Structuring Distributed Shared Memory with the π Architecture

Dinesh C. Kulkarni, Arindam Banerji, Michael R. Casey and David L. Cohn
dcrl@cse.nd.edu

Distributed Computing Research Laboratory
University of Notre Dame, Notre Dame, IN 46556, USA

Abstract

The classic system software structure that hides implementation details behind an interface specification limits the effectiveness important paradigms. For example, distributed shared memory has long been recognized as a promising, but flawed, approach to distributed cooperation. The problem is that its optimal implementation depends heavily on the application interaction specifics. π is a new approach to the design of system software which allows subsystems, such as distributed shared memory, to be tailored to the varying needs of applications and hardware configurations. This paper describes the π architecture and discusses its use in a DSM subsystem for clustered workstations.

1: Introduction

This paper examines the use of a new architecture for the design of system software. The architecture allows the construction of flexible system software components, and the resulting realizations can be tailored to the needs of various applications. This paper focuses on the application of that architecture to distributed shared memory.

The system software architecture, called π, defines elements of system software as generalized objects. These π objects have two interfaces: one for usage and one for additional control. On the one hand, π enables system software components to evolve as computer hardware evolves and on the other it addresses the demands of emerging classes of applications like OODBs and multimedia. Evolutionary support is particularly important as mobile computing and dynamic interconnection topologies become an integral part of computing systems [11].

Distributed shared memory is an attractive programming paradigm particularly when it is supported by language level tools. When it is integrated into a language, such as Linda [2], assumptions are made about its "expected use", which colors its implementation [10]. On the other hand, application specificity of shared memory realizations is critical for achieving acceptable performance [6],[3]. Indeed, the problems of coherency and consistency controls seen in DSM [5] also occur in file systems and databases. A π-based distributed shared memory subsystem can be flexible enough to accommodate these wide ranging needs.

The next section outlines how the π architecture defines and supports flexibility. It is followed by a section that summarizes five different views of distributed shared memory, which each offer a different interface. The fourth section presents a graph model for software design and shows how maps this model to the computational reality. The application of the π approach to a clustered DSM implementation is presented in Section 5. Section 6 speculates on where π could take system software design and the final section relates π to other work.

2: A new architecture for system software

Changing application demands and ever-evolving hardware are making it hard to build suitable system software. Therefore, successful system software architectures must be *flexible* enough to span a broad range of uses [20] and accommodate change on a continuing basis.

2.1: The role of flexibility in system software

The implications of flexibility in hardware design are clear: it allows realizations of the same architecture across a broad price/performance spectrum. Flexibility for system software is more complex. No one-dimensional measure, such as manufacturing cost, defines a spectrum. Instead, its ability to adapt to a variety of application requirements and hardware configurations is its measure of flexibility.

Different classes of applications have different needs. No single set of primitives will satisfy them all, and no single implementation will be the best in all cases. For example, when data is shared between remote elements of a distributed application, the optimal coherency control mechanism depends on how the application is using the data. However, application data usage can change, and the coherency policy may need to change with it. Application diversity and the need for change, in general defy assumptions about expected usage pattern.

A flexible system software architecture must also accommodate a wide range of hardware platforms. The range includes not only various processor speeds, memory sizes and peripheral devices, but also multi-processor systems, diverse network topologies and new types of input/output media. In the future, as radio-based networking becomes common, these structures will change dynamically, requiring the operating system to adapt as elements of the system move into and out of range. For example, a file system for portable computers must be able to tolerate the absence of connection to the file server [12]. Indeed, with mobile computing, flexibility will no longer be desirable, it will be mandatory.

2.2: Constraints and conditions

Flexible system software should scale. That is, the same software architecture should be effective in embedded processors, in distributed multiprocessors and in large main-frames. This does *not* mean that one size fits all machines. Rather it says that flexibility allows tailoring to the needs of many different computing situations.

Scalability and flexibility require modularity. It should be possible to effectively combine system software components from different sources. A very simple example of this is the installable file system structure; this makes it possible to replace Microsoft's High Performance File System in OS/2 with IBM's Journaled File System. Modularity also allows increased functionality, such as adding distributed shared memory support to a virtual memory subsystem. Finally, the conventional concept of modularity should be extended to localize the visibility of changes made to a system software component.

2.3: Views of the system software

A programmer creating an application or a new system software component needs clearly defined *interfaces*. The interface should be high-level and hide implementation details, but fine-grained control is required to allow customiztion. As we shall see, the trick to achieving this is to provide control over implicit elements without losing the transparency that high-level interfaces offer.

Starting with this idea of dual interfaces, one for functionality and one for control, π makes special provisions for change at the architecture level. Event management that allows flexible binding between objects and controlled evolution of interfaces through change propagation ensure its goals.

Making the implementation of a system software component controllable is the first step toward making it adaptive. A monitoring element can be added which assesses the effect of modifications and decides what further changes to try. However, even though the concept of adaptability is compelling, it requires a full understanding of mechanisms for controllability.

3: Views of distributed shared memory

In the traditional form of distributed shared memory, the data is seen by an application program as bytes in a region of memory. It acts essentially like physically shared memory in that a process can alter its contents and an underlying DSM support system uses virtual memory techniques to preserve coherency between replicas.

Traditionally, shared memory is viewed as a range of addresses which can be read or written. Alternatively, we can model it as an object with read and write operations as shown in Figure 1. We will refer to the read and write operations as the *application interface* to the DSM object.

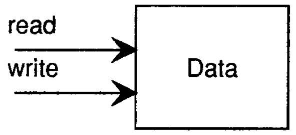

Figure 1 - Basic view of shared data

For classic DSM, issues of coherency and sharing are often hidden from the application. The use of page faults to shuttle the data back and forth is not visible, and the fact that remote processes see the same data is only implicit. Such a view of DSM is appropriate for simple applications when performance is less important than ease of use. However, the lack of DSM's acceptance as a distributed programming paradigm clearly indicates that this view is insufficient.

Of course, the simplistic view does not fully describe the DSM object. The support system maintains replica and coherency information about the shared data which is not normally seen by the application. It also may maintain security information for access control and structural type information for data translation between machine architectures.

Although this *meta-data* is normally used only by the operating system, flexibility will be increased by making it more accessible. For example, a run-time support system could then query the data's structure to do type checking, and a security subsystem could alter the data's security level. The process of making an implicit concept such as meta-data explicit is known as *reification*. As Figure 2 shows, reifying the meta-data expands the inter-

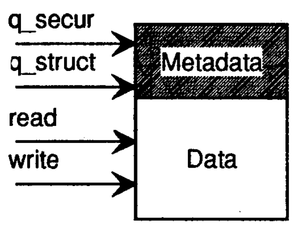

Figure 2 - Metadata-aware view

face to the DSM object. It creates a *meta-object* with what we will call a *system interface*. This does not mean that the interface is available only to system software; the π approach does not distinguish between applications and system software elements, but it does enforce protection domains.

In a similar way, the data's distribution can be made explicit. Suppose, for example, that there are consistency constraints on the data within the DSM object. Therefore, an application would want to perform atomic updates. Figure 3 shows an interface through which an application

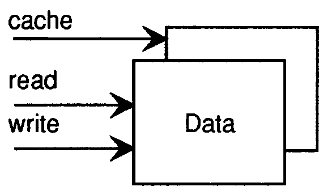

Figure 3 - Distribution-aware view

can ask that its local replica of the data be *locked* while changes are made. It could make the changes and insure that the consistency constraints are satisfied before releasing the lock and allowing the changes to propagate to other replicas. Although this allows the replicas to be temporarily incoherent, it lets the application guarantee that the data will always be consistent.

These two aspects of distributed shared data can be combined into a single system interface as shown in Figure 4. Here, the system interface has inherited the

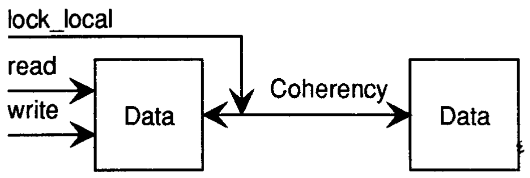

Figure 4 - System interface to shared data

meta-data-aware interface and the distribution-aware interface.

The Munin [5] project has clearly demonstrated the value of tailoring the coherency policy for distributed shared data. It has shown that different kinds of shared data benefit from different policies. Thus, it can be valuable for the application itself to control coherency [3]. For example, an asynchronous algorithm that can use stale data can avoid the delays involved in synchronization. Thus, if the system default coherency control policy is strict, the asynchronous application would benefit by changing to a weak policy. Reifying the coherency control policy as an object with an interface, such as shown in Figure 5, produces a meta-object to control the

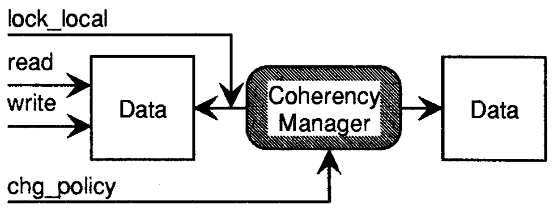

Figure 5 - Metaobject for coherency policy

shared data object.

This example illustrates the π design philosophy. It starts by specifying an interface to the basic abstraction. This interface is adequate for many purposes and presents the classic view of the object. Greater control over the object is provided by reifying its implicit aspects and then manipulating them. This may require the addition of services, a change in the policy which defines the services, a new policy-support mechanism or even a different implementation.

4: The π approach

Traditionally, an application programmer describes his or her application in a programming language which then translates the description into machine operations. π understands that the underlying application can be modeled as a directed graph and that it is the job of system software to map this model to the computational reality. This section shows how π presents this reality to the application.

4.1: Abstract model

Both applications and system software, can be effectively modeled as *dynamic directed graphs*. The nodes of the graph represent *objects* and the arcs are *relationships* between the objects. The objects can be active processes, activatable code or even passive data. The arcs represent a variety of possible relationships, including object references, activation protocols, delegation targets and so forth. The graph is dynamic since new nodes can be created, old ones can be purged and relationships can be changed.

Subgraphs can provide a convenient way of hierarchically structuring the model. They also provide the means for expressing properties associated with an aggregate of objects. Just as the property of persistence can be associated with a node to represent the storage for the object, the property of atomicity could be associated with a subgraph to indicate consistency constraints that span multiple objects. A node which is in fact a subgraph is particularly useful for associating resources with a set of objects.

The reference links in the graph may cross boundaries: Links between active entities may represent cooperation between machines. A data object may have been created by a C++ tool and may be currently accessed by one written in CLOS. A currently-used data object may have been created by a now-deceased program. Numeric data may have been created on a big-endian machine and may be in use by a process on a little-endian machine. Thus, the boundaries can be between machines, languages, time and representation conventions.

While heterogeneity and interoperability are essential to run an application, they only confuse the programmer. He or she should see a clean model of objects and relationships. Clearly, there must be meta-data to support boundary crossings, but it should not be the concern of the programmer. The system software should be able to take care of the details.

The job of operating system is to map the abstract application model onto the computational reality. A language gives the programmer the tools to express the application while the run-time environment and the operating system control the resources. Figure 6 shows

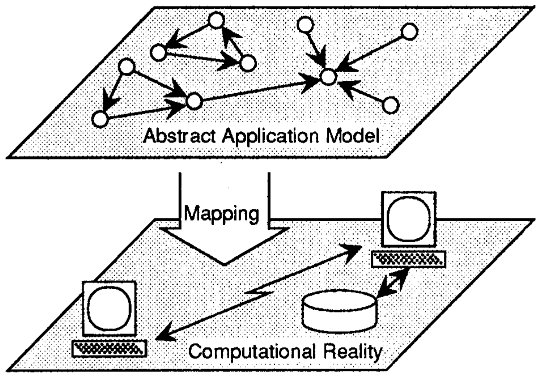

Figure 6 - Mapping model to reality

the relationship between the graph model and the resources. It does not, however, depict the events which make the model dynamic. Events cause state changes in both the operating system and the application.

The mapping should hide a variety of changes in the computational reality. For example, as technology evolves, the implementation should continue to present a consistent model. As the details of a particular implementation change, through variations in load and problem structure, the implementation should adapt.

Finally, the abstractions supported by the model should be *flexible* enough to accommodate the needs of a broad range of applications. The existing mapping of the model onto resources could also influence the realization of an abstraction. An arc in a graph model can be realized in several ways. It could be a name resolved by a name server in the remote case, or a pointer if both the objects are on the same machine. Thus, the underlying system should support multiple realizations, and programmers should be able to convey additional needs.

4.2: π architecture

We have seen that system software maps an abstract application model to the computational reality. The π architecture defines a structure or *framework* for this mapping. It specifies how the elements of this mapping look to each other and how they interact. An operating system based on the π architecture has the capabilities to support applications modeled as graphs.

π removes the traditional division between operating systems and language run-time support. The operating system typically will include high-level data models and language-interoperability tools as well as hardware resource management and transaction support. Any additional support needed by a language is written as π elements and becomes indistinguishable from the base operating system.

π defines a uniform object model for both applications and system software. Its architecture is based on three main concepts:

Resources - elements of the computational reality.
Events - the causalities of computation.
Interfaces - structured presentation of resources and events.

π specifies how these three ideas are combined into a flexible object-based framework.

4.3: Resources

The computational reality consists of three basic types of *resources*: computational, communication and storage. π requires that they be encapsulated into *resource objects* which hide the physical reality and conform to a software model. In the same way, abstract resources like locks and semaphores are also presented as resource objects. The π architecture defines the rules by which various resource objects can be combined to form other resources. Thus, the π architecture does not assume a monolithic operating system structure; indeed, there may not even be a micro-kernel structure. *All* π objects are peers, including those built by application programmers.

4.4: Events

Resource objects alone form a static model; their interactions make the model dynamic. These interactions are called *events* and include interrupts, method invocations and exceptions. Indeed, π considers any *important* state change an event. The implementor of each resource object determines what constitutes an important state change. π defines events as *typed entities.* Each event has three parts: the *generation* when the event occurs, the *notification* by which other objects learn of the event and the *handling* step where its effects are generated. Each resource object clearly specifies in its interface the events it may generate and the ones it can handle. The notification concept allows π to formally decouple event generation and event handling.

As an example, consider an application which wants to open a file. The application object generates a file open event; a function call or RPC serves as the notification mechanism, and a file system object handles the request. The latter two steps may actually consist of a number of steps, each of which involves event generation, notification and handling.

4.5: Interfaces

Application programmers are not concerned with raw resources and low-level state changes. Rather, they are interested in an application-oriented view seen through *interfaces.* Interfaces present the functionality of each object. They do not indicate *how* the object implements the functionality, just what it is and how to use it.

4.6: π Environment

A realization of these objects is based on an *environment* which is the glue that binds them together. The environment allows an application to tailor subsystems like the file system or the communication subsystem. It provides services related to object instantiation, inter-object communication, change propagation etc. In addition to compile/setup time and start-up time changes, the environment allows dynamic changes.

We have argued for a flexible operating system architecture. π assures flexibility by incorporating three features. Unlike classical objects where an event generator must specify the handling object, the *generalized object model* [19] permits the environment to make the decision dynamically. *Meta-computing* [13] [8] allows the manipulation of the reified aspects of a subsystem. As a subsystem is modified, the evolution of related interfaces can be handled through *version set interfaces* [18].

4.7: Generalized objects

The generalized object model promotes the separation of interfaces from their implementations. Rather than requiring the generator of an event to include a target object identifier in a method invocation, it lets the environment dynamically resolve the activation target. The resolution can be based on the parameters and the context of the invocation. For example, a request to draw a circle overlapped by a triangle could be sent to the circle object, the triangle object or an overlapping object. The π architecture *permits* the use of generalized objects, but does not *require* it.

4.8: Meta-computing

Normally, application objects use environmental services without seeing the service provider. For example, an object sending a message to a peer does not "see" the transport mechanism; rather, it is implicit. However, for true flexibility, it must be possible to make these implicit components explicit and to modify them. The former is called *reification* and is necessary for the latter. *Meta-objects* are the reification of normally implicit services and their interfaces allow the services to be changed.

4.9: Version set interfaces

When interfaces change, client objects using the interface are affected. If the new version does not support the services the client expects, system integrity is compromised. In a distributed environment, it may be hard to identify all of the clients without costly global information. π addresses these problems with the version set mechanism. A version set interface is actually a sequence of interfaces tracking the evolution. A client can continue to use an old version even though a new one is presented to new clients.

Since there are multiple versions of an interface, it is not necessary that *all* potential clients know of the latest version. This allows *lazy propagation* of changes. A meta-object associated with the changed object propagates the change information to a set of *acquaintances* [1]. The acquaintances in turn, inform their acquaintances. This way, change information eventually gets to many, if not all, objects. In fact, a change need not be made publicly available and universal propagation of its information may note be required.

Apart from handling modifications, version set interfaces also allow an object to export multiple views through different interfaces. Views are particularly useful in database applications because of their ability to simplify an interface and control access to services provided by the object [17].

5: π-based DSM subsystem design

This section describes the design of a DSM subsystem using the π approach. Figure 7 shows its main

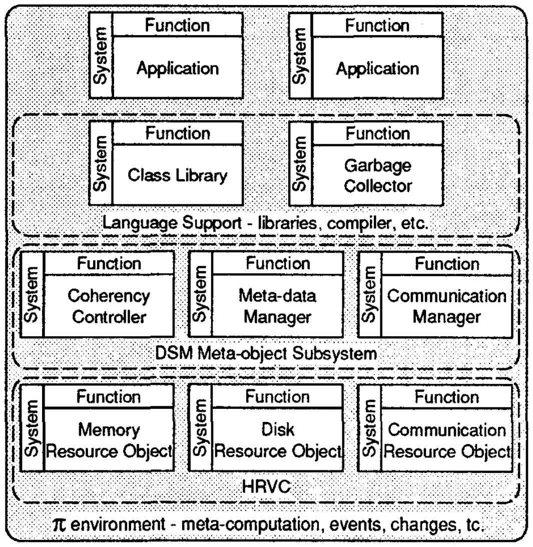

Figure 7 - DSM subsystem implementation

components. Each subcomponent is shown as an object with two interfaces: a *Function* interface that exports the normal functionality and a *System* interface that allows control over the implementation. Although the components in the figure are shown as layers, all objects are actually peers.

5.1: Components

The bottom layer is a Hardware Resource Virtualization Component, or HRVC, that is the effective interface to the system's hardware resources. In the DSM realization, the HRVC provides access to the main memory, virtual memory, disks, communication links and other peripherals on machines in an interconnection. The interconnection may itself be composed of clusters of machines. The organization of the HRVC is flexible; for a cluster with high-bandwidth communication, there may be only one HRVC per cluster; in other cases there may be one HRVC per machine.

The HRVC realizes resource objects and thereby provides a concise interface to other system software components. It supports dual interfaces to present both functionality and control. For storage resources like main memory and disk, its functionality interface provides basic memory-mapped data objects. The second interface is device specific; e.g. the resource object for a multimedia device allow specification of timing constraints. A logical object from an application domain is suitably mapped onto one or more such resource objects. For example, a directory object could be mapped onto multiple main memory and disk-based objects on different machines.

The DSM meta-object subsystem reifies the implementation of DSM. It is a client of the HRVC and serves applications and language support elements by providing services such as allocation/de-allocation of shared objects and coherency control. It has several sub-components:

- **DSM meta-data manager** maintains, updates and can restructure the meta-data for shared data objects.
- **Coherency controller** maintains coherency and can change coherency policy and policy implementation for selected objects.
- **Replication controller** replicates data objects and can change the replication policy and policy implementation for selected objects.
- **Communication manager** implements cooperation between DSM subsystems through the device virtualization layer and can change communication media or implementation specifics.
- **Link manager** manages and updates global data references. It can change the target of references, the nature of information maintained or the reference implementation.

These subcomponents can be replaced or new subcomponents can be added using the π environment.

The π environment, shown as a box enclosing other components, defines the world in which π objects exist. It provides generic services for meta-computation and handles events and change management. For example, a new meta-object installed to enhance the functionality of a DSM subsystem could ask to be notified whenever certain events are generated by other meta-objects constituting the subsystem. Similarly, a client could request information about the changes made to an interface exported by some server.

The Language Support components elevate the services provided by other components to the language level. If the distributed objects supported by DSM are presented as C `structs` or C++ objects of a certain class, they are easier to use [7]. A variety of classes could be bundled with the DSM subsystem as libraries to provide features related to performance, fault tolerance and customized applications. Further control can also be provided through inheritance from these classes which would be translated as appropriate meta-computations on the subsystem. Also included in this component is an IDL compiler similar to that in CORBA [15]. In general, it is this component that defines the API and the general look and feel within the limits imposed by the other components.

5.2: Sample modifications

The components described above allow a wide range of modifications to the DSM subsystem. The modification may involve a change in the implementation, mechanism or the policy behind an abstraction. One lock manager could be replaced with a more efficient one that exports identical functionality. A link abstraction could use a pointer mechanism if the connected objects are in the same address space or it may use additional meta-data if the link goes across machines. An application may chose a garbage collector based on the copy mechanism instead of the in-place mechanism. Such changes require type checking of interfaces for compatibility which is performed by the π environment.

Many meta-computations go beyond module replacements. Additional functionality is often provided through additional modules. These modules need to be notified when certain events occur. Consider for example a DSM implementation based on locking and invalidations. Further assume that the implementation provides the latest version on demand. In such an implementation, whenever a write lock is released, invalidations are sent out to other machines in the interconnection that have stale replicas of the object. To modify the subsystem so that the latest version is piggybacked onto the invalidations, a new object could be introduced into the subsystem to perform the piggybacking. However, such an object needs to know when the invalidations are generated. In π, such an object can request access to the invalidation events generated by the existing lock manager. Such performance improving strategies have been known in the DSM area for a while. However, what remains to be established is how a modification can be accomplished on-line and to what extent multiple strategies can coexist.

5.3: Implementation

The components described above are being implemented on IBM RS/6000s running AIX 3.2. The workstations are connected together to form a cluster by point-to-point serial optical links that provide high bandwidth and reasonable latency. The HRVC is implemented as a kernel extension that can access page faults. It converts the cluster into a shared memory multiprocessor and exports a uniform memory-mapped interface for storage resources within the cluster. A fast datagram protocol has been implemented for intra-cluster communication. The implementation of other components is underway.

6: Looking forward

So far, we have developed the core ideas of the π architecture and are in the process of experimenting with them. The DSM implementation will allow us to tailor an abstraction using reification and meta-computing. In the process, the architecture will be verified and will acquire a more definitive shape.

π is intended to be a framework for component-based, flexible operating systems. Its focus is on integration of components and on modification rather than specific algorithms for subsystems. Existing algorithms and implementation strategies for subsystems can be used in the verification of these two goals. X-kernel communication subsystem [16] and Coda file system for disconnected operation [12] are two such well-developed implementation technologies.

Incrementality and protection are the two main challenges in producing flexible operating systems as discussed in [11]. The π project intends to address these challenges by choosing appropriate granularity for modification and controlling the extent of modifications. Object-level granularity used in π is more suitable than task-level granularity used in microkernels. Change propagation and control often involves multiple address spaces and requires a sophisticated environment that can keep track of the interested recipients of the change and propagate the change to them.

In summary, π architecture includes the following salient features:

- A meta-computing environment to facilitate reification of implicit aspects of subsystems like DSM, file system.
- Event specification through an interface that documents the events generated and handled by the object.
- Support for interface compatibility checks to ensure that the handled events conform to the specification of generated events.
- Interfaces which can evolve and a mechanism to assure the evolution is propagated in a controlled manner.

7: Related work

Work related to the ideas discussed in this paper falls into two categories, DSM which is used as an example subsystem to illustrate π architecture, and meta-computation. The former is a very well researched topic [14]. Several researchers in this area have argued for application specificity [3],[5],[6],[7] and developed the necessary implementation technology.

The other area, that of meta-computation in operating systems is relatively new. The Apertos project at Sony Computer Science Laboratory [21],[22] has done some pioneering work in this area. Objects in Apertos run on virtual processors realized using meta-objects. A change in the virtual processor is effected using invocation of methods on meta-objects in the meta-object hierarchy through reflectors. Reflectors act as entry-points to meta-objects.

Significant work has been done in developing meta-object protocols for languages [10],[13] and in the theoretical aspects of reflection. Emerald [4] has shown the importance of type-conformity in distributed systems.

Even industrial projects are paying increasing attention to flexible interaction between objects and facilities for dynamic modifications. One of the most important is CORBA by OMG [15],[19]. CORBA is a software platform for cooperation between objects in a heterogeneous distributed environment. It is based on the classical object model and uses a subset of C++ for interface specification. A second project viz. IBM's D-SOM [9] aims at inter-language interoperability and provides basic facilities for meta-computing to application programs in OS/2 and AIX. However, these projects do not directly deal with subsystems of operating systems that handle hardware resources. They are oriented towards application-level meta-computing, not managing low-level resources.

References

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, Cambridge, MA: MIT Press, 1987.

[2] S. Ahuja, *et al*, Linda and Friends, *Computer*, 19(8), IEEE, 1986, pp. 26-34.

[3] R. Ananthanarayanan, *et al*, Application Specific Coherence Control for High Performance Distributed Shared Memory, *Proc Symposium on Experiences with Distributed and Multiprocessor Systems*, USENIX, 1992, 109-128.

[4] A. Black, *et al*, Distribution and Abstract Types in Emerald, *IEEE Trans. Software Eng.*, 13(1), Jan. 1987, pp. 65-76.

[5] J. Carter, *et al*, Implementation and Performance of Munin, *Proc 13th ACM Symp on Operating System Principles*, ACM, 1991, pp. 152-164.

[6] D. Cheriton, Problem-oriented Shared Memory Revisited, *Fifth ACM SIGOPS Workshop on Models and Paradigms for Distributed Computing*, ACM, 1992.

[7] D. Cohn, *et al*, Basing Microkernel Abstractions on High-Level Language Models, *Distributed Computing, Practice and Experience, Proc. 1992 OpenForum Tech. Conf.*, Utrecht, 1992, UniForum, Santa Clara, CA, pp. 59-72. Also, Tech. Report 92-2, Dept. of Comp. Sci. & Eng., Univ. of Notre Dame, IN.

[8] J. Ferber, Computational Reflection in Class Based Object Oriented Languages, *Proc OOPSLA '89*, ACM, 1989, pp. 317-326.

[9] IBM *OS/2 2.0 Technical Library, System Object Model Guide and Reference*, IBM, 1991.

[10] G. Kiczales, *et al*, *The Art of the Metaobject Protocol*: MIT Press, 1991.

[11] G. Kiczales, *et al*, A New Model of Abstraction for Operating System Design, *Proc Int'l Workshop on Object Orientation in Operating Systems*, IEEE, 1992.

[12] J. Kistler & M. Satyanarayanan, Disconnected Operation in the Coda File System, *ACM Transactions on Computer Systems*, 10(1), ACM, 1992, pp. 3-25.

[13] P. Maes, *Computational Reflection*, Ph.D. dissertation, Tech Report 87-2: Vrije Universiteit Brussel, 1987.

[14] B. Nitzberg & V. Lo, A Survey of Issues and Algorithms, *Computer*, 24(8), August 1991, IEEE, pp. 52-60.

[15] OMG, *The Common Object Request Broker: Architecture and Specification*, OMG Document No. 91.12.1: Object Management Group, 1991.

[16] L. Peterson, *et al*, The x-kernel: A Platform for Accessing Internet Resources, *Computer*, IEEE, May 1990, pp. 23-35.

[17] J. Shilling & P. Sweeney, Three Steps to Views: Extending the Object-Oriented Paradigm, *Proc OOPSLA '89*, ACM, 1989, pp. 353-361.

[18] A. Skarra & S. Zdonik, The Management of Changing Types in an Object-Oriented Data Base, *Proc OOPSLA '86*, ACM, 1986, pp. 483-495.

[19] R. Soley (ed), *Object Management Architecture Guide*, OMG TC Document 90.9.1: Object Management Group, 1990.

[20] M. Wilkes, The Case for a New Approach to Operating Systems for Personal Computers and Work Stations, *Proc. WWOS-III*, IEEE, 1992, pp. 164-167.

[21] Y. Yokote, *et al*, The Muse Object Architecture: A New Operating System Structuring Concept, *Operating Systems Review*, 25(2), ACM, 1991, pp. 22-46.

[22] Y. Yokote, The Apertos Reflective Operating System: The Concept and Its Implementation, *Proc. OOPSLA '92*, ACM, 1992, pp. 414-434.