition to providing the capability to statically bind an encoding scheme (stream) to an I/O device (buf), *stream manipulators*[3] are provided to allow encoding schemes to be switched on the fly. By inserting (or extracting) a `z_on` manipulator into a stream, the previous formatting/encoding scheme is suspended and replaced with the *z* encoding. Inserting (or extracting) a `z_off` manipulator restores the original formatting/encoding scheme.

GEncode/XDR: GEncode/XDR is a class library of primitive base classes which correspond directly with the standard eXternal Data Representation (XDR)[36], as shown in Figure 4. This provides a set of Basic Encoding Rules that allow objects to be shared across diverse host platforms. For each *built-in* data type (*e.g.*, char, int, double) a type-conversion operator is provided to convert between language/compiler dependent types and formats to their corresponding GEncode/XDR base class. By leveraging off of C++ type management mechanisms, the presence of the GEncode layer can be completely transparent to the application programmer. GEncode also supports collection classes (*e.g.*, Lists, Dictionaries, Sets), C++ references, and pointers.

[3]Manipulators are "functions" that can be inserted or extracted into/from a stream. Inserting/extracting a manipulator has the effect of calling the manipulator's corresponding function with the target stream as the function's first argument. Manipulators allow otherwise complex encoding/formatting expressions to be written as a simple series of insertions/extractions.

GEncode/XDR Name	Representation	C++ Data Type
SignedInteger	32bit 2's Comp. (Big Endian)	char,short,int,long
UnsignedInteger	32bit unsigned (Big Endian)	u_char,u_short,u_int,u_long
Enumeration	32bit 2's Comp. (Big Endian)	enum
Boolean	32bit 2's Comp. (Big Endian)	int
FloatingPoint	IEEE 32b FP (1s/8e/23m)	float
DPFloatingPoint	IEEE 64b FP (1s/11e/52m)	double, [long double]
FixedLengthOpaqueData	Arbitrary Data (zero padded)	any opaque range of bytes
VariableLengthOpaqueData	4B Len + Data (zero padded)	any opaque range of bytes
CharacterString	4B Len + Ascii String (zero padded)	char *
FixedLengthArray	n * element representation	T[n]
VariableLengthArray	4B Len + n * element representation	T[n]
DiscriminatedUnion	4B Tag + Anon. Union	long + union

Figure 4: GEncode/XDR Fundamental Base Classes

GEncode/ASN.1: An OSI Abstract Syntax Notation.1 version of GEncode/XDR. GEncode/ASN.1 is more complex than GEncode/XDR, as it has to address overflow issues (*i.e.*, reading an 8 octet integer into a 4 octet `long`). Also, it is difficult to directly support the ASN.1 notion of Set using only C++ constructs.

GEncode/MAX: A set of primitive base classes that represent Multimedia Activity eXtensions. These classes allow multimedia objects and basic application activities to be represented in a host platform independent manner. The Multimedia eXtensions we are currently implementing include support for

- 8KHz, 8 bit μ-law PCM audio
- 44.1KHz 16 bit linear PCM audio
- 44.1KHz 16 bit linear PCM audio (multi-channel)
- Indexed and Direct Color Pixmaps
- JPEG Still Image
- MPEG Motion Image

The Activity eXtensions we are currently investigating include non-blocking and asynchronous remote procedure calls, C++ pointer-to-member-function semantics, and language-independent procedure name binding.

3.4 Distribution Layer

From the Presentation layer down, the support for distribution consists primarily of efficient mechanisms for copying objects to and from heterogeneous systems. The Distribution layer is the

layer that creates an infrastructure for *transparently* migrating objects without explicit initiation from the programmer. The application programmer can simply specify the location where the object should be located (if desired) and can then access the object as if it were located in the local address space. The Distribution layer consists of the following 2 sublayers:

Location Management Layer: The Location Management layer orchestrates the migration of objects based on both *explicit* (*i.e.*, the object's `existOn` member function is explicitly invoked) and *implicit* (*i.e.*, a member function declared as *remote* is invoked) events. Object locations are managed through the use of:

- *Distributed Instantiation* – a technique where by overloading the language's `new` and `delete` operators, objects can be instantiated on remote hosts simply by passing an additional argument to the `new` operator.
- *Object Directory Service* – a distributed directory service from which application entities can capture a capability for an object based on an *object identifier tuple*.

Remote Delegation Layer: The Remote Delegation layer manages and execution of application object's member functions and arbitrates multiple accesses to a single object. By using a technique called *remote delegation*, an object's member functions are automatically invoked on the proper host system without programmer intervention.

4 Conclusions

We have presented a new architecture for the design and implementation of high performance distributed applications. We are currently experimenting with, conformance testing, and benchmarking our initial prototype implementation based on XDR, TCP/sockets, `mmap` shared memory, and ADAPTIVE, while implementing additional `bufs` and GEncode/ASN.1. The initial implementation has provided an expressive environment for specifying and implementing distributed applications.

References

[1] E. J. Berglund, "An Introduction to the V-System," *IEEE Micro*, vol. 10, pp. 35–52, August 1986.

[2] A. S. Tanenbaum, R. V. Renesse, H. V. Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. V. Rossum, "Experiences with the Amoeba Distributed Operating System," *Communications of the ACM*, vol. 33, pp. 46–63, December 1990.

[3] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System," *IEEE Computer*, vol. 21, pp. 23–36, February 1988.

[4] P. Dasgupta, R. LeBlanc, M. Ahamad, and U. Ramachandran, "The clouds distributed operating system," *IEEE Computer*, vol. 24, no. 12, pp. 34–44, December 1991.

[5] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure," *Communications ACM*, vol. 23, pp. 92–104, February 1980.

[6] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "Overview of the CHORUS Distributed Operating Systems," Tech. Rep. CS-TR-90-25, Chorus Systems, 1990.

[7] J. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *IEEE Computer*, vol. 19, pp. 26–34, Aug. 1986.

[8] A. D. Birrell, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, vol. 13, pp. 65–76, January 1987.

[9] M. B. Jones and R. F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," Tech. Rep. CMU-CS-87-150, Carnegie Mellon University, September 1986.

[10] B. Liskov, "Distributed Programming in Argus," *Communications of the ACM*, vol. 31, March 1988.

[11] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Transactions on Software Engineering*, pp. 190–205, Mar. 1992.

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

[13] Sun Microsystems, "RPC: Remote Procedure Call Protocol Specification," Tech. Rep. RFC-1057, Sun Microsystems, Inc., June 1988.

[14] IEEE, New York, NY, *Information Technology – Portable Operating System Interface (POSIX) – Part xx: Protocol Independent Interfaces (PII) (draft)*, 1992.

[15] D. F. Box, D. C. Schmidt, and T. Suda, "ADAPTIVE: An Object-Oriented Framework for Flexible and Adaptive Communication Protocols," in *Proceedings of the IFIP Conference on High Performance Networking*, (Belgium), IFIP, 1992.

[16] J. Case, M. Fedor, M. Schoffstall, and C. Davin, "Simple Network Management Protocol (SNMP)," *Network Information Center RFC 1157*, pp. 1–36, May 1990.

[17] I. O. for Standardization, "Information Processing Systems - Open Systems Interconnection - Basic Reference Model," Tech. Rep. ISO 7498, ISO, 1984.

[18] International Organization for Standardization, *Information processing systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, May 1987.

[19] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "A Distributed Implementation of the Shared Data-Object Model," in *Proceedings of the First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 1–19, IEEE, 1988.

[20] Object Management Group, "The Common Object Request Broker: Architecture and Specification," Tech. Rep. Revision 1.1, OMG, Dec. 1991.

[21] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, "An Overview of the ARJUNA Distributed Programming System," *IEEE Software*, pp. 66–73, Jan. 1991.

[22] J. Postel, "Transmission Control Protocol," *Network Information Center RFC 793*, pp. 1–85, Sept. 1981.

[23] OSI Standard, "OSI Transport Protocol Specification," Tech. Rep. ISO-8073, ISO, 1986.

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

[25] D. D. Clark, M. L. Lambert, and L. Zhang, "NETBLT: A Bulk Data Transfer Protocol," *Network Information Center RFC 998*, pp. 1–21, Mar. 1987.

[26] Protocol Engines Incorporated, Santa Barbara, *XTP Protocol Definition*, September 1990.

[27] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *SIGCOMM Symposium on Communications Architectures and Protocols*, (Philadelphia, PA), pp. 200–208, ACM, Sept. 1990.

[28] C. Huitema and A. Doghri, "A High Speed Approach for the OSI Presentation Protocol," in *Protocols for High Speed Networks* (H. Rudin and R. Williamson, eds.), IFIP, North-Holland, 1989.

[29] J. Postel, "User Datagram Protocol," *Network Information Center RFC 768*, pp. 1–3, Aug. 1980.

[30] 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.

[31] IBM Corp., *NetBIOS Application Development Guide*, Apr. 1987.

[32] *Windows Sockets – An Open Interface for Network Programming under Microsoft Windows*, June 1992.

[33] Bjarne Stroustrup, *The C++ Programming Language, 2nd Edition.* Addison-Wesley, 1990.

[34] *AT&T C++ Language System Library Manual*, 1990.

[35] C. Partridge and M. T. Rose, "A Comparison of External Data Formats," in *Proc. IFIP TC6 Working Symposium*, Oct. 1988.

[36] Sun Microsystems, "XDR: External Data Representation Standard," Tech. Rep. RFC-1014, Sun Microsystems, Inc., June 1987.