УДК 681.142.2

ARCHITECTURAL DESIGN OF E1 DISTRIBUTED OPERATING SYSTEM

L.B. RYZHYK, A.Y. BURTSEV

This paper presents the 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. Copy coherence is ensured by replication algorithms. For each object the most efficient access algorithm, taking its semantics into account, can be applied. All E1 subsystems are designed to support replication, which makes E1 a convenient platform for developing reliable distributed applications.

INTRODUCTION

In modern operating systems distributed computations support is usually limited to network protocol stack. However, construction of distributed applications requires more advanced communication facilities such as remote procedure calls, distributed synchronization primitives and distributed shared memory. The growing complexity of software systems necessitates a new software layer, providing developers with efficient, reliable and secure access to network resources.

Currently, this layer is most frequently implemented by middleware systems. Middleware is defined as a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system [1, 2]. For example, in distributed data processing systems, component-oriented middleware, which supports the common object model in different network nodes, is widely used [3, 4, 5, 6].

The alternative approach consists of integrating distributed computations support into the operating system. Nowadays, the advanced communication facilities have become an essential software component like file system or inter-process communication facilities. OS-level implementation allows the construction of the most effective architecture, supporting the unified set of primitives for access to local and remote resources.

Distributed OS 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.

This paper presents architectural design of E1 distributed operating system. Such OS should meet three major requirements:

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 centralized ones. Among the complexity factors one can name: heterogeneity of access to local and remote resources, high probability of faults, asynchronous communication environment, non-uniform memory access. To enable computations in such an environment, the distributed OS 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.

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

3. **Reliability.** In the absence of fault tolerance mechanisms, a single node or network connection failure can put the whole distributed system out of order and cause loss of data. Therefore the distributed OS should provide reliable computations support, including redundant storage and execution, as well as fault recovery.

1. E1 CONCEPTS

This section presents our approach to implementation of the above requirements.

Convenient interface

To provide applications with convenient interface to all computer network resources, E1 implements a Single System Image abstraction, which implies that for application software the distributed system looks like a centralized one. This feature allows a developer to ignore the physical layout of resources but instead focus on the functionality they provide.

Implementation of single system image in E1 is based on abstraction of the **distributed object**. Distributed objects encapsulate state and functionality of all OS components. Each object exposes a set of well defined interfaces that can be invoked by other objects. Objects are globally accessible by their interfaces from all nodes of a system.

Both OS components and application software relies on a single E1 object model, i.e. E1 applications are constructed as a collection of distributed objects. To an application programmer the computer network looks and fills like a single virtual computer, its software structured as a set of objects. Access to the hardware resources, as well as the interaction between software components are reduced to invoking methods on the corresponding objects.

Efficiency

The distributed software systems consist of interacting components located in different network nodes. As the operations, performed in each node, often depend on instructions and data received from remote components, the communication latencies eventually affect the performance of the entire system. Two popular techniques, used to overcome this effect are: replacing remote communication by local operations, and removing remote communication beyond the critical execution paths. Replacing remote communication by a local interaction implies that the state of a server object is cached in the client nodes. In this case read operations are performed locally on the cached copy of an object. Modifications can sometimes also be applied locally with the subsequent delayed delivery of changes to a server. Removing the remote communication beyond the critical paths allows the reduction of the time spent by main computational threads waiting for remote messages. For this purpose additional helper threads, that speculatively obtain the data, required by main computations, are used.

Object replication constitutes a generalization of the indicated approaches. In E1 a 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 synchronized (replicated) among nodes. Each invocation of an object method is handled by its replica in the node, where the call originates. Communication with the remote replicas occurs only when required by the replication protocol, for example, when it is necessary to obtain a missing part of an object state.

Thus, the distributed communication in E1 is moved inside the distributed object. Hence, efficiency of access to an object is determined by efficiency of the replication strategy. Obviously, there is no single replication strategy, equally effective for all types of objects. Therefore E1 does not impose the use of any specific strategy or a collection of strategies. Instead, E1 provides services and tools to simplify the construction of replicated objects. 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.

Reliability

E1 provides support for reliable distributed applications development through replication and persistence. Replication can appear not only as a means of efficient access to an object, but also as a redundancy mechanism. For example, by supporting consistent copies of an object in *n* different nodes, it is possible to tolerate up to $n - 1$ node crashes [7]. Thus, replication utilizes hardware redundancy of the distributed system to provide reliable execution of applications.

Persistence is the ability of the objects to exist for unlimited time, irrespectively of whether a system functions continuously. For this purpose a copy of an object is kept in nonvolatile storage and is being synchronized with an active copy. The stored object state is always correct, even in the face of hardware failures. (As for now, support for persistence in E1 is not designed in sufficient details. Therefore, it is not covered in this paper.)

Another important principle underlying the E1 architecture is **component model support**. According to this principle, the replicated objects model is extended to a valid component model. Such architecture makes E1 a convenient platform for the development of distributed applications.

On the low level, the E1 component model relies on the execution primitives, which are essentially different from the ones used by the conventional OS. The primary execution abstraction in the conventional systems is process or task, representing an instance of a program, loaded into memory. Each task runs in a separate address space. Within a task several execution threads can exist. This model does not appropriately support interacting objects of medium granularity [8]. Therefore, we abandon it for the new execution model, tailored for component systems. In E1 all executable code and data belong to objects. All objects reside within a single 64-bit address space. E1 supports the migrating threads model [8], in which execution of a thread, invoking an object method, is transferred to the context of the invoked object. Migrating threads allow the departure from a server-style object design, where an object runs one or several threads to process incoming method invocations.

Since both OS services and application software are developed within the framework of a single E1 component model, the model has to be highly flexible,

while introducing minimal overhead. These requirements have guided the design of E1 component services, presented in the following sections of this paper.

2. COMPARISON WITH OTHER SYSTEMS

Modern distributed operating systems can be divided into two classes, based on the method of access to distributed system resources: client/server systems and distributed shared memory (DSM)-based systems. E1 implements a third approach, based on replicated objects. This section presents a brief characteristic of existing architectures and compares them to the E1 architecture.

In client/server OS'es, similar to E1, 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. Two well-known examples of client/server distributed OS'es are Mach [9] and Chorus [10]. The advantage of client/server architecture is its relative simplicity. 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. Another disadvantage of client/server architecture is the lack of reliability mechanisms. Failure of a single node can cause a wave of software failures all over the system, a phenomenon known as the «domino effect». Furthermore, client/server architecture lacks scalability, as it does not support load balancing among nodes.

DSM-based OS'es [11, 12, 13, 14] use essentially different approach to implement the distributed object model. The main idea underlying these systems 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 OS creates local copies of its pages. The copies are synchronized using memory coherence algorithms [15]. These algorithms can be thought of as universal replication strategies, applicable for 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, we observe a natural trade-off between generality and efficiency of the replication strategy.

In E1, the efficient and reliable access to each object is ensured by selecting replication strategy on the basis of the object's semantics.

Two examples of standard E1 replication strategies 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 OS'es. Thus, E1 can be considered a generalization of these architectures.

3. E1 OVERVIEW

3.1. Distributed object

Distributed objects are first-class citizens in E1. All OS services, as well as application software are constructed from distributed objects.

All objects reside in a single virtual 64-bit address space. Each object exposes one or several interfaces consisting of a set of methods. Each distributed object interface is identified by its unique 64-bit address. Any object, knowing this address, can invoke the interface methods from any network node. All interfaces in E1 contain a standard method of navigation between the interfaces of the same distributed object.

Objects in E1 can be physically distributed, i.e. 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 synchronization is called object **replication.**

The E1 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. At the same time, objects of the same class can be replicated using different strategies.

To achieve the goal above, we put forward the distributed object architecture, in which object semantics and replication strategy are implemented by separate structural units. In E1, distributed objects are composed of **local objects**. A local object is limited to one node of the distributed system. Note that a similar distributed object architecture has been implemented by Globe object-oriented middleware [6].

The E1 local object resembles the structure of a C++ object [16]. It 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 heap. In terms of C++, the interfaces of the local object are purely virtual base classes, from which the object is inherited. A similar approach is taken by COM [4].

The distributed object architecture is shown in Fig. 1.

In a simple case when the distributed object has only one replica, it is identified with a single local object, **semantics object**. Semantics object contains the distributed object state, exposes the distributed object interfaces and implements its functionality.

When the reference to the distributed object is created in the node, where there is no replica of the given object yet, a new replica is created in this node. The structure of distributed object with several replicas is shown in Figure 1. A copy of the semantics object is placed in each node, where the distributed object is represented. To ensure global accessibility of the distributed object interfaces by their virtual addresses, semantics objects are placed to the same virtual memory location in all nodes. The distributed object integrity is maintained by **replication objects,** complementing the semantics objects in each node. Replication objects implement the distributed object replication protocol. Replication object substitutes implementations of semantics object interfaces by its own implementations, which allows it to process the distributed object method invocations. (Detailed discussion of interface substitution technique lies beyond the scope of this paper.) While processing the invocation, replication object 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 synchronization and remote execution of operations. Interface substitution is transparent for other ob-

jects 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 are not actually distributed, i.e. have only one replica. If an object with several replicas eventually remains with only one replica, its replication object is deleted.

The presented distributed object architecture has two important advantages. First of all, it effectively separates the object's semantics and replication strategy. Secondly, it 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.

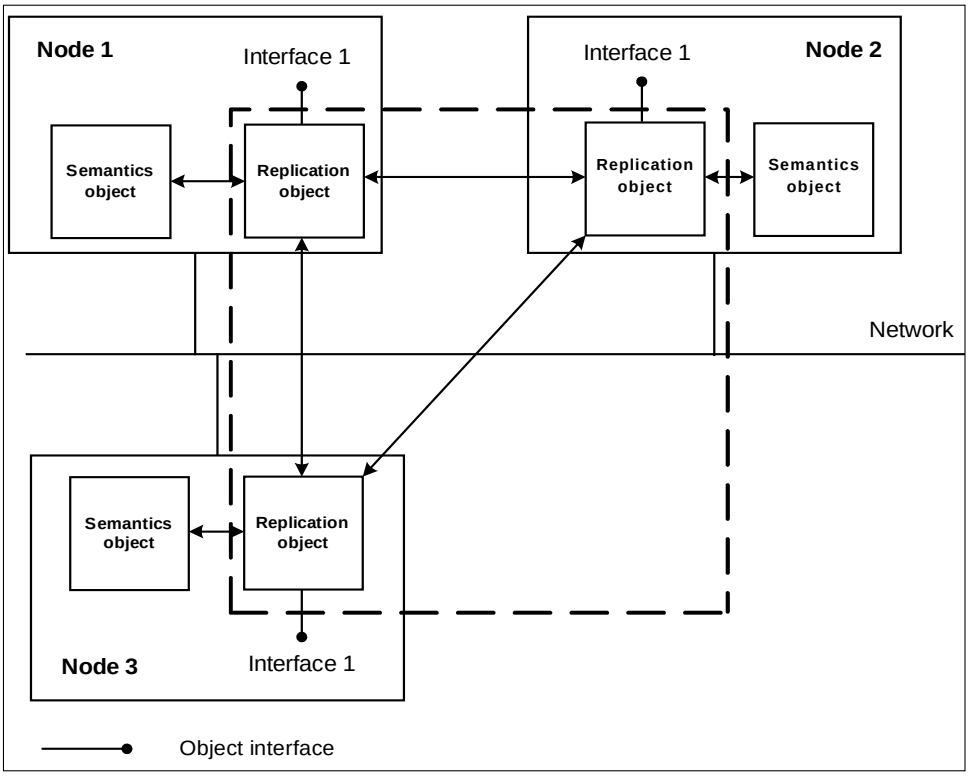

Fig. 1. The distributed object architecture

3.2. E1 architecture

Fig. 2 shows a generalized E1 architecture. E1 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 OS construction, such as: address spaces, threads, IPC and interrupts dispatching. All operating system and application functionality is implemented by objects.

Microkernel-based design has a number of advantages. First, it is potentially more reliable than conventional monolithic architecture, as it allows the major part of operating system functionality to be moved beyond the privileged kernel. Second, 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 kernel, it is possible to replace or update certain services at run-time, or even start several versions of a service simultaneously. Third, and finally, some of the existing microkernels achieve an IPC performance an order of magnitude over monolithic kernels [17]. Among these are microkernels of the L4 family [18, 19, 20, 21]. For object-oriented operating systems, like E1, it is extremely important to minimize the latency of control transfer between address spaces; therefore, L4 has been selected as the microkernel of E1.

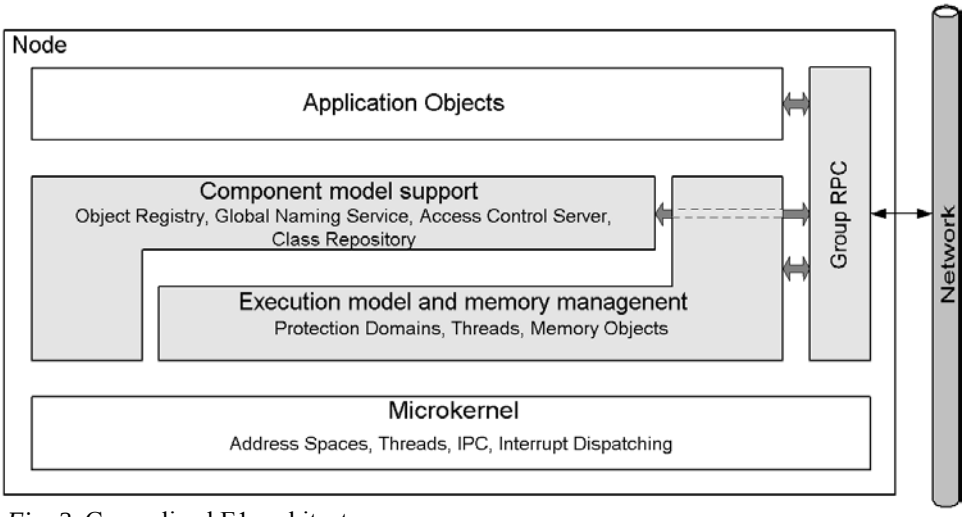

Fig. 2. Generalized E1 architecture

4. OBJECT INTERACTION AND PROTECTION

In order to perform useful operations, objects interact by means of method calls. Safe execution of applications is provided by a protection mechanism which guarantees that object interaction is controlled by precisely defined access control policy.

4.1. Protection Domains

OS protection model has to be based on facilities provided by the underlying hardware platform, primarily, virtual memory mechanisms in modern microprocessors these are. Therefore, object protection is closely related to virtual memory organization.

In E1 all objects reside in a single virtual address space. Object interfaces are invoked directly by their virtual addresses, just as in C++ methods are invoked through the pointer to an object. The major advantage of such a virtual memory organization is a convenient programming model, which greatly simplifies the communication between objects. A single E1 address space spans the whole distributed system. Hence, all objects in the system are accessible by their unique virtual addresses from any network node.

Nevertheless, in order to provide effective object isolation in E1, we introduce the notion of **protection domain**, representing a part of a single virtual address space, within which one or several distributed objects are located. Each object in E1 belongs to exactly one domain. Associated to 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. Method calls inside domain do not require the protection context switch. Thus, domains offer a trade-off between efficiency and safety of interaction.

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. Therefore, if two objects are isolated from each other, i.e. reside in different domains, then their replicas will be placed in different domains in all nodes. Like other E1 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

Objects in E1 interact via method calls. 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).

In E1 all method calls are executed by the local replica of the invoked object. In order to guarantee that such a replica will exist and will not be deleted by the garbage collection system, one must create a reference to an object before using any of its methods.

Object methods are invoked through a pointer to one of its interfaces. Since all objects in E1 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 work very similar to C++ language: arguments are placed in stack and registers, and the control is transferred to the address specified in the method table of an invoked object.

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's task is to prepare the stack, containing the invocation arguments, which will be mapped into the target domain and on which the method will be executed. All arguments are copied directly to the new stack. Although crossdomain call mechanism does not explicitly support passing large data arrays without copying, a similar functionality can be achieved by passing pointers to objects, representing shared memory regions.

Having created the call stack, the CA transfers control to the microkernel to complete the call. The kernel then refers to the Object Registry (see Section 5.1) 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. Return from crossdomain call occurs in a similar way, through the target domain CA.

4.3. Threads

The E1 execution model is based on the migrating threads concept [8]. 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. As shown in

[8], migrating threads are more appropriate for object-oriented environment, than traditional static threads.

Since in E1, distributed object invocation is actually an invocation of its local replica, it does not cause the transfer of thread execution to a remote node. There is, however, one particular situation, when such transfer occurs. It is when the replication strategy requires migration of object replica between network nodes. The object state is then moved to the target node, along with all of its threads. After completing execution within the migrated object replica, threads return to their home nodes.

5. COMPONENT SERVICES

This section describes the E1 services, which extend the distributed object model to a full-featured component model. Among these are Object Registry, Access Control Server, Global Naming Server, and garbage collection system. (Detailed description of dynamic class loading mechanism lies beyond the scope of this paper.)

5.1. Object Registry

Object Registry lies at the heart of the E1 component model. It maintains the information about all local replicas of distributed objects, including their types, virtual addresses, host domain IDs and reference counting information. The Registry coordinates execution of such operations as creation and deletion of the distributed objects and their replicas, crossdomain calls and garbage collection.

Crossdomain calls validation

At the time of crossdomain call, the microkernel refers to the Object Registry through an *IAccessValidator* 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 itself does not implement access control policy. Instead, for the verification of call legitimacy it refers to the Access Control Server, which will be discussed in the next section.

To improve the efficiency of crossdomain communication, information on objects and rights can be cached by the microkernel, which avoids having to look up the Registry for each crossdomain call.

5.2. 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.

Selection of an operating system access control model is very challenging. Having its own limitations and drawbacks, none of the existing protection models can be considered generally optimal. Therefore E1 does not impose any specific access control policy to be implemented by ACS. Nor does it limit the ACS replication strategy or data structures used to store information on rights. However, the ACS must implement the *IAccessControl* interface, used by the Object Registry for crossdomain calls validation. The main method of the IAccessControl interface, namely *ValidateAccess* of this interface confirms or

denies the validity of a call, based on the thread identifier, caller and callee identities, and the invoked method.

A variety of protection models can be implemented within the framework of the presented approach, including various capability [22, 23, 24, 25] and access control list (ACL) [26] models. It is also possible to select subjects and objects of the model in different ways. Some possible choices for objects are: distributed object, a single interface or even method. While for the role of subjects, one can use distributed object, protection domain or user.

5.3. Global Name Server

The Global Name Server (GNS) implements a distributed object location protocol, which maps the object's virtual address to the list of its contact points, i.e. network nodes, containing the object's replicas. GNS is used by the Object Registry to initiate the creation of a new distributed object replica in a local node.

The choice of a specific object location algorithm, implemented by GNS, should be based on the scale of the system and on the frequency with which nodes are added to and removed from it. For small systems a centralized protocol with one or several name servers is preferable. For large-scale systems with stable structures the hierarchy of domain servers [27] is usually used. While for highly dynamic systems decentralized naming protocols, e.g. [28], are most effective.

5.4. Garbage collection

The purpose of the E1 garbage collection system is to detect and delete of unused distributed object replicas.

In each node, the garbage collection system maintains only the information concerning local replicas. For each replica, the list of strong references to it, as well as the list of references it holds to other replicas, are stored. Both distributed and local references are taken into account. This information is sufficient to trace any changes in the reference graph, including those caused by node or network connection failures, while the simple references counting does not account for such situations correctly.

Cyclic distributed garbage collection in E1 is based on the partial reference graph tracing procedure, which verifies the reachability of some specified replica from **Root Object Set** [29]. The Root Set consists of system objects, which by definition are never regarded as garbage.

6. REPLICATION

In E1 the efficiency of the access to distributed object is determined by the efficiency of the replication strategy. The most efficient strategies are those that take into account the properties of certain object categories.

6.1. Distributed object replicas communication

Any non-trivial replication strategy requires some communication layer to organize the interaction between distributed object replicas. In E1, such a layer is provided by the **Group RPC** (GRPC) service, supporting transparent invocation of remote replication objects. GRPC in turn relies upon the **Group Communication**

mechanism which supports the exchange of unicast and multicast messages with various delivery ordering and reliability properties.

Group communication mechanism. For the purpose of this discussion, a group is a communication-level abstraction, which corresponds to a set of a single distributed object's replicas. The E1 group communication system includes two main services: **group membership service** and **message delivery service**.

Group membership service allows the addition and removal of object replicas dynamically. In addition, it is responsible for maintaining the consistency of the group in the face of hardware and software failures, which might cause unexpected replica crashes or group fragmentation. This is a nontrivial task, since in an asynchronous distributed environment it is impossible to distinguish a node crash from temporary inaccessibility caused by network delays [30]. To overcome this obstacle, one can use a distributed algorithm, determining accessible group members and reaching a consensus concerning a new group structure among its surviving members [31]. Such an algorithm is implemented by a special membership service component – **Failure Detector** (FD)**.**

If some of the group members become inaccessible as a result of network partitioning, rather than node failures, group fragmentation occurs. In this case, group membership service initiates formation of a new group in each fragment. Later on, the fragments may remerge into a single group again.

Message delivery service provides primitives for sending unicast and multicast messages between group members. For each message session, the delivery protocol properties can be specified. The most important ones are reliability of delivery and message ordering guarantees.

We plan to build the E1 group communication system on one of the existing implementations [32, 33, 34]. Such systems are initially designed to provide replication support within more complex software systems. Therefore they can be relatively easily integrated into E1. Also, being highly modular, they can be easily extended to support new message delivery properties [33].

Group RPC. Message-oriented communication primitives form the basis for distributed object replicas interaction. However, it is desirable to provide the replication strategy developer with a more convenient procedural model, allowing direct access to methods of the remote replication objects. In the case of point-to-point communication, the remote procedure call (RPC) mechanism is generally used to invoke operations on remote objects. The group remote procedure call (GRPC) is the generalization of RPC for the case of multicast communication. On the basis of group communication services described above, the GRPC implements a single primitive allowing a simultaneous invocation of several remote objects.

Like regular RPC, GRPC implements remote invocation with the help of client and server stabs. Stubs are compiled automatically from IDL definitions of objects'. Client stubs locally expose interfaces of remote replication objects. Each call to a client stub is converted into a message, sent to one or several remote replicas by means of a group communication system. The message is delivered to a server stub, which transforms it into a call of an appropriate replication object method. The result of the invocation is sent back to the caller's client stub. Having obtained the necessary number of responses (determined by the semantics of the call), the client stub returns control to the calling object.

6.2. Serialization interface

Since replication object is generally unaware of the semantics object's structure, the semantics object must implement serialization operations itself. These operations are available to the replication object through the *ISerializable* interface.

Some languages, e.g. Java and C#, provide support for automatic object serialization/deserialization, based on run-time type information. We expect that these languages will be widely used for application programming in E1.

However, in order to support languages, such as C++, in which automatic object serialization is not generally possible either at the time of compilation, or at run-time [35], it is desirable to develop a language-independent method for generating serialization interface.

In E1 the support for automatic objects serialization is provided by the memory management system. Each local object consists of a static part and dynamically allocated data. The dynamic memory allocation interface is exposed by **Heap** objects. Heap object represents a continuous virtual memory area, upon which allocation and deallocation operations are defined. Each domain provides the default local heap, which can be used by all its objects. Besides, any object can create a separate heap and allocate memory only from it. To serialize such object, it is enough to store the structure of the heap plus the object's static part in some data packet. At object deserialization, the heap is restored in a new node in the same virtual address. So, the problem of serialization/deserialization of the semantics object is reduced to a far simpler problem of serialization/deserialization of a Heap object. This approach is language-independent and can be used for objects of any type. However, it introduces certain memory overhead, since using a separate heap per object implies that the object's dynamic data occupies an integral number of physical memory pages.

CONCLUSION

This paper presents the architectural design of the E1 distributed operating system. In E1 the abstraction of the replicated distributed object is used as a building block for both OS components and application software. Since the 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 according to its semantics allows for 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 programming of a local (non-replicated) object.

Remote object access protocols are implemented by the developers of replication strategies. Most replication strategies are universal, i.e. can be applied to objects of various types. However, the replication strategy can be designed for a particular type of objects, which allows it to maximize the efficiency of access by taking type-specific properties into account. E1 provides support for the replication object development, including group communication system, object persistency, and special synchronization primitives.

E1 runs on top of a microkernel which supports a minimal set of primitives like address spaces, threads, IPC, interrupts dispatching. All operating system and

application functionality is implemented by distributed objects. 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 systems oriented at intensive communication of medium-grained objects.

Further work on E1 includes extending the presented architecture with support for object persistence. After that we plan to proceed to implementation of the first E1 prototype and its subsequent analysis.

REFERENCES

1. *Bakken E.D.* Middleware // Encyclopedia of Distributed Computing. — Kluwer Academic Press. — 2003. — 1110 p.
2. *Bernstein P.A.* Middleware // Communications of the ACM. — **39**. — № 2. — Feb. — 1996. — P. 68– 98.
3. *Object* Management Group (OMG). The Common Object Request Broker: Architecture and Specification. Version 3.0 // July. — 2002. — P. 1150.
4. *Microsoft* Corporation and Digital Equipment Corporation. The Component Object Model Specification. Version 0.9 // October. — 1995. — P. 274.
5. *Sun* Microsystems, Inc. EJB specification 2.0. — 640 p.
6. *Van Steen M., Homburg P., Tanenbaum A.S.* Globe: A Wide — Area Distributed System // IEEE Concurrency. — January — March. — 1999. — P. 70–78.
7. *Schneider F.* Implementing fault — tolerant services using the state machine approach: a tutorial // ACM Computing Surveys. — 22(4). — December. — 1990. — P. 290–319.
8. *Ford B., Lepreau J.* Microkernels Should Support Passive Objects. // Proc. I — WOOOS'93. — December — 1993. P. 226–229.
9. *Mach*:A New Kernel Foundation for UNIX Development / M.J. Accetta, R.V. Baron, W. Bolosky et al. // Proc. of the Summer 1986 USENIX Conference. — July. — 1986. — P. 93–113.
10. *Chorus* Distributed Operating Systems / M. Rozier, V. Abrossimov, F. Armand et al. // Computing Systems. — 1988. — **1**, № 4. — P. 82–98.
11. *Chase J. S.* An Operating System Structure for Wide-Address Architectures // PhD Thesis, Department of Computer Science and Engineering, University of Washington. –August. — 1995. — P. 144.
12. *Heiser G., Elphinstone K., Russell S., Vochteloo J.* Mungi: A distributed single address — space operating system // Technical Report 9314. School of Computer Science and Engineering, The University of New South Wales. — 1993. — P. 271–280.
13. *The ARIAS* Distributed Shared Memory: an Overview / P. Dechamboux, J.-P. Fassino, Hagimont D. et al. // Lecture Notes in Computer Science. — 1997. — **1175**. — P. 56–73.
14. *The Design* and Implementation of the Clouds Distributed Operating System / P. Dasgupta, R.C. Chen, S. Menon et al. // Computing Systems Journal. — **3**. — USENIX. — Winter. — 1990. — P. 11–46.
15. *Li K.* Shared Virtual Memory on Loosely Coupled Multiprocessors // PhD thesis, Yale. — September. — 1986. — P. 213.
16. *Stroustrup B.* The C++ Programming Language (3rd Edition) // Addison — Wesley. — 1997. — P. 911.
17. *Achieved* IPC performance (still the foundation for extensibility) / J. Liedtke, K. Elphinstone, S. Schönberg et al. // Proc. 6th Workshop on Hot Topics in Operating Systems (HotOS). — Chatham (Cape Cod), MA. — May. — 1997. — P. 28–31.

18. *Liedtke J.* L4 reference manual (486, Pentium, Pro) // Research Report RC 20549. — IBM T. J. Watson Research Center, Yorktown Heights, NY. – September. — 1996. — P. 53.
19. *Elphinstone K., Heiser G.*, L4 Reference Manual // Technical Report UNSW — CSE — TR — 9709. — School of Computer Science and Engineering, University of New South Wales. — December. — 1997. — P. 79.
20. *Potts D., Winwood S., Heiser G.* L4 Reference Manual: Alpha 21x64 // Technical Report UNSW — CSE — TR — 0104. — University of New South Wales. – Sydney. – March. — 2001. — P. 79.
21. *The L4Ka* team. L4 experimental kernel reference manual, version X.2 // — February. — 2002. — P. 182.
22. *Tanenbaum A. S., Mullender S.J., Renesse R. van.* Using Sparse Capabilities in a Distributed Operating System // Proc. Sixth International Conference on Distributed Computing Systems. — IEEE. — 1986. — P. 558–563.
23. *Gong L.* A Secure Identity — Based Capability System // Proc. IEEE Symposium on Security and Privacy. — 1989. — P. 56– 65.
24. *Jones A. K., Lipton R. J., Snyder L.* A Linear Time Algorithm for Deciding Security // Proc. 17th Symposium on Foundations of Computer Science. — Houston, Texas. — 1976. — P. 33–41.
25. *Bishop M., Snyder L.* The transfer of information and authority in a protection system // Proc. 17th ACM Symposium on Operating Systems Principles. — 1979. — P. 45–54.
26. *Ritchie D. M., Thompson K.* The UNIX time sharing system // Comm. ACM 17:7. — July. — 1974. — P. 365–375.
27. *Mockapetris P., Dunlap K. J.* Development of the Domain Name System // Proc. ACM SIGCOMM. — Stanford, CA. — 1988. — P. 123–133.
28. *Stoica I., Morris R., Karger D., Kaashoek M. F., Balakrishnan H.* Chord: A scalable peer — to — peer lookup service for Internet applications // Technical Report TR — 819. — MIT. — March. — 2001. — P. 149–160.
29. *Wilson P.R.* Uniprocessor garbage collection techniques // Technical report, University of Texas. — January. — 1994. — P. 1–42.
30. *Ricciardi A., Schiper A., Birman K.* Understanding Partitions and the “No Partition” Assumption // Proc. 4th IEEE Workshop on Future Trends of Distributed Systems. — Lisboa. — September. — 1993. — P. 12–26.
31. *Schiper A., Ricciardi A.* Virtually — Synchronous Communication Based on a Weak Failure Suspector // Proc. 23rd International Symposium on Fault — Tolerant Computing Systems. — Toulouse, France. — June. — 1993. — P. 534–543.
32. *Birman K. P., Joseph T. A.*, Exploiting Virtual Synchrony in Distributed Systems // Proc. 11th ACM Symp. on Operating Systems Principles. — Austin, TX. – November. — 1987. — P. 123–138.
33. *Renesse R. van, Birman K. P., Glade B., Guo K., Hayden M., Hickey T. M., Malki D., Vaysburd A., Vogels W. Horus*: A Flexible Group Communication Subsystem // Technical Report TR 95 — 1500. — Cornell University, Ithaca, NY.— 1995.— P. 14.
34. *Amir Y., Dolev D., Kramer S., Malki D.* Transis: A communication sub — system for high availability // Proc. 22nd IEEE Fault — Tolerant Computing Symposium (FTCS). — July. — 1992. — P. 76–84.
35. *Shapiro M., Gautron P., Mosseri L.* Persistence and Migration for C++ Objects // Proc. Third European Conference on Object — Oriented Programming. — 1989. — P. 191–204.

Received 08.05.2003

From the Editorial: The article corresponds completely to submitted manuscript.