ARCHITECTURAL DESIGN OF E1 DISTRIBUTED OPERATING SYSTEM

Leonid Ryzhik, Anton Burtsev
{leonid, anton}@e1os.org

ABSTRACT

Modern distributed operating systems provide users with transparent access to all resources of the computer network by means of distributed object abstraction. The access reliability and efficiency of such systems are determined by the internal implementation of the given abstraction. The existing architectures are based either on the centralized storage of the object state in one of network nodes, or on a state distribution among several nodes by means of distributed shared memory. The indicated approaches do not take into account semantics of a specific object. As a result, for the majority of objects the access efficiency in the distributed environment is considerably lower, than in a local case.

This paper presents the 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. 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. For example, in distributed data processing systems, component-oriented middleware, which supports the common object model in different network nodes, is widely used [35, 33, 53, 50].

The alternative approach consists in 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. Operating system - level implementation allows the construction of the most effective architecture, supporting the unified set of primitives for access to local and remote resources.

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.

This paper presents architectural design of E1 distributed operating system. Such operating system 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 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.
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.
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 operating system 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 property 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 operating system 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 operating system 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 feels like a single virtual computer, with its software structured like 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 modules 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 is involved 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 [46]. 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[1].

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

Before proceeding to the discussion of the use of component models in distributed operating systems, we will briefly outline the concepts underlying the component software development paradigm.

Component-oriented approach to software development is based on the idea of constructing software systems from prefabricated reusable components. Components should be independently deployable, i.e. a component can be used by a third party, which was not engaged in design and implementation of the given component.

Software component is defined as a unit of composition with contractually specified interfaces and explicit context dependencies only [54]. Components inherit essential concepts from object-oriented programming: encapsulation, polymorphism and availability through interfaces. However, components have additional properties not inherent to objects in object-oriented programming languages. Unlike objects, components are software products. In particular, it means that components can be developed and used independently by different sites. Component is an executable unit, rather than a programming language entity. Therefore, implementation inheritance is not supported for components. Component reuse is achieved by composition and aggregation. For two components to be interoperable, it is sufficient that they fit the requirements of a single component model, whether they were developed using the same or different programming languages. Components are characterized by higher degree of independence, than objects, and

---

[1] As for now, support for persistence in E1 is not designed in sufficient details. Therefore, it is not covered in this paper.

consequently, they have coarser granularity. As a rule, a component is constructed from several programming language objects.

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, e.g. persistence, transactions, replication, object trading, etc. (see, for example, [36]).

Extending a component model across the network, yields a convenient environment for distributed applications development which, besides other advantages of component-oriented architecture, provides network transparency, i.e., the components, located in different nodes, can invoke each other the same way as in local interaction. This approach is implemented by middleware systems, e.g. COM [33], Corba [35], EJB [53].

It is remarkable that modern distributed operating systems often provide abstractions and services resembling distributed component models of middleware systems. Apparently, it can be explained by the fact that both classes of systems are intended to serve as software platforms for distributed computing. Like middleware systems, distributed operating systems generally provide unified access to distributed system resources by means of object-oriented interface. In some implementations, objects are first-class citizens ([55], [21], [11]) while other systems support more simple primitives, e.g. message ports in Mach [4] and Chorus [43] or portals in Opal [10] above which the notion of the object is introduced by the object-oriented application run-time. Distributed operating systems provide a number of services for maintaining distributed objects, which are quite similar to key component services. First of all, it is a protected interaction mechanism, supporting the uniform invocation of object methods from any network node, provided that the caller possesses sufficient capabilities. Besides that, distributed operating systems include global naming services that enable binding to an object by its unique identifier. Some systems also support persistence of objects [11, 10, 12, 49, 15].

Despite indicated similarities, today's distributed operating systems do not provide valid component models. In these systems object abstraction serves primarily as a convenient means of interprocess communication, rather than application structuring paradigm. Both operating system services and application software are structured as a set of server processes that expose entry points for communication with other programs. Through an entry point a server exports operations for access to a certain resource or a group of resources. These operations are invoked with object semantics. Client specifies the identifier of an entry point, plus the required operation code and a parameter set. In response to a call, a server can return one or more values. Thus an object serves mainly as a communication abstraction.

At the same time, component software development paradigm regards objects as independent software entities with private state, explicit context dependencies and contractually specified functionality. Such notion of components doesn't fit the framework of modern distributed operating systems. Implementing a component model on top of these systems would require an intermediate software layer, similar to traditional middleware.

We believe 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 component model support should be initially designed into operating system. Following this approach, E1 implements a distributed component model, based on the abstraction of replicated object.

On the low level, the E1 component model relies on the execution primitives, which are essentially different from the ones used by the conventional operating systems. 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 [18]. 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 [18], 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.

Another feature of E1 component model is that it is based on replicated objects. The ability to replicate is a generic property of all objects. E1 provides extensive support for replication, including flexible replica communication service and extensible library of replication strategies.

Besides these services, E1 component model provides:

- Protected interaction mechanism, supporting the transparent invocation of object methods from any network node. In E1 all invocations are processed by the local replica of an object. Legitimacy of each call is verified by the distributed Access Control Server (ACS).
- Class Repository and Dynamic Class Loader.
- Global Naming Service, providing mapping of a unique object identifier to one ore more contact points of a given object.
- Garbage collection system, which detects and destroys unused object replicas on the basis of reference graph analysis.
- Support for persistence, which provides object lifetime control, based on reliable storage of a consistent object state in nonvolatile memory.
- Component development tools, including E1 Interface Definition Language compiler and Replication Strategies Compiler.

Since both operating system 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 E1.

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. Two well-known examples of client/server distributed operating systems are Mach [1] and Chorus [43]. 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 operating systems [10, 21, 13, 11] take 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 operating system creates local copies of its pages. The copies are synchronized using memory coherence algorithms [29]. These algorithms 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, we observe a natural trade-off between generality and efficiency of the replication strategy.

To illustrate inefficiency of distributed shared memory, let us consider the hypothetical Message Queue object. Objects in different network nodes can post to and receive messages from queue. Each client of the Message Queue receives all messages, which have been posted to it, so that in each node the identical set of messages is eventually obtained. However, the order, in which messages are delivered in different nodes, should not necessarily be identical – the only requirement is that causal message ordering is preserved [27]. Let us assume that Message Queue is implemented as a simple unidirectional list of small fixed-size entries. Let us also assume that shared access to Message Queue is provided by DSM algorithm [29]. Posting a message to a queue, then, requires exclusive locking of at least one object page (containing the pointer to the tail of the queue). To retrieve a message from the queue, one must acquire shared lock on at least one object page. Each locking operation requires broadcasting a message to all nodes, containing a copy of the queue, and receiving confirmation. In case of intensive use of the queue, when messages are posted and retrieved simultaneously from different nodes, each operation requires synchronization of the local object copy with one or several remote copies. In other words, each reference to the queue requires exchange of several broadcast and unicast messages with remote nodes.

Since memory coherence algorithms fail to take into account semantics and granularity of the performed operations, there is no way to essentially improve the efficiency of access to Message Queue within the framework of DSM-based approach. On the other hand, existing replication algorithms [26, 19] are capable of providing local execution of both read and write operations, while preserving causal message delivery ordering. Messages are received from and posted to a local copy of the queue. Delivery of messages to remote replicas is performed in asynchronous manner. Causal ordering is achieved with the help of logical timestamps, attached to each message [27]. This example illustrates, how replication strategy, which exploits object's specific properties, can be considerably more efficient, than memory coherence algorithm.

In summary, both client/server and DSM-based systems use universal methods to access objects in distributed environment (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 E1, the efficient and reliable access to each object is ensured by selecting replication strategy on the basis of the object's semantics. Of course, there is no need to design special replication protocols for each class of objects. E1 includes extensible library of replication strategies, from which one can select efficient strategy for virtually any type of objects. For example, the above-mentioned Message Queue replication algorithm is a variant of *active replication*, which can be effectively applied to a broad class of objects (see Section 6.1).

Two other 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 operating systems. Thus, E1 can be considered a generalization of these architectures.

3. E1 overview

3.1. Distributed objects

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

All objects reside in a single virtual 64-bit address space (its structure will be discussed in Section 4.1). 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 methods of the interface 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 [50].

The E1 local object resembles the structure of a C++ object [52]. 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 [33].

The distributed object architecture is shown in Figure 1. In a trivial case when the distributed object has only one replica (Figure 1a), 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 on 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 1b. 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[2]. 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 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 are not actually distributed, i.e. have only one replica. If an

---

[2] Detailed discussion of interface substitution technique lies beyond the scope of this paper

object with several replicas eventually remains with only one replica, its replication object is destroyed.

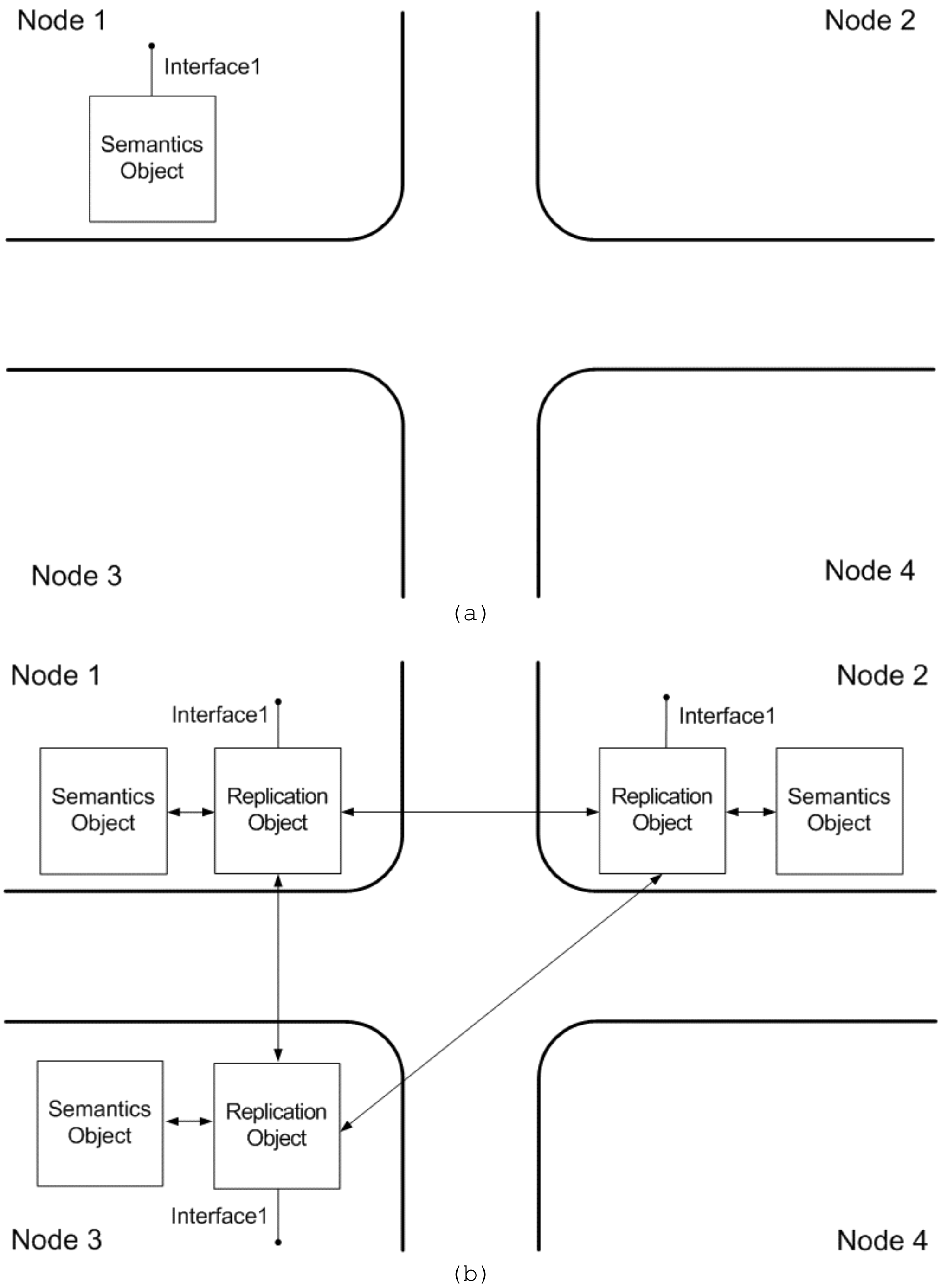

Figure 1. The distributed object architecture. a. distributed object with one replica; b. distributed object with several replicas

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.

3.2. Class objects

Classes of local objects in E1 are described by objects of the special type – **class objects**. Encapsulation of class properties by objects allows implementing dynamic class loading. For the same purpose serve class factories in COM [33] and Corba [37].

Class object 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 E1. Instead, a distributed object can be identified by the class of its semantics object, since it is the semantics object that encapsulates the distributed object's functionality.

3.3. E1 architecture

Figure 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 operating system construction, such as: address spaces, threads, IPC and interrupts dispatching. All operating system and application functionality is implemented by objects.

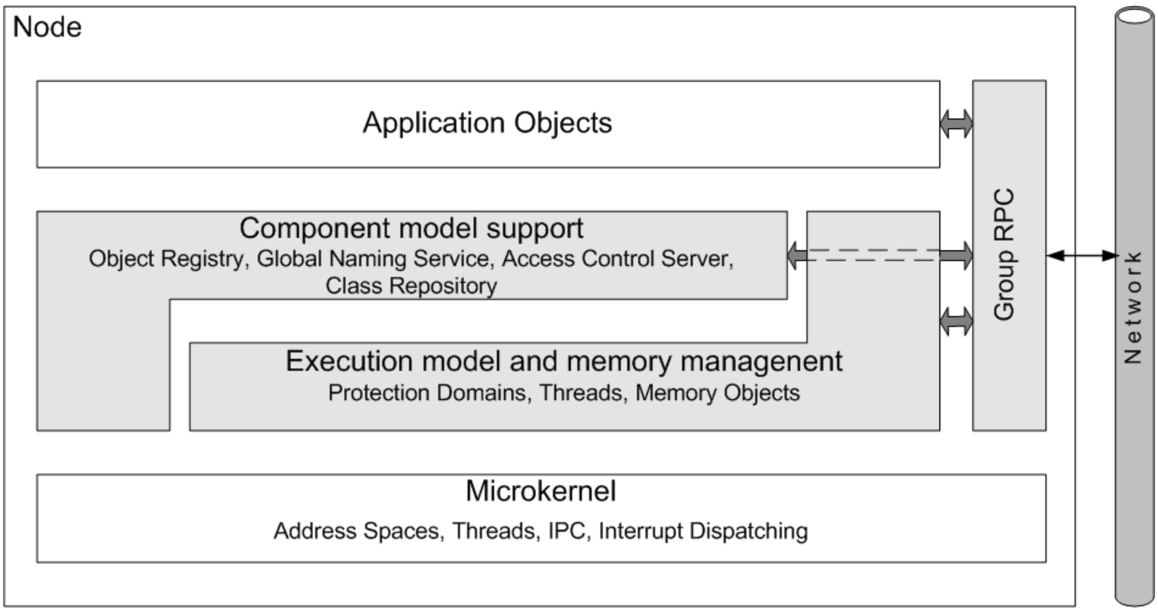

Figure 2. Generalized E1 architecture

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 [32]. Among these are microkernels of the L4 family [31, 14, 39, 25]. 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.

The E1 system components can be divided into three groups, represented with grey bars in Figure 2:

1. Objects, implementing the E1 execution model and memory management.
2. Component model support.
3. Replication support (“Group RPC” in Figure 2).

4. Object interaction and protection

In order to perform useful operations, objects interact by means of method calls. To provide safe execution of applications, it is necessary to ensure that object interaction is governed by some precisely defined access control policy.

The E1 protection model is based on three assumptions:

1) Object methods have no immediate access to the internal state of other objects (object isolation requirement).
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.

These assumptions are provided, accordingly, by three mechanisms: **protection domains**, **crossdomain calls** and **access control mechanism**. This section describes domains and crossdomain calls, while access control mechanism will be discussed in Section 5.

4.1. Protection Domains

Operating system protection model has to be based on facilities provided by the underlying hardware platform, primarily, virtual memory mechanisms in modern microprocessors. Therefore, geared towards 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.

The 4-gigabyte address space of modern 32-bit processors is, obviously, insufficient for holding all code and data of the distributed system. Therefore, single address space operating system (SASOS), like E1, requires a hardware platform with wide virtual address space.

Generally speaking, the E1 distributed object model could be implemented within the traditional private address space approach. In this case, smart pointers could be used to reference objects beyond the local address space. However, the single address space approach results in a simpler and easier-to-use architecture. Further analyzes of SASOS advantages can be found, for example, in [10, 22].

Let us return to the problem of distributed objects protection. The E1 protection model is based on object isolation requirement. According to that, the object’s state is not directly accessible to other objects. To achieve such kind of isolation within a single address space, it is necessary to place each object into a separate protection context, so that virtual memory outside this context is unavailable to object methods. Although this scheme provides correct object isolation, it introduces a significant overhead. First, since memory management units of modern processors operate at page granularity, protection context can consist of an integral number of pages only, which results in extremely inefficient use of physical memory, especially in the case of small objects. Secondly, within the outlined approach every method invocation results in protection context switch, which requires a number of additional CPU cycles. During intensive object interaction context switching would consume a significant part of CPU time.

In order to provide effective object isolation in E1, we introduce the notion of **protection domain**, offering a trade-off between efficiency and safety of interaction. Protection domain represents a part of a single virtual address space, containing one or several distributed objects. 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. Intradomain method invocations do not require the protection context switch.

While arranging objects in domains, one must take into account the following factors:

- placing objects in different domains protects them from accidental or deliberate attempts of unauthorized 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, one should place intensively communicating objects which jointly implement some functionality to 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. 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 destroyed by the garbage collection system, one must create a reference on an object before using any of its methods (see Section 5.4).

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 (both passed by value and by reference) 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.

Each node contains exactly one instance of CA, mapped to all domains. Thus, CA operates as a universal proxy object, handling all crossdomain calls in a system. To prepare the call stack, the CA needs to know the called method's parameter types, which can be obtained from its class object through a special reflective interface.

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 this stack is aligned to page boundary and the resulting address is interpreted as the bottom of a new stack (see Figure 3), which is then mapped to a target domain, so that the content of the calling object's stack is not accessible to the called object.

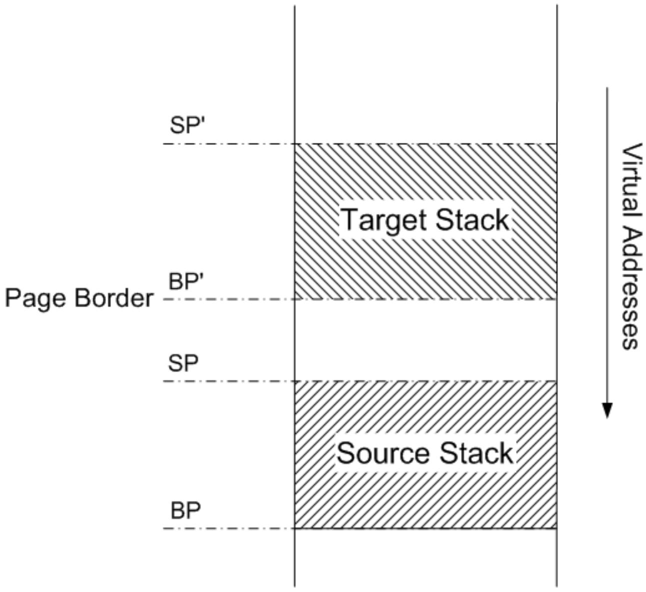

Figure 3. Stack management during crossdomain call.

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 [18]. 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 [18], migrating threads are more appropriate for object-oriented environment, than traditional static threads.

Migrating threads eliminate the need of starting a separate thread for processing each call or queuing calls for sequential processing. This results in increased efficiency of object interaction, as well as a simpler and more lightweight object architecture.

To start a new thread, one specifies its initial object and method. While executing this method, thread can perform nested calls to other objects. The thread is terminated on return from the method, in the context of which it was started.

If a thread performs an illegal operation while running in the context of some distributed object replica, this replica will be destroyed and the thread will return to the previous element in the stack of nested calls just like if it had completed execution of a method with an error code.

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 (see Section 6.1). 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.

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, i.e. a set of register values to be restored on return from the call. 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, e.g. 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. 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[3].

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.

Creating and destroying objects

Distributed objects in E1 are created by means of the *CreateObject* method, exposed by Object Registry. It accepts class name, target domain identifier and, optionally, the name of the replication strategy to be applied to the new object. *CreateObject* method performs the following sequence of operations:

1. If the required class object does not exist in the target domain, it will be loaded from Class Repository.
2. Calls the class object to create a semantics object. Since the new distributed object is represented only in one node, it does not require a replication object.
3. Registers the new object in internal Object Registry data structures. The caller of *CreateObject* method obtains the first strong reference on a new object.
4. Registers the new object in global naming system.
5. Returns the pointer to one of the newly created object's interfaces.

Distributed object is automatically destroyed when all of its replicas turn to garbage (see Section 5.4). One can also force the destruction of an object by calling the *DeleteObject* method of Object Registry.

[3] Detailed description of dynamic class loading mechanism lies beyond the scope of this paper

Creating replicas of existing distributed objects

The newly created distributed object consists of only one replica. Subsequently more replicas can be created and destroyed. Replica creation is initiated when a strong reference on a distributed object is created in the node where there is no replica of the given object yet (see Section 5.4). Object Registry then performs the following sequence of operations:

1. Obtains information about the object from the global naming system: its class name, contact points and replication strategy.
2. Loads the class objects for semantics and replication objects to be created, from Class Repository to target domain.
3. Creates a semantics object and the associated replication object.
4. Initializes replication object with the list of contact points, required to execute a group join protocol. This protocol is a part of the replication strategy.

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. In Figure 4 an optimized crossdomain call path is shown with red dashed line. Cache consistency is maintained by the Object Registry through the *IAccessCache* interface.

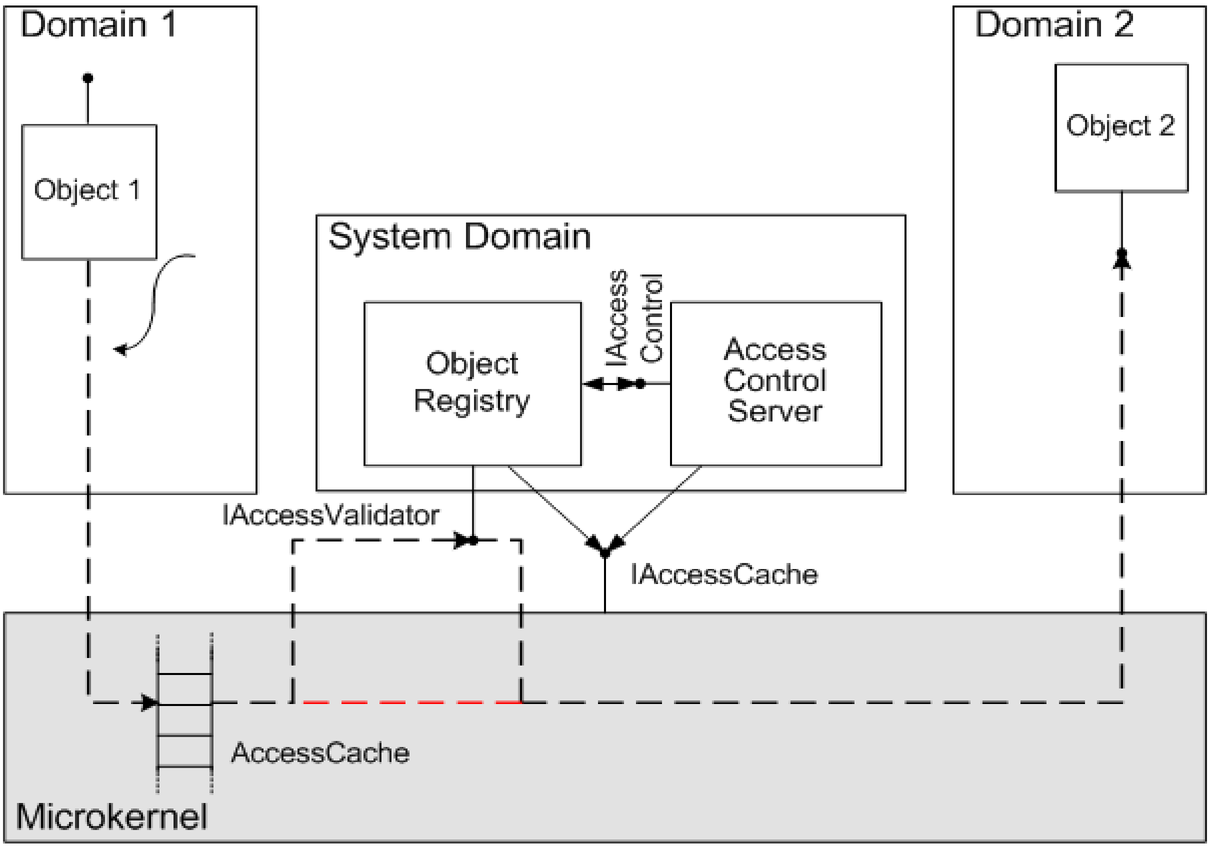

Figure 4. Communication between microkernel and Object Registry during crossdomain call. The dashed line shows normal and optimized (red) crossdomain call path.

Other important Object Registry functions – reference management and garbage collection are described in section 5.4.

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 task. 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*, 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. For example, ACS, implementing Take/Grant capability model [7, 8], can provide *Take*, *Grant* and *Revoke* methods, while emulation of UNIX access control list model would require methods like *Chown* and *Chmod*.

Global fulfillment of access control rules is provided by the ACS replication strategy. For example, 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.

A variety of protection models can be implemented within the framework of the presented approach, including various capability [56, 20, 24, 7] and access control list (ACL) [42] 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. The last possibility is rather interesting. Until now, we have not introduced user abstraction in E1. Nevertheless, the models in which rights belong to users or roles are in wide use today [44, 42]. In such models each thread operates on behalf of some user. Therefore, though the concept of users is not explicitly supported in E1, it is possible to implement it at the ACS level by associating users with groups of threads.

5.3. Global Naming 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, on 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 join and leave 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 [34] is usually used. While for highly dynamic systems decentralized naming protocols, e.g. [51], are most effective.

5.4. Garbage collection

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

In conventional operating systems there is normally no need for a separate garbage collection subsystem. Instead, every operating system component uses its own resource management mechanism, based on a simple reference counting. Such approach is easy to implement and it results in minimal overhead. However, in an asynchronous distributed environment, reference management becomes a substantially more complicated task [38]. In E1 it is further complicated by the possibility of having several object replicas in different nodes. Garbage collection in such systems requires sophisticated distributed algorithms and data structures. Since it is inefficient to design and implement them separately for each operating system component, E1 provides a single garbage collection system for all distributed objects.

Garbage collection in E1 is based on the analysis of a reference graph between distributed objects replicas. Two types of references correspond to two types of object interaction: local interaction between replicas of different distributed objects, and remote interaction between replicas of a single distributed object within its replication strategy. Correspondingly, there are **local references** between different distributed objects and **remote references** between replicas of one object.

We will also distinguish **weak** and **strong** references. Weak reference is simply a pointer to an interface of an object or RPC-pointer to one or several remote replicas, used to perform local and remote invocations, respectively. Weak references are not traced by garbage collection system or taken into account while detecting unused object replicas. To convert a weak reference to a strong reference, one must execute *AddRef* operation over it. In E1, *AddRef* method is exposed by the garbage collection system, rather than the object itself. As a result of the *AddRef* operation, new strong reference is registered in the garbage collection system. If the replica addressed by the given reference does not exist yet, it is created by the Object Registry, as described in Section 5.1. Every subsequent *AddRef* operation increments the value of a counter, associated with the given reference. The counter is decremented by the *Release* operation. When it drops to zero, the strong reference is deleted. Deleting the last strong reference to the replica initiates the replica's removal.

In each node, the garbage collection system maintains only the information concerning local replicas. For each replica, the list of strong references on it, as well as the list of references it holds to other replicas, is 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.

The majority of information about the references is stored in the Object Registry. Registries in different nodes communicate in order to manage remote references. Besides the Object Registry, the E1 garbage collection system includes **Reference Monitors** located in each domain. Reference Monitor carries out reference counting within its domain, which minimizes the number of crossdomain calls to the Registry. It exposes the *IRefMonitor* interface, containing *AddRef* and *Release* methods. To create and delete local references, application objects interact with Reference Monitor, which, if necessary, calls the Object Registry reference management methods.

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** [58]. The Root Set consists of system objects, which by definition are never regarded as garbage. All objects reachable from ROS are not garbage either. All other objects are considered garbage. To perform partial reference graph tracing, a suspect replica must be selected using some heuristic procedure. This replica will become a starting point for graph scanning. As a result, either reachability of the given replica from the ROS will be proven, or a set of replicas forming the garbage cycle will be detected.

Figure 5 shows a reference graph fragment. Solid arrows denote strong references, while dashed arrows correspond to weak references. For the distributed object *A* the client/server replication strategy with one primary and one backup server is used. Client replica A_2 holds strong reference on primary server replica A_4, which maintains the object state and performs operations upon it. In its turn, A_4 holds the strong reference on the backup server A_1, which stores a secondary copy of the state. Client A_2 forwards all method calls to primary server A_4, while A_4 communicates with backup server A_1 to keep it in a consistent state.

Object *B* uses an active replication strategy. Each method invocation is broadcasted to all replicas, which perform corresponding operations over the local copy of object state. To execute these remote invocations, each replica holds weak references on all other replicas (in the figure, object *B* has only two replicas). In the case of active replication each replica has to exist only as long as it is used in its local node, and can be safely destroyed afterwards, i.e. replicas do not depend on each other. Hence, no replica needs to hold strong reference on any other replica.

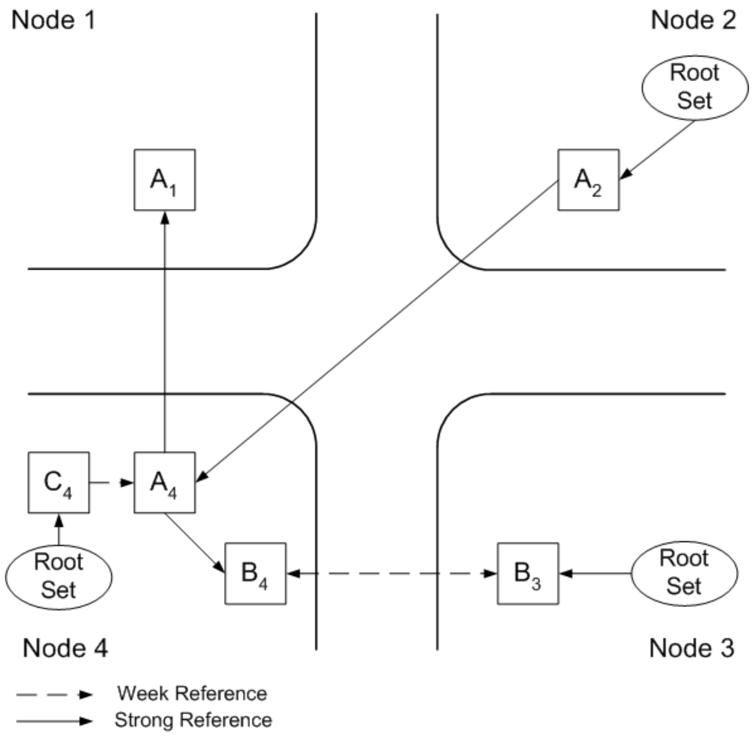

Figure 5. Reference graph fragment

This example clarifies the semantics of strong and weak references. Strong reference reflects the dependent relationship between replicas, that is when some replica (or object) requires another replica (or object) for its correct operation. Weak reference is simply a means of interaction. It is used when communicating replicas do not depend on each other, and therefore destruction of one does not cause the other to malfunction.

6. Replication

In E1 the efficiency of the access to distributed object is determined by its replication strategy. The most efficient strategy is usually the one that takes into account the properties of particular object or object category. To enable the use of such strategies, E1 does not impose any limitations on the internal architecture of replication object, neither on replication algorithms used. Instead, it provides a set of services, helping developer to solve the most complicated tasks, arising from implementation of the majority of replication strategies.

6.1. Survey of replication strategies

This section provides brief description of several widely used classes of replication algorithms, which form the basis of the E1 library of replication strategies. The purpose of this section is to present an introduction to object replication. For a detailed description of various strategies, the reader may consult the following papers [9, 5, 28, 16, 2, 57, 19, 26].

Client/server replication

Client/server is a trivial replication strategy. A single copy of the object state is maintained by a *server* replica (Figure. 6). Other replicas are *clients*. All client invocations are forwarded to the server.

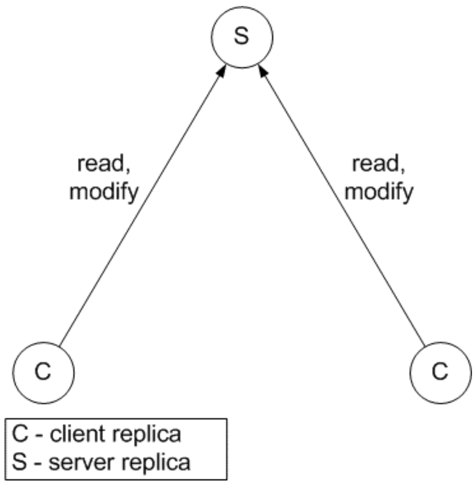

Figure 6. Client/server replication

This strategy is in most cases inefficient, since it does not provide local access to resources. Another disadvantage is low reliability due to centralized access to objects.

Passive replication

In the case of passive replication [9,5], each replica stores a copy of an object state (Figure 7). One replica is assigned as *primary*. Read operations are executed locally in each node. Modifications are forwarded to the primary replica, which executes the required operations and updates all other replicas.

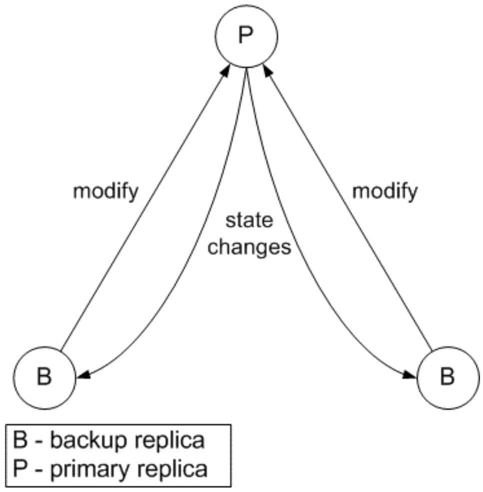

Figure 7. Passive replication

Active replication

Each replica stores a copy of an object state (Figure 8). Both read operations and modifications are performed locally in each node. To ensure replica consistency modifications are broadcasted to all replicas.

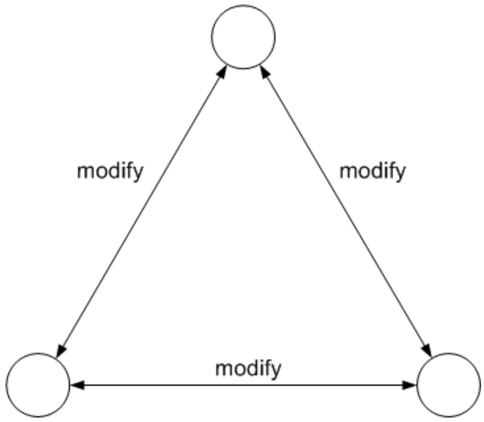

Figure 8. Active replication

Within the framework of active replication a variety of algorithms, providing different types of replica consistency (sequential consistency [28, 16], causal consistency [2], temporal consistency [57], weak consistency [19], and lazy consistency [26]), have been developed.

Migration

Migration in E1 refers to the transfer of object replica between nodes. Migration is not an independent replication strategy. It is used in conjunction with other strategies to improve the efficiency of access to resources by means of load balancing.

Migrating a replica, without any threads running in its context, is a rather trivial task. However, it is sometimes necessary to migrate replicas of objects which have methods that execute for a long time or even for the object's entire lifetime. The E1 port of a traditional UNIX program is the example of such an object. 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. As mentioned in Section 4.3, E1 provides for such a capability.

6.2. 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 maintains consistent group membership lists, or **views**, for all object replicas. It allows replicas to join and leave the group dynamically. In addition, it is responsible for maintaining the consistency of the group in the face of hardware and software failures, which might cause replicas destruction 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 [41]. 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 [45]. 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 merge into a single group again.

Message delivery service provides primitives for exchanging 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. Table 1 summarizes some possible values of these properties.

Property	**Description**
DELIVERY RELIABILITY	
Unreliable delivery	Does not provide any message delivery guarantees.
Atomic delivery	Guarantees that each message will be either delivered to all its destinations, or to none of them
DELIVERY ORDERING	
Unordered delivery	Does not impose any restrictions on message delivery order
FIFO-ordering	All messages from a group member are delivered in the order in which they were sent
Causal ordering	Preserves causal relations [27] between messages
Total ordering	Each member receives all messages in the same order

Table 1. Message delivery properties

The development of a group communication system from scratch is a rather complicated task, which comprises implementation of message delivery and group membership algorithms. Therefore, we plan to build the E1 group communication system on one of the existing implementations. Currently the services described above are implemented in a number of group communication systems [6, 40, 3]. Such systems are 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 [40].

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. Figure 6 depicts the architecture of E1 GRPC mechanism.

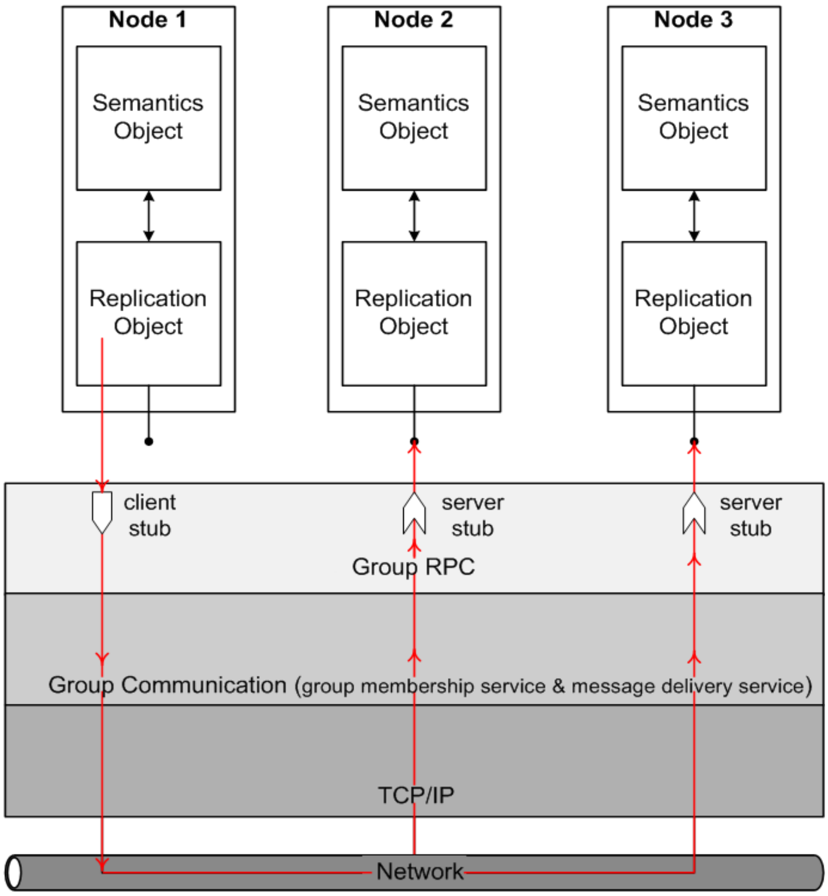

Figure 9. Execution of the remote call by GRPC system.

Like regular RPC, GRPC implements remote invocation with the help of client and server stubs. Stubs are compiled automatically from the IDL-definitions of objects (see section 7.1). 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.

6.3. Change of group structure

As discussed in the previous section, the group structure can change either when new replicas join or leave the group, or as a result of communication or node failures. On the level of group communication system such events are handled by group membership service, which delivers consistent views to all replicas. The group communication system, then, notifies replication objects about the changes in the distributed object structure through the *IReplicaGroup* callback interface. On the level of replication strategy, the handling of this event can involve distributed communication between replication objects, including replica synchronization, creation and deletion of the remote references and even creation or destruction of replicas. This process yields a new distributed object configuration, which meets the consistency requirements, imposed by the replication strategy used.

The following sample scenario of distributed object recovery after network partitioning illustrates that the distributed protocol which is handling the changes in group structure is an essential part of any replication strategy.

Initially, an object consists of client replica *C*, primary server *P* and backup server *B* (Figure 10a). The arrows indicate strong references between the replicas. As a result of the network partitioning the object divides into three fragments (Figure 10b). If the replication strategy does not provide for a future fragment re-attachment and doesn't apply any special efforts to preserve object integrity, all object replicas will be destroyed: server replicas (*P* and *B*) will be destroyed by the garbage collection system, since there are no strong references on them; while client replica must self-destruct, since it does not store a consistent copy of the object state and therefore cannot successfully process incoming method calls. Suppose however, that replication strategy tolerates object fragmentation in the following way: in order not to be destroyed by the garbage collection system, the primary server replica creates a strong reference on itself from the Root Set (Figure 10c). Additionally, since the given replication strategy implies the existence of a backup server with a secondary copy of the object state, the primary server creates this backup replica. All invocations of a client replica will return an error indicating that the object is currently fragmented. Thus, only the secondary server *B* will be destroyed as garbage. If the network connection is subsequently restored, the surviving object replicas will remerge (Figure 10d) and continue normal operation.

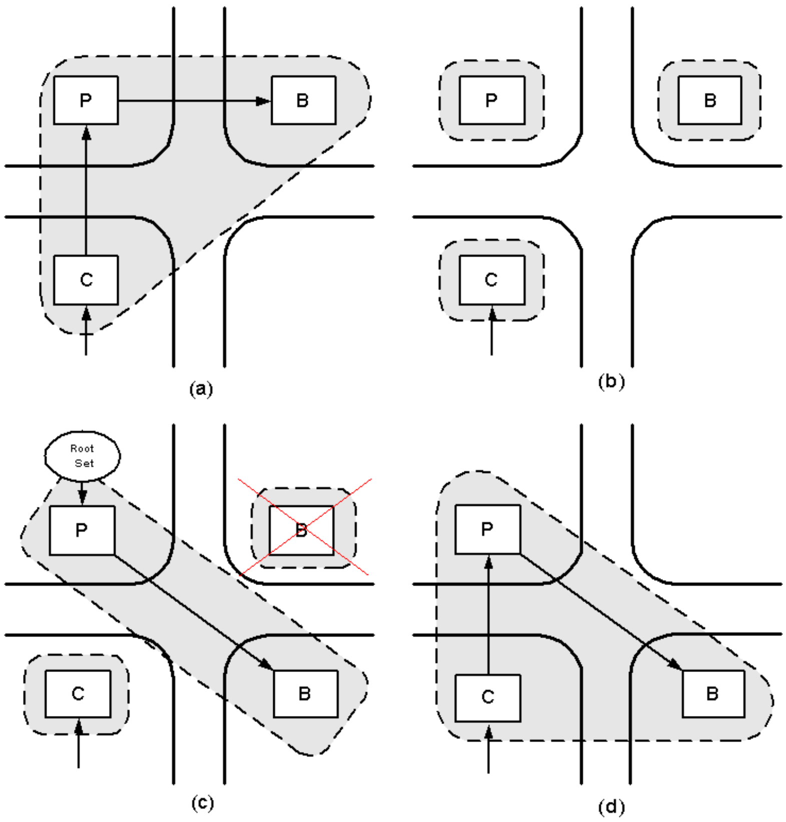

Figure 10. Fragmentation of the distributed object

Note that immediately after fragmentation (Figure 10b), replicas *P* and *B* could be destroyed as garbage. In order to process these kinds of situations correctly, the notion of **transitional distributed object state** is introduced. As soon as the group communication system notifies the replication object of a view change, the replica is transferred into a transitional state in which it cannot be destroyed by the garbage collection system. While in transitional state, the object can freely change its structure. For example, the object can adjust strong references between replicas or create external references, as in the example above. As soon as all required operations are complete, the object will leave the transitional state.

6.4. Serialization interface

This section discusses another important aspect of the distributed object development – implementation of the serialization interface. All replication strategies rely on some operations, which replication object can execute only in cooperation with the semantics object. The most important examples are serialization and deserialization of an object state. Almost all replication strategies employ these operations to transfer object state between the replicas and to synchronize replicas. Since the replication object is generally unaware of the structure of the semantics object, the semantics object must implement serialization operations itself. These operations are exposed to replication object through the *ISerializable* interface. *ISerializable* is similar to the CORBA *Checkpointable* interface, which also supports object replication [35].

In other words, the serialization of the distributed object state is delegated to the semantics object developer. Note that the serialization/deserialization procedures are generally rather cumbersome. It is therefore desirable to generate them automatically. This is an intricate problem, with no general solution, that would work efficiently for all types of objects while being language-independent.

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

However, along with them, other languages, in particular, C++, should be supported, for which automatic object serialization is not generally possible either at the time of compilation, or at run-time [48]. It is therefore desirable to develop a language-independent method for generation of the 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 part. The dynamic memory allocation interface is provided 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 an object, it is enough to store the structure of the heap plus the object's static part in some data packet. On object deserialization, the heap is restored in a new node in the same virtual address. So, the problem of serialization and deserialization of the semantics object is reduced to a far simpler problem of serialization and deserialization of a Heap object. This approach is language-independent and can be used for objects of any type. However, it introduces a memory overhead, since using a separate heap per object implies that the object's dynamic data occupies an integral number of physical memory pages.

Figure 11 compares two approaches. Figure 11a shows the serialization of the object, developed in a programming language with run-time type information support. Such object uses a default heap for dynamic memory allocation. To perform object serialization, language run-time components parse its structure by tracing the intra-object reference graph. On deserialization, the object structure is restored in the target node's default heap. This procedure preserves the object's referential integrity, while separate object fragments are placed at virtual addresses, distinct from their original ones. Figure 11b shows the serialization of the object, written in C++, and using private heap for memory allocation. After deserialization all object fragments are allocated at their original virtual addresses.

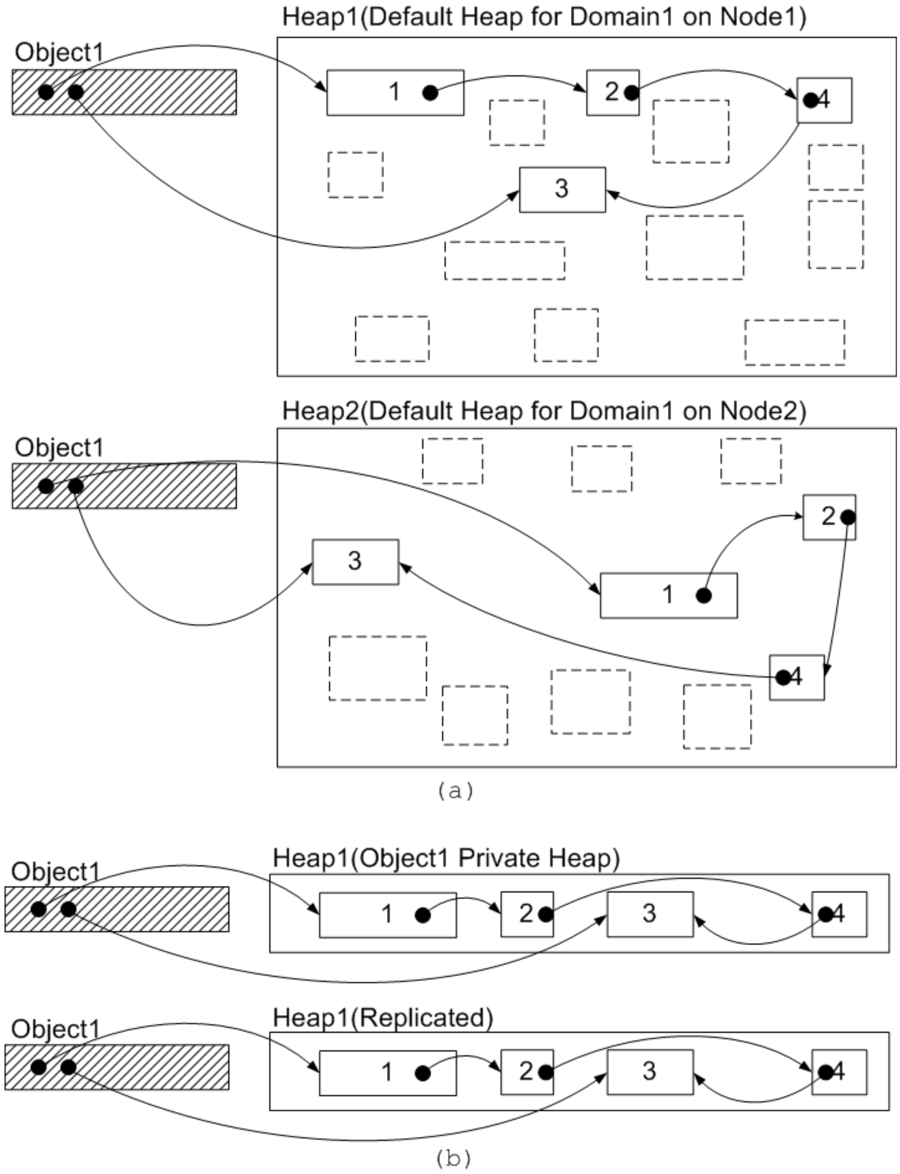

Figure 11. Two approaches for automatic serialization and deserialization of the semantics object. a. Using language run-time support. b. Using a private heap for dynamic memory allocation.

If the programming language does not support automatic serialization of the object state, and if the use of a private heap for each object results in an unacceptable waste of the physical memory, then the object developer should implement *ISerializable* interface himself. This approach will be used for the E1 system objects.

7. Programming in E1

E1 applications are constructed as a collection of distributed objects. This section briefly discusses the methodology of distributed objects development and the tools needed.

A distributed object consists of semantics and replication objects developed, as a rule, by different sites. Normally, an object developer implements the semantics object and specifies one or more standard replication strategies, which can be applied to it. In rare cases is the replication strategy developed for one particular semantics object.

E1 supports the component paradigm of software development. For this purpose, the distributed object model is extended by a set of services: Object Registry, Access Control Server, Global Naming Server, and garbage collection system. Also included in the E1 component model is the **Interface Definition Language (IDL)**, making posiible binary interoperability of components written in various programming languages. Figure 12 illustrates the C++-based development cycle of a distributed object. Object development is carried out by two sites – the site implementing the replication strategy, and the site implementing the semantics object, i.e. the component developer. The semantics object developer prepares the IDL-definition of object interfaces, using which the IDL compiler generates abstract C++ classes from which object implementation is inherited. The IDL-definition of the object contains the following information:

- definitions of the data structures used as parameters or return values by the object methods;
- unique class and interface identifiers;
- interface declarations: method names, types of arguments and return values, directional attributes and additional meta-information which can be used by the Replication Strategies Compiler. For example, method declaration can have one of the ([read], [modify]) attributes.

Replication strategy can be implemented independent of the semantics object. However replication object can not be compiled until the information about the interfaces of the specific semantics object is available. This contradicts the existence of universal replication strategies, e.g. active replication, which can be applied to various objects. To alleviate this contradiction, the replication strategy is described using a special scripting language denoting the semantics object's interfaces and methods by some abstract identifiers. When the replication strategy is applied to a certain class of semantics objects for the first time, the appropriate replication object is compiled on the fly from the replication strategy description and the semantics object's IDL-definition by the **Replication Strategies Compiler (RSC)**. Besides replication objects, the RSC generates GRPC-stubs, that are required for communication between distributed object replicas.

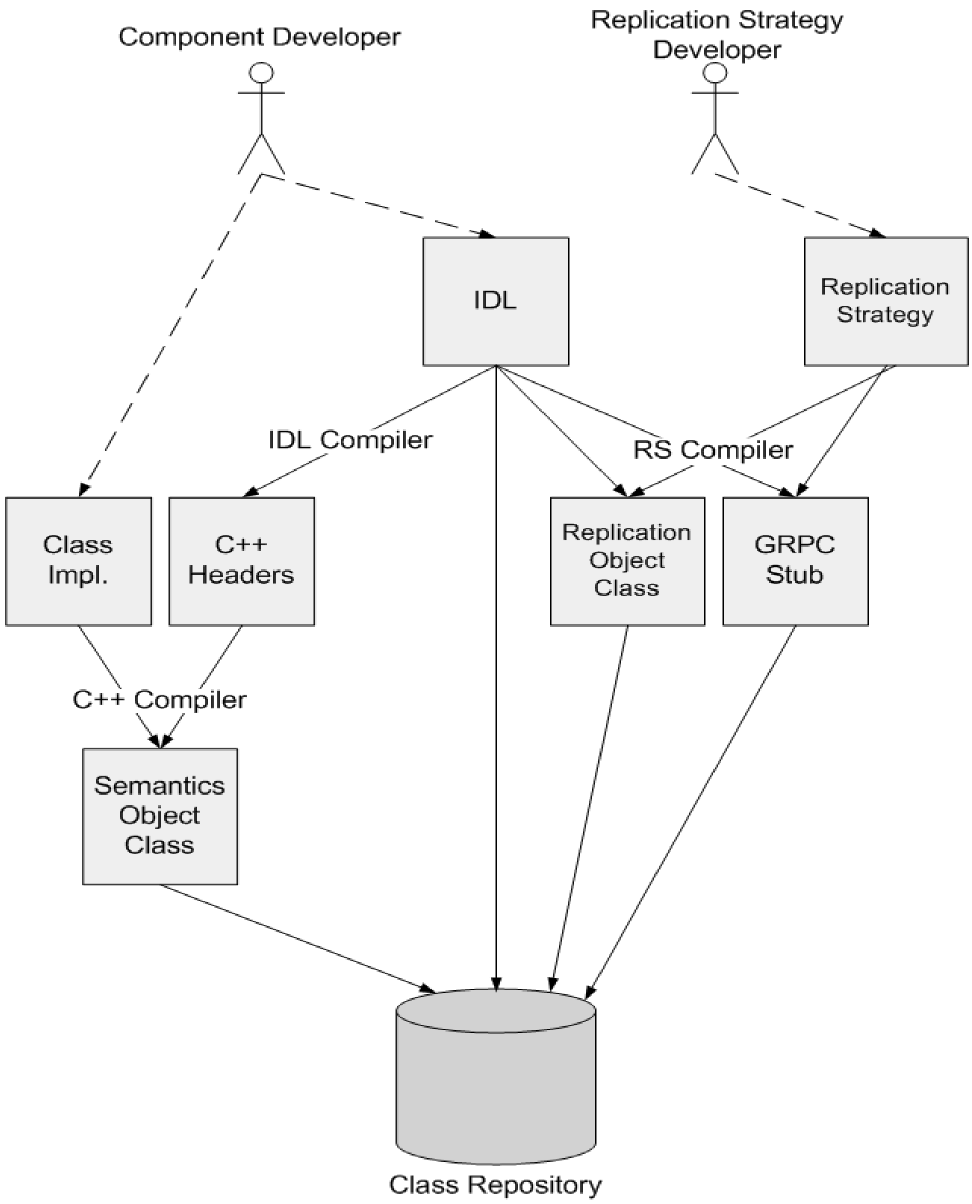

Figure 12. Development cycle of a distributed object

The distributed objects can be developed not only in C++, but also any other programming language, which can be mapped to the E1 object model.

Conclusions

We have presented 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 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.

Distributed 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 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, and object persistence.

E1 runs on top of a microkernel which supports a minimal set of primitives like address spaces, threads, IPC, and 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 the systems oriented at intensive communication of medium-grained objects.

All E1 objects reside in a single virtual 64-bit address space divided into isolated protection domains. Addresses of the object interfaces serve as globally unique identifiers, using which object can be invoked from any network node. Such virtual memory organization provides convenient object communication environment.

The E1 execution model is based on the migrating threads concept. In E1 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.

Following the current trend towards component software, E1 provides support for component-oriented programming. Development and use of components require a component model, defining a set of services, interfaces and conventions, all of which constitute the execution and communication environment for components. Current component models are usually implemented as middleware systems. In contrast, E1 implements component model by extending the distributed object model with component-oriented services and tools, thus avoiding the use of an additional software layer. Besides being highly efficient, such architecture simplifies the development and use of components, since in E1 distributed objects (or components) are first-class citizens, like files in UNIX. The E1 component model provides services for object protection, global object naming, dynamic class loading, garbage collection, as well as component development tools like IDL compiler and replication strategy compiler.

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. M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, Jr. A. Tevanian, M. W. Young. Mach: A New Kernel Foundation for UNIX Development. *Proc. of the Summer 1986 USENIX Conference* pp.93-113, July 1986.
2. M. Ahamad, M. Raynal, G. Thia-Kime. An Adaptive Protocol for Implementing Causally Consistent Distributed Services. *Proc. 18th International Conference on Distributed Computing Systems*, Amsterdam, pp.86-93, 1998.
3. Y. Amir, D. Dolev, S. Kramer and D. Malki. Transis: A communication sub-system for high availability. *Proc. 22nd IEEE Fault -Tolerant Computing Symposium (FTCS)*, July 1992.
4. J. S. Barrera. A fast Mach network IPC implementation. *Proc. of the Second USENIX Mach Symposium*. pp.1-12, 1991.
5. K. P. Birman, T. Joseph, T. Raeuchle, and A. El Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, 11(6):502–508, June 1985.

6. K. P. Birman, T. A. Joseph, Exploiting Virtual Synchrony in Distributed Systems, *Proc. 11th ACM Symp. on Operating Systems Principles*, pp. 123-138, Austin, TX, November 1987.
7. M. Bishop, L. Snyder. The transfer of information and authority in a protection system. *Proc. 17th ACM Symposium on Operating Systems Principles*, 1979.
8. J. Biskup. Some variants of the take-grant protection model. *Information Processing Letters*. 3, №19. 1984.
9. N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, Distributed Systems, ACM Press Books, chapter 8, pages 199–216, Addison-Wesley, second edition, 1993.
10. J. S. Chase. An Operating System Structure for Wide-Address Architectures. PhD Thesis, Department of Computer Science and Engineering, University of Washington, August 1995.
11. P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. Jr. LeBlanc, W. Applebe, J. M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, C. J. Wileknloh. The Design and Implementation of the Clouds Distributed Operating System. *Computing Systems Journal*, Vol 3, *USENIX*, Winter 1990.
12. A. Dearle, R. di Bona, J. M. Farrow, F. A. Henskens, A. Lindström, J. Rosenberg, F. Vaughan. Grasshopper: An Orthogonally Persistent Operating System, Technical Report, Departments of Computer Science, Universities of Adelaide and Sydney, 1993.
13. P. Dechamboux, J.-P. Fassino, D. Hagimont, J. Mossière, X. Rousset. The ARIAS Distributed Shared Memory: an Overview. *SOFSEM Seminar*, Prague, November 1996.
14. K. Elphinstone, G. Heiser, L4 Reference Manual, Technical Report UNSW-CSE-TR-9709, School of Computer Science and Engineering, University of New South Wales, December 1997.
15. K. Elphinstone, S. Russell, G. Heiser. Supporting Persistent Object Systems in a Single Address Space. Technical Report 9601, School of Computer Science and Engineering, The University of New South Wales, February 1996.
16. A. Fekete, M. F. Kaashoek, N. A. Lynch. Implementing Sequentially Consistent Shared Objects Using Broadcast and Point-to-Point Communication *Proc. International Conference on Distributed Computing Systems* pp.439-449, 1995.
17. B. Ford, J. Lepreau. Evolving Mach 3.0 to use migrating threads. Technical Report UUCS-93-022, University of Utah, August 1993.
18. B. Ford, J. Lepreau. Microkernels Should Support Passive Objects, *Proc. I-WOOOS'93*, December 1993.
19. R. Golding. Weak-Consistency Group Communication and Membership. PhD thesis, University of California, Santa Cruz, 1992.
20. Li Gong. A Secure Identity-Based Capability System. *Proc. IEEE Symposium on Security and Privacy*, pp.56-65, 1989.
21. G. Heiser, K. Elphinstone, S. Russell, J. Vochteloo. Mungi: A distributed single address-space operating system. Technical Report 9314. School of Computer Science and Engineering, The University of New South Wales, 1993.
22. G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, J. Liedtke. The Mungi single-address-space operating system. *Software Practice and Experience*, 28(9), July 1998.
23. P. Homburg. The Architecture of a Worldwide Distributed System. PhD thesis Vrije University, Advanced School of Computing and Imaging, Amsterdam, 2001.
24. A. K. Jones, R. J. Lipton, and L. Snyder. A Linear Time Algorithm for Deciding Security. *Proc. 17th Symposium on Foundations of Computer Science*, Houston, Texas, pp. 33-41, 1976.
25. The L4Ka team. L4 experimental kernel reference manual, version X.2. February 2002.
26. R. Ladin, B. Liskov, L. Shirira, S. Ghemawat. Providing Availability Using Lazy Replication. *ACM Transactions on Computer Systems* 10:4, pp.360-391, 1992.

27. L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Comm. ACM* 21:7, pp.558-565, July 1978.
28. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), September 1979.
29. K. Li Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD thesis, Yale, September 1986.
30. J. Liedtke. On μ-Kernel Construction. *Proc. of the 15th ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain Resort, CO, pp.237-250, 1995.
31. J. Liedtke. L4 reference manual (486, Pentium, Pro). Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.
32. J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, T. Jaeger. Achieved IPC performance (still the foundation for extensibility). *Proc. 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pp.28–31, Chatham (Cape Cod), MA, May 1997.
33. Microsoft Corporation and Digital Equipment Corporation. The Component Object Model Specification. version 0.9., October 1995.
34. P. Mockapetris, K. J. Dunlap. Development of the Domain Name System. *Proc. ACM SIGCOMM*, Stanford, CA, 1988.
35. Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification. version 2.6.1, May 2002.
36. Object Management Group (OMG). CORBAservices: Common Object Services Specification*,* 1998.
37. Object Management Group (OMG). Life Cycle Service Specification. version 1.2. September 2002.
38. D. Plainfosse, M. Shapiro A Survey of Distributed Garbage Collection Techniques *Proc. Int. Workshop on Memory Management Kinross*, Scotland (UK), September 1995.
39. D. Potts, S. Winwood, G. Heiser. L4 Reference Manual: Alpha 21x64. Technical Report UNSW-CSE-TR-0104, University of New South Wales, Sydney, March 2001.
40. R. van Renesse, K. P. Birman, B. Glade, K. Guo, M. Hayden, T. M. Hickey, D. Malki, A. Vaysburd, W. Vogels. Horus: A Flexible Group Communication Subsystem. Technical Report TR 95-1500, Cornell University, Ithaca, NY, 1995.
41. A. Ricciardi, A. Schiper, K. Birman. Understanding Partitions and the "No Partition" Assumption. *Proc. 4th IEEE Workshop on Future Trends of Distributed Systems*, Lisboa, September 1993.
42. D. M. Ritchie, K. Thompson. The UNIX time sharing system. *Comm. ACM 17:7*, pp 365-375, July 1974.
43. M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser. Chorus Distributed Operating Systems. *Computing Systems*, 1988.
44. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, Volume 29, No 2, pp.38-47, February 1996.
45. A. Schiper, A. Ricciardi. Virtually-Synchronous Communication Based on a Weak Failure Suspector. *Proc. 23rd International Symposium on Fault-Tolerant Computing Systems*, Toulouse, France, pp.534–543, June 1993.
46. F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4). pp.290–319, December 1990.
47. M. Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pp.198-204, May 1986.
48. M. Shapiro, P. Gautron, L. Mosseri. Persistence and Migration for C++ Objects. *Proc. Third European Conference on Object-Oriented Programming*, pp.191-204, 1989.

49. A. C. Skousen, SOMBRERO: A Very Large Single Address Space Distributed Operating System. MS Thesis, Computer Science and Engineering Department, Arizona State University, December 1994.
50. M. Van Steen, P. Homburg, A. S. Tanenbaum Globe: A Wide-Area Distributed System. *IEEE Concurrency,* pp.70–78 January-March 1999.
51. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. Technical Report TR-819, MIT, March 2001.
52. B. Stroustrup. The C++ Programming Language (3rd Edition). Addison-Wesley, 1997.
53. Sun Microsystems, Inc. EJB specification 2.0.
54. C. Szyperski Component Software – Beyond Object-Oriented Programming. Addison-Wesley, 1998.
55. A. S. Tanenbaum, M. F. Kaashoek, R. van Renesse, H. Bal. The Amoeba Distributed Operating System - A Status Report. *Computer Communications,* vol. 14, pp.324-335, July/August 1991.
56. A. S. Tanenbaum, S. J. Mullender, R. van Renesse. Using Sparse Capabilities in a Distributed Operating System. *Proc. Sixth International Conference on Distributed Computing Systems,* IEEE, pp. 558-563, 1986.
57. F. Torres M. Ahamad, M. Raynal. Timed Consistency for Shared Distributed Objects. *Proc 18th ACM Int Symposium on Principles of Distributed Computing (PODC 99)*, Atlanta, pp 163-172, 1999.
58. P. R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994.