π: A New Approach to the Design of Distributed Operating Systems

Dinesh C. Kulkarni, Arindam Banerji, David L. Cohn

Distributed Computing Research Laboratory
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556

Technical Report 92-5

April, 1992
Revised: August 1992

The two page version prepended to the main document will be published in *OOPSLA '92 Addendum to the Proceedings* as a part of the poster session submissions.

π: A New Approach to the Design of Distributed Operating Systems

Dinesh C. Kulkarni, Arindam Banerji and David L. Cohn
Distributed Computing Research Laboratory
University of Notre Dame
Notre Dame, IN 46556, U.S.A.
contact: dlc@cse.nd.edu

A Vision of Computing's Future

Modern computer hardware technology allows a single architecture to be used for a range of machines from supercomputers to toasters. Hardware designers have recognized that an architecture should be *scalable* and system software developers must address the same issue. Scalability means that the architecture must be *flexible* enough to satisfy diverse hardware and application needs. The key to operating system flexibility is to base the architecture on a high-level graph model and to use the generalized object model and metacomputing. The resulting framework is a uniform object model for application and system software.

Classically, operating system flexibility means that the architecture must handle various processor capabilities, differing memory and disk sizes and a vast range of peripheral devices. Today, it also includes dealing with multiprocessor systems, diverse network topologies and multimedia input/output. In the future, as radio-based networking becomes common, changes will be dynamic, requiring the flexible operating system to adapt as elements of a distributed system move into and out of range.

Different classes of applications have different requirements. The valuable data managed by an OODB must be maintained over long periods of time while a multimedia application requires guaranteed bandwidth. The necessary services and the relative emphasis on different functionalities vary drastically.

The π architecture is designed as a framework for flexible operating systems and support software. It is based on this vision of the future and is a first step towards adaptive operating systems. It starts with a high-level model of applications and incorporates facilities for changing abstractions as a fundamental feature.

Graph Model for Applications

We contend that applications 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 may be active, activatable or simple data objects. The arcs may be 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.

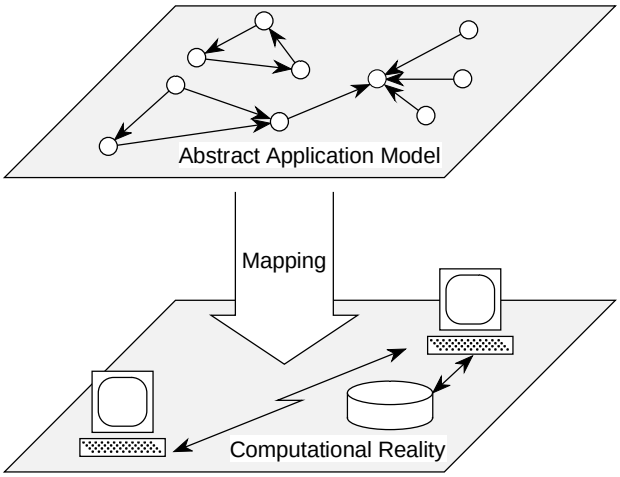
For example, a CAD model of a robot could have a set of data objects containing the descriptions of the robot's components and a set of links that tie these data objects together. The links would represent the "is composed of" relationship. Thus, the robot arm "is composed of" a hinge and a gripper and so on. For another application, an arc connecting two nodes could represent a client-server relationship between two objects.

Events make the graph model dynamic. An event could simply be a method invocation by a client object or it could be an asynchronous activation caused by an interrupt or an exception.

Mapping onto Resources

System software maps abstract application models onto the computational reality as shown in the figure. The programmer first expresses the application through a language and then the run-time environment and operating system control the resources which realize the application. The system software mapping should hide possible changes in the computational reality. Thus, if more processors become available at run time, they should be used them effectively.

The services offered by the abstractions and their implementation must be tailorable. For example, a reference which crosses a machine boundary may use a name server but when its target is on the local machine, it should be possible to use a simple pointer.



A Framework for Mapping - π

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

- 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.

The elements of the computational reality, such as the computational, storage and communication resources, are encapsulated into *resource* objects. Similarly, abstract resources like locks and messages are also presented as resource objects. A programmer can use these objects in various ways. They can be used as is, or several can be combined or additional services can be defined.

Events are typed entities which encapsulate important state changes in the system. They have three aspects: generation, notification and handling. Each object clearly specifies the events that it might generate and the events that it can handle. Notification mechanism allows decoupling of event generation and event handling.

Applications see resource objects and events through interfaces. The interfaces contain the type specification for the services provided by the object and also for the events generated by the object.

These three components are glued together by the π environment. The environment provides services traditionally found in operating systems, language tools and run-time support systems. It is not a rigid structure and applications can alter it by incorporating value-added services. The generalized object model gives the environment flexibility in the resolution of activation targets. Reification of otherwise implicit features supports metacomputation which allows modifications to the environment. Finally, version set interfaces streamline the management of changes caused by metacomputation. The changes are propagated by metaobjects to their *acquaintances*, thereby providing a lazy propagation mechanism suitable for distributed systems.

The flexible features can be illustrated with a sequence of distributed shared data abstractions. At the basic level, the distributed shared data appears to a programmer like a normal region of memory. Any support for distribution, such as replication and coherency control, is invisible. Similarly, any associated metadata, such as type information for translation between architectures, would be hidden. The abstraction is refined if the metadata is reified and made available for modification. A further refinement is to make the application aware that the data is *virtually* shared through coherent replicas. Finally, the coherency manager itself could be reified to allow the programmer to change the coherency policy to improve performance.

Goals

The long term goal of the π project is to develop a framework within which operating systems can be assembled using basic components and their refinements. It focuses on change as the most important aspect and handles it through metacomputation and interfaces.

π: A New Approach to the Design of Distributed Operating Systems

Dinesh C. Kulkarni, Arindam Banerji and David L. Cohn

Distributed Computing Research Laboratory
Computer Science & Engineering
University of Notre Dame
Notre Dame, IN 46556
USA
Contact: dlc@cse.nd.edu

Abstract

Operating systems need to be flexible to meet diverse application needs and to exploit continuously evolving hardware technology. They must present an application-oriented view for ease of use and handle resources effectively for efficiency. Applications can be modeled as dynamic directed graphs that the system software maps onto resources. Events, which generalize method invocations, interrupts and exceptions, contribute to the dynamic nature of the graph. A new operating system architecture, called π is proposed to achieve a flexible mapping. Using the architecture, operating system components can be tailored for specific applications and various hardware platforms. The π architecture specifies consistent service interfaces while metacomputing allows changes to service set implementations and the service set itself.

1. A VISION OF THE FUTURE OF COMPUTING

As the number of computing systems continues to explode, the number of computing architectures does not. The number of computers now is an order of magnitude than a decade ago, but there are probably fewer architectures. Therefore, successful architectures, both for hardware and software, must be *flexible* enough to span a broad range of uses. The same processor that is the heart of a super computer may also run a toaster. Hardware designers have long recognized the need to specify an architecture and then build a family of realizations. System software developers are just beginning to grapple with the same challenge.

1.1 Flexibility in Operating Systems

The implications of flexibility in hardware design are clear: it must be possible to do the "same thing" across a broad cost spectrum. Flexibility was at the heart of IBM's System/360 and many other systems. The implications of flexibility for operating systems is more complex. There is no one-dimensional measure, such as manufacturing cost, that defines a spectrum. Instead, operating systems must cope with a variety of application requirements and hardware configurations. The ability of an operating system architecture to optimize in these situations is a measure of its flexibility.

Different classes of applications have different needs. No single set of primitives will satisfy them all and no single implementation will be best in all cases. For example, when data is shared between remote elements of an application, one of several coherency control mechanisms could be used. The best choice depends on how the application uses the data. However, an application may use the same data differently at different times and hence want to dynamically alter coherency control.

A flexible operating system architecture must accommodate a wide range of hardware structures. Classically, this means various processor speeds, differing memory and disk sizes and sundry peripheral devices. Today it would also include multiprocessor systems, diverse network topologies and a new types of input/output media. In the future, as radio-based networking becomes common, these changes will be dynamic, requiring the operating system to adapt as elements of the system move into and out of range.

1.2 Constraints and Conditions

A major goal of all system software is to relieve the programmer from the burden of solving non-application domain problems. The sequence of tools from programming languages through run-time support to operating systems must make it easy for the programmer to express what is to be done and then to do it.

For simplicity, this discussion will combine run-time support, which is normally associated with a particular language, with operating systems, which are more generic. There is no clear distinction between them and a good operating system architecture should be able to offer whatever run-time tools a language needs. Indeed, one measure of an operating system's flexibility is whether it can provide those tools.

A flexible operating system should scale. That is, the same software architecture should be effective in embedded processors and in large main-frames. This does *not* mean that one

size of operating system fits all machines. Rather it says that a flexible operating system can be tailored to the needs of many different computers.

Scalability and flexibility require modularity. It should be possible to effectively combine system software components from different sources. We see this today in a limited way with installable file systems. It is theoretically possible to replace Microsoft's High Performance File System in OS/2 with IBM's Journaled File System. The idea is extendable to allow increased functionality such as adding distributed shared memory support to a virtual memory subsystem. What is missing is an overall framework for constructing operating systems tailored to specific needs.

1.3 Programmer's View of the Operating System

A programmer creating an application or a new system software component needs clearly defined interfaces. Therefore, the operating system functionality should be defined in terms of interfaces, not implementations. If functionality is presented in terms of an implementation, it is more difficult for the programmer and makes portability problematic. Thus, the interface should be high-level and hide the implementation details. Unfortunately, high-level interfaces can breed inefficiency. It might seem that inefficiency is an inescapable byproduct of flexibility. For example, the Mach micro-kernel is designed to be highly flexible, but it suffers from efficiency problems and is also difficult to use.

A good operating system architecture must permit the fine-grained control that efficiency requires but still present a high-level interface. As we shall see, the trick is to provide control over implicit operating system elements without losing the transparency that high-level interfaces offer.

1.4 Flexibility and Adaptability

System software, including programming languages, run-time support and the operating system, can be viewed as a function which maps a problem from the application domain to the computational reality. Figure 1 shows the classic picture of an operating system as a function.

If the operating system is flexible, there must be a mechanism for changing it. Figure 2 modifies the picture of the operating system function to add this mechanism.

Eventually, once flexibility is well understood, there is another level of operating system design. The operating system itself should be able to monitor what it is doing and, perhaps

with some hints from the application, decide which of the changes to make. This would lead to the *adaptive operating system* as pictured in Figure 3.

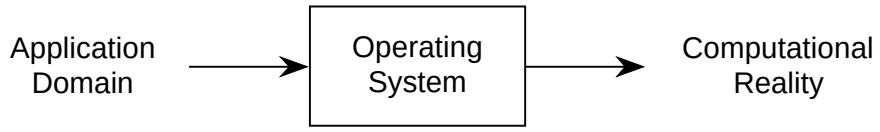


Figure 2 - System Software as a Function

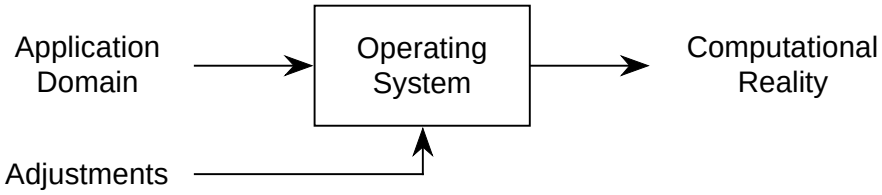


Figure 3 - Flexible Operating System

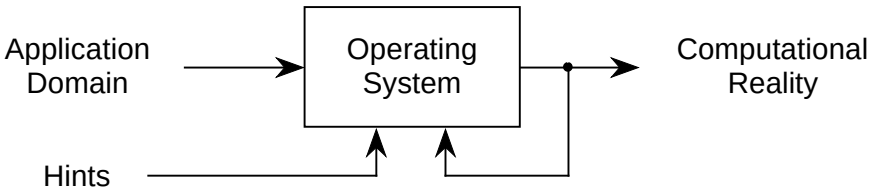


Figure 4 - Adaptive Operating System

However, an adaptive operating system can only be built on top of one which is flexible. To understand what such an operating system should provide to its clients, we begin with a close look at what application and system software programmers really need.

2. AN ABSTRACT MODEL FOR APPLICATIONS

We contend that applications 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, activatable or even in the form of passive data and there are a variety of relationship options. The arcs may be 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. The approach is best explained with an example.

2.1 A VLSI CAD Example

Consider the VLSI circuit simulator example described by Brockman and Director [Brockman, 1992]. The goal is to evaluate the performance of a particular VLSI circuit that has been laid out by some design tool. The circuit includes devices whose characteristics have been determined and it is to be driven by some test input. Thus, simulator object has references to four passive objects: a netlist indicating circuit topology; the device model parameters; a test pattern of stimuli, and the resulting circuit performance. The simulator uses the first three as inputs to produce the fourth as output. The resulting model, including other links to the data objects, is shown in Figure 4.

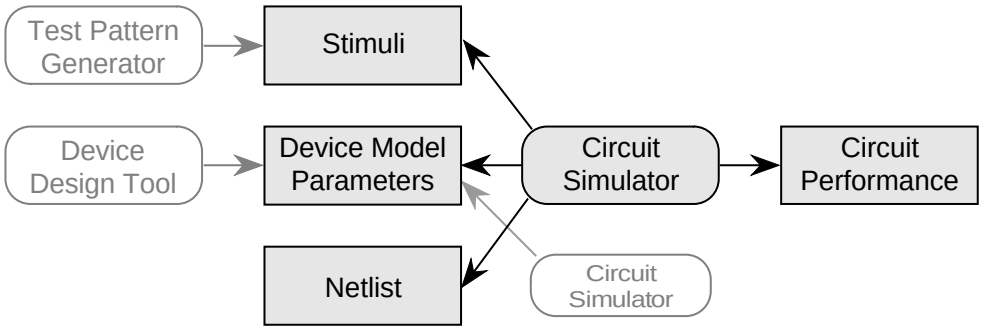


Figure 5 - Graph Model of a VLSI Circuit Simulator

The input data objects are persistent since they are the output of other tools. Once the simulation is run, the result object should also become persistent. However, it can not stand alone. It must retain reference links to the three input objects so applications can know how it was generated. Thus, the saved object looks like that in Figure 5. It is interesting that the

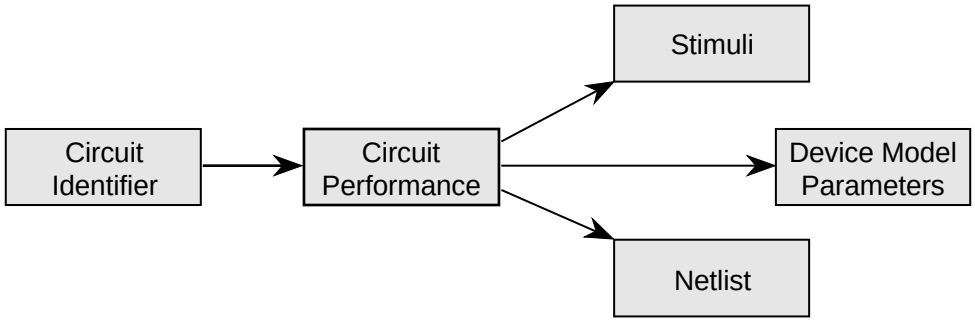


Figure 6 - Persistent Data Object with References

same type of graph model describes the passive persistent object and the active computation.

2.2 Modeling Objects

An object which is an encapsulated entity can be modeled as a node in the graph. Both the data as well as applicable methods are subsumed in the node. Its relationship with other objects is depicted using arcs in the graph. Relationships include but are not limited to paradigms like client-server interaction and peer level cooperation.

A graph may represent a system of active objects, activatable objects, data objects or a combination of all three. Each object in turn may be an aggregate object and hence can be represented as a subgraph itself. For example, a data object representing a CAD model of a robot can be modeled as a directed graph. The CAD model could have a set of data objects containing the descriptions of robot parts and a set of *links*, that tie these data objects together. The links represent the "is composed of" relationship, so that the arm "is composed of" a hinge and a gripper and so on.

The decomposition of a system of objects into these two components is universal, regardless of the application domain. However, the relative mix of nodes and arcs depends on the application. Typically, a matrix computation would require far fewer references than a sophisticated VLSI CAD schematic tool.

Notice that the graph model provides a view that is close to the application domain as in software engineering or programming languages. This is in contrast with hardware oriented view provided by operating systems with abstractions like pages and IPC messages.

2.3 Events

A model graph evolves as a result of events. Events could mark a normal request from a client to a server or unusual conditions like exceptions. They may be handled synchronously as in case of function calls in sequential languages or they could be asynchronous like hardware interrupts. Events may cause creation of new nodes, purging of existing nodes, changes in graph topology or even changes in the composition of a node which may itself be a subgraph.

2.4 Subgraphs

Subgraphs provide convenient way of hierarchical structuring in 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 subgraphs is particularly useful for associating resources with a set of objects.

2.5 Boundaries

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 metadata to support boundary crossing, but it should not be the concern of the programmer. The system software should be able to take care of the details.

2.6 Mapping

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

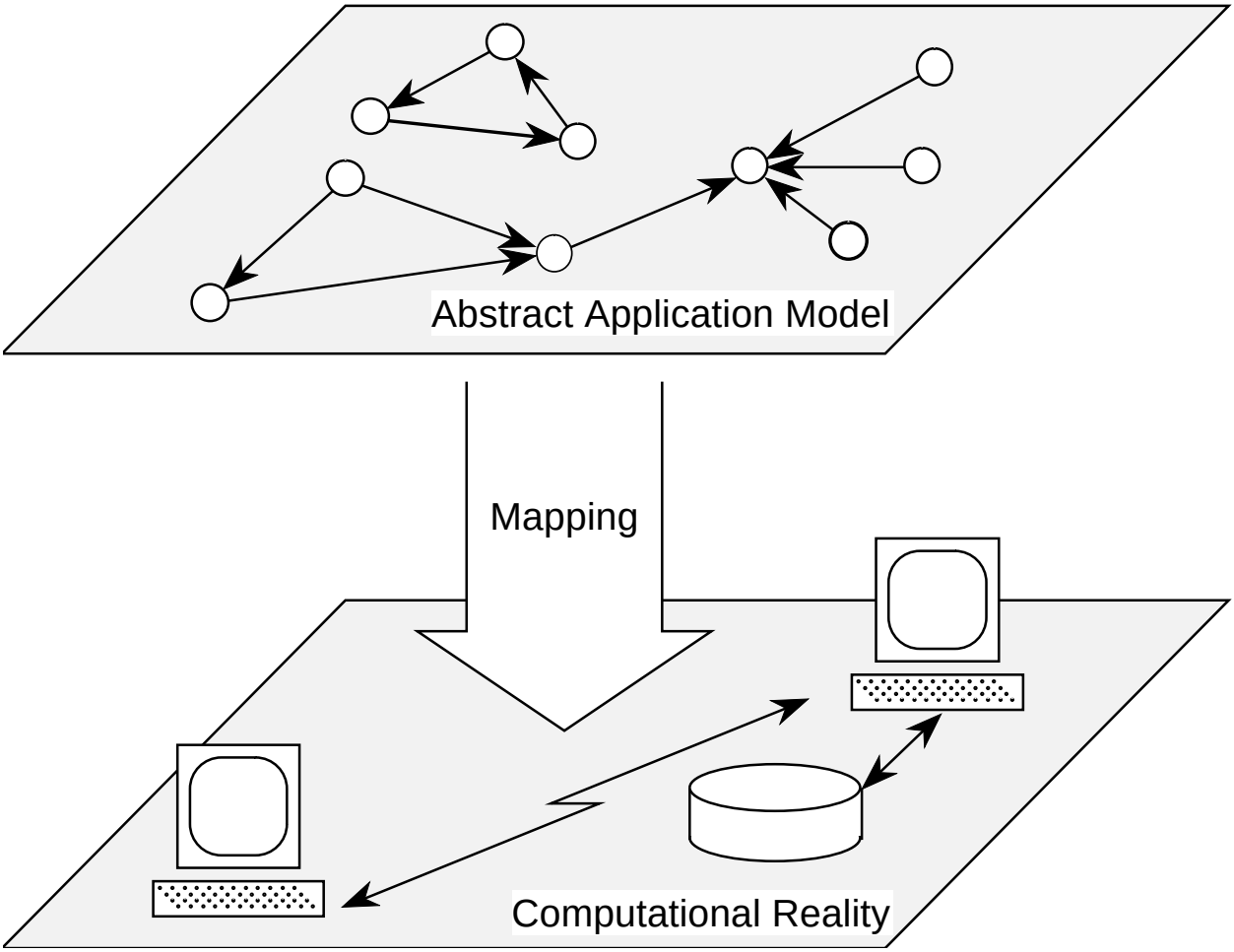


Figure 7 - Mapping Abstract Model to Computational Reality

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. For example, the coherency needs of different applications may vary; some cooperating entities require that atomicity be maintained while others relax coherency constraints for better performance. The existing mapping of the model onto resources could also influence the realization of an abstraction. An arc in a graph model could be realized as a name resolved by a name server if the connected nodes map to objects on different machines while the same arc could be optimally realized as 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 their needs to it.

3. WHAT IS π

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 application modelled as nodes and edges.

The traditional division between operating systems and language run-time support does not exist in a system that conforms to π. 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 notions:

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.

3.1 Resources

The computational reality consists of three basic types of *resources*: computational, communication and storage. These are all hardware resources which must be made available to software. 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 not even be a micro-kernel structure. *All* π objects are peers, including those built by application programmers.

3.2 Events

Resource objects 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*, thereby raising them to a higher level of abstraction. Each event has three parts: the *generation* when the event occurs, the *notification mechanism* by which other objects learn of the event and the *handling* step where its effects are generated. Each resource object clearly specifies the events it may generate and the ones it can handle. The notification mechanism 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, themselves, actually consist of a number of event generation, notification and handling steps.

3.3 Interfaces

Application programmers are not concerned with raw resources and low-level state changes. Rather, they are interested in an application-oriented view which they see through *interfaces.* Interfaces present the functionality of each object. They specify the events that a given object can generate and those it can handle. They do not indicate *how* the object implements the functionality, just what it is and how to use it. Interfaces provide the glue that allows applications to use the resources and events supported by the operating system.

3.4 π Environment

A realization of these notions is based on an *environment* which is the glue that binds them together. The environment provides services traditionally found in operating systems, language tools and run-time support systems. For example, these include inter-object communication, compilation and garbage collection. Eventually, we will see that the elements of the environment could also be objects that adhere to the π architecture. The environment is not a rigid structure and applications can alter it by incorporating value-added services.

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* [OMG, 1991] permits the environment to

make the decision dynamically. *Meta-computing* [Maes, 1987][Ferber, 1989] allows the manipulation of the environment itself. As the environment changes, the interfaces can evolve in a uniform and controlled way by making them *version set interfaces* [Skarra, 1986].

3.5 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. A particular implementation, perhaps for a very small system, may be better served with the classic model.

3.6 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 [Kiczales, 1991]. The former is called *reification* and is necessary for the latter. *Metaobjects* are the reification of normally implicit services and their interfaces allow the services to be changed.

3.7 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 metaobject associated with the changed object propagates the change information to a set of *acquaintances* [Agha, 1987]. 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 in which case, eventual propagation of the change across the system is not 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 [Shilling, 1989].

4. EXAMPLE

Let us consider and example that will illustrate how the π approach generalizes an application's view of and control over a resource. We will look at data that is replicated across multiple machines.

In its simplest form, normally referred to as distributed shared memory, the data is seen as bytes in a region of memory. It acts much like physically shared memory in that other processes can change it and an underlying DSM support system uses virtual memory techniques to handle communication and consistency. The application sees a section of memory within an address range with operations such as read and write as shown in Figure 7. All issues of coherency and sharing are hidden from the application. The use of

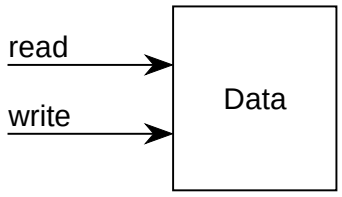


Figure 8 - Basic View of Shared Data

page faults to shuttle the data back and forth is not seen. Such a view may be appropriate for some simple applications when performance is not critical.

Normally, there is information about the data associated with shared data. For example, security information may be used for access control and structural type information may be need to translate the data as it moves between machine architectures. Although this *metadata* is normally used by the operating system, there are application-level operations that are valuable. For example, an application may be allowed to query the structure and alter the security level. Figure 8 shows the application's view when the metadata has been reified as a metaobject with an application-level interface.

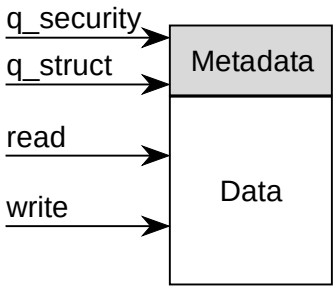


Figure 9 - Metadata-Aware View of Shared Data

In the same way, the distribution subsystem could be made explicit to the application. Consider a situation where there are consistency constraints on the data within the data object. Figure 9 shows an interface through which the application can ask that its local copy

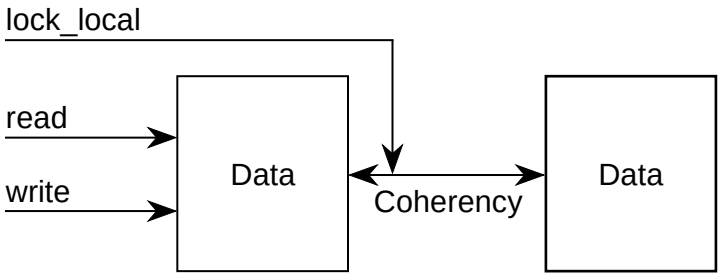


Figure 10 - Distribution-Aware View of Shared Data

of the data be locked while changes are made. Only when the lock is released will changes be propagated to other replicas. Although this allows the replicas to be temporarily incoherent, it lets the application guarantee that the data will be consistent.

These two aspects of distributed shared data can be combined into a single system interface as shown in Figure 10. The system interface has inherited the metadata-aware interface and the distribution-aware interface.

With the system interface on the data object, the application is aware of coherency control and can optimize the particular coherency policy that is provided. However, sometimes even greater control over coherency is valuable [Ananthanarayanan, 1992]. For example, an asynchronous algorithm can use stale data and avoid the delays involved in synchronization. If the default coherency control policy is strict, the asynchronous algorithm would benefit from a change in policy. Reifying coherency control as an object with an interface, such as shown in Figure 11, produces a metaobject for the shared data object.

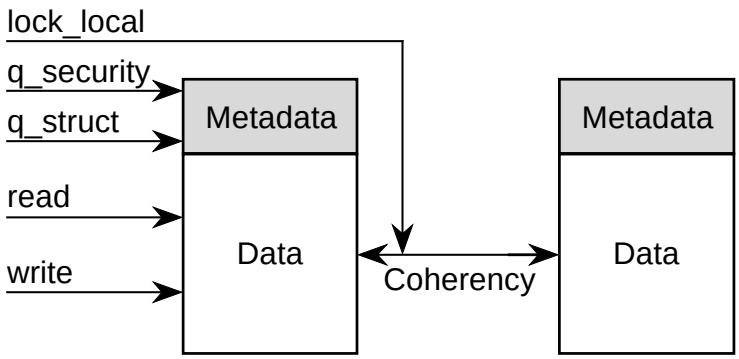


Figure 11 - System Interface to Shared Data

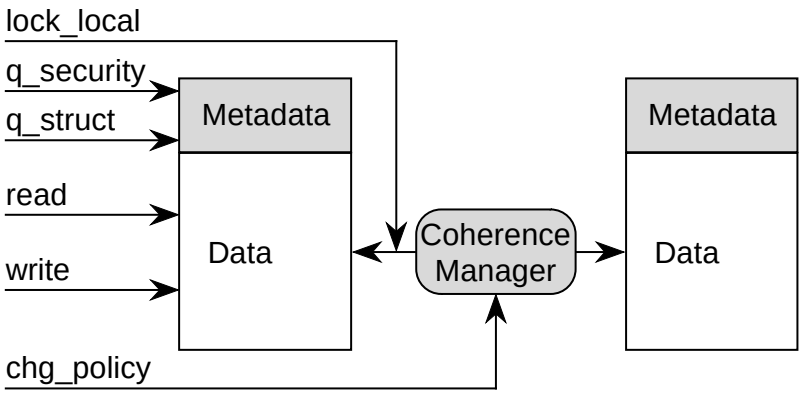


Figure 12 - Metaobject for Coherency Policy

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, or a new policy-support mechanism or even a different implementation.

5. Looking Forward

So far, we have developed the core ideas of the π architecture and are preparing to do some experiments with them. We have designed a prototype of flexible distributed shared data as

described in the foregoing example and are in the process of writing code. This will allow us to experiment with the use of reification and metacomputing to tailor an abstraction. 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. It should also promote the use of new, but more conventional operating system technologies like the X-kernel communication abstractions [Peterson, 1990] and Coda's disconnected file system operation [Kistler, 1992]. Eventually, the development of tailorable operating systems will lead to truly adaptive system software.

In summary, an implementation of the π architecture includes the following salient features:

- A metacomputing environment based on reification of implicit aspects such as address spaces and inter-object communication.
- An interface specification method which allows description of an object's expected and offered services.
- Support for interface compatibility checks which may involve interface repositories and dynamic type checking.
- Interfaces which can evolve and a mechanism to assure the evolution is propagated in a controlled manner.

6. Related Work

Several other ongoing projects are also developing methodologies and tools for interoperability, flexible modification and management of change. One of the most important is CORBA by OMG [Soley, 1990]. 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.

The System Object Model (SOM) product available for OS/2 [IBM, 1991] aims at inter-language interoperability and provides basic facilities for metacomputing to application programs. It is complementary to operating system services such as those in OS/2 and AIX since it deals with application-level metacomputing, not managing low-level resources.

The Muse project at Sony Computer Science Laboratory [Yokote, 1991] is exploring the use of metacomputing in an operating system. Objects in Muse run on virtual processors realized using metaobjects. 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.

7. PROJECT GOALS

The π research project has four long-term goals:

- Develop a framework for true component engineering and code reuse.
- Build efficient implementations of metaobject-based system software.
- Demonstrate lazy change propagation for object versions.
- Take a meaningful step toward truly adaptive operating systems.

REFERENCES

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

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

J. Brockman & S. Director (1992) A Schema-Based Approach to CAD Task Management, *Electronic Design Automation Frameworks*, to appear: Elsevier/ North Holland.

J. Ferber (1989) Computational Reflection in Class Based Object Oriented Languages, *Proc OOPSLA '89*, 317-326: ACM.

IBM (1991) *OS/2 2.0 Technical Library, System Object Model Guide and Reference*: IBM.

G. Kiczales et. al. (1991) *The Art of the Metaobject Protocol*: MIT Press.

J. Kistler & M. Satyanarayanan (1992) Disconnected Operation in the Coda File System, *ACM Transactions on Computer Systems*, Vol 10, No 1, 3-25: ACM.

D. Kulkarni, A. Banerji & D. Cohn (1992) Operating System Support for Cooperation in Distributed OODBs, *Distributed Object Management*, to appear: Morgan Kaufmann.

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

OMG. (1991) *The Common Object Request Broker: Architecture and Specification*, OMG Document No. 91.12.1: Object Management Group.

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

J. Shilling and P. Sweeney (1989) Three Steps to Views: Extending the Object-Oriented Paradigm, *Proc OOPSLA '89*, 353-361: ACM.

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

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

Y. Yokote et. al. (1991) The Muse Object Architecture: A New Operating System Structuring Concept, *Operating Systems Review*, Vol. 25, No. 2: ACM.