Journal of the Nigerian Association of Mathematical Physics
Volume **11 (November 2007), 475 - 484**

Improving access, reliability and efficiency of a distributed operating system using replicated distributed object abstraction

Tola John Odule
Department of Mathematical Sciences,
Olabisi Onabanjo University.
Ago-Iwoye, Ogun State, Nigeria.

Abstract

This paper presents an alternative distributed operating system architecture based on the concept of replication of distributed objects. A complete or partial copy of distributed object's state is placed in each node where the object is used. Replication algorithms ensure copy coherence. For each object the most efficient access algorithm, taking its semantics into account, can be applied. This makes our proposed architecture a convenient platform for developing reliable distributed application.

Keywords*:* Distributed object, replication, semantics object, client/server, distributed shared memory, migrating thread, microkernel.

1.0 Introduction

Distributed operating system is a software platform providing applications with common execution environment within distributed system, including means of access to hardware and software resources of the system and application communication facilities. Such operating system should meet three major requirements: c*onvenient interface, efficiency* and *reliability*.

1.1 Convenient Interface

Due to the nature of distributed systems, it is more difficult for users and software developers to work in them than in centralised ones. Among the complexity factors are: heterogeneity of access to local and remote resources, high probability of faults, asynchronous communication environment and non-uniform memory access. To enable computations in such an environment, the distributed operating system must support a set of abstractions, isolating developers from the listed complexities and providing a convenient interface to all the resources of a distributed system.

This can be achieved by implementing a Single System Image (SSI) abstraction, which is based on abstraction of the *distributed object* that encapsulates state and functionality of all operating system components. Objects are globally accessible by their interfaces from all nodes of a system. Applications are constructed as a collection of distributed objects. Access to the hardware resources, as well as the interaction between software components are reduced to invoking methods on the corresponding objects.

1.2 Efficiency

Operating system efficiency is determined mainly by temporal characteristics of access to various resources. In the distributed environment network latencies become a productivity bottleneck. Therefore, distributed operating system should minimize the influence of remote communication on software operation.

1.2.1 Object replication

Is a generalisation of the preferred implementation approaches in which complete or partial copy of a distributed object's state can be placed in each node where the object is used. The state of an object is synchronised (replicated) among nodes. Its replica in the node handles each invocation of an object method, where the call originates. Communication with the remote replicas is involved only when required by the replication protocol, such as, when it is necessary to obtain a missing part of an object state.

Distributed communication is thus moved inside the distributed object such that efficiency of access to an object is determined by efficiency of the replication strategy. Definitely, there is no single replication strategy, equally effective for all types of objects. Therefore, the use of any specific strategy or a collection of strategies should not be imposed. Instead, services and tools to simplify the construction of replicated objects should be provided. In effect, for each class of objects the most efficient access algorithm, which takes into account its semantics, can be applied. Such algorithm can be either selected from a set of existing replication strategies or designed specifically for the given class of objects.

1.3 Reliability

Support for reliable distributed application development can be provided through replication and persistence. Replication can appear not only as a means of efficient access to an object but also as a redundancy mechanism. Providing consistent copies of an object in n different nodes, for instance, makes it possible to tolerate up to n-1 node crashes [1]. Thus, replication utilises hardware redundancy of the distributed system to provide reliable execution of applications.

Persistence is the ability of the objects to exist for unlimited time, irrespective of whether a system functions continuously, by keeping a copy of it in non-volatile storage and synchronising it with an active copy. The stored object state is thus always correct, even in the face of hardware failures.

Another notable architectural concept is component modelling [2], which is based on the idea of constructing software systems from prefabricated reusable components. Components should be independently deployable by a third party, which is not engaged in design and implementation of the given component. The component model specifies the environment in which components operate, including protected method invocation mechanism, naming service, late binding support, garbage collection service, component development tools, as well as a number of additional services like persistence, transactions, replication and object trading [3,4,5,6,7].

It is hereby noted that implementation of a distributed component model at the operating system level has potential advantages over the middleware approach. The designer of a component-oriented middleware inevitably arrives at the implementation of some virtual machine over the operating system abstractions, which, naturally, results in significantly reduced performance. In order to get rid of this overhead, we suggest that distributed component model support should be initially designed into operating system based on the abstraction of replicated object.

On the low level, the distributed component model should rely on the execution primitives, which are essentially different from the ones used by the conventional operating systems: process or task. This execution abstraction does not appropriately support interacting objects of medium granularity [8]. Therefore, we propose a new execution model, tailored to suit component-based systems.

In our proposal, all executable code and data belong to objects. All objects reside within a single 64-bit address space. Also, there is support for the migrating threads model [8], in which execution of a thread, invoking an object method is transferred to the context of the invoked object. This allows the departure from a server-style object design, where an object runs one or several threads to process incoming method invocation.

2.0 Comparative overview of distributed operating systems

Modern distributed operating systems can be classified into two categories, based on the method of access to distributed system resources: *client/server* systems and *distributed shared memory* (DSM)-based systems. Our model proposes a third approach, based on replicated objects.

In client/server operating systems, all resources of the distributed system are represented by objects, which are uniformly accessible from all nodes. However, objects are not physically distributed. Each object is located in one of system nodes under control of a server process. Global availability of objects is provided by the remote method invocation mechanism, which hides the distributed nature of interactions from the client [9].

This architecture is relatively simple; however, it does not provide a locality of access to resources and, therefore, does not eliminate the influence of network latencies on the performance of the system. Also, it lacks reliability mechanisms: failure of a single node can cause a wave of software failures all over the system (*domino effect*). Furthermore, client/server architecture lacks scalability, as it does not support load balancing among nodes.

The main idea underlying DSM-based operating system [10] is to emulate common memory in the distributed environment. The state and executable code of each object are globally accessible from each node by their virtual addresses. On the first access to an object, the operating system creates local copies of its pages. The copies are synchronised using memory coherence algorithms [11], which can be thought of as universal replication strategies applicable to any types of objects. Unfortunately, they often fail to provide acceptable efficiency of access. To implement efficient access to an object the replication algorithm should take into account its semantics. Algorithms working on the level of virtual memory pages are obviously unaware of object semantics. Thus, there is a natural trade-off between generality and efficiency of the replication strategy.

In summary, both client/server and DSM-based systems use remote methods invocation and memory coherence algorithms correspondingly. These methods have limited efficiency, as they do not take into account semantics of a particular object.

In our proposed model, selecting replication strategy on the basis of the object's semantics ensures the efficient and reliable access to each object. Of course, there is no need to design special replication protocols for each class of objects. An extensible library of replication strategies can be included from, which one can select efficient strategy for virtually any type of objects. In addition, other replication strategies applicable in our model are client/server replication and memory object replication. These strategies reproduce the types of access to distributed objects used respectively by client/server and DSM-based operating systems; thus making our model a generalisation of these architectures.

3.0 Distributed object architecture

3.1 Preliminaries

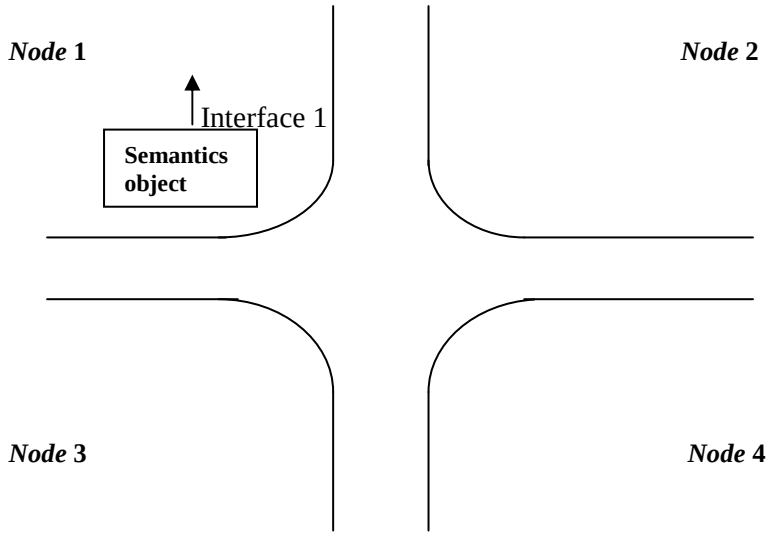
All operating system services as well as application software are constructed from distributed objects, which reside in a single virtual 64-bit address space. Each object exposes one or several interfaces, identified by its unique 64-bit address, consisting of a set of methods. Any object, knowing this address, can invoke methods of the interface from any network node. Objects can be physically distributed by keeping partial or complete copies of the state in several nodes. The copy of an object's state in one of the system nodes is called distributed object *replica*. The distribution of the state among replicas and replica synchronisation is called object replication.

Distributed object architecture aims to separate an object's semantics and replication strategy. An object developer implements only the object's semantics or functionality in local (non-replicated) cases, while a replication strategy supplier implements the replication algorithm. Replication strategy can be universal i.e., applicable to objects of various classes. Also, objects of the same class can be replicated using different strategies. To achieve this goal, we propose distributed object architecture similar to Globe [12], shown in Figure 1.

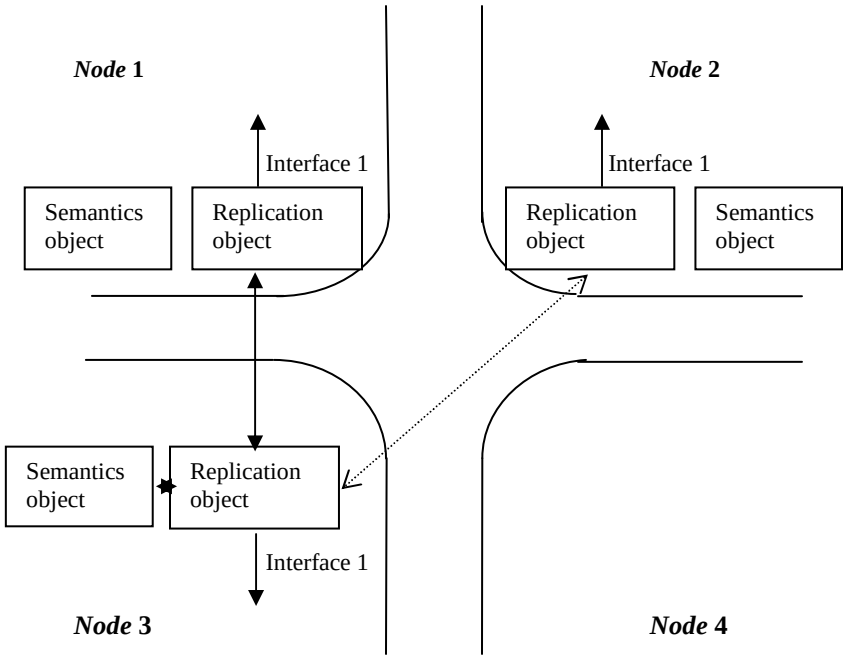
In this architecture, distributed objects are composed of *local objects*. A local object, *semantics object*, is limited to one node of the distributed system and consists of a fixed-size section containing data members and pointers to interfaces (method tables) and the data structures dynamically allocated by the object from the heap. Semantics object contains the distributed object state, exposes the distributed object interfaces and implements its functionality.

To ensure global accessibility of the distributed object interface by their virtual addresses, semantics objects are placed to the same virtual memory location in all nodes. Replication objects, complementing the semantics objects in each node, maintain the distributed object integrity. Replication object implements the distributed object replication protocol and substitutes implementations of semantics object interfaces by its own implementations, which allows it to process the distributed object method invocations.

While processing the invocation, it can refer to the semantics object to execute necessary operations over the local object state, as well as communicate with remote replication objects to perform synchronisation and remote execution of operations. Interface substitution is transparent for other objects and can be thought of as aggregation of the semantics object by the replication object. Such architecture eliminates the overhead of supporting replication objects for the distributed objects that have only one replica. This architecture effectively separates the object's semantics and replication strategy and does not impose any essential limitations on replication algorithms used. Hence, for each object the access protocol providing high efficiency, while preserving required reliability guarantees can be applied.



(a) Distributed Object with one replica



(b) Distributed object with several replicas

Figure 1: The Distributed Object Architecture

3.2 Class Objects

Objects of the special type *class objects* describe classes of local objects in our model presentation. Encapsulation of class properties by objects allows implementing dynamic class loading. Class objects stores interface implementations and exposes methods for creating and destroying instances of the given class. Classes are stored in a single system-wide *class Repository*, which guarantees the use of coherent versions of class objects in different nodes. Before creating class instances, a corresponding class object must be loaded from Repository to memory. There is no concept of distributed object classes in this model; instead, the class of its semantics object can identify a distributed object, since it is the semantics object that encapsulates the distributed object's functionality.

3.3 Model Architecture

Figure 2 shows a generalised architecture of our model. It consists of a microkernel and a set of distributed objects acting at the user level. The microkernel supports a minimal set of primitives that are necessary for operating system construction such as address spaces, threads, inter-process communication (IPC) and interrupts dispatching. Objects implement all operating system and application functionality. Microkernel-based design offers a number of advantages. One, it is potentially more reliable than conventional monolithic architecture as it allows the major part of the operating functionality to be moved beyond the privileged kernel.

Two, microkernel implements a flexible set of primitives, providing a high level of hardware abstraction, while imposing little or no limitations on operating system architecture. Therefore, building an operating system on top of an existing microkernel is significantly easier than developing from scratch. Besides, since operating system services run at user level rather than inside the microkernel, it is possible to replicate or update certain services at run-time, or even start several versions of a service simultaneously.

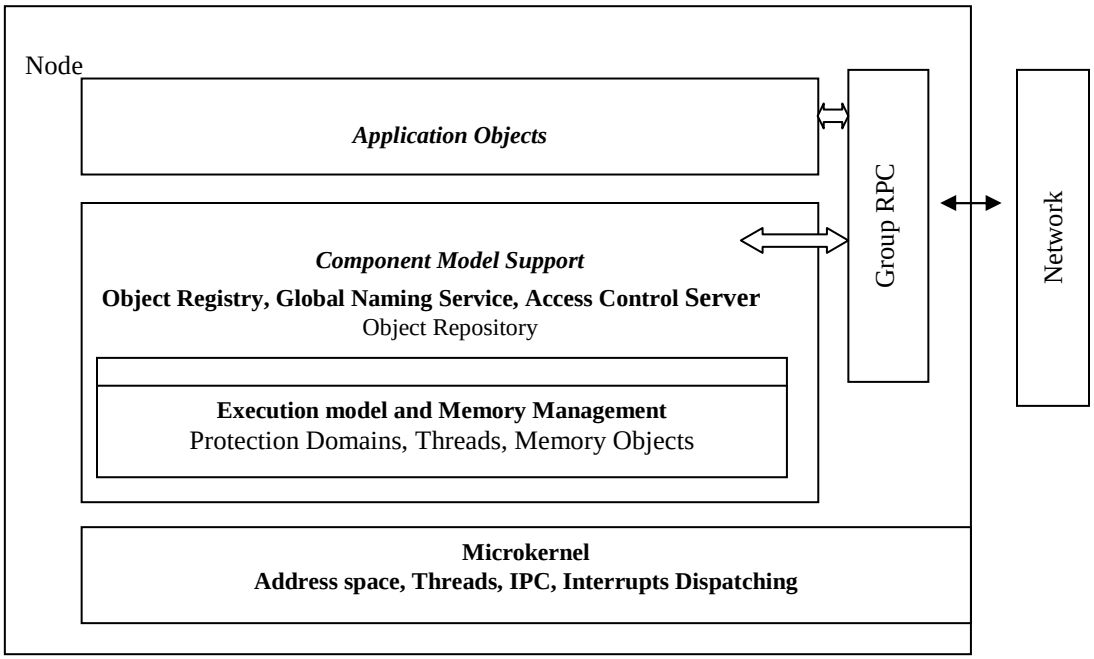


Figure 2: Generalised Distributed Object Architecture

Finally, some of the existing microkernels achieve an IPC performance an order of magnitude over monolithic kernels [13]. Among these are microkernels of the L4 family [14,10,15,16].

4.0 Object interaction and protection

Objects in our model interact via method call. This type of communication is synchronous. Each call is accompanied by a set of input and output parameters, specified by the object developer by means of *Interface Definition Language* (IDL). The local replica of the invoked object executes all method calls. In order to guarantee that such a replica will exist and will not be destroyed by the garbage collection system, a reference is created on an object before using any of its methods. Object methods are invoked through a pointer to one of its interfaces, and, since all objects in our model are located in a single address space, this pointer is valid in any system node and in any protection domain. Within the domain boundaries, method calls arguments are placed in stack and registers, and control is transferred to the address specified in the method table of an invoked object.

The protection model of the architecture presented in section 3.3 above is based on three assumptions:

1) Object methods have no immediate access to the internal state of other objects
2) Objects can interact only by method calls
3) Method calls are monitored by the operating system, which validates each call within effective access control policy.

The following three respective mechanisms: protection domains, crossdomain calls and access control mechanism provide for these assumptions and form the basis of our discussion in the following sections.

4.1 Protection Domains

The protection model of our architecture is based on object isolation requirement; this means the object's state is not directly accessible to other objects. A single address space spans the whole distributed system; hence, all objects in the system are accessible by their unique virtual addresses from any network node as in [4,17].

In order to provide effective object isolation, we introduce the notion of protection domain. Protection domain represents a part of a single virtual address space, containing one or several distributed

objects. Each object belongs to exactly one domain. Associated with each domain is a separate protection context, isolating internal domain objects from the other objects in the system. However, objects inside domain are not protected from each other and intradomain method invocations do not require the protection context switch. While arranging objects in domains, we consider the following factors:

- Placing objects in different domains protects them from accidental or deliberate attempts of unauthorised access;
- Method invocations within domain are more efficient than crossdomain calls;
- Objects use physical memory more efficiently inside a common domain than when placed in separate domains.

Due to the above conditions, intensively communicating objects, which jointly implement some functionality, should be placed in common domain. Domains provide global isolation of objects within the framework of a distributed system. If the object has several replicas, then in every node its replica resides in the same domain and at the same memory address. Thus, if two objects are isolated from each other, then their replicas will be placed in different domains in all nodes. Like other primitives, domains are distributed objects. A replica of each domain is placed in each node, where there is a replica of at least one object, belonging to this domain.

4.2 Crossdomain Calls

Implementation of crossdomain calls is more complicated, although for the interacting objects the difference is transparent. An attempt to access an object outside the local domain triggers a page fault exception, handled by *Crossdomain Adapter* (CA), located in the same domain as the object where the exception occurs. The CA prepares the stack, containing the invocation arguments, which will be mapped into the target domain and on which the method will be executed. All arguments both passed by value and by reference, are copied directly to the new stack. To avoid the creation of a separate stack segment for each crossdomain call, CA uses the stack that the calling thread was running on before the call. The top of the stack is aligned to page boundary and the resulting address is interpreted as the bottom of a new stack, as shown in Figure 3, which is then passed to a target domain, so that the content of the calling object's stack is not accessible to the called object.

Having created the call stack, the CA transfers control to the microkernel, which then refers to the *Object Registry* for the validation of the caller's capabilities to invoke the given operation, and finally maps the call stack to the target domain and transfers control to the called object to complete the call.

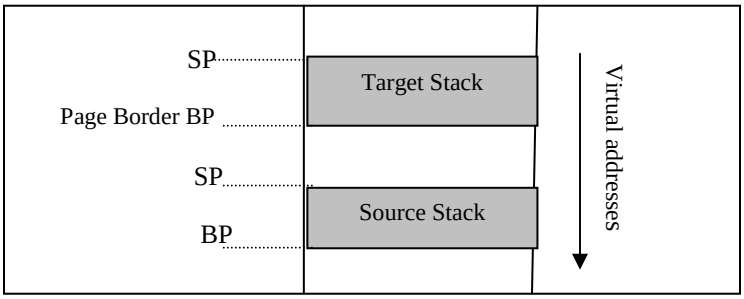


Figure 3: Stack Management during Crossdomain call

4.3 Crossdomain call validation

At the time of crossdomain call the microkernel refers to the Object Registry through an *AccessValidate* interface to assure the existence of the invoked object's replica in a local node, and also to validate the caller's rights to perform the given operation. The Registry verifies call legitimacy through the *Access Control Server* (ACS). Information on objects and rights is cached by the microkernel to avoid having to look up the Registry for each crossdomain call, as shown in Figure 4, in order to improve efficiency of crossdomain communication.

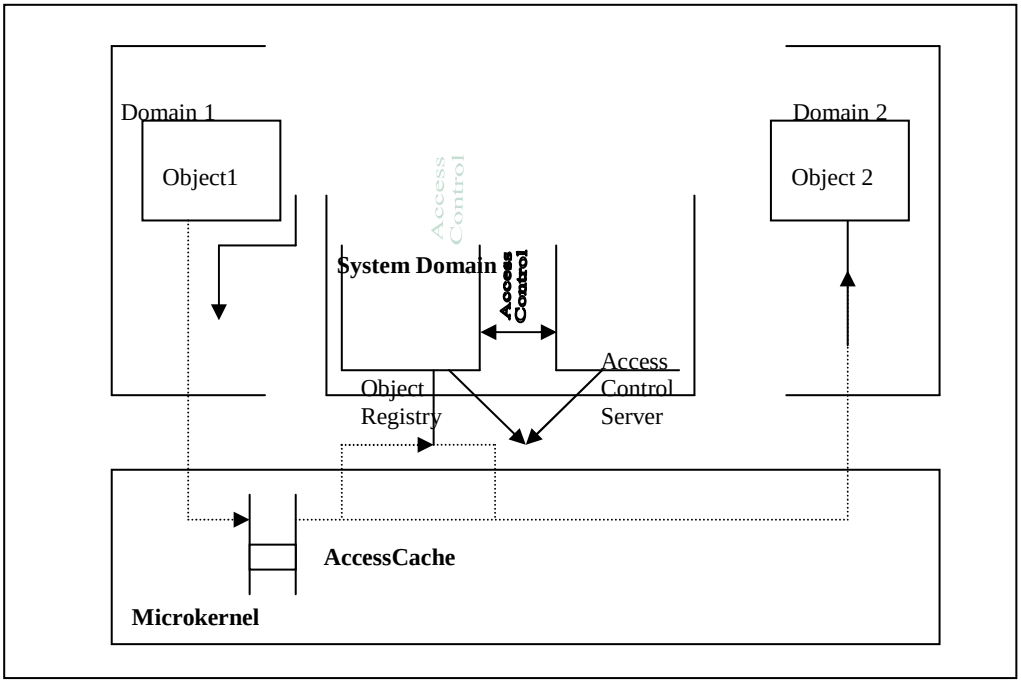


Figure 4: Communication between Microkernel and Object Registry during Crossdomain call. (*Dashed line shows optimised crossdomain call path*)

4.4 Access Control Server

Access Control Server (ACS) is a distributed object, which enforces a single access control policy across the distributed system by verifying the legitimacy of each call. The ACS implements the *IaccessControl* interface, used by the Object Registry for crossdomain call validation. The main method of the *IaccessControl* interface, namely *ValidateAccess,* confirms or denies the validity of a call, based on the thread identifier, caller and callee identities and the invoked method. ACS can also expose additional interfaces, depending on the particular access control model it implements. Global fulfilment of access control rules is provided by the ACS replication strategy. For instance, on capability revocation, corresponding notification must be delivered to all ACS replicas, which contain outdated information. The overhead introduced by ACS replication is one of the important factors to be considered when selecting an access control model.

4.5 Basic Execution model

The execution model of our architecture is based on the migrating threads concept. At any point in time each thread runs in the context of a specific object. During method invocation, execution of a thread is transferred to the target object. Thus, the thread is not permanently bound to any specific object or domain. This execution model eliminates the need to start a separate thread for processing each call or queuing calls for sequential processing. This results in increased efficiency of object interaction as well as simpler and more lightweight object architecture. Since in our model architecture distributed object invocation is actually an invocation of its local replica, it does not cause the transfer of thread execution to a remote node, except when the replication strategy requires migration of object replica between network nodes*. After completing execution within the migrated object replica, threads return to their home nodes.

Associated with each thread is an *activation stack*, which describes the sequence of nested calls, both intradomain and crossdomain, performed by the given thread. Each element of the activation stack stores the address of the object, which performed the invocation. For crossdomain calls, the activation stack also stores the processor context. This information allows the thread to correctly return from method

invocations. In addition, by placing special instructions to the elements of the activation stack, the operating system can control the thread's behaviour, for example, suspend it, transfer to remote node or terminate. Execution of these instructions is deferred until the thread returns from method invocation, having finished all possible modifications of an object's state.

5.0 Conclusion

In our model, the abstraction of the replicated distributed object is used as a building block for both operating system components and application software. Since distributed object's interfaces are globally uniformly accessible across the network, the distributed nature of the system is hidden from application developers and users. Selecting replication strategy for each object on the basis of its semantics allows achieving efficient access, while providing the required degree of reliability. The internal architecture of the distributed object effectively separates its semantics and replication algorithm, which actually reduces the task of distributed object development to the development of a local, non-replicated, object.

Our design runs on top of a microkernel, which supports a minimal set of primitives like address spaces, threads, IPC and interrupt dispatching. Distributed objects implement all operating system and application functionality. We believe that microkernel-based architecture improves modularity and reliability of the system, as well as reduces control transfer costs via the kernel, which is especially important for the system oriented at intensive communication of medium grained objects.

The execution model of our architecture is based on the migrating threads concept. This means a thread is not permanently bound to any specific object or domain, but transfers execution between objects on method calls. The migrating threads model simplifies object development, results in more lightweight objects and improves the efficiency of object communication.

References

[1] Schneider F. (1990), Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM computing Surveys* 22(4) pp.290-319.

[2] Microsoft and Digital Equipment Corporation (1995). The Component Object Model Specification. version 0.9.

[3] Object Management Group (1998). CORBAservices: Common Object Services Specification.

[4] Chase J.S. (1995). An Operating System Structure for Wide-Address Architecture. PhD Thesis. Department of Computer Science and Engineering, University of Washington.

[5] Dasgupta R.C. et al (1990). The Design and Implementation of the Clouds Distributed Operating System. *Computing Systems Journal*, Vol. 3, *USENIX*.

[6] Elphinstone K., Russell S., Heiser G., (1996). Supporting Persistent Object Systems in a Single Address Space. Technical Report 9601. School of Computer Science and Engineering, The University of New South Wales, Sydney.

[7] Skousen A.C., (1994). SOMBRERO: A Very Large Single Address Space Distributed Operating System. Msc. Thesis. Computer Science and Engineering Department, Arizona State University, Arizona.

[8] Ford B., Lepreau J. (1993). Microkernels Should Support Passive Objects. *Proceedings of International Workshop on Object Oriented Operating Systems*.

[9] Rozier M., et al (1998). Chorus Distributed Operating Systems. *Journal of Computing Systems*.

[10] Dechamboux P., et al (1996) The ARIAS Distributed Shared Memory: An Overview. *SOFEM Seminar*, Prague.

[11] Li K., (1986). Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD Thesis, Yale University, Yale.

[12] Van Steen M., Homburg P., Tanenbaum A.S. (1999). Globe: A Wide-Area Distributed System. *Proceedings of IEEE Conference on Concurrency*. pp. 70-78.

[13] Liedtke J. et al (1995). Achieved IPC Performance (still the foundation for extensibility). *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pp.28-31. Chatham, Cape Cod, MA.

[14] Liedtke J. (1996). L4 Reference Manual (486, Pentium, Pro). Research Report RC 20549. IBM T. J. Watson Research Centre, Yorktown Heights, NY.

[15] Potts D., Winwood S., Hediser G., (2001). L4 Reference Manual: Alpha 21x64. Technical Report UNSW-CSE-TR-0104. University of New South Wales, Sydney.

[16] The L4Ka team (2002). L4 Experimental Kernel Reference Manual, version X.2.

[17] Heiser G. et al (1998). The Mungi single-address-space operating system. *Journal of Software Practice and Experience*, 28(9).

*For example the port of a traditional UNIX program. Its main () method is called right after the object is created and executes until the object is destroyed. Such an object must be moved to a remote node, along with all the threads that are executing within it.