ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

Rainer Manthey and Viacheslav Wolfengagen (Eds)

Advances in Databases and Information Systems 1997

Proceedings of the First East-European Symposium on Advances in Databases and Information Systems, (ADBIS'97), St Petersburg, 2-5 September 1997

Portability of STDL on Top of the TUXEDO Transaction Monitor

Malik Saheb

Published in collaboration with the
British Computer Society

ISBN: 3-540-76227-2

Portability of STDL on top of the TUXEDO Transaction Monitor

Malik Saheb[1,2]
[1]INRIA Rocquencourt, B.P 105
F-78153 Le Chesnay,France
[2]ENST, 46 rue Barrault
F-75634 Paris Cedex 13, France
Malik.Saheb@inria.fr

Abstract

The Structured Transaction Definition Language (STDL) is a language based programming interface to transactional protocols and runtime systems, designed to resolve the industry problem of incompatible Transaction Processing (TP) monitor programming interfaces. STDL defines a three-group model, in which application procedures are grouped according to the type of work they perform: presentation, transactional flow control and error handling, and data access. A separate interface definition is created for each group of procedures and one procedure calls another procedure via this interface. STDL does not define protocol for the procedure calls except for remote task calls, which use the X/Open TxRPC protocol or DCE. Some existing TP monitors do not use TxRPC to invoke remote service but other protocols, such the Application Transaction Manager Interface (ATMI), the Common Programming Interface for Communication (CPI-C), or even CORBA/OTS in an object environment.

This paper describes an STDL compiler on top of the TUXEDO monitor, translating STDL applications into XATMI client/server applications, supported by TUXEDO. Then it illustrates the possibility provided by STDL to achieve interoperability through TP platforms.

1 Introduction

The usage of transactions [14] is a very popular concept for the management of large data collections. Transactions guarantee the consistency of data records when multiple users or processes perform concurrent operations on them. In general, the properties of transactions are known as the ACID properties (Atomicity, Consistency, Isolation, Durability) [16].

An important aspect of distributed transaction processing applications is communication. Within the product domain for Distributed Transaction Processing tools, there are several popular communication paradigms in common use today. The communication paradigm chosen can significantly influence the architecture of the application.

Because it is not possible to choose a single communication paradigm applicable to the entire broad range of DTP applications, the X/Open consortium has provided Application Programming Interfaces or API for the most popular paradigms in order to bring the benefits of open systems to the widest possible range of transaction processing applications.

For applications choosing to communicate using a conversation, X/Open offers a Communication Programming Interface for Communication (CPI-C)[8]. For applications already running on open systems and using communication paradigm based on service requests, X/Open specifies the XATMI interface; Finally, for distributed applications using the remote procedure call (RPC) mechanisms, X/Open provides the TxRPC interface.

To provide communication subsystem independence for transactional applications, the *Structured Transaction Definition Language* (STDL) [13] has recently been specified by X/Open as its high-level language [11] for

Distributed Transaction Processing (DTP) [12]. STDL is a procedure-oriented language designed specifically for distributed transaction processing to resolve the industry problem of incompatible TP monitor programming interfaces. Using STDL for the development of TP applications for multiple platforms allows programmers to concentrate on business solutions rather on the complex notation of programming interfaces.

STDL defines a three-group model, in which application procedures are arranged according to the type of work they perform: user or device access (*presentation procedures*), transactional flow control and error handling (STDL *task procedures*), and data access (*processing procedures*). A separate interface definition is created for each group of procedures and one procedure calls another via this interface definition. STDL defines a protocol for remote task calls, which use the X/Open TxRPC protocol [9] (an extension of the Remote Procedure Call (RPC) for the *Distributed Computing Environment* (DCE) [22], and the OSI Transaction Processing protocol [15]) and plain DCE RPC for an external client to the TP system or for remote non-transactional task calls.

The focus of the STDL design is the concept of the RPC. Consequently STDL includes an interface definition language (IDL) that is used to create interface definitions separately from the procedures themselves. The STDL IDL is called a *task group specification*.

The specification of STDL has been implemented differently by different vendors and publicly demonstrated at Telecom '95 [4]. However most of these implementation are based on the Remote Task Invocation (RTI) protocol described in the TxRPC specification for remote task calls, on which STDL is mapped, while there are other protocols, widely used such XATMI or CPI-C on which there is no mapping.

This paper describes an STDL compiler on top of the TUXEDO monitor, translating STDL applications into XATMI client/server applications, supported by TUXEDO. The following three sections give a short overview of the X/Open Distributed Transaction Model, the STDL language, and the TUXEDO transaction monitor, respectively. Section 5 shows the mapping of STDL into the XATMI interface and the STDL TUXEDO components. Section 6 illustrates the possibility provided by STDL to achieve interoperability through Transaction Processing (TP) platforms. Finally section 7 concludes the paper.

2 The X/Open Distributed Transaction Processing Model

The software architecture called X/Open Distributed Transaction Processing (DTP) model [12] illustrated in Figure 1 allows multiple application programs (AP) to share resources provided by multiple resource managers (RM), such as database systems, transactional file systems or queue managers, and allows their work to be coordinated into global transactions by a transaction manager (TM) among these resource managers .

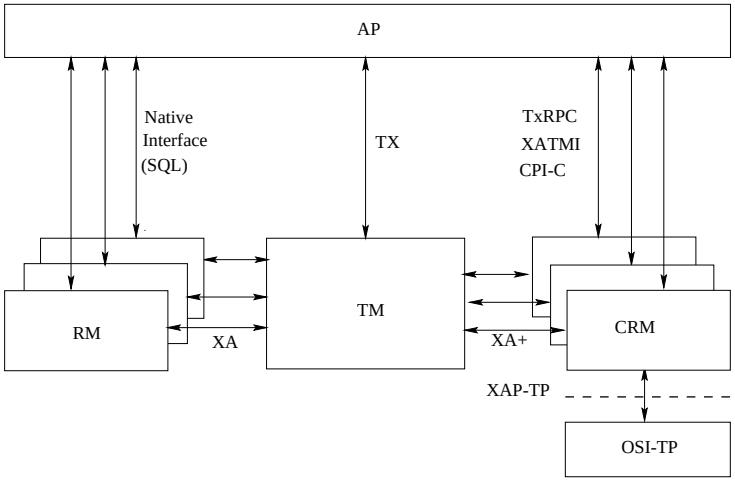

Figure 1: The X/Open DTP model

The X/Open DTP offers to the AP the TX interface [10] by which it calls the TM to demarcate global transactions and direct their completion. X/Open also defines a subroutine library for the RMs to register with their local TM, and for the TM to invoke them at system restart and at transaction begin, commit and abort. This TM-RM interface is called XA-interface [5].

Within the product domain of DTP tools, there are several common communication paradigms in use today. The communication paradigm chosen can significantly influence the whole architecture of the application. Because it is not possible to choose a single communication paradigm applicable to the entire broad range of DTP applications, X/Open provides APIs for the most popular paradigms.

X/Open offers a CPI-C Communication Resource Manager (CRM) API [8] for applications choosing to communicate using a conversational paradigm, where communication takes place through an exchange of messages.

Many applications already running on open systems use a communication paradigm based on service requests. X/Open specifies for this request/response the XATMI interface [7], which also offers the conversational paradigm.

For distributed applications using the remote procedure call (RPC) mechanisms, X/Open offers a Transactional Remote Procedure Call (TxRPC) [9]. TxRPC allows application writers to invoke remote procedures in the same form as local procedures, but with transaction semantics.

3 The Structured Transaction Definition Language

The Structured Transaction Definition Language (STDL) [13, 11] is a block-structured language specialized for transaction processing. STDL provides transactional features including transaction demarcation, transactional remote procedure call, transactional task and data record queuing, transactional display management, transactional exception handling, and transactional working storage called workspaces.

STDL divides an application into three parts: presentation, transactional flow control, and processing. The presentation part interfaces with display devices using a presentation manager, such as Motif, or Windows. The transaction flow control part is written in STDL and controls the flow of execution, including transaction demarcation, exception handling, and access to queues. The processing part is written in traditional languages, such as C, COBOL, and SQL, and provides computation and access to resource managers such as databases and files.

The application functions in the three parts of the STDL application model are referred to respectively as presentation procedures, tasks, and processing procedures. The application functions are packaged into groups for the purpose of compilation and execution. The groups are referred to as presentation groups, task groups, and processing groups.

A group specification describes the functions in the group and their interfaces. The interface specification includes the arguments that are passed to the function and an indication of whether an argument is input only, output only, or both input and output. For a task, the interface specification also indicates whether the task begins a new transaction (NONCOMPOSABLE) or joins the caller's transaction (COMPOSABLE). STDL does not define protocol for the procedure calls except for remote task call, which use the X/Open TxRPC or DCE , as illustrated in Figure 2.

The different groups define the scope of context that can be shared among executions of procedures. A task group defines the scope of task context, created by an executing task, that can be shared among executions of tasks. This context is called *task group context*. Two tasks share the same task group context if one of the following conditions is met :

- One task execution was caused by the other task execution through task calls to composable tasks in the same task group on the same TP system.
- Both task executions are caused by another task execution through task call to composable tasks in the same task group on the same TP system.

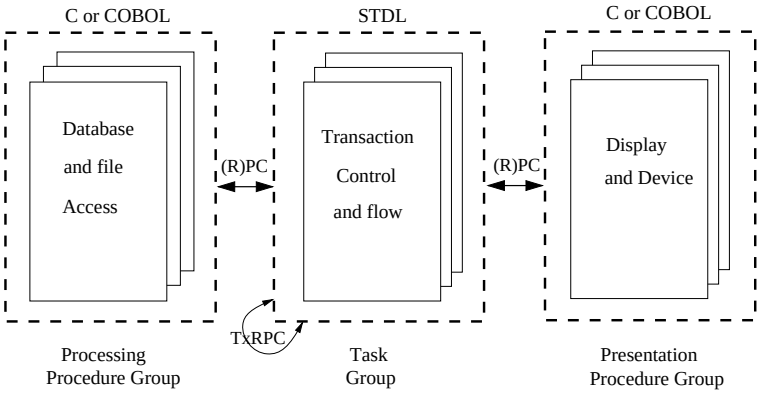

Figure 2: STDL Three-Group Model

- The task executions are caused by task calls made sequentially on the same transactional dialogue, which is a dialogue over which a transaction spanning two TP systems is coordinated.

The task group context consists of a transaction context (retained for the duration of the transaction), including a processing group context and a transactional dialogue context, and a non-transaction context minimally, including a presentation group context (retained for duration of a task execution).

A *transactional dialogue* is a dialogue over which a transaction spanning two TP systems is coordinated. Calls to composable tasks within a transaction can be performed over a transactional dialogue. A dialogue server can optionally accept multiple calls over a transactional dialogue. A transactional dialogue always terminates at the end of the transaction. A *non-transactional* dialogue is a dialogue over which no transaction coordination is done. Calls to non-composable tasks use a non-transactional dialogue.

Transactional dialogues used for task calls are part of the task group context at the client TP system and at the server TP system. Any two task calls made sequentially from tasks executing in the same task group context at the client TP system to the same server TP system for task in the same task group must use the same dialogue.

A processing group defines the scope of context sharing the execution of processing procedures. The context that a processing procedure creates can be used by another processing procedure, belonging to the same processing group, and a fortiori by the procedure itself when it is called several times. This context is called the processing group context and may contains file context, such as file position indicator, and SQL context, such as cursors.

3.1 Mapping STDL to the X/Open DTP Model

Figure 3 illustrates the relationship of STDL to the X/Open DTP model. Basically, STDL represents an application in the X/Open model. STDL is a full-function TP language and includes features to those defined within the X/Open DTP model.

A C or COBOL processing procedure provides an interface to a resource manager, while access to the transaction manager is accomplished through STDL. Also, access to the communication resource manager is accomplished directly via STDL.

4 Overview of the TUXEDO System

The Tuxedo system is a transaction monitor and database system designed to run on the UNIX operating system [1, 6]. Tuxedo is built around a main component called Tuxedo System/T, which provides the critical

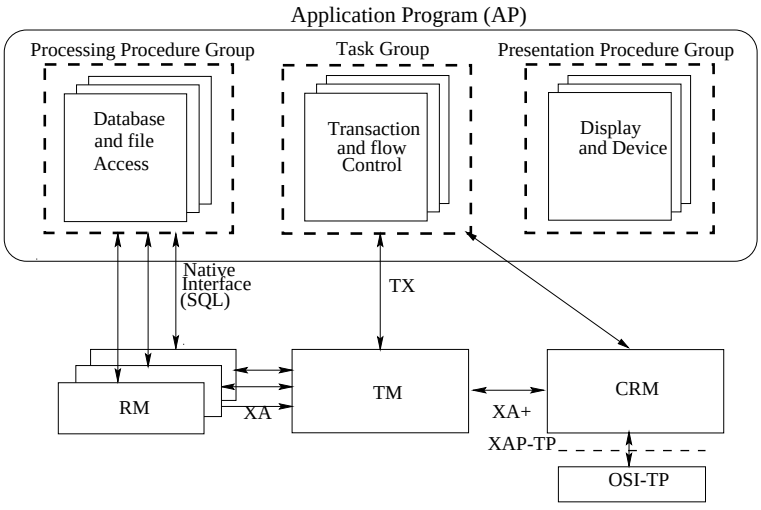

Figure 3: Mapping STDL to the X/Open DTP Model

distributed application services: naming, message routing, load balancing, configuration management, transaction management, security, and application development tools (Application Transaction Manager Interface (ATMI) a superset of XATMI). In addition to System/T, Tuxedo offers the following components: Tuxedo /WS, which extend the client side capability to intelligent workstations; Tuxedo /Q, providing store-and-forward queue management; Tuxedo System /D, an SQL database system which acts as a resource manager for System /T; and Tuxedo /Domain (Tuxedo Version 5), which provides a framework for interoperability.

To locate servers, Tuxedo provides a server called "Bulletin board Link (BBL)" which manages a shared-memory or "bulletin board (BB)" recording information about local and remote servers. The BB serves as the name service database for distributed application, providing location information for the distributed application as illustrated in Figure 4.

In order to start the server and to initialize the bulletin board of the TUXEDO System/T, a specific administration command called tmboot is performed, which uses information contained in a binary configuration file called the TUXCONFIG file (loaded, via the command tmloadcf, from a text file "UBBCONFIG", created by a programmer). Within TUXEDO, servers are built via the administrative command called buildserver, while clients are built via the buildclient command.

Messages are passed to servers in typed buffers. The advantage of typed buffers is that programmers do not have to worry about converting data being sent to machines having different data representation formats. The TUXEDO System performs transparently data format conversions. In order to associate typed buffer to structures used by an application, a program must know the format of the incoming data. This is done through a set of view descriptions created and stored in source viewfiles. The description maps fields in the view description to members in a C structure or COBOL record.

Viewfiles are source files for descriptions of one or more C data structures, or "views". When used as input to the viewc command, the viewfile forms the basis for a binary file (view_filename.V) used for coding and decoding typed buffers, and a header file (view_filename.h) needed to be included in programs using the typed buffer.

5 The TUXEDO-STDL compiler

The concept of IDL, on which STDL is based, is not supported by the XATMI interface. Therefore encapsulation of services via an interface and possible attributes such as transactional quality assigned to services are not taken into account by XATMI. The XATMI interface provides sufficient ways for applications to interoperate,

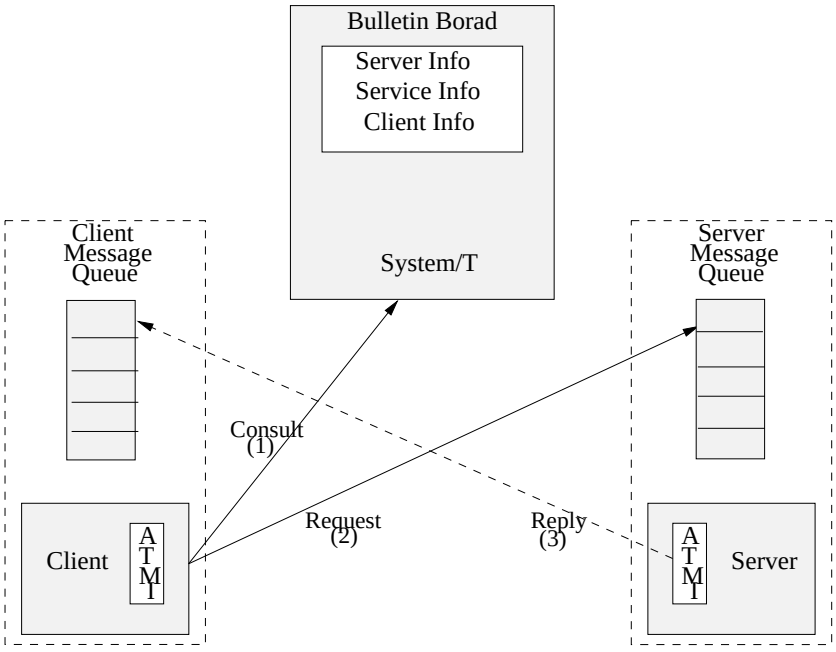

Figure 4: The Bulletin Board as a name service database

but not transparency regarding to the fact that a particular API is used instead of an oriented procedure call to invoke a service which can be local or remote. The lack of transparency provided by an interface and its encapsulation is filled by a language-based approach to TP, provided by STDL, instead of a system-service approach. The language-based approach has the advantage of allowing a language compiler or precompiler to be introduced as an intermediate step in creating a TP application. The API provided by XATMI will be hidden, enabling a transparency for a call.

Figure 5 illustrates the development of an STDL application over the XATMI Communication Resource Manager, in which client stub and server stub files generated from the task group specification and linked for the generation of executable client and server, allow respectively to invoke XATMI services enabling to make a request, and to use XATMI services to receive a request and to return the result.

5.1 Task Invocation Mapping

Two types of services paradigms are defined by the XATMI interface:

- Request/response service: XATMI supports both synchronous and asynchronous request/response. tpcall() provides a synchronous call, while tpacall() enables an asynchronous call. To tpacall() is associated tpgetreply() allowing the program to get a response to the request previously sent via tpacall()

- Conversational service: the conversation takes place in a half-duplex manner. By using the tpconnect() function, the requester initiates conversational communication. The tpconnect() function returns to the requester a descriptor that it shall use to refer the newly established connection during communications. The tpsend() and tprecv() functions allow programs to exchange data over an open connection.

  A communication is terminated by the communication RM in an orderly manner after the service calls the tpreturn() function. If the requester wishes to terminate the conversation abortively, it can call the

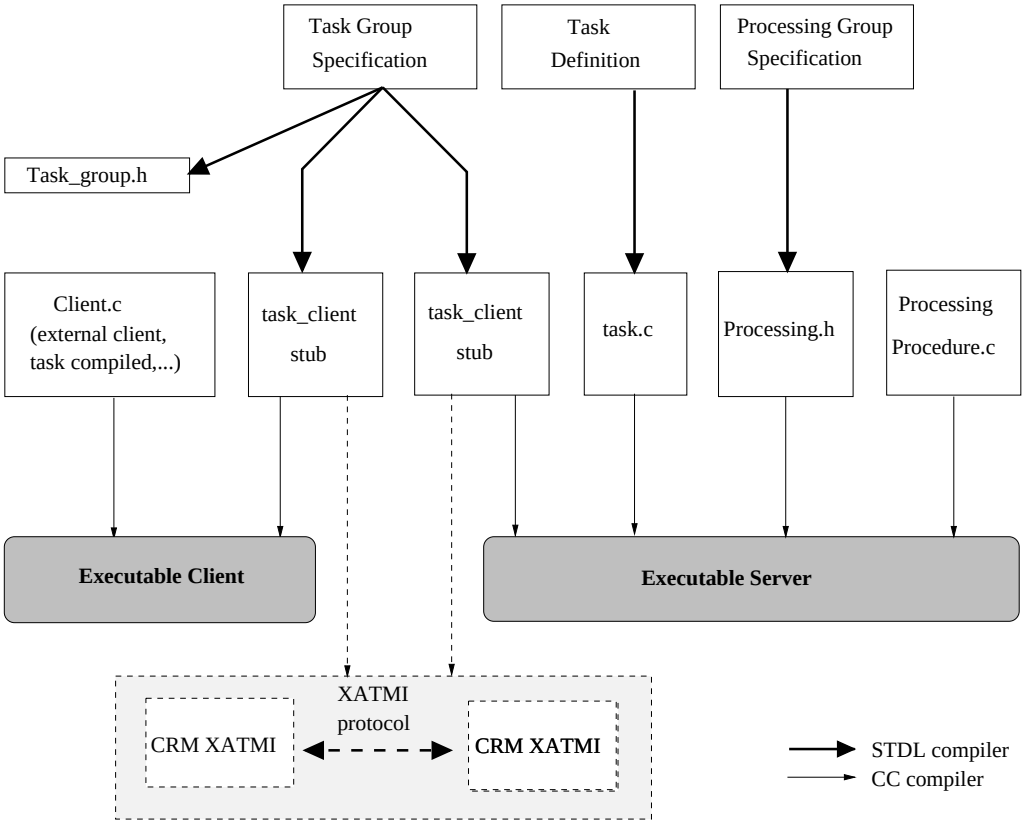

Figure 5: Developing an STDL application on the top of XATMI

tpdiscon() function, which terminates a connection in such manner that data in transit may be lost and any active transaction associated with that connection is rolled back.

The choice of development with either one of paradigms is influenced by the application that one would implement, and the relationship between a client and a server, according to whether a context has to be maintained or not between these two elements. In the case of XATMI the paradigm allows safeguarding a context between a client and a server through a conversation enabling. A conversation is identified by a descriptor used in both client and server, respectively to send data to the same server, and to identify the same client.

An example needing the maintenance of a context between a client and a server is a service giving the list of the accounts of a bank whose credit would exceed an amount given. Due to the fact that it is often difficult to display in a once this list, successive calls will be necessary. If the counts are stocked in a database, the function offering the service will have to manipulate a cursor via a query language such as SQL.

The only paradigm provided by STDL, at the application level, is RPC. As said above, a task call has to invoke, transparently to the application, services offered by the underlying CRM, in our case XATMI. To map a task call, we can think, at first sight, that the request/response service is the most appropriate since the RPC paradigm is also viewed as a request/response paradigm.

A STDL task calling another task to list accounts is susceptible to make successive calls. In the server side, a processing group context (belonging to the task group context created by the task server) has to be created and will be used for successive accesses to a resource such as a database. Besides indicating the end of service routine, the XATMI tpreturn() function also causes the server performing the request to release a context created for the requester, by which it can recognize this requester through successive calls. When this function is issued by the service routine, the requester will receive a return code indicating that it cannot send another request on the same connection. Finally the XATMI request/response service seems not appropriate to enable several task calls to the same server which must keep a context for a requester. For this reason the conversational service is preferred on which a call task is mapped.

Although the problem of maintaining a context between client and server is resolved by a conversational service, the tpreturn() function is needed in any case to complete properly a service routine especially when it is in a transaction mode. Indeed, in transaction mode, tpreturn() places the service's portion of the transaction in a state where it may be either committed or rolled back when the transaction is completed. In any case, in order that the transaction commits at the server the tpreturn() function has to be issued by the server. Otherwise a protocol error is returned for the initiator of the commit. Although the tpdiscon() function allows a requester to disconnect the conversation, this is done abortively rather than orderly. Any data that has not reached its destination may be lost and any transaction in which the conversation has been initiated must be rolled back.

Considering that a tpreturn() has to be used to complete properly a service routine, when is it issued? Indeed, all possible XATMI functions reside in the stubs generated automatically by the stdl compiler. That is, the service routine located in the generated server stub does not know in advance when it can issue a tpreturn() function since it does not have enough information about a called task, for instance, does it need to maintain a context (such an SQL cursor) or not and at what time this context is released by the application.

The first solution proposed is to use the way in which a transactional context is retained, for the duration of the transaction. Because the completion of a transaction is ordered by the task which initiates the transaction, it can notify the generated service routine in the server stub to issue the tpreturn(), in such way it allows the server to receive the commitment messages. The server waiting for a message expects to receive either a new call for a task or an event indicating that it can issue the tpreturn() function.

A monothreading process such as TUXEDO's means that an associated server is dedicated for a particular client until the completion of the service routine. There is no possible parallelism for this server to perform requests of another client. The fact that I have choose to issue the tpreturn() when it is ordered by the client means the server is dedicated for the same client even if there is no longer data exchanged. For this reason a second solution is proposed where the responsibility to issue the tpreturn() is taken at the server side. In this case, the server can know if there is a context maintained for the client or not. If yes, the server replies to the client with a tpsend() function, in such way that the conversation still maintained to receive another request. If

no, the server issues a tpreturn(). Both cases are expressed in Figure 6.

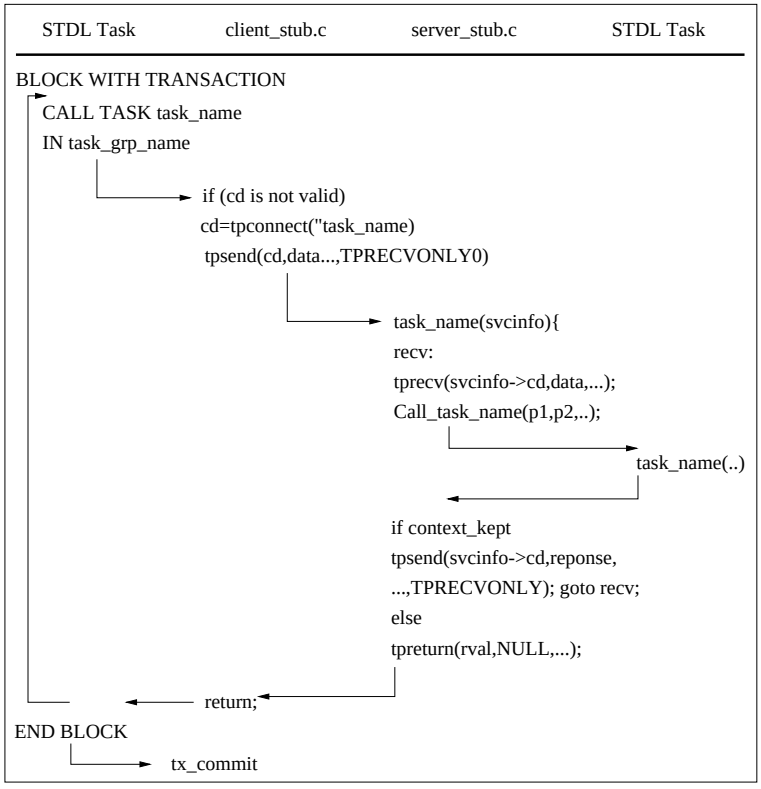

Figure 6: STDL task mapped into XATMI conversational service

5.2 The STDL compiler architecture

The TUXEDO STDL compiler translates STDL definitions (Record, Task) and STDL specifications (Processing group specification, Presentation group specification, Task group specification) into an object form, suitable for linking into an executable form. The compiler generates all the code necessary for supporting the application in the distributed environment, including server initialization, and application context propagation.

A programmer writes only task, processing procedures, presentation procedures, and the corresponding group specifications. This allows him to concentrate on the application flow control, computation, and data access while relying on the STDL compiler to generate required initialization and system flow control operations.

Internally, the TUXEDO STDL compiler consists of a series of steps that run under the control of a driver program. This processing takes place in the steps shown inside the dotted-line box of Figure 7. The STDL compiler first reads STDL specifications or definitions and constructs internal structures that represent each STDL entity in the source file. Once an entity has been completely parsed and the syntax has been checked for errors, the compiler generates intermediate files by translating:

- STDL Task groups into TUXEDO client and server stubs
- STDL tasks into C and TUXEDO run-time service calls

- STDL record definitions into view descriptions translated into C structures contained in C header files or COBOL copy files with the viewc compiler.

The TUXEDO client and server stubs are similar in concept to the "classical" TUXEDO client and TUXEDO server. The client stub is linked with others applications that invoke this group's tasks. The server stub is combined with application code to create the application server image. In other words, the client stub allows to invoke XATMI services enabling to make a request, while the server stub allows to use XATMI services to receive a request and to return the result.

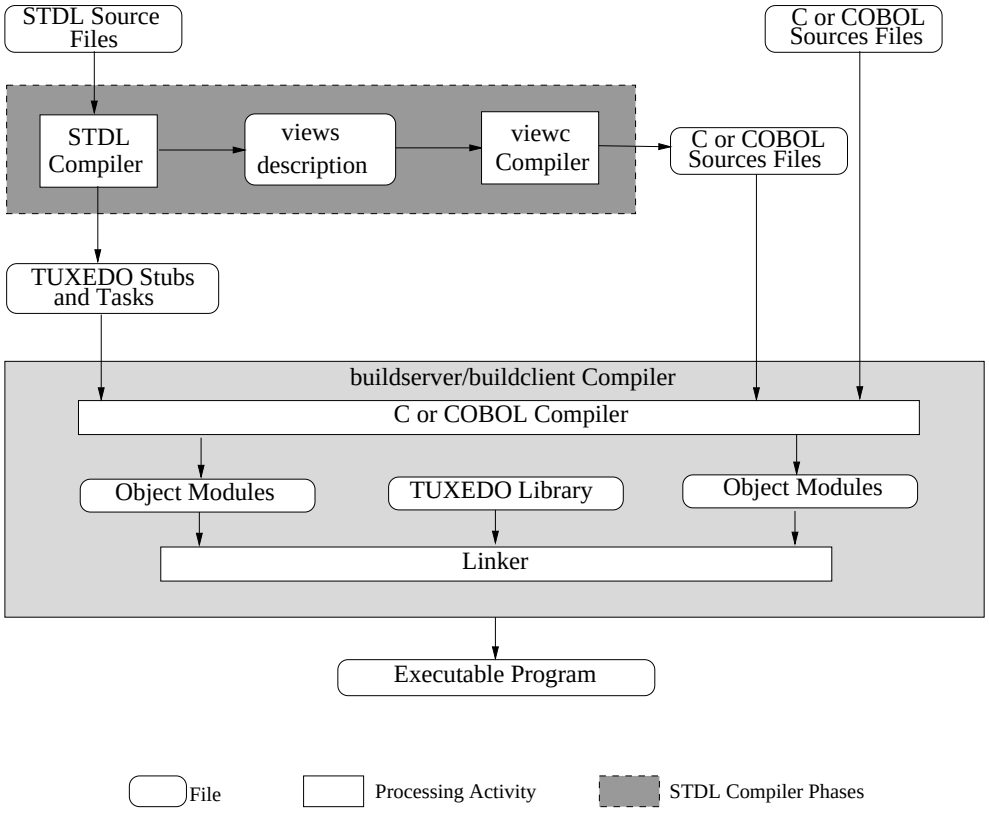

Figure 7: TUXEDO STDL Compiler Flows

After the STDL compiler has generated all the intermediate files, appropriate language processors are invoked to convert the files into object files by buildserver to build the executable server and buildclient to build the executable client. User-written program is built with the buildclient command as follows:

```
buildclient -o CLIENT -f client.c \
-f task_group_client1_stub.c \
-f task_group_client2_stub.c
```

buildclient used to construct a TUXEDO System/T client module allows options. The -r rmname option used to specify the resource manager associated with the user-written program should not be used because resource managers are accessed only by task servers linked to processing procedures, as defined in the STDL specification.

Task servers, which are compiled by the STDL compiler, execute the written STDL tasks and define the flow of control in an STDL application. Task servers can act as clients of other task servers. To build a task server

the buildserver command is used as follows, where the -o option specifies the name of the file the output load module is to have:

```
buildserver -s task_group_name1 -o TASK_SERVER_1 \
-f task_group_server1_stub.c \
-f "task1_1.c task1_2.c" \
-f "deposit.c withdraw.c..."
```

where the entities needed to be linked are :

- default task group server stub object
- referenced task group client stubs objects
- referenced processing header file and processing objects
- tasks definition objects and TUXEDO libraries

Tasks are invoked via the task group in which they are defined. That is, a particular server providing tasks of a task group means it provides this task group and vice-versa. A task group can be viewed as a TUXEDO service routine which calls the appropriate task invoked by a client via a simple procedure call as illustrated in Figure 8.

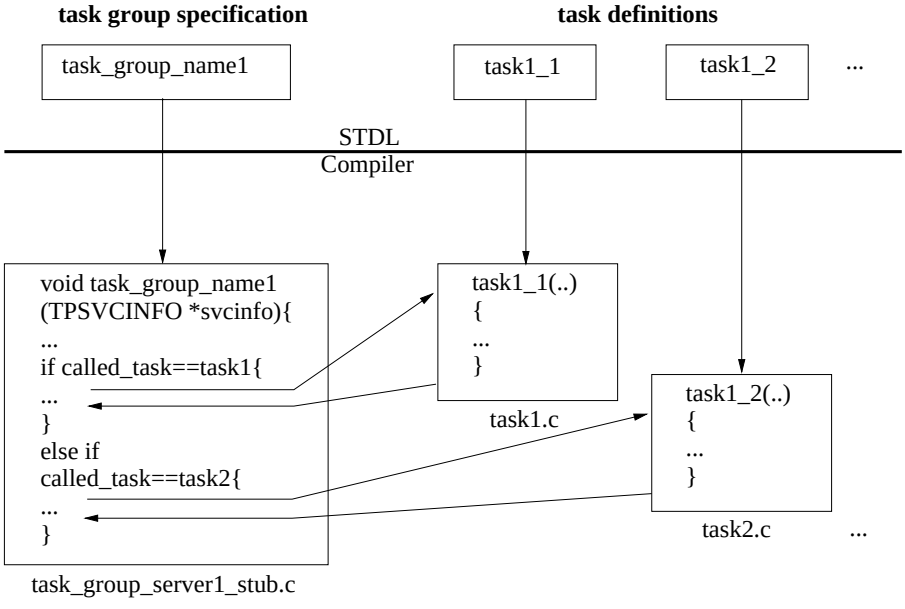

Figure 8: Task group name as a TUXEDO service routine

6 Interoperability through STDL

Although STDL provides a framework at the programming level, the problem of incompatibilities among various communication protocols is still not resolved. To get from closed transaction systems to an open transaction processing environment some new and existing concepts for the interoperability must be realized. Interoperability concerns both interoperability between different communication resources, as defined by X/Open

(TxRPC, XATMI, CPI-C), and interoperability between OMG OTS [3] and X/Open. An important issue is to close the gaps in different specifications for the sake of a useful and practicable realization. This gap can be closed by *bridge* acting as a *proxy*, which converts a request from an environment to a different environment as described in [23].

6.1 OMG Object Transaction Service

The OMG has specified several object services for its object-oriented service platform CORBA. The Object Transaction Service (OTS) is the CORBA service for object-oriented distributed transaction processing. The object transaction service provides operations

- to control the context and the duration of a transaction
- for the participation of multiple objects in a single transaction
- to combine internal changes of object states within a transaction
- for the coordination of the 2PC protocol at the end of a transaction

Figure 9 shows the coherence of the different components and objects of OTS.

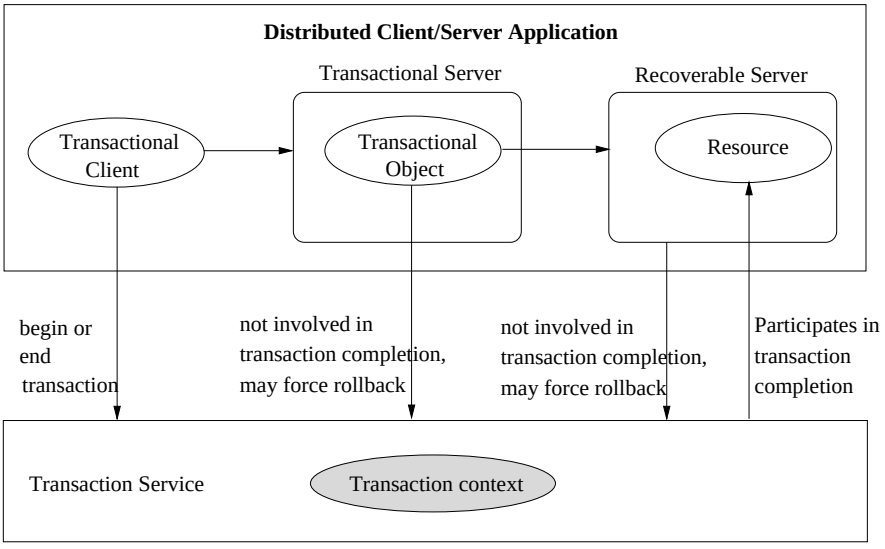

Figure 9: The Components and Objects of an OTS

- An object whose methods can be called in a transactional context is called a *transactional object* (TO). A TO is characterized by including some persistent data or pointers to persistent data, which can be modified by its methods.
- A call to a TO need not be transactional, even if the call is within the context of a transaction. It is left up to the object to determine which calls behave transactionally. Transactional servers and recoverable servers are implemented using TOs.
- A TO which is affected by a commit or a rollback of a transaction is called a *recoverable object* (RO).

- A *transactional server* (TS) consists of one or more objects involved in a transaction, but doesn't have any state information about the transaction.
- A *recoverable server* (RS) includes at least one RO.
- A *transactional client* (TC) can be any program which calls methods of transactional objects in the context of a single transaction.

In the context of the *ACTS* (Advanced Communication Technology and Service) research programme of the European Commission, the *ACTranS* project (A Transaction Processing Toolkit for ACTS, AC081) [21] demonstrates

the interoperability of different DTP systems in heterogeneous environments [17]. ACTranS has achieved an interoperability of the two DTP standards of X/Open and the OMG by using a half bridge based on X/Open compliant Communication Resource Manager (CRM) of the ACTranS toolkit [18]. A demonstration of the ACTranS half bridge implementation is shown at the *4th International Conference on Intelligence in Services and Networks (IS&N 97)* in Como, Italy [2].

To realize a global view on the different DTP concepts for the development of transactional applications, STDL is used as a high-level interoperability and portability concept in the ACTranS project [19]. Starting with the same task-group-specification, the different STDL compilers generate the corresponding stubs. Due to the different protocols of X/Open and OTS a proxy is placed between the two domains to enable interoperability. Figure 10 illustrates this concept.

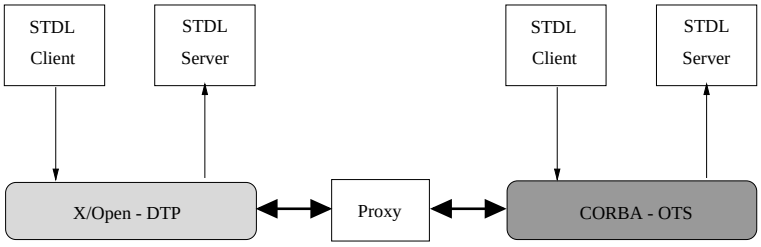

Figure 10: Interoperability between X/Open and OTS through STDL applications

For the end user of the ACTranS toolkit it is transparent if he develops transactional applications either for a single DTP standard, or beyond the boundaries of different standards, as the underlying protocol differences are hidden by the language.

Beyond interoperability between STDL applications on the top of TxRPC and OTS, tested in ACTranS, STDL will enable interoperability between STDL applications over TxRPC or OTS with STDL applications written on the top of a CRM not supporting originally the concept of IDL and encapsulation, such as XATMI.

Starting with the same task-group-specification, the different STDL compilers (for XATMI, or for OTS/ or TxRPC) generate the corresponding stubs. Due to the different protocols of XATMI and OTS (or TxRPC) a half bridge (proxy) is placed between the two domains to enable interoperability. Figure 11 illustrates the different compiler steps to generate an STDL client on top of XATMI calling an STDL server on top of OTS, while in Figure 12 the call is in the other direction.

A similar configuration can be applied for interoperability between STDL applications over XATMI and TxRPC. The generated CORBA IDL can replaced by TxRPC IDL, which can in turn produce a DCE IDL and stubs according to the different TxRPC implementation described in [20].

The proxy includes two kinds of bridges: an application bridge and a transaction bridge. The application bridge is responsible to translate CORBA request (or TxRPC call) into XATMI requests and vice versa. It acts, within the client domain, as a server representing the service, and acts, within the server domain, as a client. The proxy receives a client operation invocation, locates the service and transforms the parameters and their types into a call recognizable by the server domain. Because needed files for interoperability (stubs and IDL files) are generated from a task group specification, and because STDL functionalities and data types are

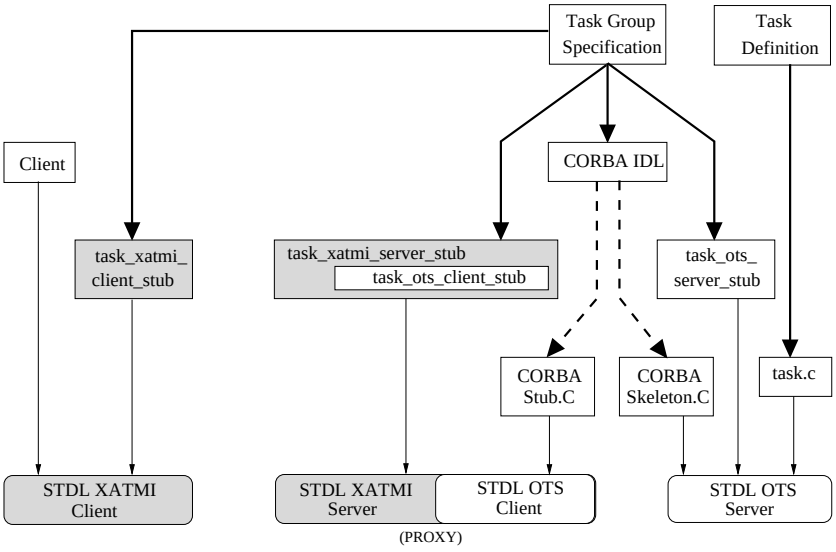

Figure 11: STDL Client over XATMI calling STDL Server over OTS

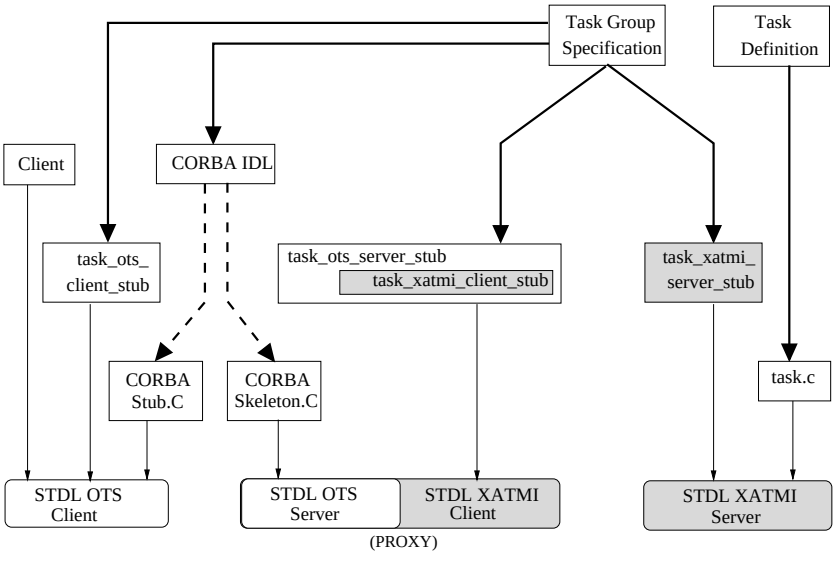

Figure 12: STDL Client over OTS calling STDL Server over XATMI

portable through X/Open and OMG, the bridge or proxy application is generated automatically, preserving a programmer to write it manually and to reimplement it each time the task group specification is modified.

The transaction bridge has the responsibility to control the transaction propagation from one domain to another, by translating the transactional functions. This propagation is meaningful due to the concept of the Communication Resource Manager based on the OSI TP protocol [15], which in the X/Open model is very well suited for the half bridge kernel [17, 18].

Figure 13 illustrates the transaction bridge concept similar to the one used in ACTranS and demonstrated in IS&N, in which TxRPC is replaced by XATMI.

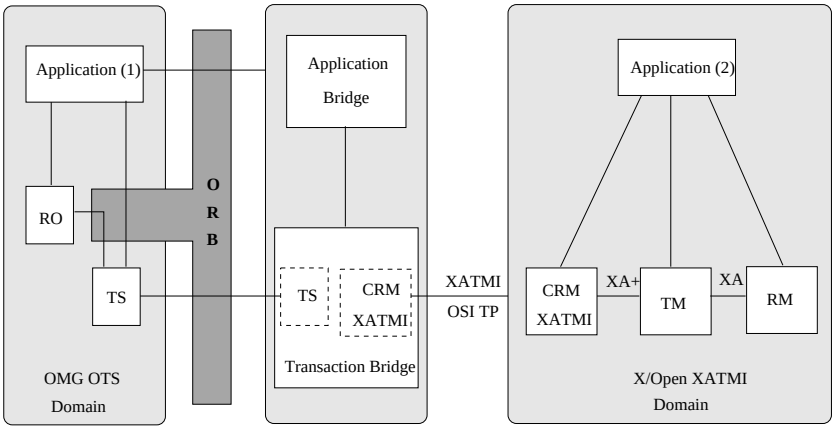

Figure 13: Transaction Bridge for Transaction Systems

Coming from the OTS domain and calling a server in the X/Open, the transaction bridge performs a proxy function, which acts both as a recoverable object and as a XATMI client. Furthermore the module provides a resource object interface to the superior OTS coordinator.

Coming from an X/Open client and calling a recoverable object in the OTS domain, the transaction bridge performs as a proxy, acting as both an XATMI server and an OTS transactional client, and a component which addresses the subordinate OTS by the interposition mechanism.

7 Conclusion

STDL is a procedure-oriented language which has been successfully layered on top of existing TP monitors, and has been adopted by the X/Open consortium for its DTP model as the high-level language. In this paper we have presented how it can be ported on the top of a CRM not supporting the concept of IDL and its encapsulation, such as XATMI. By this way STDL represents the possible track of allowing users to write transactional applications regardless of the underlying TP platform.

Interoperability of STDL applications on top of different commu nication paradigms can be achieved via a bridge which translates the requests from one domain into a format recognized by the other domain. Portability and interoperability provided by STDL makes applications independent of the underlying communica tions mechanism, and enables users to describe transactional applications by using only a single language available for different platforms.

8 Acknowledgments

I would like to thank for their comments on this paper, Eric Newcomer from Digital (DEC), who is also editor and coauthor of the X/Open STDL specification, and Simone Sedillot from INRIA.

References

[1] BEA Systems, inc. products: BEA TUXEDO http://www.beasys.com/product/tuxedo.htm.

[2] IS&N97 - http://www.at.infowin.org/ACTS/IENM/CONCERTATION /ISN/cfp.htm.

[3] Object management group: Object transaction service, 1996.

[4] Telecom '95 - http://www3.itu.ch/TELECOM/wt95/.

[5] X/open CAE Specification: The XA interface specification. x/open company ltd., December 1991.

[6] Novell inc; tuxedo system release 5.0: Application programming, 1994.

[7] X/open CAE specification (working draft 2). distributed transaction processing: The XATMI specification. x/open company ltd., December 1994.

[8] X/open preliminary specification. distributed transaction processing: The CPI-C specification, version 2. x/open company ltd., October 1994.

[9] X/open CAE Specification: Distributed transaction processing: The txRPC specification, x/open company ltd., November 1995.

[10] X/open CAE Specification: The TX (transaction demarcation) interface specification. x/open company ltd., April 1995.

[11] X/open CAE Specification: Structured transaction definition language (STDL), x/open company ltd., 1996.

[12] X/open guide: Distributed transaction processing: Reference model, version 3, x/open company ltd., 1996.

[13] P. A. Bernstein, P. O. Gyllstrom, and T. Wimberg. STDL - a portable language for transaction processing. pages 218–229. Proceedings of the VLDB Conference, Dublin, Ireland, 1993.

[14] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing,*. Morgan Kaufmann Publisher, 1996.

[15] International Organization for Standardization. *Open Systems Interconnection Transaction Processing (*OSI TP*)*, April 1992.

[16] J.Gray and A.Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publisher, 1993.

[17] T. Kunkelmann, H. Vogler, and S. Thomas. Interoperability of distributed transaction processing systems. Proc. Int. Workshop on Trends in Distributed Systems (TREDS 96 Aachen), Springer Verlag LNCS 1161, 1996.

[18] J. Liang, S. Sedillot, B. Traverson, H. Lejeune, G. Vandome, and S. Thomas. Interoperability ots-txrpc, specification, deliverable d2e. Technical report, EEC ACTS ACTranS, 1996.

[19] E. Newcomer, H. Vogler, T. Kunkelmann, and M. Saheb. STDL as a high-level interoperability concept for distributed transaction processing systems. pages 145–154. Proc. 4th Int. Conference on Intelligence in Services and Networks, Springer Verlag, 1997.

[20] S.Sedillot, J.Liang, and J. L. Chimia. Integrating dce rpc with osi tp to offer transactional rpc. First International Workshop on High Speed Networks and Open Distributed Platforms, St Petersburg, June 1995.

[21] F. Vogt. Werkzeuge fier die transaktionsverarbeitung heute - morgen. Proceedings of the 19th European Congress Fair of Technical Communication - ONLINE'96, Congress VI, Hamburg,, February 1996. (only German title, paper is written in English).

[22] W.Rosenberry, D.Kenney, and G.Fisher. *Understanding DCE*. O'Reilly & Associates, Sebastopol, California, 1992.

[23] Z.Yang and A.Vogel. Achieving interoperability between CORBA and DCE applications using bridges. Proceeding of the ICDP'96 - IFIP/IEEE International Conference on Distributed Platforms, Dresden, February 1996.