Alexander Schill (Ed.)

DCE – The OSF Distributed Computing Environment

Client/Server Model and Beyond

International DCE Workshop
Karlsruhe, Germany, October 7-8, 1993
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 6980
Vincenz-Priessnitz-Straße 1
D-76131 Karlsruhe, Germany

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Volume Editor

Alexander Schill
Institut für Telematik, Universität Karlsruhe
Postfach 6980, D-76128 Karlsruhe, Germany

CR Subject Classification (1991): C.2.1, D.1.3, D.2.6, D.4.2-3

ISBN 3-540-57306-2 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-57306-2 Springer-Verlag New York Berlin Heidelberg

Printed in Germany

Typesetting: Camera-ready by author
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Preface

Client/server applications are of increasing importance in industry; they are a significant first step towards a global distributed processing model. A very recent response to this trend is the Distributed Computing Environment (DCE) of the Open Software Foundation (OSF), the emerging new industry standard for distributed processing. The papers in this volume discuss the client/server approach based on DCE, illustrating and analyzing the functionality of important DCE components and applications. Moreover, a number of contributions also focus on new models beyond traditional client/server processing and beyond DCE.

The material summarized in this volume was presented at the International Workshop on the OSF Distributed Computing Environment on October 7 and 8, 1993 in Karlsruhe, Germany. This workshop was organized by the German Association of Computer Science (*Gesellschaft für Informatik*, GI/ITG), together with the University of Karlsruhe and the Nuclear Research Center in Karlsruhe.

Major subject areas of the workshop were analysis and overview of DCE, methods and tools for DCE applications, extensions of the DCE remote procedure call, and distributed object-based systems on top of DCE, including the Object Request Broker (ORB) of the Object Management Group (OMG). Most papers are of practical orientation but typically have a strong technical and conceptual background. A more detailed overview of the papers is given at the end of the first contribution which gives a survey of distributed systems, DCE, and approaches beyond DCE.

We would like to thank all people who contributed to the success of this workshop. The members of the program committee did a very good job in reviewing about 10 papers per committee member. The Institute of Telematics of the University of Karlsruhe, especially Prof. Dr. Gerhard Krüger, made the workshop possible by providing a lot of organizational support. The university supported the workshop by making the required lecturing halls available. The background organization of the workshop was made possible by the *Gesellschaft für Informatik*, especially by its working groups on operating systems and on distributed systems (FA 3.1 and 3.3). We would also like to thank the speakers and authors and the colleagues who did the industry demonstrations on DCE; their technical contributions were a major prerequisite for this workshop. Moreover, the work force who helped with the local organization, especially the colleagues and students from the Institute of Telematics did an excellent job.

Finally, we would of course like to thank all companies that supported the workshop in various ways, including Daimler-Benz AG, Digital Equipment Corporation, Hewlett-Packard, IBM, the Open Software Foundation, Siemens-Nixdorf, and SUN Microsystems. The local organization was particularly supported by Dr. Lutz Heuser of Digital Equipment's Campus-based Engineering Center (CEC) in Karlsruhe and by the Volksbank Karlsruhe. Moreover, we would like to thank all other colleagues who supported this workshop in one way or the other during the last few months.

Karlsruhe, August 1993 — Alexander Schill

Workshop Organization

General organization:

German Association of Computer Science (GI/ITG), particularly the working groups "Operating Systems" and "Communication and Distributed Systems"

University of Karlsruhe, Institute of Telematics

Nuclear Research Center Karlsruhe

Workshop chair:

Alexander Schill
University of Karlsruhe, Institute of Telematics,
Postfach 6980, 76128 Karlsruhe, Germany
e-mail: schill@telematik.informatik.uni-karlsruhe.de

Program committee:

Martin Bever (IBM European Networking Center ENC, Heidelberg)
Kurt Geihs (University of Frankfurt)
Lutz Heuser (DEC Campus-based Engineering Center CEC, Karlsruhe)
Elmar Holler (Nuclear Research Center, Karlsruhe)
Winfried Kalfa (Technical University of Chemnitz)
Klaus-Peter Löhr (Free University (FU) of Berlin)
Klaus Müller (R&O Software Technology, Chemnitz)
Max Mühlhäuser (University of Karlsruhe)
Alexander Schill (University of Karlsruhe)
Peter Schlichtiger (Siemens-Nixdorf, Munich)
Gerd Schürmann (GMD Research Center (FOKUS), Berlin)

Local organization:

Lutz Heuser (DEC Campus-based Engineering Center CEC, Karlsruhe)
Monika Joram (University of Karlsruhe)
Ludwig Keller (University of Karlsruhe)
Dietmar Kottmann (University of Karlsruhe)
Gerhard Krüger (University of Karlsruhe)
Markus Mock (University of Karlsruhe)
Max Mühlhäuser (University of Karlsruhe)
Alexander Schill (University of Karlsruhe)
Jörg Sievert (University of Karlsruhe)

Table of Contents

RPC Extensions

Object-based Systems

Invited Talk

Appendix

Distributed Systems, OSF DCE, and Beyond

M. Bever[1], K. Geihs[2], L. Heuser[3], M. Mühlhäuser[4], A. Schill[5]

1) IBM European Networking Center, Vangerowstr. 18,
69115 Heidelberg, Germany; e-mail: bever@dhdibm1.bitnet

2) University of Frankfurt, Dept. of Informatics, P.O. Box 111932,
60054 Frankfurt, Germany; e-mail: geihs@informatik.uni-frankfurt.de

3) Digital Equipment GmbH, CEC Karlsruhe, Vincenz-Prießnitz-Str. 1,
76131 Karlsruhe, Germany; e-mail: heuser@kampus.enet.dec.com

4) University of Karlsruhe, Institute of Telematics, Postfach 6980,
76128 Karlsruhe, Germany; e-mail: max@tk.telematik.informatik.uni-karlsruhe.de

5) University of Karlsruhe, Institute of Telematics, Postfach 6980,
76128 Karlsruhe, Germany; e-mail: schill@telematik.informatik.uni-karlsruhe.de

Abstract. This introduction paper presents basic foundations of distributed systems and applications and then shows how OSF DCE addresses the requirements imposed by distributed environments. The DCE architecture is illustrated, the basic functionality of the DCE components is explained, and the DCE RPC as the major base for client/server applications is presented in closer detail.

The paper also discusses requirements and new models beyond DCE in order to enable even more advanced distributed applications. In particular, distributed object-oriented DCE extensions are outlined and directions towards distributed multimedia applications are pointed out. Moreover, other requirements and trends such as advanced tool support or distributed transaction facilities are also discussed. Finally, an overview of the papers within these proceedings is given.

1 Introduction and Overview

The potential benefits of distributed processing systems have been widely recognized [1,2]. They are due to improved economics, functionality, performance, reliability and scalability. In order to explore the advantages of distributed processing, appropriate support is needed that enables the development and execution of distributed applications. A distributed application consists of separate parts that execute on different nodes of the network and cooperate in order to achieve a common goal. A supporting infrastructure should make the inherent complexity of distributed processing transparent as much as possible. The infrastructure is required to integrate a wide range of computer system types and should be independent of the underlying communication technology.

The Open Software Foundation (OSF) has presented such an infrastructure called Distributed Computing Environment (DCE). It is a collection of integrated software components that are added to a computer's operating system. DCE provides means to build and run distributed applications in heterogeneous environments.

Let us illustrate the role of DCE by an example: Figure 1 shows a distributed office / manufacturing procedure that implements a product management scenario. Several distributed activities are performed by a collection of processes. We assume that each process is allocated to a different network node, and that nodes are connected by a physical network. The processes cooperate as shown by the arrows by forwarding forms or control data between each other. Some of the activities can be executed in parallel (such as the manufacturing and marketing activities) while others are sequential, or alternative (such as regular quality control, simplified quality control or bypassing according to the product type). Each activity can be subdivided hierarchically.

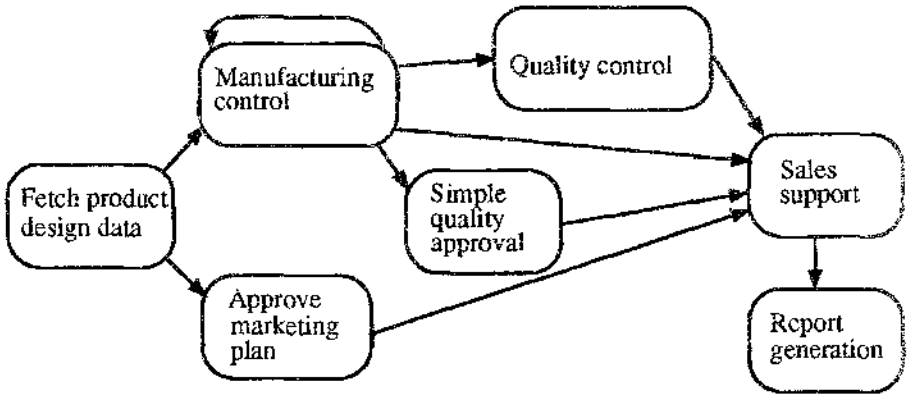

Fig. 1 Example of a distributed office procedure application

An example of an underlying distributed system is shown in figure 2. Two hosts and three workstations are interconnected via an Ethernet and a Token Ring. The two networks are coupled via a gateway. Each computer system offers local resources (at least CPU and main memory, but possibly also printers and secondary storage). These resources can also be accessed remotely and can be shared among different computers. Resource control is performed in a decentralized and mainly autonomous way. On each computer system, a set of application processes are operating - as found in our distributed application. These processes can communicate over the interconnected networks via basic interaction mechanisms such as remote procedure call. At this level, the underlying physical network topology is already considered to be relatively transparent.

Role of DCE and client/server-model: The OSF Distributed Computing Environment (DCE) can now be classified as being a distributed system, while also offering a set of services that support the development of distributed applications. Basically, DCE closes the gap between the physical components of a distributed system and the application components.

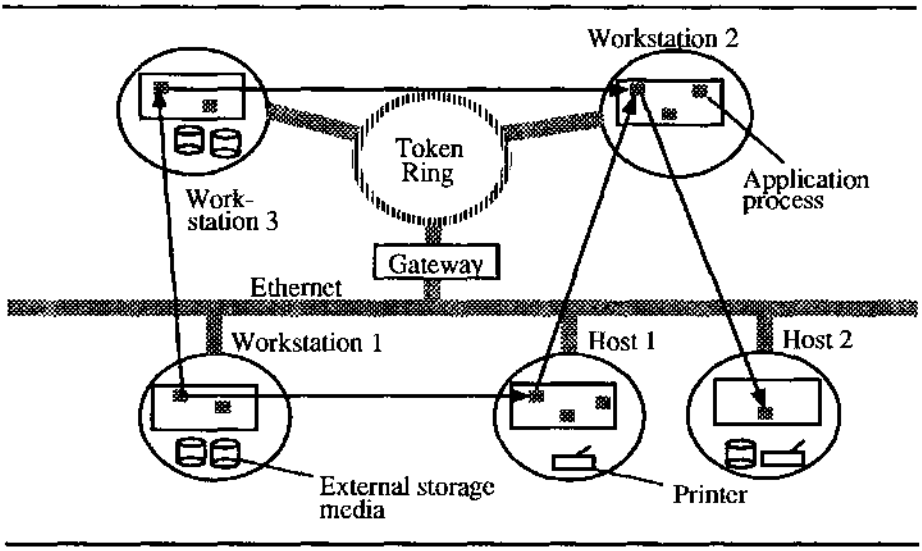

Fig. 2 Distributed system with communicating application processes

DCE internally works with the client/server model (see fig. 3), and is particularly well-suited for the development of applications that are structured according to this model: A server typically offers some service to a population of clients; typical examples are print services, computational services or name translation services. A client can make use of a service by sending a service request message to a suitable server. The request can contain input parameters (e.g. data to be printed). The server performs the requested service and finally sends a service response back to the client. The response can contain output parameters (e.g. a status indication).

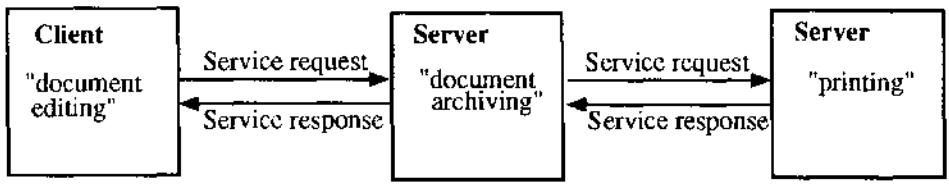

Fig. 3 Client/server model

As shown in the figure, a server can also act as a client of another service, i.e. delegate parts of a service request to a peer server. For example, a document archiving server could request a print service in order to offer a more complete document management functionality to its clients.

2 DCE: Strategy and Architecture

Based on the introduced foundations, this section presents the general strategy of the Open Software Foundation towards products for open systems and then illustrates DCE as one of these products in more detail.

2.1 Goals and Strategy of the Open Software Foundation

The Open Software Foundation (OSF) is a not-for-profit research and development organization. Its members comprise computer hardware and software vendors, end users, universities and other research institutions. One of the major goals of the OSF is to enable global interoperability among heterogeneous systems by providing a practical open computing environment [3].

To achieve this, the OSF solicits proposals for open systems software technology, then evaluates the submissions, and finally licenses the selected solutions for incorporation into the OSF open computing environment. That environment is a collection of technologies that provide for interoperability of diverse systems as well as application portability.

Its main parts are currently

- the OSF/1 Unix operating system,
- the OSF/Motif graphical user interface,
- the OSF Distributed Computing Environment (DCE) and
- the OSF Distributed Management Environment (DME).

From a distributed systems point of view, DCE and DME are of primary importance. While DCE is the base for building distributed applications and also offers a set of distributed services directly to the end user, DME addresses the issues of network and system management; it should suffice to mention that it offers an object-oriented infrastructure for distributed management applications, together with support for the management protocols SNMP and CMIP. It also provides a management user interface and several supplemental management services [4]. Moreover, DME uses certain DCE components. The DME development has not yet reached the same mature stage as DCE.

In the meantime, DCE tends to become an industry standard for distributed processing; most of the major computer vendors are members of the OSF and offer (or have announced) DCE compliant products for their computing platforms. As opposed to other standards, the implementation of the components existed first, and standardization was performed by the OSF thereafter. This seems to have major advantages concerning the resulting functionality, system performance and timeframe of delivery.

2.2 DCE Architecture and Services

Fig. 4 shows the overall DCE architecture [5-6]. All DCE components 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 *Thread 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, non-blocking remote invocations and multi-threaded servers.

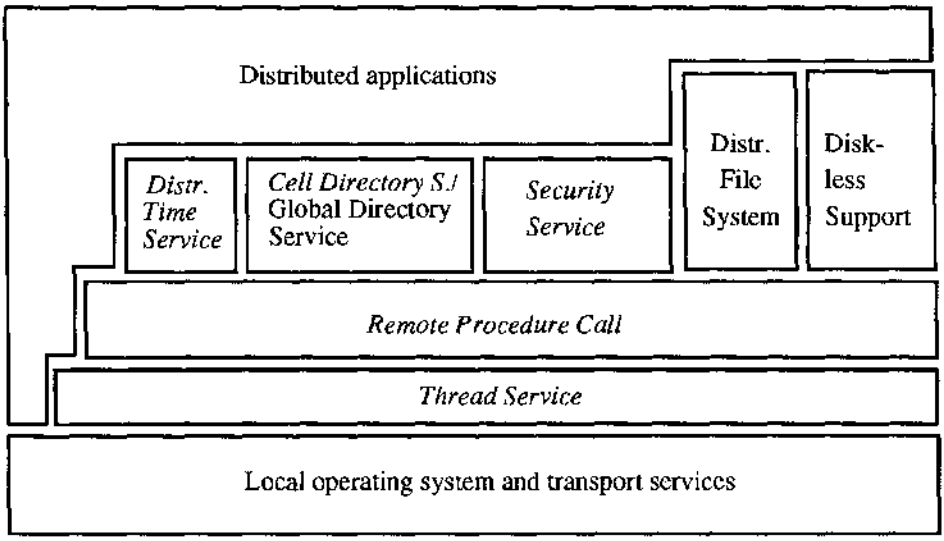

Fig. 4 DCE architecture

The *DCE RPC* is the major base for heterogeneous systems communication. Based on RPC, a client request for a remote procedure (i.e. a service request) is transferred to the server, mapped to a procedure implementation, executed, and finally acknowledged by sending back results to the client. All input data and results are encoded as RPC parameters similar to local calls. All parameter conversion and transmission tasks are handled by call marshalling facilities that are part of so-called RPC stub components at both sites. This way, the remoteness of a call be be masked to a large degree at the application level. The stubs are generated automatically from an interface description which specifies the signatures of the invoked procedures. DCE offers a C-based *Interface Definition Language (IDL)*, various kinds of 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 *Cell Directory Service (CDS)* supports distributed name management within dedicated management domains. Name management basically comprises mapping of (attributed) names to addresses, and update of name information. Most important, it is the base for mapping RPC server addresses to client requests. Its functionality is integrated into the DCE RPC programming interface via *NSI (Name Service Interface)*.

CDS exploits replication and caching to achieve fault tolerance and efficiency. 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 tighly integrated with DCE RPC; for example, RPC clients and servers can be mutually authenticated, servers can dynamically check access control lists for proper client authorization, 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 also directly exploited by other DCE components.

Other DCE services: The *Global Directory Service (GDS)* extends CDS by global naming facilities across administrative domains. It is based on the X.500 directory service standard. Therfore, it enables interoperability not only with other DCE directory servers but also with other X.500 servers worldwide. As an alternative, the *Internet Domain Name Service* can also be used for global 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 similar to a local Unix file system. File access is 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 [7]. 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.

In summary, DCE provides a rich and integrated functionality for distributed applications. Moreover, DCE supports heterogeneous systems interoperability and is offered in product quality.

2.3 DCE System Configurations and Application Example

DCE supports structuring of distributed computing systems into so-called cells in order to keep the size of administrative domains manageable. A cell can consist of all nodes attached to a local area network but is usually defined according to organizational considerations rather than physical network structures. Therefore, it is basically a set of nodes that are managed together by one authority.

Cell characteristics: Most DCE services are especially optimized for intra-cell interactions. While cross-cell communication is possible, interactions within a cell are usually much more frequent, and can therefore benefit from such optimization significantly. Moreover, cell boundaries represent security firewalls; access to servers in a foreign cell requires special authentication and authorization procedures that are different from secure intra-cell interactions. Finally, the distributed file system within a cell provides complete location transparence; as opposed to that, explicit cell names must be specified for file access across cells.

Example: Fig. 5 shows an example of an application framework based on DCE to implement an office / manufacturing scenario as discussed above. It consists of three cells A-C for product data management, manufacturing and marketing / sales. Within each cell, various nodes with dedicated application services exist (such as manufacturing control, machine management, and quality control processes on three different nodes in cell B). Moreover, each cell has a set of DCE system servers, including security, directory, time, and file servers. Typically, two or more servers of each kind are configured within a cell in order to improve availability of DCE services and performance of service access. One or several global directory servers are available in the example to enable cross-cell naming, e.g. to identify and access an application server in a remote cell. Finally, a diskless workstation pool is part of cell A and is linked to DFS and other DCE services via the diskless support component of DCE.

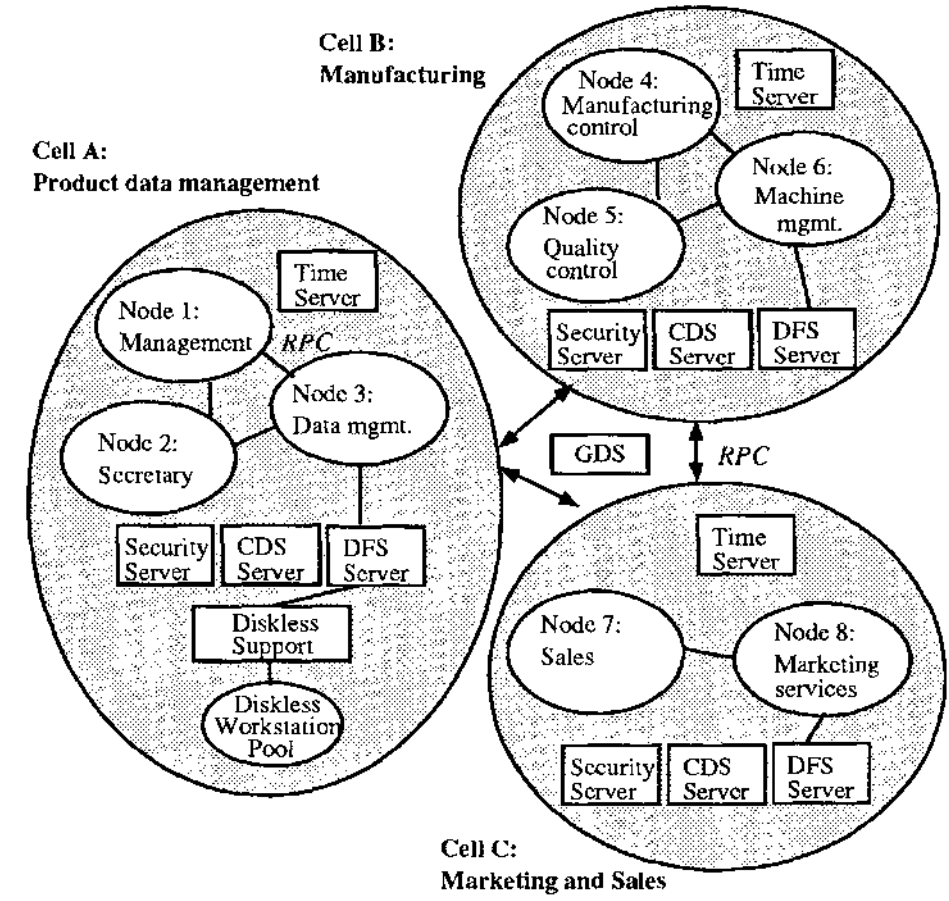

Fig. 5 DCE application example and cell structure

All nodes, respectively the application processes, and also the DCE components interact via DCE RPC. For example, this is indicated within cell A and between cell B and C in the figure. RPC servers are located via CDS based on logical names, and via GDS across cells. RPC communication can be made secure by the protocols offered by the security servers. Each process can comprise a number of threads to serve multiple RPCs concurrently (server site) or to issue multiple RPC requests in parallel (client site).

Data management can be based on the distributed file system. This way, different processes such as the management, secretary, and data management components of cell A can share file data in a location-transparent way. On the other hand, these files can also be accessed from remote cells upon request, provided that the accessing client is properly authorized and authenticated in both cases.

3 DCE Remote Procedure Call

As the RPC tends to be the most important mechanism within DCE, it shall be described in more detail, augmented with practical examples.

3.1 Properties of DCE RPC

Language integration and data representation: The implementation of DCE RPC is based on the C programming language; all interface specifications are given in a specific *Interface Definition Language (IDL)* that is a superset of the declarative part of C, corresponding to C header file code portions. Moreover, the RPC programming interface is offered as a C library - similar to the interfaces of other DCE components.

IDL allows the specification of arbitrary parameter data types with virtually the same facilities as found in C. The RPC runtime system, namely the stubs generated from IDL, are able to handle nested data structures by flattening them recursively, transmitting them to the server, and rebuilding them there. All differences concerning data representations at the client and server sites are masked by DCE by converting data formats accordingly. This principle is called "receiver makes right" and means that data are transmitted in the sender's representation and are adapted to the receiver's format at the destination site. The DCE implementation of a particular vendor must therefore know all other possible data formats of peer nodes - however, in practice, only a few different formats actually exist.

Call semantics: The application programmer can choose between different kinds of call semantics. For example, the default, at-most-once, makes sure that a call is executed once even if communication messages are temporarily lost. This is achieved by message retransmission combined with the detection of duplicate messages. Although node failures cannot be tolerated, message loss can be masked this way. Other selectable semantics provide weaker guarantees in the case of failure but achieve an improved efficiency.

Thread support: Based on threads, it is possible to implement multithreaded servers; this just requires an appropriate parameter setting during server initialization. Then a (static) pool of concurrent server threads is allocated initially. The application programmer, however, must take care of correct thread synchronization in case of shared

data modifications. On the client site, threads must be started explicitly to do concurrent, asynchronous calls to multiple servers. Within its body, each thread then performs a synchronous call while different threads are mutually asynchronous.

Security: As mentioned above, secure RPC communication is possible based on the security service. First, the application client and server run a distributed authentication protocol in cooperation with a security server. In this phase, they mutually validate their identity based on a private key encryption approach. In a second phase, the actual call is executed; before the server starts acting upon it, it checks the proper authorization of the client based on a local access control list. Finally, the call data can optionally be encrypted in order to enable complete privacy during communication.

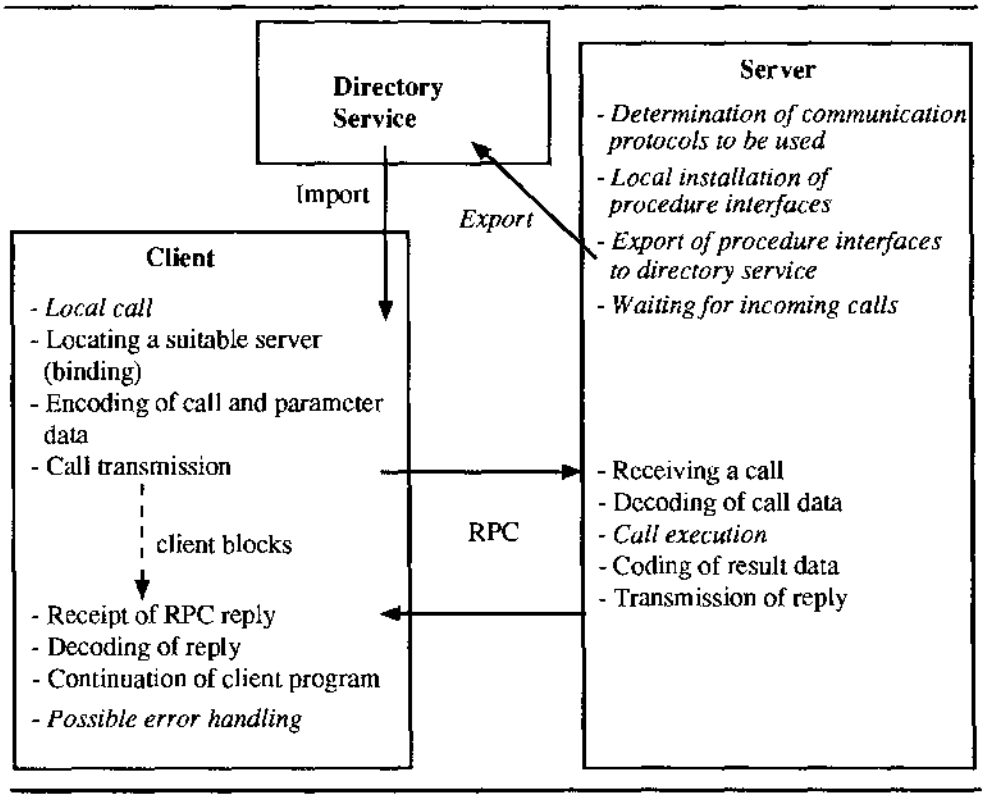

Fig. 6 Typical DCE RPC runtime scenario

3.2 Building Applications with DCE RPC

Building distributed applications with DCE RPC requires the following steps:

- *Interface definition:* An IDL interface must be specified with all procedures that shall be offered by a server.

- *Server implementation:* The server procedures must be implemented as ordinary C code. Moreover, DCE-specific server initialization steps must be performed by the implementation.
- *Client implementation:* In the simplest case, the client site is implemented as a standard C program. Advanced DCE features such as explicit selection among a group of servers or execution of secure RPC require additional code, however.

RPC runtime aspects: A typical DCE RPC runtime scenario is illustrated in fig. 6. All functionality that has to be implemented explicitly by the application developer is shown in italics, everything else is or can be performed automatically by DCE RPC.

The first step is the server initialization. The servers determine which communication protocols to use (such as TCP/IP or UDP/IP), installs its offered procedure interfaces with the RPC runtime system, exports the procedure interface information to the directory service (i.e. CDS), and finally waits for incoming calls.

To invoke an RPC, a client calls the corresponding procedure locally. However, based on the stubs that are generated from IDL, an internal handler routine is executed instead of a local application procedure implementation. It contacts the directory service for locating a suitable server. The input is a logical name for the server and the required procedure interface, the output is a server address, a so-called binding handle. This whole process is called RPC binding. Then the remote call and its input parameters are encoded and transmitted to the server. While the server executes the call, the client blocks. The remaining steps of call decoding, execution and result transfer have already been explained earlier. Finally, the client should include some error handling due to possible transmission problems etc.

Example: A program example shall illustrate the required code; it implements a remote client query against a server that manages product data. The interface definition consists of a header with a unique interface number (generated automatically) and with versioning information. The interface body comprises the required C type definitions and procedure interfaces with fully typed parameter specifications. Some attributes beyond C are required to distinguish between input and output parameters, for example.

```
[
uuid(765c3b10-100a-135d-1568-040034e67831),
version(1.0)
]
interface ProductData {                          // Interface for product data
  import "globaldef.idl";                        // Import of general definitions
  const long maxProd = 10;                       // Maximum number of products
  typedef [string] char *String;                 // String type
  typedef struct {
     String productName;                         // Product name
     String productAnnotation;                   // Textual annotation
     Plan manufacturingPlan;                     // ...Type defined in globaldef.idl
  } ProductDescription;                          // Product description data type

  long productQuery (                            // Remote query procedure
     [in] String productName[maxProd],           // -> Product names
     [out] ProductDescription *pd[maxProd],      // <- Product descriptions
     [out] long *status );                       // <- Call status
}
```

Server: The server initialization implements the steps discussed above by calling a number of DCE RPC system functions. A simplified example program looks as follows:

```
#include "productdata.h"                    // Generated by the IDL compiler
#define entryName "/.:/ProductServer"       // Name of server's directory entry
#define maxConcCalls 5                      // Max. number of concurrent invoc.

main ()    {
  unsigned status;                          // Return status
  rpc_binding_vector_t *bVec;               // Vector of binding handles

  // *** Perform some local initializations (not detailed here)
  // *** Get the actual vector of binding handles: ***
  rpc_server_inq_bindings (&bVec, &status);

  // *** Register the interface with a machine-local RPC manager process: ***
  rpc_ep_register (ProductData_v1_0_s_ifspec, bVec, NULL, NULL, &status);

  // *** Export the interface to the directory service under the given name: ***
  rpc_ns_binding_export (rpc_c_ns_syntax_default, entryName,
          ProductData_v1_0_s_ifspec, bVec, NULL, &status);

  // *** Now be ready to accept incoming invocations concurrently: ***
  rpc_server_listen (maxConcCalls, &status);
  }
```

The implementation of the server's application procedures, i.e. of *productQuery* in this case, is identical with a local implementation and is therefore not detailed here.

Client: The client site is also independent from DCE or distributed systems aspects (if no advanced RPC functionality is desired):

```
#include "productdata.h"                    // Interface definition header file

main ()    {
  String product[maxProd];          // Product names
  ProductDescription *pd[maxProd];  // Requested product descriptions
  long rc, status;                  // Status values

  inputProductNames (product);      // Input function (appl. specific)
  rc = productQuery (product, pd, &status);   // RPC
  // ... check status value and handle errors
  }
```

In summary, building DCE applications based on the client/server model is a relatively straightforward task for C programmers. However, the use of advanced features is more difficult. In the following, such features are summarized briefly; for deatails, see [6].

3.3 Advanced DCE RPC Features

Binding: During binding, the client can control the selection of a specific server explicitly; this mode is called *explicit binding* as opposed to the *automatic binding* applied above. The implementation of explicit binding is based on directory service interaction procedures to be called by the client via system RPCs. Moreover, DCE offers facilities to register groups of servers with the directory service and to specify client-specific search paths through the directory entries. This way, the server selection process can be controlled in detail.

Callback: With DCE RPC, it is possible for a server to issue a callback to a client during remote procedure execution; the client must offer an appropriate call interface for that. This way, a server can deliver intermediate results or can request further input data.

Pipes: For bulk data transfer, logical pipes can be established between client and server by passing pipe references as RPC parameters. A server can then request large chunks of data via the pipe dynamically from the client, and can also send bulk data back to the client this way.

Context: For multiple client/server interactions in a row, it is sometimes useful to establish some context information between both sites. An example is information about an open file of a file server that is read by a client by several RPCs. DCE RPC offers explicit mechanisms to handle such context information.

Other features such as asynchronous RPCs and secure RPCs have already been discussed. Altogether, a quite rich RPC functionality is provided by DCE.

4 Challenges and Models Beyond DCE

While DCE is a major step towards open distributed computing, the approach is still limited to relatively conventional client/server applications. This presents a number of ongoing research and development challenges to provide advanced support and new models beyond DCE. Examples are advanced development and management tools, distributed object-oriented systems, distributed transaction support, and multimedia extensions.

4.1 Advanced Method and Tool Support

Only few commercially available dedicated tools exist for client/server type applications. As a consequence, programmers use design methods, debugging tools and other software development aids that were developed for the sequential programming languages used as RPC host languages. The situation is similar for management components such as source code control tools. The distribution and parallelism exhibited in client/server applications, however, requires dedicated development aids. This requirement becomes even more important as we move to programming paradigms beyond client/server, such as the ones described later.

We will, for the remainder, focus on a number of important aspects of tool support which can be divided into three categories: development tools (here we will discuss formal specification and design), runtime-level tools (debugging and cooperation), and

management tools (runtime management and distributed system management). For some of these topics, we will separately discuss support for DCE-like client/server applications and for advanced distributed applications.

Formal specification techniques have gained a lot of attention in the context of communication protocols. This is due to the fact that such protocols are complex (i.e. hard to describe unambiguously with informal techniques) and that implementations of different vendors have to interoperate in open distributed systems. The formal techniques used for communication protocols can be applied to distributed applications as well [8]; the application of Lotos, Z, and SDL to distributed systems is described in [9]. However, the corresponding formal specification techniques usually do *not* focus on any of the most interesting aspects of distributed applications, such as dynamic reconfiguration of the network of processes (active entities and communication links beeing added and removed at runtime), hierarchical decomposition, and asynchronous communication. Moreover, the feasibility of formal techniques for large software projects, and the close coupling of formal techniques to DCE (in the sense of automatic code transformation) have not been achieved to a satisfactory degree yet. Moreover, in the client/server context, formal techniques for reasoning about the correctness of RPCs as such ought to be included.

Design: Specific design tools for distributed applications are hardly in use, either. The software engineer would want them to support visual programming, early animation of coarse designs, and automatic code generation. An example for a prototype tool with these features, VDAB, is described in this volume. It also supports the design of the dynamic behaviour of distributed applications "by example", based on dedicated *call scenarios*. The tool translates into a distributed version of C++, which is in turn implemented on top of DCE.

Debugging: Distributed debugging is widely recognized as one of the most important issues to be resolved on the way to cost-effective development of distributed applications. This owes to the fact that sequential debuggers do not help resolve some of the most predominant problems with testing distributed programs: the interference of the debugger with the code (which can make it impossible to detect the effects of race conditions), the presence of indeterminisms (which hinder the reproducability of subsequent debugging sessions), the vast amount of parallel events to be perceived by the user, and the lack of support for notions like "distributed breakpoint" and "distributed single-step". While uncountable contributions to these issues can be found in the literature, hardly any commercially available distributed debugging tool exists in the wider DCE context.

Runtime management: For sequential program development, the management of different versions and branches of source code and executable code in the development environment has been considered an important problem; to resolve this issue, code management tools were developed. In contrast, the *installation* of the final software version in a target environment (i.e. at the customer site) was usually deferred to a one-shot installation procedure. The installation of a distributed application however, i.e. the management of executable code in a distributed environment, is often an iterative and cumbersome task. Executables have to be copied to all sites, parameters and input files have to be considered at these sites, nameserver, network, and operating system setups have to be adjusted, etc. During execution, performance monitoring is desired, e.g., as a base for reconfiguration decisions. Such tasks are mostly carried out by hand today. But with the increasing deployment of distributed applications and the

continuing sophistication of these applications, the need for user-friendly (e.g., graphics-based) and highly automated runtime management tools will increase drastically.

Distributed system management: As DCE shows, distributed system management is a rather complex task. For example, CDS or Security servers must be installed, managed, and replicated. Security information such as passwords or access control lists must be maintained. DFS management comprises an even wider variety of different tasks. Therefore, graphical management tools are required to provide simplified management user interfaces. Beyond this, further higher-level programs are desirable which automate other routine management tasks in a distributed system such as backups or software upgrades.

To summarize, tool support for distributed programming has, for the most part, not yet left the academic stage. With DCE, however, the development of distributed applications *has* left this stage and more and more commercial sites get involved in distributed programming. The expected aggravating effects on the software crisis will probably lead to rapid changes in the scene of support tools in the years to come.

4.2 Distributed Object-Oriented Systems

A step beyond RPC are distributed object-based systems as extensions of programming languages like C++, Smalltalk, Trellis, Modula-2 or Eiffel. An object can be defined as a data structure associated with a set of operations. The data structure could refer to other objects by which an object graph can be built representing a so called complex object. Thus, as opposed to RPC servers, the granularity of objects is scalable and ranges, in general, from a couple of bytes up to thousands of bytes.

The fine granularity of objects and their capability to form complex objects lead to a unit of mobility, i.e. the object, which is easy to handle. Objects interact with each other through "message passing", i.e. an object sends a remote message to a peer object to initiate the execution of one of the provided operations. At the communication level, message passing is performed by RPC-style location independent object invocations whereby interacting objects can reside at different nodes.

However, as opposed to RPC with call-by-value parameter semantics, object references (i.e. pointers to objects) can also be passed remotely, leading to call-by-object-reference semantics. In addition, single objects or even complete object graphs could be passed as parameters to the callee [10].

The resulting approach is more flexible than RPC and, in particular, enables a more natural modeling of distributed applications. Due to the mobility of objects, they can be relocated dynamically. This way, communicating objects can be co-located in order to reduce communication costs or to increase availability during execution of a joint action of a set of objects.

Example: The discussed office / manufacturing system can directly be mapped to a distributed environment this way (see fig. 7). Distributed office procedures can be represented as task objects transferred between server objects. Typical operations of task objects are start, stop, suspend, or status inquiry while servers provide actual service invocations, status inquiries, or accounting functions. Data/document objects attached to an office procedure can also be modeled as objects. Moreover, since each document

has a certain structure like chapters, paragraphs and so on, it is likely that such document objects are object graphs as mentioned earlier.

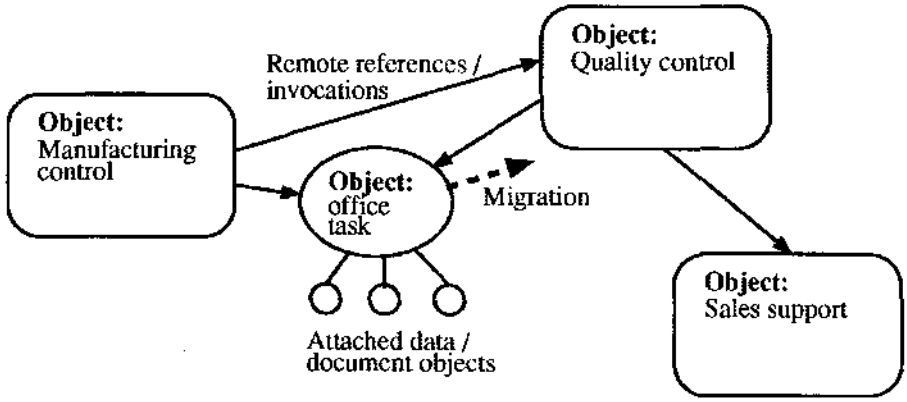

Fig. 7 Distributed object-oriented office procedure modeling

Class structure: The mechanisms to create remote objects, to locate and invoke them, and to relocate them dynamically can be implemented by superclasses from which all relevant application classes inherit. As an example, a C++ superclass is given below [11]; it also offers methods to fix objects at a certain location in order to prevent migration, and to unfix them later:

```
class DistributedObject {
// ... Instance variables like location, size etc.
public:
DistributedObject (Location*); // Constructor to create at a given location
~DistributedObject ();          // Destructor
Location *locate ();            // Locate the object
boolean move (Location*);       // Move to a given location
boolean fix ();                 // Fix at current location (prevent from moving)
boolean unfix ();               // Release for migrations
void Invoke (//...);            // Perform generic
invocation
};
```

Such functionality can be implemented on top of DCE RPC as illustrated by other papers in the proceedings. This requires sophisticated additional mechanisms for locating mobile objects (e.g. via forwarding addresses), for synchronizing migrations and computations, for controlling object migrations according to a given goal, and for monitoring and controlling the overall system behaviour.

OMG/CORBA: The industry consortium Object Management Group (OMG) has defined the Object Management Architecture (OMA) for managing objects in distributed systems. This approach aims at providing support for distributed object interaction in a

heterogeneous environment. In OMA, objects usally tend to be much larger than they are at programming language level, i.e. a whole application could be an object. This approach differs from object graphs in the sense that these "coarse-grained objects" are treated as monoliths.

A key component of OMA it is the Object Services Architecture [12-13] that offers the required services with a very broad spectrum of functionality. In general, these services provide a higher level of abstraction than DCE does and cover a broader technological area. Examples are database- and transaction-oriented services, version control of software objects, concurrency control, and distributed object replication. Location independent object interactions are supported by the Common Object Request Broker Architecture (CORBA); it supports mechanisms for identifying, locating, and accessing objects in a distributed environment. However, the Object Management Architecture has not yet reached the same level of maturity that DCE has. Moreover, some functionality of distributed object-oriented systems mentioned above, namely mobility, is not yet supported by CORBA. Several vendors implement - at least partially - CORBA on top of DCE.

Open Distributed Processing (ODP): Distributed system technology has become a major focus in international standardisation and harmonisation activities, too. As an important example, work in ISO and related standardisation committees is in progress to define a reference model for Open Distributed Processing (ODP). The reference model will include a descriptive as well as a prescriptive part. The descriptive part defines terminology and modeling gear that can be used to model arbitrary distributed systems. The prescriptive part specifies when a distributed system may be called an ODP system. It prescribes architectural properties that an ODP system must have. After the ODP reference model has been finished, individual ODP standards conforming to the reference model will be defined. Most likely, one will first work on standards for infrastructure components similar to those that we find in OSF DCE today. ODP and OSF DCE are two projects that are completely unrelated from an organisational point of view. However, the ODP work on an abstract reference model benefits significantly from the design of an infrastructure such as OSF DCE. The latter shows what functionality is needed in distributed processing systems and how components can be integrated into a common framework. Furthermore, when individual ODP standards will be sought for, the OSF DCE technology will certainly be a suitable and promising starting point. OSF and OMG (see above) have expressed their interest in advancing the ODP standardisation.

4.3 Distributed Transactions and Workflows

Transactions are a well-known approach found in database systems. It guarantees so-called ACID semantics (atomicity, consistency, isolation, and durability). Atomicity means that an operation is only performed as a unit; it is either fully completed (commit) or its effects do not become visible (abort). Consistency means that a transaction transforms data from one consistent state into another consistent state. Isolation means that concurrent transactions execute like in a sequential system without interference. In particular, a transaction T1 will never see any intermediate state of data caused by another transaction T2 before T2 is completed. Finally, durability means that the effects of a transaction (data manipulations etc.) remain persistent after transaction completion; for example, they do not get lost after a system crash.

Distributed transactions: The transaction properties have proven very useful for implementing processing functionality with strict consistency requirements (like credit/debit transactions). Therefore, extensions of the basic concept towards distributed systems have been developed [14]. Some of them are based on RPC or distributed object interactions, others on pure message passing. The implementation typically relies on the two-phase commit protocol. In the first phase, all transaction participants are polled by a coordinator whether they are able to successfully commit. In the second phase, the uniform decision is propagated to them in order to commit or abort jointly.

This concept has also been extended towards nested transactions with hierarchical subtransactions. This allows for running subtransactions in parallel, and for selective rollback and restart of subtransactions. Standards and implementations on top of DCE: Meanwhile, there are several emerging product-level implementations of distributed transactions that conform to new standards. Most notably, the *X/Open* consortium has defined the *X/Open Distributed Transaction Processing (DTP)* application programming interface named XA [15]. This approach is designed to work with standardized ISO/OSI transaction protocols, namely CCR and TP.

Conforming to the XA standard, there are several implementations available. The origins of XA have evolved according to the *Tuxedo* system [16] of Unix System Laboratories that is available on numerous hardware platforms and operating systems. The *Encina* transaction processing system [17] of Transarc is an open, XA standards-based family of components that provide online transaction processing based on DCE. Transactional integrity is added to DCE programs through Transactional-C, Transactional RPC (TRPC), two-phase commit and the management of recoverable data. Transactional-C consists of C language extensions to indicate transaction demarcation, concurrency control and exception handling. TRPC adds exactly-once semantics to DCE RPC. When a remote procedure is called from within a transaction, it is executed exactly once, if the transaction commits and not at all if the transaction aborts. Besides this basic support for transactional integrity, Encina offers a Structured File System (SFS) and a Monitor as an administrative, runtime and development environment for transactional applications. Functionality of the Monitor comprises, for example, monitoring active clients, performing load balancing and connecting front-end tools (like OSF/Motif). SFS is an record-oriented file system (in contrast to DCE DFS) that meets the requirements of transactional systems for record-style and recoverable resource managers.

Workflows are a rapidly emerging technology area that deals with long-lived, well-defined activities like office procedures. A workflow system controls the execution of the global control flow and in some cases provides certain reliability support by using transaction mechanisms. Workflows are distributed by nature and thus are one of the key application domain for distributed processing. On the other hand, workflows introduce a new style of programming since execution order and principles should be extracted from the single application part and be moved to a separate workflow program.

Workflow systems could benefit from all DCE services. Especially RPC for communication between the workflow and the application parts, authorization and authentication of users performing the individual steps, and directory services for locating workflow servers are important.

4.4 Distributed Multimedia Systems

A multimedia system is characterised by the computer controlled generation, manipulation, presentation, storage, and communication of independent discrete media such as text and graphics and continuous media such as audio and video [18,19]. Application domains for multimedia systems are, e.g., multimedia e-mail, multimedia-supported teaching, virtual reality simulation systems, and workstation conferencing systems. Many of these domains are inherently distributed. A workstation conferencing system, for example, allows sharing of window based applications among participants at different locations supported by multimedia services for audio communication as well as video conferencing.

Distributed multimedia systems impose new challenges for the communication of continuous data. Whereas discrete media have time independent values, the values of continuous media change over time and these changes contribute to the media semantics: Each single value in an audio or video stream represents stream information for some fraction of time. Changes in the times at which values are played or recorded result in the modification of the original data semantics and must not happen unintentionally. The timing demands of continuous media require operating and transport system support for connections with guaranteed quality-of-service (QoS) for the transmission of continuous data [20]. This is achieved by allocating some fraction of the end-system and network resource capacity and scheduling these resources appropriately. Another meaningful requirement is the support of multicast since a continuous stream often must be transmitted from one source to multiple sinks.
For the development of distributed multimedia applications it is reasonable to model sources (e.g. a microphone) and sinks (e.g. a loudspeaker) of continuous streams as objects. A source object, for example, offers operations to connect itself to a sink object and to start, stop or suspend the production of a stream. This kind of control operations is performed by conventional DCE RPC communication. When control operations must be submitted to multiple sinks, a multicall extension to the RPC is convenient. Directory and security infrastructure is crucial to identify appropriate sources and sinks. The establishment work for the respective connections, however, that comprises the negotiation of the required QoS, and the transmission of the data itself can be left to the "multimedia" transport system. As a result, sources and sinks must be able to cope with the coexistence of dedicated runtime systems for conventional RPC communication as well as processing of continuous data.

Special support for producing, processing and consuming continuous data is needed, when the data get manipulated within the application. Manipulations are, for example, encrypting or compressing audio or video information, or mixing and synchronising different but related streams. A useful construction in this context is to conceive the connection between a source and its sink as an object in its own right. Such an object exhibits two categories of operations: a lower level category acting on the established transport connection and a higher level for controlling it.

5 Overview of the Technical Contributions

The technical contributions in this volume are concerned with DCE implementation issues, with applications and tools, but also with models and approaches beyond DCE. In particular, the following areas are covered:

DCE analysis and comparison: DCE is compared with the ANSAware environment developed in the UK. Moreover, DCE RPC is compared with SUN RPC and with other RPC approaches. Two performance analysis studies evaluate DCE RPC and CDS in detail. This also leads to concrete recommendations, for example concerning the configuration of a CDS name space. This way, the conceptual overview given above is augmented with practical DCE analyses and experiences.

Application support: This part focuses on DCE application support tools and on actual DCE applications. Two different tools for enabling Fortran access to DCE services are presented. The first tool supports the conversion of existing non-distributed Fortran applications to distributed DCE applications. The other approach enables direct program-level access from Fortran to C-based DCE functions. One paper presents a practical DCE application for stock broker support. This contribution emphasizes how DCE is used for real-world applications and reports experiences with DCE application development. Another application implements print services and a heterogeneous interface to various mail systems based on DCE RPC, using a generic services architecture.

Methods and tools: Several other DCE support tools are presented. A formal method and tool approach focuses on the problem of converting monolithic, non-distributed programs to distributed applications on top of DCE. A similar transition approach is presented for DCE Security, helping to incorporate conventional Unix security environments into a DCE framework. Finally, advanced tools for resource monitoring in DCE cells are presented. Altogether, tool examples from all three categories of development, system management and runtime-level are discussed.

RPC extensions: This part covers direct extensions of DCE RPC. The first example is multimedia support based on new media types and quality of service attributes in IDL, and on runtime mechanisms for time-constrained RPC and realtime thread scheduling. Another paper introduces an integration of RPC and message passing, leading to a more flexible set of communication facilities. Moreover, optimized RPC server selection is also addressed in order to relieve the application developer from the selection process discussed above. Finally, an ambitious distributed object model is implemented by an object-oriented RPC extension. It provides the facilities discussed above, but also supports dynamic typing of objects. This allows a more natural integration of common generic services in a DCE environment.

Object-based systems: Several other papers focus on object-based DCE extensions, too. An operating implementation of CORBA on top of DCE is presented, showing that there's a strong practical relationship between CORBA and DCE. The design and implementation of a distributed object-oriented framework with mobile objects beyond CORBA, but on top of DCE, is illustrated by another contribution. A related paper shows how the more conventional DCE functionality can at least be offered via higher-level, object-oriented class interfaces. This way, an improved abstraction is provided to the application developer. Finally, an object-based tool for graphical support of DCE applications is presented. This also extends the tool discussion towards the early phases of application design.

In summary, the papers in this volume illustrate that DCE is a practical environment for building distributed programs. However, they also make the need for higher-level tools, models and abstractions obvious. It is hoped that directions for further research and development in the context of DCE are pointed out this way, and that such work will help to make DCE a success for open distributed environments.