

DC++: distributed object-oriented system support on top of OSF DCE

To cite this article: A B Schill and M U Mock 1993 *Distrib. Syst. Engng.* 1 112

View the [article online](#) for updates and enhancements.

You may also like

- [Visualization of thermal ablation lesions using cumulative dynamic contrast enhancement MRI](#)
Andriy Shmatukha, Benu Sethi, Mohammed Shurrab et al.
- [A deep matrix factorization framework for identifying underlying tissue-specific patterns of DCE-MRI: applications for molecular subtype classification in breast cancer](#)
Ming Fan, Wei Yuan, Weifen Liu et al.
- [Respiratory motion correction for enhanced quantification of hepatic lesions in simultaneous PET and DCE-MR imaging](#)
Matteo Ippoliti, Mathias Lukas, Winfried Brenner et al.

DC++: distributed object-oriented system support on top of OSF DCE

Alexander B Schill† and Markus U Mock‡

Institute of Telematics, University of Karlsruhe, Postfach 3640, D-76021 Karlsruhe, Federal Republic of Germany

Received 24 March 1993

Abstract. The OSF Distributed Computing Environment (DCE) is becoming an industry standard for open distributed computing. However, DCE only supports client/server-style applications based on the remote procedure call (RPC) communication model. This paper describes the design and implementation of an extended distributed object-oriented environment, DC++, on top of DCE. As opposed to RPC, it supports a uniform object model, location independent invocation of fine-grained objects, remote reference parameter passing, dynamic migration of objects between nodes, and C++ language integration. Moreover, the implementation is fully integrated with DCE, using DCE UUIDs for object identification, DCE threads for interobject concurrency, DCE RPC for remote object invocation, and the DCE Cell Directory Service (CDS) for optional retrieval of objects by name. An additional stub compiler enables automatic generation of C++-based object communication interfaces. Low-level parameter encoding is done by DCE RPC's stub generation facility using the C-based DCE interface definition language (IDL).

The system has been fully implemented and tested by implementing an office application. Experiences with the existing system and performance results are also reported in the paper. Furthermore, a former, less transparent implementation of our group using DCE RPC as a pure transport-level mechanism is compared with the described approach. Related C++ extensions and standardization efforts are also compared with our work.

1. Introduction

The OSF Distributed Computing Environment (DCE) [OSF92a-d] is becoming an industry standard for open distributed computing. It offers RPC as its basic communication mechanism both among its decentralized system components and within applications. Moreover, DCE provides a number of supplemental system services including concurrency support, distributed name management, security aspects, and distributed file management. This environment has reached a stable, product-level stage, is becoming widely available, operates in heterogeneous systems, and is the base for many higher-level services such as distributed transaction support. For these reasons, DCE has been the choice for our research and development projects, too. However, like other authors [LET91], we have also observed several deficiencies of the traditional client/server-model supported by DCE:

- *Granularity.* Clients and servers are heavyweight instances. Therefore, it is costly to install them dynamically and it is virtually impossible to relocate them at runtime.

- *Communication.* The communication paradigm in asymmetric: invocations are usually client-to-server round-trip. Server-to-client invocations require cumbersome implementation techniques but are desirable within many applications.

- *Parameter semantics.* RPC reference parameters are dereferenced and their contents are copied by value into the peer's address space. This can lead to anomalies in case of concurrent access to client and server copies. Moreover, parameter passing by remote reference would also be more efficient in some cases.

- *Remote data access.* Data structures managed by a server can only be accessed by invoking the server's data management operations. The structure and implementation code of many applications could be facilitated by enabling direct access to remote data via accessor methods being called remotely. Data abstraction will be maintained then since object access is still via operations only, but no intermediate server entity is involved any more. Although the corresponding implementation will generally not yield significant performance gains, a higher level of programming abstraction is achieved.

- *Entity identity.* Data objects do not have a globally unique identity. Therefore, they cannot be arbitrarily

† E-mail address: schill@ira.uka.de.

‡ E-mail address: mock@ira.uka.de.

addressed from remote locations, one of the reasons for the lack of direct remote data access. Client and server entities only have a global identity by application-specific composition of low-level address and identifier information.

For these reasons, we designed and implemented a distributed object-oriented extension of DCE that addresses these problems. It supports the following features:

- *Fine-grained distributed objects.* The programming model is based on fine-grained, dynamically created C++ objects located at several distributed network nodes. An initial remote location can optionally be specified at object creation time. C++ objects therefore are the basic units of distribution. However, objects can also contain nested C++ data structures, leading to objects of arbitrary granularity.
- *Systemwide identity.* All distributable objects are internally referenced via systemwide unique identifiers based on DCE's universal unique identifiers (UUIDs).
- *Location independent invocation.* Objects communicate by method invocations, no matter whether the peer object is local or remote. The task of locating peer objects is performed by our system. Remote invocations are internally mapped onto DCE RPC. This is achieved by an own stub generation facility working together with DCE's IDL-based stub compiler.
- *Dynamic object migration.* Upon request by the application, objects can dynamically move between nodes, e.g. to co-locate communicating objects or to distribute parallel computations onto different nodes. An important property of our approach is that moved objects can still be accessed in a uniform way, and that concurrent migration and invocation requests are synchronized.
- *Concurrency support.* Object invocations at a given node can be performed concurrently based on multithreaded RPC servers. Moreover, applications can explicitly create concurrent computations by using a thread-related class library; this class library of our system is internally mapped onto DCE threads.
- *Decentralization and dynamics.* The implementation is based on a decentralized architecture. Especially the algorithm to locate objects is fully decentralized. Moreover, object creation and deletion is fully dynamic, and the node structure can also be reconfigured dynamically. Based on these properties, there are no system-inherent scalability limitations.
- *Full integration with DCE.* One of the most important and distinguishing properties of our system is its full integration with DCE mechanisms. It solely uses DCE RPC for implementing interobject communication, and DCE threads for concurrency. Moreover, UUIDs serve as object identifiers, nodes are addressed by DCE binding handles, and the DCE Cell Directory Service is used for optionally registering objects by logical names. Based on DCE, the implementation is highly portable and enables heterogeneous systems interoperability.

The approach is based on concepts introduced by earlier systems such as *Emerald* [BHJ87], *Amber* [CAL89], *Arjuna* [SDP91], and *Amadeus* [HOC91]. However, as opposed to

these systems, it is integrated with DCE mechanisms, an issue that guided many detailed design choices. Moreover, the approach does not introduce any C++ language modifications—therefore, it is a system service on top of DCE and C++ and not a new language and system environment. This fact may improve external acceptance according to the experiences with our project partners. Finally, several implementation details of our object mobility and addressing support contain new features.

It should be noted that the paper does not cover aspects of object persistence or of object-based distributed transactions. Although such features will be of significant importance in practice, for example in office applications, they have not been within a major focus of our project plan yet. Therefore, only very limited facilities for writing objects to disk and for importing them back again are offered by our prototype up to now. Nevertheless, we believe that advanced features can be built directly on top of our distributed object management techniques. Such an enhanced implementation can also be facilitated by RPC-based distributed transaction management products on top of DCE such as *Encina* [ENC92].

In the following sections, we first give an overview of the underlying OSF Distributed Computing Environment and then discuss our system architecture and design choices. Thereafter, we describe details of our implementation and discuss experiences and performance results. We also illustrate the functionality by an example application. Finally, related approaches, including OMG's Object Request Broker, are discussed in more detail and an outlook to future work is given.

2. Basic concepts and system architecture

2.1. OSF distributed computing environment

Originally, the OSF has issued requests for technology (*rft*) for important distributed systems functionality in order to assemble a practical distributed computing environment. Hardware and software vendors as well as research institutes have responded with technical submissions of operable prototypes. The OSF has selected one or multiple systems for each required functionality based on technical criteria and on maturity. Finally, the OSF has integrated all selected components and has determined standardized C programming interfaces for them. Finally, the source code of the resulting DCE has been offered to vendors for porting it to their platforms [OSF92c-d]. As of 1993, the OSF DCE is commercially available on several major systems, including OSF/1, DEC/Unix, IBM/AIX, SunOS, VAX/VMS, and—with limited functionality—on PC/DOS, OS/2, and OS/400; this list is not exhaustive. Interoperability of DCE applications on these systems is guaranteed by common programming and communication interfaces, and by automatic data translation between heterogeneous data formats.

DCE architecture and services. Figure 1 shows the overall architecture of the OSF DCE. All DCE components

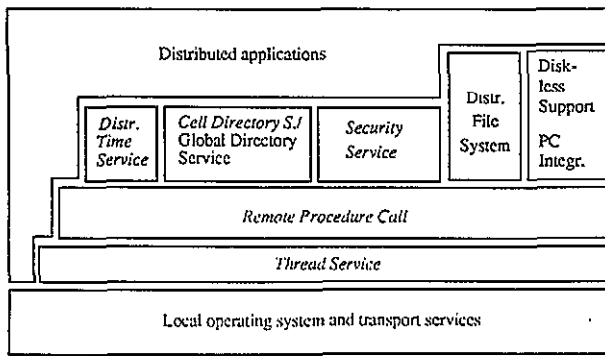


Figure 1. DCE architecture.

are based on local operating system services (e.g. Unix) and transport services (e.g. TCP/IP). Distributed applications make explicit use of *fundamental DCE services* (in *italics* in the figure) via C programming interfaces. The other DCE services are used implicitly via the fundamental services or via modified operating system services.

Fundamental DCE services. The *Threads Service* provides a portable implementation of lightweight processes (*threads*) according to the *POSIX Standard 1003.4a*. Threads enable concurrent processing within a shared address space, and are especially used by RPC for implementing asynchronous invocations and multithreaded servers.

DCE RPC is the major base for heterogeneous systems communication with DCE. It offers significant functionality, including the C-based *Interface Definition Language (IDL)*, explicitly selectable call semantics, nested parameter structures, secure RPC with authentication and authorization based on the DCE Security Service, global (up to worldwide) naming of servers based on the X.500 directory service standard, backward calls from servers to clients, and bulk data transfer based on typed pipes (logical channels). The overall functionality is quite rich, and DCE RPC seems to be increasingly accepted as a *de facto* standard in practice. Most important, it is operable here and now. Like all other DCE components, it is based on an existing implementation that has been submitted after an OSF request for technology (*rft*); this implementation has been part of the *Network Computing System (NCS)* of HP/Apollo and DEC. The OSF adapted the system and determined a set of interface procedures for RPC usage. Within the context of our work, RPC is the most important component. In the following sections, we will make use of *RPC binding handles* that represent addresses of RPC servers.

The *Cell Directory Service (CDS)* supports distributed name management. It is the base for RPC binding and its functionality is integrated into the DCE RPC programming interface via *NSI (Name Service Interface)*. As mentioned above, it provides a robust and efficient directory service implementation for DCE cells, i.e. for domains consisting of logically related network nodes. CDS exploits replication and caching for achieving this functionality. An

advanced CDS programming interface is offered by the standardized *X/Open Directory Service Interface*.

The *Security Service* implements authentication, authorization, and encryption. These mechanisms are tightly integrated with DCE RPC; for example, RPC clients and servers can be mutually authenticated, servers can perform ACL-based access control, and all RPC messages can be encrypted on demand.

Finally, the *Distributed Time Service (DTS)* implements distributed clock synchronization, a common problem in distributed environments. It guarantees that local clocks of participating nodes are synchronized within a given interval. Moreover, synchronization with exact external time sources (e.g. with radio clocks) is supported. This functionality is important for implementing timestamp-based distributed algorithms; it is directly exploited by other DCE components, for example by timestamp-based name update synchronization.

Other DCE services. The *Global Directory Service (GDS)* extends CDS by global, cross-cell naming. It is based on the X.500 directory service standard. Therefore, it enables interoperability not only with other DCE directory servers but also with other arbitrary X.500 servers worldwide. As an alternative, the *Internet Domain Name Service (DNS)* can also be used for cross-cell naming.

The *Distributed File System (DFS)* implements cell-wide transparent distributed file management. Files can be stored at different servers and can also be replicated. Clients, i.e. application programs, can access files by location-transparent names as in a local Unix file system. File access quite efficient based on whole-file caching at the client site. This technique also supports scalability by offloading work from file servers to clients during file access. Interoperability with the widely used *Network File System* is enabled via an *NFS/DFS interface*. DFS is augmented with a *Diskless Support* component; it provides boot, swap, and file services for diskless workstations; this way, they can be integrated into a DCE environment.

Finally, the *PC Integration* provides access to DCE services from personal computers. In particular, print service access, file system access, and RPC communication is supported by this component.

In summary, DCE provides very rich and integrated functionality for distributed and cooperative applications. Moreover, DCE supports heterogeneous systems interoperability and is offered in product quality. DCE can simplify distributed programming significantly already today. However, several extensions of DCE seem to be desirable in the near future. Extensions towards distribute object management have been motivated in section 1 and are the subject of the rest of this paper. Other desirable extensions are integrated distributed transaction support, advanced communication protocols such as atomic broadcast, and integration with other emerging standards.

2.2. DC++ architecture and basic concepts

DC++ uses fundamental DCE services, namely threads,

RPC and CDS. The time service is also operating in our environment but is only exploited internally, especially by CDS timestamps for name entries. The other services can be integrated with our approach in the future. Figure 2 shows the extended architecture of DC++ based on a simple example configuration.

On each node, a DCE RPC demon is installed and serves RPC invocations. Moreover, threads are used for handling concurrent invocation requests and can be exploited by the application with object-oriented class capsules. Distributed objects are allocated at various nodes and have local and remote interobject references. A remote reference is implemented by a proxy indirection; a proxy contains a location hint for the referenced object and transparently forwards invocations based on DCE RPC. Each node maintains a hash table for mapping the global object identifiers within incoming invocations onto actual storage addresses of C++ objects.

One or more DCE CDS servers are also part of the environment. They store proxies for objects that have been registered by a logical name. This way, peer objects can acquire a proxy for a remote object by handing its name to CDS. Possible replication of CDS is made transparent by DCE.

2.3. Proxy management and object access

A proxy is installed whenever a node learns about the existence of a remote object. This is the case when a reference to a remote object is passed as a parameter of an invocation. In addition, when an object moves and has references to remote objects, proxies must be installed at the destination node for each reference. Moreover, a moving object leaves a proxy at its former location. This results in forwarding chains of proxies that are followed when an object is invoked. The location information within the whole chain is updated upon stepwise return of the call. This way, forwarding chains will usually have a length of only one hop—assuming that invocations are more frequent than migrations.

The alternative of immediately updating all remote proxies whenever an object moves would improve

invocation performance of mobile objects and is found in some distributed *Smalltalk* implementations (see [DEC86], for example). However, it has two major problems: (1) migrations are more expensive, and the approach is not scalable since migration costs increase significantly in large systems; (2) each object would have to maintain backward references to all proxies; this requires significant storage space and leads to orphaned references in case of node failures.

As a trade-off between a pure forward addressing technique and an immediate proxy update approach, we integrated an additional technique: objects register their current location at their ‘birthnode’, i.e. at the node where they were created. That is, after having performed a migration, an RPC is sent to the birthnode containing the new location. From each proxy, the birthnode’s address can be derived by explicitly registering the node identifier with each proxy. Therefore, an object can be located by either following the forwarding chain or by querying the birthnode. The first option is used in the fault-free case. However, if a forwarding chain is broken by a failed intermediate node, the birthnode is queried for an object’s location. In the normal case, forward addressing is more efficient—it requires one RPC if the location information is up-to-date, while the birthnode option would require at least two RPCs for locating the object at a third-party node.

Example. Figure 3 shows an example of a dynamic object management scenario with four functional steps. The boxes represent network nodes, and the arrows between them show references to a given mobile object, say O, based on proxies and birthnode addressing. The figure represents the status of the references at the end of each step. It is assumed that O was created at N1. It then moved to N2, N3, N4, and finally N5 in step 1. The birthnode, N1, maintains a direct reference to the object’s location while the other nodes use proxy-based forwarding addresses. In step 2, the object has been invoked from a caller at node N2. It is located by following the proxy chain. Thereafter, the whole chain is updated and now all proxies contain the correct location of O. The object is then moved from N5 to N2 in step 3. A forwarding address is installed at N5, and

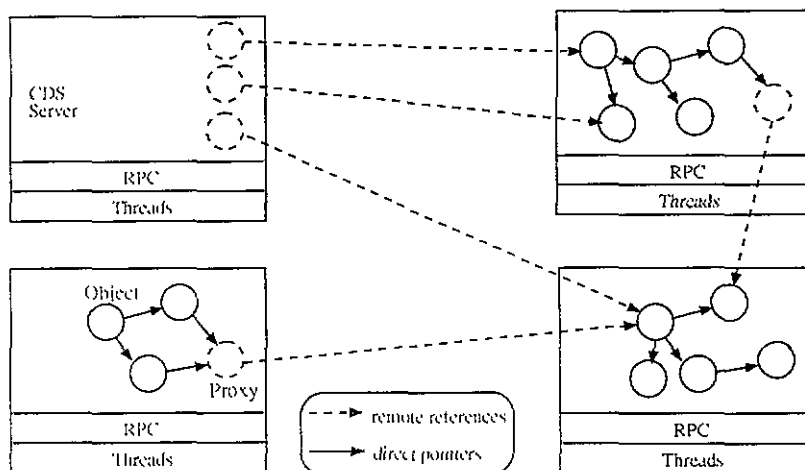


Figure 2. DC++ architecture.

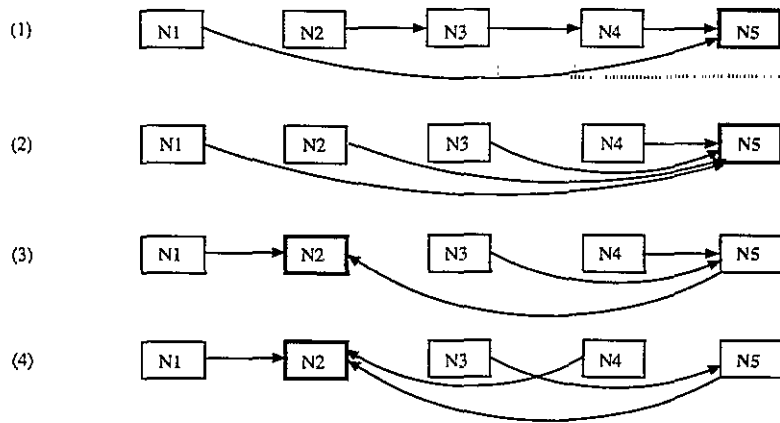


Figure 3. Example of the forwarding chain update mechanism.

the birthnode N1 is informed about the new location—therefore, N1 is able to update its location information as shown in the figure. Finally, O has been invoked from N4 by locating it via the forwarding address, i.e. via N5. The resulting proxy update of N4 after the return of the call is given in the figure.

As an additional extension, the obligations of an object's birthnode can be transferred dynamically and explicitly to an alternative node; this node is then called *guardian node* of an object. For example, when an object permanently moves into a different network domain or, more general, into a different organizational area such as a foreign DCE cell, it might perhaps hardly ever return to its birthnode. By transferring responsibility for the object to a different guardian node, unnecessary dependencies concerning the former birthnode in a remote cell are avoided. For example, a network failure between the corresponding cells will not leave the object unlocatable any more. Moreover, inefficient wide area network access to a distant birthnode can be avoided this way.

2.4. Object mobility

Object migrations are requested by the application by calling an automatically generated method of an object. Basically, a migration consists of the following internal operations (see figure 4 for an example of moving an object O1 from node 1 to node 2). (1) First, the object to be moved is locked by a semaphore. This is required for

synchronization with ongoing invocation requests. (2) Then the object is replaced by a proxy at the source node and unlocked; however, the object data is still kept for failure recovery. (3) Next, an RPC installation operation is invoked remotely at the destination node, passing the object's data as an RPC parameter. All object data structures are defined in IDL so that marshalling and unmarshalling can be done completely by DCE RPC. (4) The destination node installs the object and inserts its identifier into its hash table. If there has been a proxy before, it is replaced by the object. (5) Upon receiving the reply of the remote installation RPC, (6) the source node of the migration informs the birthnode about the new location. (7) Finally, the original object data is deleted at the source node. As a prerequisite, we assume that the moving object's class is available at the destination node based on class replication.

This approach has some interesting characteristics: although migrations and invocations are synchronized by semaphores, locks are not held at the source node until the migration has fully completed. This is not necessary as the source node can immediately forward invocations when the proxy has been installed. The birthnode is informed only when a migration is completed so that it does not receive incorrect information if a migration fails. If an object should be located via the birthnode in the meantime, the operation would still work: the birthnode would direct the invocation to the former location of the object which then already has a proxy pointing to the new destination.

Migration requests can also go to remote objects. In this case, the request is forwarded like a usual method invocation until it reaches the destination node. Then the migration is performed as discussed above.

Instead of specifying an absolute destination, a relative migration method is also supported. It takes a peer object as a relative destination specification, locates the given object as discussed above, and then performs the regular migration to the found location.

Migration with heterogeneous class structures. A special problem is object mobility in applications with heterogeneous class structures, i.e. in environments where implementations of a given class differ at various nodes or

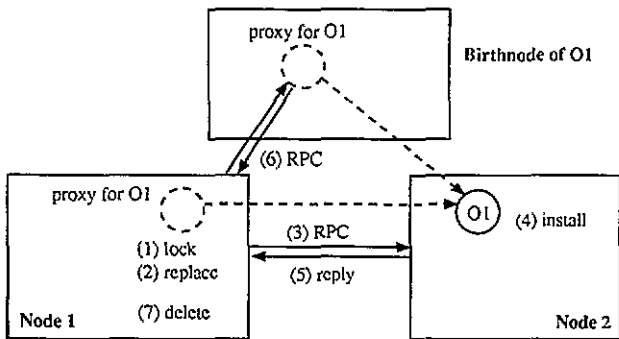


Figure 4. Object migration.

where specific classes are not available at all nodes. This is also an important issue in practice as it is not possible to replicate all classes everywhere for large applications.

Our current prototype allows a first, simple solution to this problem: classes need not be totally replicated throughout the system, but object mobility is affected by such limited class replication. In particular, an object can only be moved to a location where its class and all classes required by associated object references are present. Based on a careful application design, functional domains with selected subsets of classes can be configured, whereby different subsets may share a small number of common classes in order to interact. As argued later, dynamic class installation at new locations would improve this approach with enhanced flexibility. However, as it requires dynamic code installation and imposes specific problems at the operating system level, we did not provide such a feature yet.

The coexistence of different implementations of a given class is difficult to support in the context of C++ due to the limited separation of class definition and implementation. Based on language modifications and significant compiler extensions, such functionality could be provided, but was not part of our work up to now. We plan to focus on this problem within future research efforts in a wider context, considering distributed versions of object classes. Such versions may evolve dynamically based on the lifecycle of objects. An overall solution will be required to convert mobile objects between versions of limited compatibility, to control distribution of versions, and to manage version updates in the context of existing object instances.

2.5. Class structure

The described functionality is offered by a set of classes shown in figure 5 together with the most important relationships with application and system components. The *Object_Reference* implements all required data and basic functionality for remote object access and object migration. For each application class with distributed instances, an auxiliary wrapper class is required. This class (*<Wrapper_Class>* in the figure) is derived from *Object_Reference*. It mainly implements the proxies with code to distinguish between local and remote invocations. However, an instance of a wrapper class is also present for each local object as an external capsule. The wrapper class

offers the required code to migrate objects with application-specific data structures, too. In the case of remote invocations and migrations, it makes direct use of DCE RPC as indicated in the figure. Most importantly, this class can be generated automatically based on an interface description as described below.

The actual implementation of each application class, denoted *<Application_Class>* in the figure, is identical with a regular class implementation as found in a corresponding non-distributed application. Each auxiliary object of a wrapper class has a local reference to the associated object of the application class. As the application classes shall remain unaffected by the aspect that they will be distributed, they are not derived from a common superclass. Instead, all required distribution functionality is provided by the wrapper classes.

Network nodes are also represented by objects, for example to specify destination locations of migrations. The derived class *Node* offers the corresponding functionality. In particular, each object of class *node* contains the required address information as a DCE RPC binding handle. An application only uses objects of class *Node* and of wrapper classes directly. Several other auxiliary classes are part of the system, namely classes for threads, semaphores, hash tables, and directory service name entries.

The concrete structure, use, and automatic generation of these classes are described below.

3. Implementation

The implementation was done on a network of DECstations 5000 and 5240 under Ultrix 4.2, using AT&T C++ 2.1 and DEC's C++ compiler, named cxx. Our DCE prototype has been provided by DEC (version 1.0). Basic transport-level communication is performed by TCP/IP, UDP/IP or DECnet on an Ethernet. The actual communication protocol can be selected at RPC initialization time. For implementing the stub generation, the Unix tools *awk* and *sed* have been used; the reasons for this choice were mainly pragmatic as we did not have the resources for implementing a complete compiler from scratch. The non-standard language extensions on which the interface definition and stub generation is based are also a consequence of this choice. For a product-level solution, the direct use of C++-style interface definitions would of

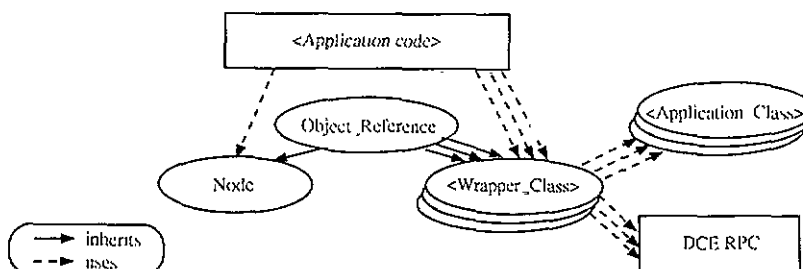


Figure 5. Class/module structure.

course be recommended. Actually, this also represents a future direction of our work. However, with our simplified approach, C++ parsing would not have been possible.

The following subsections describe our implementation. We first discuss the system classes provided by our approach and then show how auxiliary application classes are generated. Then, performance and experiences are discussed, after presenting an example application.

3.1. System classes

Object_Reference. Much of the functionality of our approach is given by class *Object_Reference*. It has the following (simplified) structure:

```
class Object_Reference {
private:
    uuid_t    object_id; // object UUID
    char      *object_name; // object name
    Node      *suspected_loc; // suspected (NOT
                           // necessarily current) location
    Node      *creating_node; // creating node of
                           // object
    pthread_mutex_t mutex; // semaphore
public:
    Object_Reference(char*); // used for application
                           // objects and nodes
    Object_Reference(RPC_Obj_Ref*); // used for
                           // migrated objects and location hint evaluation
    ~Object_Reference(); // destructor
    void lock(); // lock semaphore
    void unlock(); // unlock semaphore
    uuid_t get_oid(); // return id
    char* get_name(); // return name
    Node* get_cre_loc(); // return birthnode
    Node* get_sus_loc(); // return location hint
    void update(Location*loc); // update location
                           // hint
    int migrate(Object_Reference*); // relative
                           // migration
    virtual int migrate(Node*); // absolute
                           // migration
    virtual Location* locate(); // locate objects
};
```

Objects of this class contain a DCE UUID to identify them (*object_id*). It is generated by the constructor using a DCE system function. They also have an optional name (*object_name*) that is registered with CDS. The location hint of proxies and the birthnode of the corresponding object are stored in separate instance variables, *suspected_loc* and *creating_node*, respectively. In principle, it would be possible to derive the birthnode from the object UUID (the node address would be part of the UUID to make it globally unique); however, this did not work with the given DCE implementation. The semaphore for synchronizing invocations and migrations is also part of *Object_Reference*.

Most of the methods are pretty straightforward. It may be worthwhile to note that the second constructor is used to install proxies when a new object reference is passed to a given node. The required address information is provided via a parameter of type *RPC_Obj_Ref* that contains the internal RPC address information for an object's location. The *update* method is called when a proxy chain is updated

upon return of a remote invocation. The relative migration method is application-independent as it only calls the absolute migration method after having located the object. However, the absolute migration method that performs the physical migration must be provided by the application-specific subclass and is therefore virtual. The method to locate an object is implemented differently by application objects and nodes and is therefore also virtual.

Node. An object of class *Node* is created locally for each node that is known by a given peer node, including itself. It provides the required information to invoke an RPC at a suspected object location. This includes a unique identifier for the node, and a corresponding RPC binding handle.

```
class Node : public Object_Reference {
private:
    uuid_t loc_id; // id from binding handle
    rpc_binding_handle_t binding_handle; // DCE
                           // binding handle
public:
    Node(char*); // nodes defined by application
    ~Node(); // destructor
    Location* locate(); // return suspected_loc
                           // from base
    void Shutdown(); // stop RPC listener
    uuid_t get_id(); // get node id
    rpc_binding_handle_t get_bh(); // get binding
                           // handle
};
```

The constructor of this class creates a representative for foreign nodes if a node name is given. In this case, a CDS inquiry is performed for importing the required binding handle and identifier information (using the CDS interface operations *rpc_ns_binding_import_begin*, ..., *_next*, ..., *done*). Otherwise, the representative for the local node is generated. In this case, the constructor exports the local binding information to CDS (using *rpc_ns_binding_export*) so that other nodes can import it. The *locate* method just returns the suspected location of the superclass component as nodes never move. In addition to basic access operations for instance variables for internal use, a method to shut down the RPC server of a node is provided. It is useful for remote housekeeping within an application. It is implemented by calling a remote DCE RPC management function at the actual location. Note that all other methods can be implemented locally—except the interaction with CDS within the constructors. The implementation of the other application classes, namely of the threads and hash tables are relatively straightforward and they are therefore not described in closer detail.

3.2. Application classes

Class structure. The actual implementation of the 'real' application classes is similar to ordinary C++. However, the auxiliary application classes, i.e. the capsule classes around the real classes, are generated automatically. They basically have the following class structure (of the wrapper class *gen<A>*) for an application class *<A>*:

```
class gen<A> : public Object_Reference
```



```

private: gen<A> (char*); // internal constructor for proxies
public:  <A> *obj_ptr; // pointer to application object
        gen<A> (<A>_data*, RPC_Obj_Ref*); // used
        within manager after migration
        gen<A> (RPC_Obj_Ref*); // used within location
        hint evaluation
~gen<A> (); // destructor
int migrate (Node*); // absolute migration
static gen<A>* get_ref_by_name (char*); // get
        reference to existing object
// for all application-specific constructors:
gen<A> :: gen<A> (. . . tt>, Object_Reference *or
= here, char* name = ""); // regular application-specific
        constructor
// for all application-specific methods:
<result_type> gen<A>::<method_name> (. . . ,
Object_Reference *or = NULL, RPC_call_data *cd = NULL);

```

Each object has an internal pointer to the actual object data of class <A> (*obj_ptr*). This pointer is dereferenced for all local invocations, passing them to the real object. Two internal constructors are used for installing objects after a migration and for generating proxies, respectively. <A>_data* is a pointer to the data structure of the application class, however given in C instead of C++ for conformance with DCE's IDL. The implementation of the *migrate* method also accesses this data structure definition in order to perform the remote object installation by an RPC.

Moreover, a method is offered to retrieve an object reference by name. This is possible for all objects that have been registered with CDS when they were created. The method performs a remote *rpc_ns_binding_import* at a CDS server in order to retrieve the required binding information for the object. However, this operation is rather heavyweight due to relatively limited CDS performance. Therefore, it should only be used for selected objects that are part of the coarse-grained configuration of an application. References to other objects are learned by remote nodes via parameter passing by object reference.

Each application-specific constructor is extended by an optional initial location parameter, defaulted to the local node. This way, object creation at a remote node can be performed. A second optional parameter is used for specifying an object name to be exported to CDS, matching the import operation discussed before.

All application-specific methods also get additional parameters. The first one specifies an optional location where an invocation shall be performed. Usually, it is not used as we pursue the goal of location independent invocation. In this case, the object is located at its current location and the call is performed there. Otherwise, the object is actually moved to the given location before invoking it. This option is useful for distributing parallel computations to different nodes, for example. The second additional parameter is important for updating proxy chains. When an RPC returns, it carries the actual location of the invoked object and is evaluated by each intermediate node until the call is returned to the calling object.

Automatic generation. Obviously, the template feature of

C++ is not sufficient for generating the described code automatically. Therefore, we had to implement our own stub generation facility.

However, we did not have the resources to write a full parser and backend for general C++ or C. Therefore, our idea was to specify an interface definition notation similar to IDL, however, with some limitations in order to make it easily parseable (see below). Based on this notation, we implemented a simple parser using the Unix tool *awk*. This process generates *sed* command files that replace the required variables within predefined class templates as shown above. For example, the application class name, but also the application-specific parameters are replaced this way. Moreover, for all required method implementations, similar templates are provided.

Implementation templates. As an example for the method implementation templates, here is the template code for an arbitrary application method invocation (<A> is the application class, gen<A> the wrapper class, <M> is the method name, and <P> are the parameters):

```

//////////////////// method of class gen<A>
//////////////////// // method call within application
void gen<A> :: <M> (<P> Object_Reference *or,
                  RPC_call_data *cd) {
    error_status_t st; // status
    string str_loc; // location as string
    Location *sus_loc, *loc_from_cre; //
    suspected location, location received from creating node
    int mig_result; // status of migration
    idl_boolean called_from_manager; // whether
    called from internal RPC manager (within proxy chain)
    if (or) mig_result = this->Migrate (or); // check if
    object is to migrate first

    if (cd) called_from_manager = 0true; // keep
        caller in mind else called_from_manager = false;
    lock (); // lock call semaphore
    if (obj_ptr) { // object is local
        if (called_from_manager) {
            strcpy ((char*)cd->node, (char*)my_loc-
                >get_sb ()); // tell manager who I am
        }
        obj_ptr-><M> (<P>); // perform local
                           invocation
    }
    unlock (); // free semaphore
    else {
        sus_loc = get_sus_loc(); // remote call fi get
                                suspected location
        unlock (); // free semaphore
        CREATE_HINT_DATA // macro to pack location
                           hint data
        if (!called_from_manager) { // being called from
                                application so create data block
            cd = (RPC_call_data*)(new RPC_data (this));
        }
        gen<A>_<M> (sus_loc/Eget_bh (), cd, <P> &st);
        if (st != rpc_s_ok || *(cd->status) != OK) { // check
                                                    errors
            if (get_cre_loc() == my_loc ||
                get_cre_loc() == sus_loc) {
                cerr << "no sense to ask creating
                    node" << endl;
                exit (1);
            } // try creating node:

```

```

        get_loc_by_id (get_cre_loc ()/Eget_bh (),
                      *(cd->oid),
cd<ms>status, str_loc, &st);
        if (st != rpc_s_ok || *(cd/Estatus) != OK)
            exit (1);
        loc_from_cre =
            Location::get_loc_by_string (str_loc);
        if (loc_from_cre == sus_loc) { // see if
            there's a chance left
            cerr << "creating node has no better
                    info" << endl;
            exit (1);
        }
        lock ();
        update (loc_from_cre); // update info
        unlock ();

        gen<A>_<M> (loc_from_cre/Eget_bh (),
                    cd, <P> &st); // last chance
        if (st != rpc_s_ok || *(cd/Estatus) != OK)
            exit (1);
    }
    update (cd); // update with node where call
                  REALLY was made
    DELETE_HINT_DATA // delete data from remote
                      call
    if (!called_from_manager) {
        // being called from application
        // so delete data block previously created
        delete ((RPC_data*)(cd));
    }
}

```

The actual implementation performs some parameter modifications of <P> before filling the template variables within the body of the method, for example to manage proxy installation for object reference parameters. The method itself first checks whether the object should be migrated in order to perform the call at a specific location. Then it is checked whether the call is from an RPC manager, i.e. a recursive call within the processing of a proxy chain. Otherwise, the call comes directly from the application. There are slight differences in handling the address information in both cases.

Now the method can check if the call is local; in this case, the local representation of the class is invoked by the corresponding local method. Otherwise, an RPC is performed at the suspected location. The RPC invokes a C function with the same name as the method. At the destination node, this function eventually invokes the given method recursively. If it was the first invocation (directly from the application), the internal RPC call data must be initialized, too. If the remote invocation fails (indicated by a bad return status), the guardian node (which may be the birthnode that is also called creating node) is queried for the object's location. An alternative invocation attempt only makes sense if the returned location information is different from the existing hint and from the local node. Finally, after the call has returned, the proxy information is updated.

Similar templates have been implemented for methods that return a result type, for constructors, and for all auxiliary methods related to migration.

3.3. Example application

As a testbed for our system we implemented a small

application, modelling an office scenario, see figure 6.

A 'head hunter' creates a number of customers and for each customer a form is created that has to be filled out. After initializing and filling in some basic data such as the customer name the head hunter is done. The form filler periodically checks the forms' state and as soon as they are available for further processing it requests them being migrated to its own node and fills in more data. Likewise the 'rate computer' periodically checks whether the form filler is done, as soon as the form is in state 'form_filled_in' it requests a migration to its own node. Then it computes the rates for the customers (we model insurance policies being filled out). Finally the filled-in policies are migrated to the 'policy sender' upon request by the policy sender. Various migrations are involved in this scenario. Moreover, the form filler, rate computer and policy sender access the forms remotely to find out in which state they are. Therefore the application also makes use of remote method invocations.

To illustrate the development of an application with DC++ we will now go through the steps necessary for developing an application. As a running example we will use that office scenario example application.

To develop an application one has to go through the following steps:

- write the application classes
- write a corresponding IDL-description
- write a corresponding DC++ description

After those steps, the migration code is generated by running the DC++ stub generator and the IDL-compiler. Finally, the C++ compiler and linker are used to create the application code.

For illustration we will show an excerpt of the class headers of the application classes. The insurance form class, called OrigInsurance_Form looks as follows:

```

class OrigInsurance_Form {    uuid_t customer_id;
    State_Of_Form status;
    idl_long_int    rate_per_month;
    idl_long_int    age;
    idl_long_int    volume;
public: OrigInsurance_Form () { status = NOTHING_DONE;
                                }

    ~OrigInsurance_Form () {}
    idl_long_int get_rate      () { return
                                rate_per_month; }

    void    fill_in (idl_long_int, Customer*);
    void    compute_rate ();
    void    send_policy (Customer*);
    State_Of_Form get_status  () { return
                                status; }
};

```

An excerpt of the corresponding IDL description file follows:

```

void OrigInsurance_Form_Migrate ([in] handle_t bh,
    [in] uuid_t oid,
    [out,ref] error_status_t *status,
    [in,string] char *loc
);
void OrigInsurance_Form_fill_in ([in] handle_t bh,
    [in,out] RPC_call_data *data,

```

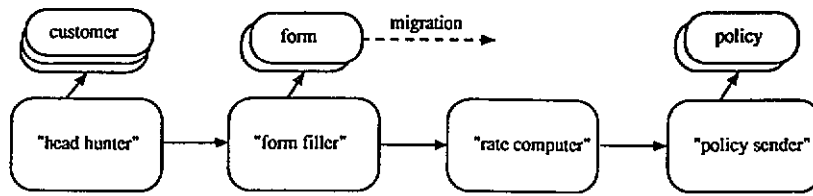


Figure 6. Example application.

```

[in] long v, /* desired insurance volume */
[in] RPC_Obj_Ref r /* reference to customer */
);

```

In `OrigInsurance_Form_Migrate` the application programmer must describe the object's data to enable parameter marshalling for migration. Apart from that, each method of the class has to be represented in the IDL interface description, shown here only for the method `'fill_in'`.

The corresponding section in the DC++ description file has the following outlook:

```

OrigInsurance_Form:Insurance_Form
                                :OrigInsurance_
                                Form_data
idl_long_int :get_rate :
void :fill_in :idl_long_int
vol, Customer* cust
void :compute_rate :
void :send_policy :Customer* cust
State_Of_Form :get_status :

```

From that description a new class will be generated†:

```

class Insurance_Form : public Object_Reference {
    //////////// parameter independent members ////////////
private:
    Insurance_Form (char*);
public:
    OrigInsurance_Form *obj_ptr;
    Insurance_Form (OrigInsurance_Form_data*,
                    RPC_Obj_Ref*);
    Insurance_Form (RPC_Obj_Ref*);
    ~Insurance_Form ();
    static Insurance_Form* get_ref_by_name (char*);
    static Insurance_Form* loc_hint_eval (RPC_Obj_Ref);
    int migrate (Location*);
    //////////// parameter dependent members ////////////
    Insurance_Form (Object_Reference *or=here,
                    char* name="");
    idl_long_int get_rate (Object_Reference
                           *or=NULL, RPC_call_data *cd=NULL);
    void fill_in (idl_long_int vol, Customer* cust,
                  Object_Reference *or=NULL, RPC_call_data
                           *cd=NULL);
    void compute_rate (Object_Reference
                       *or=NULL, RPC_call_data *cd=NULL);
    void send_policy (Customer* cust,
                      Object_Reference *or=NULL, RPC_call_data
                           *cd=NULL);
    State_Of_Form get_status (Object_Reference
                              *or=NULL, RPC_call_data *cd=NULL);
};

```

† In this case named `Insurance_Form`. The names of the generated classes can be customized in the description file.

Looking at the method `fill_in` in the generated class, one might note that it now has two more parameters, *or* of type `Object_Reference` and *cd* of type `RPC_call_data`. *Or* holds the reference to the RPC server to be contacted whereas *cd* is used to identify the object to be called. The data structure `RPC_call_data` looks like this:

```

typedef struct { uuid_t *oid; error_status_t *status; string
node; } RPC_call_data;

```

Oid is used to identify the object to be called, *status* is used to return an error status and *node* holds the string binding of the node where the method was actually executed. When the RPC call returns, this information is used to update the location hints of the nodes that were part of the forwarding chain. Since *cd* is only passed as a pointer, the `RPC_call_data` data structure can be easily changed without affecting the rest of the implementation. It is possible, for instance, to add a timestamp in order to make updates of location hints only when the returned location hint is not older than one that has been previously obtained.

As opposed to a conventional RPC-based implementation of the example, the outlined solution provides a number of benefits. First, the office procedure can be represented by a first class object itself. This allows for locating the object, for performing remote status queries, and for explicitly controlling execution by remote management commands. With a conventional client/server implementation, additional and complex functionality would have to be realized on top of RPC. Moreover, attached data objects can also move between processing sites on demand; they are also modelled as mobile objects. Alternatively, they can be accessed remotely, for example if migration of a large object is prohibitively expensive. Finally, the tight integration with C++ facilitated the overall implementation of the scenario from the software engineering point of view.

3.4. Performance and experiences

In this section we want to look at the performance of the system. Moreover, we will discuss the general experiences gained by designing and implementing DC++.

Performance. To gather performance data of our DC++ system we chose to time migration within our sample application. For that purpose we used different amounts of data within the form that is filled out and migrated in the application; the results are shown in figure 7. First, we timed the migration of the form containing only system

time [ms]	Empty	100 bytes	1000 bytes	10000 bytes
Minimum	16	16	16	59
Maximum	176	254	176	543
Median	31	31	35	82

Figure 7. Performance.

relevant data that is inherited by each application class such as the object's ID. Then we increased the additional user data from 100 bytes to 1000 bytes and finally 10000 bytes. All reported times are in milliseconds and shortest, longest and median time to complete the migration are shown. The measurements were made on lightly loaded DEC 5000 stations connected by an Ethernet. The communication protocol chosen to be used for RPC was UDP.

The figures show that the overhead incurred by DC++ is negligible. In previous measurements in the same environment we had measured about 6 ms for a raw empty RPC call. Moreover, the figures show that the migration time is not very sensitive to the amount of data being transferred. This, however, must be attributed to the tested data sizes—which all fit into a UDP packet—and the type of data used (arrays) which allows the IDL-compiler an efficient and fast encoding and decoding. A median time of 35 ms for the migration of an object containing about 1000 bytes makes DC++ suited for use in real applications.

Experiences. Based on our implementation and on the example application, we gathered a number of important experiences:

- *Object model.* The object model seems to be more suited for distributed programming than the traditional client/server approach. Within our application (and within former projects), we observed that a uniform object model facilitates application design. Location independent invocation based on globally unique object identifiers makes distribution transparent to a large degree—except the problem of failure handling, of course. Remote object reference passing contributes to this fact as it is a natural passing mechanism in local applications, too.
- *Object mobility.* Mobility is a beneficial feature of distributed object-oriented approaches. It allows for modelling physical data transfer (such as document shipping) at a very high level of abstraction. Moreover, it provides explicit control of distribution when an application requires it (e.g. to co-locate communicating objects).
- *Use of RPC.* In spite of our criticism of RPC, this mechanism has proven to be a workable base for implementing such a distributed object management facility. Based on the one-to-one mapping of method invocations onto application-specific RPCs, most of the parameter marshalling problems were just passed down to the RPC level; this facilitated our implementation significantly. Moreover, the recursive implementation of the algorithm to locate objects based on RPC has proven quite elegant and easy to test and maintain. It would be more efficient to send results back to the caller via a direct

message from the callee, but this slight disadvantage is outweighed by the chance of updating all intermediate location information.

- *Use of standards.* The use of DCE as an industry standard also had many advantages. As opposed to *ad hoc* mechanisms, the environment was rather stable. Moreover, we did not have to deal with heterogeneity problems; they are hidden by the RPC protocol. Finally, the high portability of applications based on a standardized platform is an important advantage in open systems.
- *Use of system services.* The use of system services as offered by DCE made a rapid implementation possible. In particular, we exploited CDS for node and object management and threads for concurrency support—in addition to RPC, of course.
- *Interface definition.* Our interface definition and stub generation approach is only an intermediate solution. Its capabilities regarding the language syntax are limited. Moreover, a partially redundant specification must be given. Therefore, a major goal of our future work is a full C++-based interface definition and stub generation facility. The templates defined for our stub compiler can be reused for such a solution. An eventual goal might of course be the integration of all mechanisms into DCE itself.

3.5. Limitations

Although we think that our current system is already usable for application development, it still has its shortcomings.

Most notably there are currently two description files the user has to write: the IDL-description to be used by the IDL-compiler and the DC++ description file that is used to generate support code for migration and remote access. However, this is not a design limitation, since it is possible to generate the DC++ description file from an augmented IDL-description. Work is in progress to enhance the IDL-description (by defining so-called ACF-attributes) to allow the description of C++ class interfaces. From such a description the DC++ description file could be generated automatically, maybe even by the IDL-compiler itself. This would render the need for writing a second (redundant) description unnecessary, which—apart from being a nuisance—also introduces the possibility of errors.

Another limitation of the system related to the IDL-description restricts the range of data types that can be used in migratable objects. Since the IDL-description must conform with the corresponding C++ classes it it currently impossible to support class hierarchies with virtual data member functions.

As noted above, only object instances are mobile;

object classes, especially their implementation part, cannot be transferred between nodes in our prototype. Such functionality would require dynamic code linking into the destination's address space. Moreover, it would of course be limited to relatively homogeneous, binary-compatible systems.

Finally, although IDL allows complex data types such as linked lists it is currently impossible to migrate them. The reason is that the RPC runtime system allocates some parameters of a RPC on the server's stack and deallocates them once the call has completed. This is desired behaviour for RPC and for a remote method invocation as well, however, when sending object data to another node to install the object there, i.e. when a migration is being performed, the data on the (RPC) server's side must persist. For simple flat data types DC++ can simply do the allocation itself, when more complex (user-defined) data structures are involved, though, it would be necessary to have access to the IDL-description to take appropriate action. A possible solution would be to allow an attribute for an RPC call specifying that parameter data have to persist after the call completes thus enabling migration. How to do this exactly is another topic being investigated.

Many of these problems can of course also be attributed to the very nature of distributed systems—basically, it is hard to implement functionality found in local systems via the network transparently. Our own experiences have shown that important basic functionality—such as location independent object invocation—can be achieved based on a significant implementation effort. On the other hand, we also believe that it does not make sense to provide virtually all 'local' functionality remotely at any price in terms of implementation effort.

4. Related approaches

Several other distributed object-oriented prototype systems have been implemented; a survey is given in [CHC91]. For example, distributed C++ extensions have been implemented by the *Amber* system [CAL89], the *Amadeus* system [HOC91] within the European ESPRIT project *COMANDOS*, the *Arjuna* system [SDP91], *Electra* [MAF92], *Peace* [SCP92], and *Panda* [ABB93]. A similar approach has been the base of the *ANSA* project [ANS89], leading to the *ANSAware* system.

Amadeus offers a general distributed C++ implementation but required many compiler modifications as reported by the authors. The *Amber* system integrates local multiprocessor parallelism with distributed programming. These facilities are achieved by the use of a dedicated kernel named *Topaz*. The *Arjuna* approach focuses on distributed transaction support for objects but does not provide object mobility. *Electra* offers remote method invocation by defining its own interface description language (*Snoopy*). For migration, however, the user has to write dump and undump methods to pack the object's data. The *Peace* system is based on the specialized operating system kernel with the same name. It is among the most

far-reaching approaches with full stub compiler support, mobility, and alternative implementations on a multiprocessor and on a distributed system. The *Panda* system implements distributed object management based on distributed shared memory at the object level; remote objects are fetched for invocations based on a kernel-level access fault. This makes sophisticated performance possible but requires kernel modifications.

Implementations of distributed object-oriented systems based on different or new languages are *Emerald* [BHJ87], *Distributed Smalltalk* [BEN87], and *LII* [BLA90]. They have introduced the major concepts and have shown that they can be implemented efficiently. However, due to the specialized languages, these and other systems have been limited to a dedicated domain of users.

Finally, we would also like to note the standardization effort of the *Object Management Group* (OMG) [OMG92], especially concerning the Common Object Request Broker (CORBA) [OMG91]. This ambitious work aims at providing a global distributed/persistent object management framework, including DCE technology, language bindings, but also services from other areas like databases. CORBA itself enables remote object invocations, offering a C++ language binding. However, it does not support object mobility. The other parts of OMG's proposed architecture are still within early stages of design or development.

Our work has emerged in parallel with some of these projects but focuses on rather different goals, namely on the ease of system implementation at the user level, on the use of existing, unmodified compiler and communication facilities, and—more recently—on the integration with standards. The resulting integration with DCE at the implementation level mainly distinguishes our approach from other systems. This integration provides benefits for the application programmer: heterogeneity of network nodes and protocols is handled transparently, applications are highly portable, and the underlying environment is rather stable. Moreover, many benefits for our implementation work resulted, too. Most importantly, we did not have to write low-level communication code, e.g. to access sockets; this task had turned out to be quite time-consuming within our former implementation. Another benefit was the existence of supplemental services such as threads and CDS. This way, concurrency support and object name management required just a few lines of code rather than a full thread package or directory server implementation. The achieved performance is fully acceptable within our cooperation project with DEC. It can definitely be improved by some implementation optimizations, but it will not reach the limits of microkernel implementations like *Amoeba* [TKR91], of course. This is the price to pay for the use of standards—but within our context of work, it seems to be worth paying it.

5. Conclusion

This paper described the design and implementation of a distributed object-oriented extension of the OSF

Distributed Computing Environment. The major features of the approach, location independent object invocation and object mobility, have proven very useful for application development. Moreover, the use of DCE as a standard has provided significant implementation benefits.

The described implementation grew out of experiences with an earlier system that attempted to provide the same functionality. However, we implemented this former prototype directly on top of TCP/IP [SCM93] without DCE support. Our experiences compared to DC++ can be summarized as follows: first, it was much harder to provide basic system functionality for naming, object addressing and multithreading. With DC++, it became obvious that standardized system support for such problems—such as provided by the DCE services—is a crucial prerequisite for a rapid yet stable implementation. Secondly, remote method invocations could not be offered in a transparent way as supported by DC++—the application programmer had to provide code for encoding and decoding the parameters. With DC++, much of this task can be left to the IDL compiler and stub generator. Moreover, interobject communication was only possible in homogeneous systems due to data representation mismatch problems. With DC++, DCE performs transparent data transformations based on its 'receiver makes right' scheme [OSF92c]. Finally, our former implementation did not achieve a satisfying performance due to the use of an intermediate 'message distributor' process per physical node. With DCE RPC, interprocess communication is direct as soon as a full binding handle for a peer server is available. This contributes to the acceptable performance numbers outlined above.

Future work will address an even tighter integration with DCE, especially concerning the interface definition language. Jointly with a development group within DEC, our project group will work on object-oriented, C++-based extensions of the interface specification and stub generation facilities. Another goal is the integration of a visual distributed application builder (VDAB), a graphical editor tool that has already been developed within our group. Its output will be adapted in order to match the formats of the interface definition correctly. Finally, object-oriented extension of other DCE components will be an issue for the more distant future. For example, the security service can be used for protecting objects. However, the granularity of the object model may be too fine in order to perform access checks efficiently. Therefore, new concepts will be required for protecting clusters of objects as a unit, for example. Distributed transactions as offered by new DCE-based products should also be integrated with the model of distributed object-oriented processing.

Our prototype will also be used by other projects in our distributed systems department, for example as a base for extending other object-oriented languages towards distribution.

Acknowledgments

We would like to thank Markus Person who implemented

the described concepts within his diploma thesis. We would also like to thank our project partner, Digital Equipment Corporation with its Campusbased Engineering Center in Karlsruhe for the granted funding for this project.

References

- [ABB93] Assenmacher H, Breitbach T, Buhler P, Hlbesch V and Schwarz R 1993 Panda—Supporting Distributed Programming in C++; *Internal Report* University of Kaiserslautern
- [ANS89] Advanced Network Systems Architecture (ANSA) 1989 *ANSA Reference Manual* APM Ltd, 24 Hills Road, Cambridge CB2 1JP, UK
- [BEN87] Bennett J K 1987 The Design and Implementation of Distributed Smalltalk *ACM OOPSLA Conf. (Orlando, FL)* pp 318–330
- [BHJ87] Black A, Hutchinson N, Jul E, Levy H and Carter L 1987 Distribution and Abstract Types in Emerald *IEEE Trans. Software Eng.* 13 65–75
- [BLA90] Black A and Artsy Y 1990 Implementing Location Independent Invocation *IEEE Trans. Parallel and Distributed Systems* 1 107–19
- [CAL89] Chase J S, Amador F G, Lazowska E D, Levy H M and Littlefield R J 1989 The Amber System: Parallel Programming on a Network of Multiprocessors *12th ACM Symp. on Operating Systems Principles (Litchfield Park, AZ)* (New York: ACM) pp 147–58
- [CHC91] Chin R S and Chanson S T 1991 Distributed Object-Based Programming Systems *ACM Comput. Surv.* 23 91–124
- [DEC86] Decouchant D 1986 Design of a Distributed Object Manager for the Smalltalk-80 System *ACM OOPSLA Conf. (Portland, OR)* pp 444–52
- [ENC92] Encina Transaction Processing System 1992 Transarc Corp., Pittsburgh, PA
- [HOC91] Horn C and Cahill V 1991 Supporting Distributed Applications in the Amadeus Environment *Comput. Commun.* 14 358–365
- [LET91] Levy H M and Tempero E D 1991 Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation *Software—Practice and Experience* 21 77–90
- [MAF92] Maffei S 1992 The Electra Approach to Object Oriented Programming *IFI TR 92.23* Institute for Informatics, University of Zürich.
- [OMG91] Object Management Group 1991 *The Common Object Request Broker: Architecture and Specification*.
- [OMG92] Object Management Group 1992 *Object Services Architecture*
- [OSF92a] Open Software Foundation 1992 *Introduction to OSF DCE* (Cambridge, MA: Open Software Foundation)
- [OSF92b] Open Software Foundation 1992 *DCE Users Guide and Reference* (Cambridge, MA: Open Software Foundation)
- [OSF92c] Open Software Foundation 1992 *DCE Application Development Guide* (Cambridge, MA: Open Software Foundation)
- [OSF92d] Open Software Foundation 1992 *DCE Application Development Reference* (Cambridge, MA: Open Software Foundation)
- [PER93] Person M 1993 Verteilte Objektverwaltung auf der Basis von DCE *Diplomarbeit Fakultät für Informatik der Universität Karlsruhe* (in German)
- [SCM93] Schill A and Mock M 1993 Design and Implementation of Distributed C++ *EURO-Arch '93 Conf.*

(*Munich*) (Berlin: Springer)
[SCP92] Schröder-Preikschat W 1992 PEACE—The
Evolution of a Parallel Operating System; *Reports of GMD*
No 646
[SDP91] Shrivastava S K, Dixon G N and Parrington G D An

Overview of the Arjuna Distributed Programming System
IEEE Software **8** 66–73
[TKR91] Tanenbaum A S, Kaashoek M F, von Renesse R and
Bal H E The Amoeba Distributed Operating System—A
Status Report *Comput. Commun.* **14** 324–35