# Proceedings

# Engineering Distributed Objects

# (EDO ’99)

ICSE 99 Workshop,
Los Angeles,
May 17-18, 1999

Edited by
Wolfgang Emmerich
Volker Gruhn

# Table of Contents

# Engineering Distributed Objects (EDO 99) Introduction

**Wolfgang Emmerich**
Dept. of Computer Science
University College London
London WC1E 6BT
United Kingdom
w.emmerich@cs.ucl.ac.uk

**Volker Gruhn**
Informatik 10
Universität Dortmund
44221 Dortmund
Germany
gruhn@ls10.informatik.uni-dortmund.de

## 1 THEME AND AUDIENCE

This two-day workshop will provide a forum for discussing principles, methods and techniques for the engineering of distributed objects. The workshop audience are practitioners and researchers in software architecture and distributed systems. We received 16 position papers out of which we accepted 13 that are included in this pre-print.

The position papers address various topics related to distributed objects, including the selection of object middleware, the influence of middleware on software architectures, testing distributed objects and the relationship between object middleware and internet scale event notification.

## 2 RELEVANCE

Standards for object-oriented middleware, such as OMG's CORBA, Java's Remote Method Invocation and Microsoft's DCOM have developed and matured over the last decade. They facilitate the implementation, execution and communication of distributed objects. Several products implementing these standards are available now and these products are being used in many development efforts in industry. A considerable number of projects, however, fail because they do not consider the differences between designing distributed objects and building applications based on local objects.

Several research communities in Databases and Distributed Systems have picked up on the topic. They are organizing meetings that are concerned with how to efficiently implement middleware, integrate it with databases to achieve object persistence, and how to administer the resulting distributed systems. There is, however, no established community that is looking at distributed objects from a software engineering perspective. We go a step further and argue that neither the problem, nor principles, methods and techniques for the systematic engineering of distributed objects are fully understood.

## 3 WORKSHOP GOALS

This workshop will seek to achieve several goals. We want to develop a better understanding of the differences between designing local and distributed objects. We believe that these differences complicate the engineering of distributed objects and need to be well understood. We want to identify the potential contributions of related research communities to solving the problems of engineering distributed objects and define a research agenda that will lead to principles, methods and techniques for this purpose.

The design of distributed objects is very different from the design of centralized applications. These differences arise for many reasons. Firstly, operation execution requests between distributed objects are by several orders of magnitude slower than local method calls. Secondly, a method invocation between local objects is synchronous while different forms of synchronisation are needed for distributed objects. Thirdly, local objects are active throughout their lifetime, while distributed objects might have to be deactivated when not needed for a certain period of time; hence these objects must be able to store their states persistently. Fourth, objects whose persistent state is updateable might have to be integrated with transaction monitors in order to implement distributed transactions. Finally, object interactions across public networks might have to be secured against eavesdropping, tampering and other security attacks. All these differences complicate the design of distributed objects.

The second workshop goal is the identification of results developed in related software engineering research disciplines, most notably software architecture, that can be applied to the engineering of distributed objects. On the one hand, current research in software architecture is rather general. Architectural styles and architectural description languages are defined so that they can accommodate many different implementations. This generality inhibits the application of these techniques in practice and renders architectural styles and architecture specifications less expressive. Moreover, it restricts the number of analysis techniques that can be applied. On the other hand, the distributed object paradigm is being used in an increasing number of projects. We believe that by targeting architectural styles and architectural description languages towards implementations with distributed objects, styles and descriptions become more expressive and more powerful analyses will be enabled. It would in addition be a powerful route for the transfer of research results into industrial practice and would clearly enhance the state of practice in engineering distributed objects.

The third goal is the definition of a research agenda that will eventually lead to the development of industrially applicable principles, methods and techniques for the engineering of distributed objects. Items on this research agenda may include

- Relation between requirements and distributed object architectures.
- Suitable architectural styles for distributed objects.
- Relation between architecture description languages and the interface definition languages supported by object-oriented middleware.
- Software processes for distributed objects.
- Differences between distributed and local object design.
- Extensions of object-oriented design methods and notations for engineering distributed objects.

## 4 WORKSHOP ACTIVITIES

Industrial case studies have been selected and they will be distributed to all workshop attendees before the workshop. The case study include a DCOM based architecture for exchange of real-time data feeds at a London Bank, the use of CORBA management of mechanical and electrical engineering data at Boeing, and a CORBA architecture for on-line trading at Bear & Stearns. Papers describing these case studies will be made available for participants before the workshop. Participants will be encouraged to study the papers and prepare short presentations that indicate how the principles, methods and techniques they propose for the engineering of distributed objects can be applied to one or several of these case studies.

The presentations will be used to kick off extensive discussions. They will be organized in different sessions. Though the detailed session breakdown will depend on the accepted papers, we currently foresee sessions on

- requirements engineering for distributed objects,
- architectural styles for distributed objects,
- mapping of architecture description languages to distributed objects,
- concurrency and distributed objects,
- testing of distributed objects,
- persistence and transaction management of distributed objects.

## 5 EXPECTED RESULTS

We hope that workshop will develop a better understanding of the problems that occur when engineering distributed objects and in particular the differences between designing local and distributed objects. We would like to identify routes for the application of software architecture research to distributed objects and lead to the definition of a research agenda for the engineering of distributed objects.

We hope that the workshop will become a focal point for a research community that is interested in distributed objects from a software engineering point of view. We would expect that the workshop will be a starting point for continuous interaction between workshop participants.

The workshop organizers will summarize the result of the workshop and submit a workshop report to ACM Software Engineering Notes. The Case study material and the papers accepted for the workshop will be available from the Workshop web site http://www.cs.ucl.ac.uk/EDO99.

## 6 ORGANIZERS

Wolfgang Emmerich is a Lecturer in the Department of Computer Science at University College London. His research interests include requirements engineering and distributed object-oriented software architectures. Wolfgang is Senior Consultant, Partner and Co-Founder of Zühlke Engineering GmbH. Wolfgang has consulted on several CORBA projects and given numerous industrial trainings and tutorials on OMG/CORBA and distributed object technology.

Volker Gruhn is an Associate Professor in the Department of Computer Science at University of Dortmund. His research interests are software processes for distributed systems, architecture of electronic commerce applications and workflow management. He has been chief technical officer of a German software house called LION from 1992 to 1996. In this position he was responsible for a software development department of 150 people.

## 7 SUMMARY

The workshop covers an important theme of strong interest to industry. It builds on past ICSE workshops while having strong innovative content. It is highly complementary to the main technical programme of ICSE. What is badly needed in software engineering are ways in which industrial practice and academic research can be reconciled. This proposed workshop provides a focal point for such interaction on the engineering of distributed objects.

# EDO 99
# ICSE 99 Workshop, Los Angeles

## COVER PAGE

TITLE: A LAYERED CORBA BASED DISTRIBUTED ARCHITECTURE FOR THE EXTRACTION, TRANSFORMATION, LOAD AND QUERY PROCESSES FOR A HEAVY LOADED DATAWAREHOUSE.

**Keywords:** Datawarehouse (DW) Architecture, DW processes, Functional Interfaces, CORBA distributed processing, CORBA Event Service.

Authors:

PERFECTO MARIÑO
Doctor on Telecommunications Engineering from the Polytechnic University of Madrid (Spain 1984). Professor of Electronic Technology Department (University of Vigo, Spain). Visiting scientist in the Computer Science Department of Carnegie Mellon University (Pittsburgh, USA 1988). Expert on Information Technology from the Commission of the European Communities for the SPRINT (Luxembourg 1991) and COPERNICUS (Brussels 1994) programs. Director of Digital Communications Division from the Applied Electronics Institute (University of Vigo, Spain). Member of IEEE. e-mail: pmarino@uvigo.es

CESAR A. SIGÜENZA
Telecommunications Engineer from the Industrial and Telecommunications Engineering University (Vigo, Spain 1995). PhD researcher of Digital Communications Division from the Applied Electronics Institute (University of Vigo, Spain). e-mail: csiguenza@uvigo.es.

MIGUEL A. DOMINGUEZ
Telecommunications Engineer from the Industrial and Telecommunications Engineering University (Vigo, Spain 1993). Researcher of Digital Communications Division from the Applied Electronics Institute (University of Vigo, Spain). e-mail: mdgomez@uvigo.es

FRANCISCO POZA
Electronic Engineer from the Industrial and Telecommunications Engineering University (Vigo, Spain 1986). Full Professor in the Electronic Technology Department (University of Vigo, Spain). Researcher of Digital Communications Division from the Applied Electronics Institute (University of Vigo, Spain). e-mail: fpoza@uvigo.es

JUAN B. NOGUEIRA
Telecommunications Engineer from the Industrial and Telecommunications Engineering University (Vigo, Spain 1993). Associate Professor in the Electronic Technology Department (University of Vigo, Spain). Researcher of Digital Communications Division from the Applied Electronics Institute (University of Vigo, Spain). e-mail: nogueira@uvigo.es

Name and address for correspondence:

Perfecto Mariño
E.T.S. Ingenieros Industriales
Departamento de Tecnología Electrónica (Universidad de Vigo)
Apdo. Oficial
36200 Vigo SPAIN

Tel: +34-986-812162
+34-986-812223
Fax: +34-986-469547
e-mail: pmarino@uvigo.es

Submission to EDO 99, ICSE 99 Workshop, Los Angeles

# A Layered CORBA Based Distributed Architecture for the Extraction, Transformation ,Load and Query Processes for a Heavy Loaded Datawarehouse.

**KEYWORDS:** Datawarehouse processes, Layers, Functional Interfaces, CORBA distributed processing.

## I. ABSTRACT.

The authors are involved in the implementation of a real Datawarehouse (DW) of a half terabyte of data on a Telco corporation. Firstly the driving forces in a DW are explained, and following design issues founded in this implementation are stated like different scenarios.

The discussion of these scenarios allows to authors proposing an architecture in order to overcome the founded big challenges. After introducing CORBA as a powerful tool for implementing this architecture, conclusions about achieved results and future works are stated.

## II. DATAWAREHOUSE DRIVING FORCES.

Nowadays there is a proliferation of DW tools in the market ready to install and solve any problem encountered but sometimes they overlap giving the chance to solve one problem in many places. This increases the noise in the sense that it is not always clear where to touch to change a process.

There should always be in mind that the main driving force in a DW is the Final User. His needs are mapped into the DW as table structures, functional transformations and so on until we reach the data source that we can not control in the sense that we can not introduce changes to fit user needs. With this in mind, if a DW can not follow the changes imposed by the user needs: the DW will not be useful anymore

### 1. A snapshot of a DW

Analyzing the code that supports the studied DW, we have found some useful concepts for the representation of DW related problems. The context of a DW can be seen in figure 1. There are five different parts.

**The Real Observed Systems (ROS)**
They exist independent of the DW. From the DW point of view they are pure data sources, but not ready to be read.

**Data Sources**
Generally the data consumed by a DW is being stored into an already existing data store who lives inside a control system not designed to answer complex queries on large volumes of historical data. Sometimes they are called ‘Operational Data Stores’ (ODS). From the DW point of view they are just accessible data sources. They also live independent of the DW. Frequently the data store formats have a high impedance, in terms of performance of the user queries, so they must be preprocessed before going up to the DW data store.

**The DW**

Two interfaces can be found. The first one is towards user access tools. The DW must know the ways the user is going to manage that data. This Information is transformed into structures and functional transformations into the DW system, in such a way that if the user needs changes, the DW will need them too. We can also include here the user makeup's that make more readable the final answer like changing a decimal code by a textual description.

The second interface is towards the data source, involved in three matters.

First, it must handle all the source related problems, like:

- Extraction with proprietary query-languages from the data source,
- Transport when data must be networked,
- Cleansing like the elimination of duplicated or out of range values (data validation process). A tool like 'Syncsort' [SYN96] simplifies this kind of work.

Second, it must make transformations like aggregation or summarization of data, and the more elaborated cross-referenced mappings with data coming from different data sources, in order to decrease the user-queries impedance and improve performance. On small volumes of data this process is implemented using stores procedures in the database, but when performance becomes a bottleneck this transformations must be done with file processor tools like 'Prism' [PRI97].

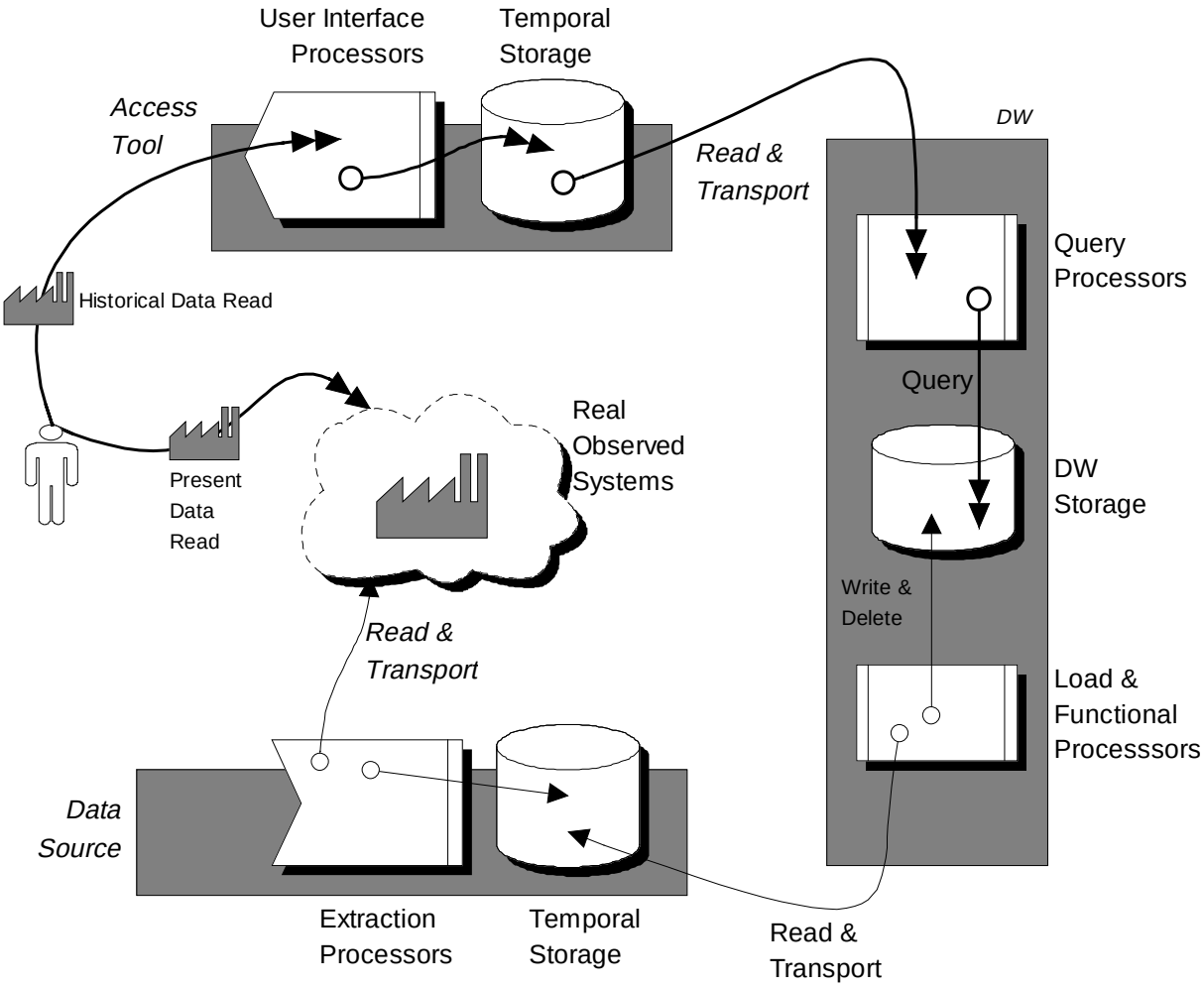


Figure 1. Context of a DW.

And third, it must load the transformed data into the database tuned structures.

It should be clear that the processes involved on the user side are absolutely different from the ones living at the data source side. The arriving of queries from the user is unpredictable, but the arrival of data from the data source is usually periodic.

**User Access Tools**
They can be custom (not a good idea if they are hard to modify at user-changes speed) or market tools in the segment of DSS, like 'BusinessObjects'[BO96]. They usually have their own temporal storage (a buffering strategy that decreases user-latency time and avoids congestion at the database server side). These tools provide a wonderful interface to the DW in such a way that if something changes, the tracking and adaptation takes minutes.

**The Final User**
Is the most important part of the puzzle and the driver of the system. From the DW point of view an user is defined by the data he wants to get, the date time window he is going to look at and the latency time between the query and the response (a matter of performance ). There is another variable that a DW must manage: the number of simultaneous users loading the system. There are different strategies to avoid bottlenecks, not seen here. From the user point of view, if latency time increases or the data is inconsistent, the DW is not worthless anymore, so there is a thin line between defeat or success.

## 2. Some Different Scenarios

Once you decide one strategy to build a DW, you build it and if it works fine, changes in the specifications from the user to the data source will start to appear. Let's see a few forces that demand instant changes to the DW. Sooner or later one of those circumstances will appear.

### CASE1: The user needs a new data view

This seems to be innocent but imagine the complications when the data that you already have in the DW does not match user needs. For example, suppose that the new backup window (the range of historical data) is greater than the one you keep at the moment. Do you change the original Backup window and affect previous systems (10 views for example) or you make a new table (the eleventh view) that must be filled by the same processes that fill other structures? there are many factors that trade off between those solutions. Different solutions drive to different parts to be changed in the system. The best one will be the one who survives to the changes. If the changes can be made on different places with no functional difference take for sure that performance may have something to say, and that someday it will be critical.

Without an architecture it's not easy to manage changes due to cross references between pieces of code if behavior is spread inside them. A better code tracking is needed to allow impact analysis.

### CASE2: More users for the same query

One day arrives a new user and the system collapses. You decide to denormalize the relationships between tables, but you can't do that because there are one hundred joins pointing to your tables. And what is worse: you don't know which ones because this kind of metadata is implicit in your SQL statements. It i not easy to answer the request :'Let me know how many processes read this table'.

This kind of relationships are more important as the system grows. Improved manage of metadata is needed.

**CASE3: Format changes on the data source**

File formats of version upgraded systems are always compatible in such a way that some code must be redesigned to fit inconsistencies. The changes could be minor but enough to blow up the hypothesis stated about a file structure. The consequence is that a new functional preprocessor must be designed in a very short time. There are worst cases, for example when data fields fit the same type but their locations in the old and new files are swapped. As the error is semantic, not syntactic, it propagates through the system increasing the noise without notice.

The use of uniform interfaces between processors will allow to test if data still meets trusted hypothesis. The identification of modifications will be easier and the propagation of lateral effects are removed from the system.

**CASE4: The data source load window**

Usually the data of a DW is refreshed periodically during the 'Load window' period. Suppose that one day some data source begins to generate more data than expected, this will mean more time for processing, time that can fall out of the Load window. Data will not be loaded. Some strategies solve this problem improving the performance of that process, but this has a limit. A variant approach is to move the functional to a different machine, generating an exception to the maintenance team. This is not easy to track over time.

The CORBA bus can be used to solve this problem eliminating transport problems between hosts reducing maintenance effort.

**CASE5: Broken sequence re-launching of data source processors**

This is very typical. Suppose that you need to summarize data provided by nine independent data sources and one fails or gets late. Data will not be loaded. If your system is better designed, it will let you launch the execution of parts of the whole chain, but this means new information to be recorded because you must be sure to be in the same state at each step of the processing algorithm as when the system executes normally (if not, you can process garbage). This implicit sequences must be made explicit and recorded somewhere to be read by a sequencer who reads this information to decide to wait, skip or resume.

There are many behaviors that can be better modeled with an component model. It is necessary to register information state. A finite state machine like a sequencer will improve performance and will allow the system to load slices of data on demmand.

The CORBA Event Service can be used to improve performance as allows the sequencer to synchronize the arriving of data with the execution of their functional processors. The distribution of congestion on the load window can be controlled.

**III. PROPOSED ARCHITECTURE. Part I: The Interfaces**

The origin of this architecture comes from an reengineering process [HAM+94][RUM95]of all the processes involved in a real DW [INM93] of a half terabyte of data on a Telco corporation. We first heard the problems the system was offering. Then we analyzed the processes in search of patterns [GAM+95][FOW97][SHA96][TAY95]. It was found that functionality was spread all over the code on different systems, and this complicated the systems with no benefit.

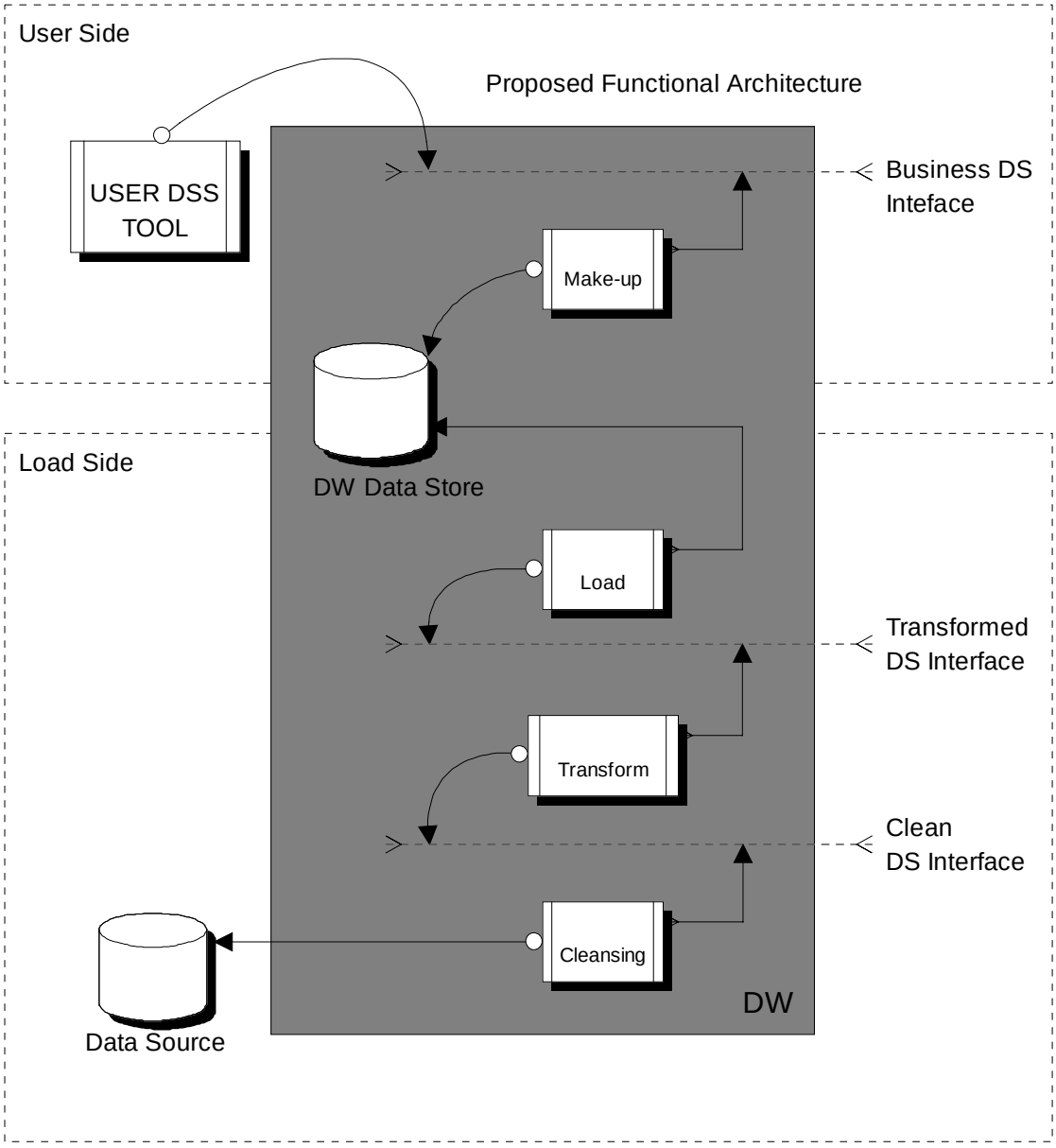


Figure 2. Proposed functional architecture.

Patterns were found where data flowed through quite similar operations but never in the same order, and there was no reason for that. The idea we propose is to identify and group the kind of operations that can be done on each of the layers created by the interfaces. reducing the kind of problems that can be found into each layer.

The great advantage of using interfaces is that it creates a point of reference for the input /output formats of data between layers. This makes code more easy to reuse as you can take for sure that once a piece of data reaches an interface, it fulfills some hypothesis that can be used to eliminate operations on upper layers. This avoids damage , and improves consistency of data, besides defining a place where to detect what kind of problem without having to navigate into a sea of code. That's the main idea. Besides, developers reduce the size of the code and as the interfaces are well defined it's more easy to define debugging or test tools for the interfaces instead a test for each kind of data that arrives to the system.

Developers also have the chance to design and document [MUL97][POW98][FOW97b] with a few statements with improved semantics, because the kind of operations that can be done on each layer are reduced and more related.

This architecture needs a sequencer, not shown in the figure. This joins explicitly all the information about when to call all the processors and manages all the trigger conditions. Another benefit is that splitting the code encapsulating functional parts it's far easy to make with out cross-references to execution matters. A sequencer helps to monitor and control the load of a DW from the data sources, a huge effort for the DW team when something goes wrong.

We have studied almost 2000 of pieces of code and searched for patterns inside them. The first behaviors permitted us build some components that substituted efficiently lots of almost similar but different code. We identified the general cases and build some components that gathered that critical information that was not so easy to record on an unique place. In this architecture we propose names to some entities. Those names improved the communication between team members.

This implicit metadata formed by the relations between files, tables and functional transformations, and by the sequencing logic it's a first order matter to deal with in a DW. The distinction between functional transformation and sequencing logic simplified the model .

The design processes involved in the solution of a new user request can be linked following the next proposed methodology (in the test phase):

1. Record user needs in terms of data selected, scanning fields and latency time.
2. With the data identified, try to find them in the accessible data sources.
3. If they are found, design the DW data store taking into account performance needs
4. Design the extraction and transport processes
5. Design the functional transformations
6. Design the makeup transformations (this reduces the amount of processing on user-tool side). this can be done after or before the user queries the DW.

As soon as data is ready (on any step) a feedback to the user must be made in order to track changing user needs. We have found, as the concurrent engineering practices states, that some information found on earlier stages of the design affects later stages and the order proposed in the methodology allows fast detection of problems.

## IV. PROPOSED ARCHITECTURE. Part II: The CORBA solution

The CORBA bus can be used to make code transparent to transport concerns between heterogeneous data sources. This simplifies code and reduces maintenance. As a result and with the use of functional interfaces, performance can be improved by moving processor to different hosts.

The target DW has one central node as a collector. This node receives data from more than 50 data sources, each with different protocols, etc. Sometimes the recollection of data must be initiated by the system and other times data comes asynchronously from the remote system. Sometimes some code must be installed in the source that communicates with matched code at the central site. This kind of heterogeneous situations result in easy lost of control. It should be better to define one interface at the central site and then create agents at the source side with the proper logic.

The distribution of behaviour in the data sources gives us a chance to simplify the code at the congested side in a N to 1 relation . This could be done developing CORBA objects serving any client code, running on the remote data source , collecting data for the DW. Even more, some functional preprocessing can be moved to the source if needed without breaking the architecture. The feedback that something goes wrong is detected very fast and problems can be arranged involving less people and less machines.

The CORBA Event Service can be used to improve performance on execution of the functional transformations. As soon as data arrives to the system. [SIE96][ORF+96] data can be processed if the adequate functional processor is activated. There also could be a smart scheduler deciding whether to launch a process or not depending on loading needs to avoid deadlocks, for example.

We have made an emulation of this behavior with some code launched periodically who detected the arrival of files, cleaned it up and triggered the functional processor. Visibroker from Visigenic and OrbixWeb from IONA are being tested. Our preliminary results show that CORBA can be one solution to those problems.

## V. RESULTS

Up to date we have identified a DW as a general component with two main behaviors on the load side: The load of periodic data and the elimination of obsolete time-slices of data on each of the tables of a DW. The execution of those processes was spread all over the system with a specialized process for each table. We detected some patterns on those behaviors and eliminate 40% of the existing code needed to handle one table in the DW by constructing a general handler of such a process. A major benefit was to find how structural information such as the size of the Backup window for each table, relayed gathered into a table in a natural place improving the reaction time to changes in this matter. With this little change we have reduced the time to develop code and increasing the tracking over the systems. this is very useful on impact analysis when analyzing possible design decisions.

The next step in this way is to identify all those structural information and the source/consumer processes that use it, and follow this line.

We have been emulating an event service with some data structures and some “daemons” to detect flaws on conceptual design. The results are better than before, allowing processes to finish inside the Load Windows, pointing out that the CORBA Event Service can be a great chance to use standard functionality. The benefits met are: improved tracking, reduced coding with reduced maintenance, and improved reaction design time when the system manages structural information on an explicit way.

## VI. CONCLUSIONS

The problems surrounding a DW are always related with changes on specifications. We proposed an architecture with layers and interfaces that pretend to isolate those problems as soon as they appear and as near to the data source as possible, to avoid error propagation through the system. We introduced DW concepts useful to identify needs on market tools and to improve communication between team members.

This architecture is beeing tested on a real DW of up to a half terabyte of data and almost 50 data sources. The data bases are implemented in ‘Sybase’ and ‘Sybase IQ’ and the hardware is a Sun Sparc 10.000.

The main benefits are:

- Improved design flexibility as interfaces allow functional parts to be moved between hosts on demand.
- Reduced design time because it is easier to find patterns to match with because each one has its own semantic and it is easy to identify the kind of operations that are being made to data. Reuse is possible.
- Improved tracking
- Execution on demand for processing broken pipes.
- Reduced code size and less maintenance by using components.

Future work deals with the implementation of CORBA as stated, pretending to simplify even more the code supporting the whole architecture.

## VII. ACKNOWLEDGEMENTS

This work was made from three R&D projects sponsored by the following entities: R&D NATIONAL SECRETARY AND CICYT, Ref. TIC97-0414, CENTRAL GOVERNMENT (MADRID, SPAIN); UNIVERSITIES GENERAL OFFICE, Ref. 64502I802, AUTONOMOUS GOVERNMENT (GALICIA, SPAIN); and RESEARCH VICE-CHANCELLORSHIP, Ref. 64102I710, UNIVERSITY OF VIGO (SPAIN).

## VIII. REFERENCES

[BO96] Business Objects: Business Objects for windows user's guide, Part number 310-10-400-01, 1996.

[FOW97b] Fowler, M.:UML distilled. Addison Wesley, 1997.

[FOW97] Fowler, M.:Analysis Patterns: Reusable object models. Addison Wesley, 1997

[GAM+95] Gamma, E., Helm, R., Johnson, R., & Vlissides, J.:Design Patterns. Elements of reusable Object Oriented Software. Addison Wesley, 1995.

[HAM+94] Hammer, M., & Champy, J.:Reingenieria de la empresa, 1994.

[INM93] Inmon, W.:Building the Datawarehouse. John Wiley, 1993.

[MUL97] Muller, P.:Modelado de Objetos con UML. Eyrolles, 1997.

[ORF+96] Orfali, R., Harkey, D., & Edwards, J.:The essential Distributed Object Survival guide. John Wiley, 1996

[PRI97] Prism solutions Inc: Prism Warehouse Executive v1.5 user's guide, Doc Number PWE15UG, 1996,1997.

[POW98] Powell, B.: Real time UML. Adisson Wesley, 1998.

[RUM95] Rummler, G., & Brache, A.: Improving Performance. How to manage the white space on the organization chart. Jossey-Bass, 1995.

[SHA96] Shaw, M., & Garlan, D.:Software Architecture. Prentice Hall, 1996.

[SIE96] Siegel, J.: CORBA fundamentals and programming. John Wiley & Sons, 1996.

[SYN96] Syncsort incorporated: Syncsort Reference guide, 1996.

[TAY95] Taylor, D.: Business Engineering with Object Technology. John Wiley & sons, 1995.

# A Comparison of three CORBA Management Tools

Bernfried Widmer and Wolfgang Lugmayr

Technical University of Vienna, Distributed Systems Group
Argentinierstraße 8/184-1, A-1040 Vienna, Austria, Europe
{bwidmer, lugmayr}@infosys.tuwien.ac.at

## Abstract

*Distributed architectures show complex dependency relationships between application components on the one hand and between application components and components of the underlying system on the other hand. Management of such systems requires means to monitor and control the behaviour and relationships of these application components. The CORBA specification does not provide such management support but proprietary developments fill that gap. We evaluated three tools that focus on management of distributed applications based on the Orbix C++ CORBA implementation: Orbix Manager, Corba Assistant and Object/Observer. We investigated the capabilities of these tools and identified the benefits and shortcomings of these according to criteria such as overhead, ease of instrumentation, management configuration capabilities and integration with standard management frameworks.*

## 1 Introduction

The Common Object Request Broker Architecture (CORBA) has been receiving an growing industrial acceptance as the middleware solution for distributed object systems during the last years. These systems typically comprise multiple distributed components that have to cooperate in order to fulfil a certain function. The number of involved components and their complex interdependency relationship require specific management support in order to detect faults or performance bottlenecks, to guarantee system availability, to predict the impact of changes in the configuration and to support other administrative tasks.

The CORBA specifications do not define any built-in management support. Systems management is currently treated at the facilities layer within the Object Management Architecture (OMA) [19], which is the framework of CORBA. The specifications so far called Common Management Facilities (XCMF) [15] focus on support for policy-driven objects. To date, low level management support, such as the performance instrumentation [4] specified for the OSF Distributed Computing Environment (DCE) is missing (see [21] for discussion). It is therefore up to ORB vendors and application developers to what extent they supply their products with management support.

We evaluated and compared three management tools for the management of *Orbix*[1] based systems. These tools require extending the source code of CORBA processes with specific management instrumentation in order to enable management. These tools are:

- *Orbix Manager*[2] *(Version 1.0c)* is a management tool developed by IONA to provide management support for their Orbix ORB implementation. Its architecture strongly adheres to common systems management models as the management application queries processes on demand for management information. Moreover, the instrumented processes may emit unsolicited events when CORBA exceptions occur. The CORBA processes may also be accessed through an SNMP gateway.
- *Corba Assistant*[3] *(Version 1.2)* is a third-party tool that adheres to network management principles. Instrumented processes are inquired on demand or periodically. It allows to monitor individual CORBA object instances in addition to CORBA processes.
- *Object/Observer*[4] *(Version 1.1)* is not a management tool in the common sense. It rather adheres to distributed software testing tools. It enables to monitor CORBA requests including parameter and return values. A setup editor enables to select the methods and parameter values to be monitored as well as the monitor points (client or server and incoming or outgoing). The monitored requests may be recorded to files in the background and inspected by a parser tool.

We used the following criteria for the evaluation: capabilities, user interface and documentation, overhead, instrumentation effort, management configuration control capabilities, resource management, integration with management standards, openness, built-in security, and measurement accuracy.

[1] Orbix is a widely deployed CORBA implementation. It is a trademark of IONA Technologies PLC (http://www.iona.com).
[2] Orbix Manager is a trademark of IONA Technologies PLC.
[3] Corba Assistant is a trademark of Fraunhofer-IITB (http://tes.iitb.fhg.de).
[4] Object/Observer is a trademark of Black&White Software (http://www.blackwhite.com).

The paper is organized as follows: Section 2 describes the evaluation criteria. In Section 3 we describe the testbed environment as well as measurement tools and applications used for the evaluation. Moreover, each tool is assessed according to the presented criteria and an overview comparison is given. A discussion on common approaches of the different tools concludes the section. In Section 4 we draw some conclusions.

# 2 Evaluation Criteria

This section describes the evaluation criteria for the tool assessment.

## 2.1 Capabilities

Leinwand and Fang [8] describe how network management tools are and their capabilities. They describe a classification, which uses the functional breakdown of management capabilities into Fault, Configuration, Accounting, Performance, and Security (FCAPS) of the OSI Systems Management framework. Moreover, it distinguishes between simple, more complex, and advanced tools. The more complex tool extends the functionality of the simple tool, and the advanced tool extends the complex tool subsequently. The following outline is used to classify the evaluated tools.

### 2.1.1 Fault Management

- **A simple tool** indicates the existence of a problem.
- **A more complex tool** detects faults. Recall that a fault is the cause of a failure or problem.
- **An advanced tool** detects and corrects faults.

### 2.1.2 Configuration Management

- **A simple tool** provides a central storage for configuration information and a search function on the stored data. The storage has to be maintained by the administrator.
- **A more complex tool** automatically gathers configuration information and stores it at a central site. Additionally, the tool is able to detect deviations between the stored and the actual configuration. Based on this information it allows changing a managed object's configuration.
- **An advanced tool** stores the configuration information in a database, thus providing extensive search functionality and queries.

### 2.1.3 Accounting Management

- **A simple tool:** allows monitoring for a metric that exceeds a quota.
- **A more complex tool:** provides billing based on given billing domains and automatically collected accounting information.
- **An advanced tool:** uses metric data and quotas to forecast the need for resources.

### 2.1.4 Performance Management

- **A simple tool:** provides real-time information about utilization, preferably in graphical form.
- **A more complex tool:** allows setting thresholds and reports if a threshold is exceeded. Additionally, it provides histories of performance information, which are useful for diagnostics.
- **An advanced tool:** supports the administrator in detecting performance bottlenecks. The tool may use historical and topical information suggesting improvements and potential future bottlenecks.

### 2.1.5 Security Management

- **A simple tool:** graphically shows the security configuration of a system.
- **A more complex tool:** monitors the system and sensitive areas in particular. It sends notification on access to sensitive data and on potential intrusions. Logging is a central service of this tool.
- **An advanced tool:** helps to predict the impact of security measures on a system concerning performance and related issues.

## 2.2 Overhead

The management systems should affect the performance of the managed entity to a minimum degree. In the network management area, a useful rule of thumb is that *"the absolute maximum allowable bandwidth consumption by management operations should be 5 percent."* [20]. Similarly the overhead of OSF DCE performance instrumentation is limited to 20 percent CPU time overhead [4]. The latter seems appropriate to CORBA management overhead.

## 2.3 Resource Management

There is usually a trade-off between performance and resource accounting. The instrumentation can be kept small and fast in moving the burden of calculating measures and storing data to persistent memory to a parallel process. Brunne [1] calls this pattern *Application-based Agent* in contrast to the *Library-based Agent*

where instrumentation and agent reside within the same address space (see Figure 1).

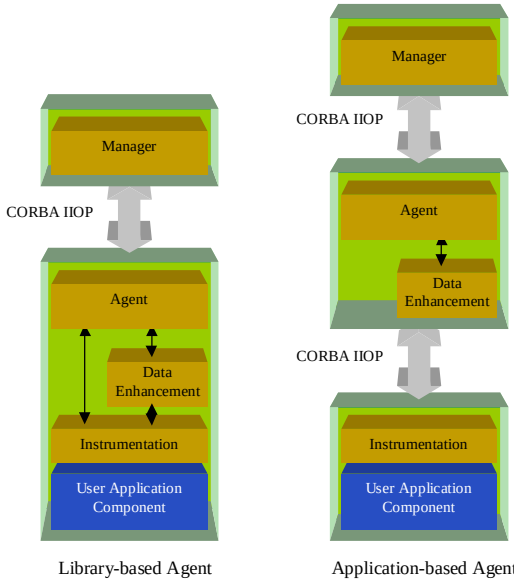


**Figure 1. Library-based vs. Application-based Agent**

## 2.4 Measurement Accuracy

Management information accuracy mainly concerns performance management. The management instrumentation should ideally not affect the timing measures. The possibility of information loss is another topic for the classification of accuracy of collected management information. Information loss can occur due to several reasons including unreliable transmission, process shutdown, and maliciously tuned storage policies.

## 2.5 Degree of Integration

The "degree of integration into managed resources" or into the application in particular [3] is the amount of modifications of a resource required for instrumentation. Three levels are identified:

- **Low: unaware to the resource.** No changes have to be done to the resource. The resource itself is unaware of the instrumentation. This is the ideal case and may be done by modifying the underlying system.
- **Medium: linking/minor source code modifications.** This level ranges from simply linking the program with additional libraries to small source code modifications. The source code changes at this level require no knowledge or understanding of the program's source code and only basic knowledge of its programming language. The changes comprise few lines of code at a single component, source file or class.
- **High: major source code modifications.** This level requires programming language knowledge and comprehension of the source code of the program. An example is the OSF DCE instrumentation that requires to insert instructions (probes) at appropriate points in the source code.

There is usually a relationship between overhead and degree of integration. Low degree of integration usually implies a higher computation overhead, as the instrumentation has to cope with unstructured data types such as message buffers or byte streams. The instrumentation has to parse these structures for the required information, which in turn requires meta information about the data structures.

## 2.6 Management Configuration Capabilities

Management imposes additional load on the system. It is therefore important to be able to configure the amount of management related load or to turn management instrumentation off completely. Significant for the workload is the *point of configuration* in the management information data flow. Turning the management information gathering process off at the management application would in general not stop it at the agent or the managed resources themselves.

Another issue is the *granularity* of the configuration. We define the level of granularity by the count of configuration setting permutations i.e. the total count of different configuration settings. The lowest granularity level is to turn management on and off.

Moreover, the *ease of configuration* is considered. We identified three levels:

- **Level 0: configuration at runtime.** An example is a CORBA object implementation providing proper methods to change the management configuration. A client can then bind to the object implementation and change the configuration through the object's interface at runtime.
- **Level 1: stop/restart configuration.** The software must be stopped and restarted in order to change its management configuration. An example for this level specifies the configuration through command line arguments.
- **Level 2: recompilation.** In this case, the management configuration requires changes in the source code of the managed object.

## 2.7 Integration into Standard Management Frameworks

The benefits from integrating CORBA managed systems into management standards are the preservation of investments in these standards. The user can use the standard management application to access the CORBA managed system. Moreover, CORBA is just part of a system and depends on other system components. The objective is to provide a common management view of a system and not to force the user to access each domain using a different application or even access policy.

The integration into or inter-working with standard management frameworks is covered in detail in [21].

## 2.8 Openness

Users should be able to integrate a management solution into their architecture and to extend the functionality according to their requirements. In addition, since distributed systems show very dynamic behaviour openness is a key requirement in order to be able to accommodate environmental changes.

In general, openness is achieved by specifying and documenting the key software interfaces of a system and publishing them [2].

## 2.9 Built-In Security

Whereas security management capabilities of a management application provide security monitoring and controlling for the managed system, built-in security deals with security management of the management application itself and the access to management information in particular. Two major security threats are identified:

- **Access to Sensitive Data.** An attacker could gain access to sensitive data by reading management information. Management information can be directly or indirectly sensitive. Indirectly sensitive information covers information, which isn't sensitive, but in conjunction with other non-sensitive information allows drawing conclusions about sensitive information. Such security threads are usually encountered in database security area commonly referred to as *inference control* [9]. More severe, management information may contain sensitive data in the clear.
- **Access to Management Instrumentation.** Unrestricted access to management instrumentation is a potential threat even if no sensitive information is revealed. An attacker could use the management system to load work on a system. This may considerably affect performance and could result in the shutdown of critical services.

Therefore, the management tool should provide encryption and authentication support. In fact, today's standard protocols such as the Simple Network Management Protocol (SNMP) provide both [18, 20].

## 2.10 User Interface and Documentation

User interface and documentation are the significant factors regarding the cost of learning process for users. After all, management systems aim to support administrators in their task by providing common means to monitor and access application behaviour. Undocumented features are simply unknown to the user and are subsequently not used.

# 3 Tool Evaluation

This section describes the environment and assessment of each tool according to the criteria described in Section 2. A more detailed description of this evaluation can be found in [21].

## 3.1 Testbed Environment

### 3.1.1 Hardware and Software Platforms

Orbix Manager and Corba Assistant were evaluated on a Sun SPARCstation 5 equipped with 64 MB RAM running Solaris[5] 2.5. Orbix Manager required Orbix multi-threaded (MT) version 2.3 C++ mapping, whereas Corba Assistant ran on Orbix MT version 2.2. The GUIs were located on a remote machine.

Object/Observer was tested on a PC architecture equipped with two Intel Pentium[6] 90 Mhz processors and 128 MB RAM. Windows NT[7] Server 4.0 was installed, running Orbix MT version 2.3 C++ mapping.

All performance related measurements were performed locally.

### 3.1.2 Overhead Measurement Policies

The different agent patterns as shown in Section 2.3 imply two different measures. First, the request latency overhead is measured. This quantity provides a user centred measure and is affected by all parts of the management environment. Second, the CPU time is measured. This quantity is more reliable and focuses on the overhead of the instrumentation within the process. The multithreaded environments imply high variances for both measurements due to different influences

[5] Solaris is a trademark of Sun Microsystems (http://www.sun.com).
[6] Pentium is a trademark of Intel Corporation (http://www.intel.com).
[7] Windows NT is a trademark of Microsoft Corporation (http://www.microsoft.com).

such as synchronization overheads and thread waiting times [7].

Overhead is measured on a per-method basis. The relative overhead imposed by the management instrumentation depends on the operation's total execution time. However, it is difficult to define an average computation time for an operation due to the variety of application domains. Therefore, the overhead measurement is restricted to communication time by implementing methods that return immediately without performing any computations. The definition of communication time in this context is the time consumed by the transmission of a message as perceived at the application layer. Thus, it includes the time consumed by the data marshalling on the sender side as well as the time consumed by data demarshalling and target object unmultiplexing on the receiver side in addition to the actual message transmission delay [5].

The overhead to be expected is therefore in general considerable lower depending on the server side computation time of the operation.

#### 3.1.3 Measurement Functions

| | **Windows** | **Solaris** |
|---|---|---|
| Latency | MicroTimer | gethrtime() |
| CPU | GetProcessTimes() | times() |

**Table 1. Measurement Functions**

The timing measurement on the Windows platform was accomplished by an implementation of the *MicroTimer* class [10], which uses high resolution performance timer provided by the Windows32 API. The CPU times were measured using the *GetProcessTimes()* Windows32 API function as described in [17].

On the Solaris platform timing measures were performed with the *gethrtime()* system call available on SunOS. The system call expresses time in nanoseconds from an arbitrary time in the past. It is very accurate since it does not drift. Measurements of CPU time were made with the *times()* system call that returns the CPU cycles distinguished by user and kernel mode since process startup. The actual CPU time was then computed using the *CLK_TCK* constant that defines the clock ticks per second and is located in the *limits.h* C++ header file.

#### 3.1.4 Evaluation Applications

The applications used for evaluation comprise two typical two-tier architectures. The first application performs the various method invocation policies that are supported by CORBA [11]. These are: static and dynamic one-way invocations (notifications), static and dynamic two-way invocations (interrogations) as well as deferred two-way and deferred multiple two-way invocations (multicast). The second application performs the overhead measurement by invoking a series of 1000 method requests. Both simple and user defined parameter types (unbounded string sequence) were used. The measurements of the second application were repeated several times, which delivered satisfactory results as the mean values of these repetitions deviated by 1 to 3 percent.

### 3.2 Evaluation Results

This section provides an overview of the evaluated management tools according to the criteria described in Section 2. Each tool is treated in more detail in the following.

#### 3.2.1 Orbix Manager

Orbix Manager requires an implementation of the Common Object Service (COS) naming service [12] as managed processes register at a naming hierarchy. Orbix Names[8] version 1.3c was used for evaluation. This naming schema allows to group hosts into managed domains. A managed domain is controlled by a single management service that provides the agent functionality (intermediary entity between management application and managed processes).

One of the main benefits of the Orbix Manager is its notification facility where process startup and termination, unexpected server shutdown and CORBA exceptions are reported. Exception reports can be assigned to severity levels on exception type basis, which allows filtering these accordingly.

Each managed process can be queried for configuration information. This information comprises merely Orbix related configuration options such as communication ports and Orbix optimization as well as Orbix environment variables. Orbix specific configuration settings such as connection timeout can be changed at runtime via the management instrumentation. Moreover, all active connections to and from a process are shown. Orbix Manager further allows to assign user specific properties to each process, whose values can be set at runtime. The properties are stored by server name at the management service. Subsequently, whenever a process starts up with this server name the property values are available.

Orbix Manager enables to monitor the performance of servers graphically at runtime. The provided information includes number of requests received and sent, number of exceptions, and throughput in bytes per second.

The tool does not support accounting and security management.

The request latency overhead of the Orbix Manager instrumentation is about 60 percent per process. This means that if both client and server are instrumented the overhead is doubled. The CPU time overhead shows a doubles invocation time.

[8] Orbix Names is a trademark of IONA Technologies PLC.

The Orbix Manager shows accurate management information, but eventually processes may fail to remove themselves from the naming hierarchy when they terminate. Consequently, these processes stay registered but not accessible. The user has to remove these entries manually in order to establish a consistent view of managed processes.

| | **Object/Observer** | **Corba Assistant** | **Orbix Manager** |
|---|---|---|---|
| (Capabilities) | | | |
| Fault | more complex | 0-simple | simple-more complex |
| Configuration | 0-simple | simple | simple |
| Accounting | N/A | simple | N/A |
| Performance | 0-simple | simple-more complex | simple |
| Security | N/A | N/A | N/A |
| (Average Communication Overhead of one process) | | | |
| Latency | - [1] | 45 % | 60 % |
| CPU | - [1] | 70 % | 100 % |
| (Resource Management) | | | |
| Components | 7 – 16 MB | 6.5 MB | 12 – 28 MB |
| Instrumentation | (+ 600 kB) | (+ 600 kB) | (+ 1600 kB) |
| Degree of Integration | medium | medium (high) [2] | medium |
| Built-In Security | N/A | N/A | N/A |
| Integration into Standard Frameworks | N/A | N/A | SNMP |
| Openness | emphasized [3] | supported | N/A |
| (Configuration) | | | |
| Location | Managed Object | Management Application | Management Application |
| Degree | runtime | runtime | runtime |
| Granularity | fine | fine | coarse |
| GUI & Documentation | ++/++ | ++/+ | +/0 [4] |
| Accuracy | | | |
| View of running processes | ++ | ++ | + [6] |
| Performance quantities | - [5] | ++ | ++ |

[1] Performance has been measured on another platform and can therefore not directly be compared to those of the other tools. The overhead strongly depends on the used parameter types.
[2] If object implementations are instrumented in addition to processes, the degree of integration is high and the Corba Assistent can be used with the utmost efficiency.
[3] The Object/Observer user guide gives a detailed description on how to build user specific components for Object/Observer.
[4] The GUI is well designed but shows annoying bugs, which shorten the capabilities. On average, the Orbix Manager documentation is good but the SNMP integration is described insufficiently.
[5] Measures are influenced by the instrumentation and times are significantly higher than that of the uninstrumented processes.
[6] Processes eventually do not deregister at the naming service and are subsequently shown in GUI although not active.

**Table 2. Tool Evaluation Overview**

The instrumentation extended a process' memory footprint by about 1600 kilobytes. The management product typically requires 12 MB of memory, the multiple GUI tools increase the memory accounting up to 28 MB.

The instrumentation requires to instantiate a C++ class in the process' *main()* function. This corresponds to the medium degree of integration.

The amount of management information gathered is not configurable. However, due to the polling mechanism the information is only requested if a process is selected in the GUI.

Orbix Manager provides a gateway to SNMP. Consequently, the managed process can be accessed by an SNMP management application. Exception notifications are mapped to SNMP traps, and managed processes are defined by a SNMP table.

The tool does not provide any interface definitions and is therefore not extensible by the user. Built-in security is not supported either.

The three GUI tools that come with Orbix Manager, the main GUI, the event viewer, and the performance graph viewer are easy to handle, but show several bugs. These are quite annoying as it is not possible to query managed client processes. The documentation is generally good, but the SNMP integration manual is scanty.

### 3.2.2 Corba Assistant

The Corba Assistant extends the process management view to CORBA objects. It shows therefore two principal types of managed objects: processes and CORBA objects. These managed objects emit events on object creation and destruction to a COS event service channel implementation. A so-called *living object service* uses this information source to provide a topical view of active managed object references. The management application called *orbas* uses these references to query the managed objects for vital management information.

The provided information includes process information as well as developer and version related information. The latter has to be supplied by the developer. In the case of client processes referenced CORBA object types are visualized, which provides valuable information for configuration management. On the server side, each object instance can be queried for management information if instrumented. Since CORBA servers may contain multiple object type implementations and multiple instances of each, this information allows to configure an application on a finer granularity than on the process level. The Orbix specific timeout mechanism can be controlled at runtime for each client.

Moreover, each server gathers accounting information on a per-user basis. For each user the amount of requests, bytes read, and bytes written is available.

The major benefit of the Corba Assistant is seen in performance monitoring. Total and average requests and bytes measures are provided, extended by response time measurement on client side. The average quantities may be viewed graphically using a well-designed graph viewer tool. The performance measures on per-object basis enable the user to enhance load balancing on a more fine-grained level.

Neither security management nor built-in security are supported by this tool.

The Corba Assistant shows the lowest overhead of the evaluated tools. The overhead for one process is about 45 percent, whereas the measure is doubled to 95 percent if both client and server are monitored. The instrumentation of both server and contained object implementations showed no significantly higher overhead. The overhead could slightly be lowered by increasing the monitoring polling interval and reducing the amount of requested information. The CPU time measurement indicated an overhead of 70 percent, whereas the more complex parameter types showed a slightly higher overhead.

The accuracy of the performance measures are very good and seem not to be affected by the instrumentation. Disappointing in this context is that the quantity named "average response time" is obviously computed by (min+max)/2. Since the statistical distribution function of request latencies is not symmetric and shows high outliers, this quantity is usually times higher than the statistical mean value.

The instrumentation extends a process' memory footprint by about 600 kilobytes. The whole management tool requires just 7 MB of memory.

Process instrumentation requires to instantiate a C++ class in the *main()* function. The object instrumentation requires to extend the object implementation and to add a C++ macro to each object instance creation within the source code. The C++ macros ease these tasks, but the user nevertheless has to know where the objects are instantiated and where the C++ object declarations are situated. Thus, the degree of integration is medium, increased to high if objects are instrumented in addition to the processes.

The amount of management information as well as the polling interval are configurable on a fine-grained basis. However, this affects only the request to the process and not the instrumentation at the process itself.

The evaluated version of Corba Assistant provides no integration support to any management standard, but makes the interface definitions of its component available to licensees. The user may therefore integrate these components and services to her specific needs.

Orbas provides a single GUI for the complete functionality. A drawback is that almost every information is shown in a new frame. The documentation is sufficient but could be structured

better. Especially, the GUI description would be more useful if structured based on task orientation.

Corba Assistant is planned to be extended with a Common Management Information Protocol (CMIP) gateway, which will enable access to the CORBA managed domain from the OSI Systems Management (OSI SM) standard. OSI SM is widely adopted in the telecommunication sector.

### 3.2.3 Object/Observer

The objective of Object/Observer differs from the other tools as it intends to provide a means to test Orbix-based applications. It provides a tracing service rather than general management support. However, monitoring is a crucial functionality for management.

The key element of the Object/Observer is the *OSMObserver* process, which serves as a registration point for instrumented processes. Moreover, it provides an agent-like functionality, as it functions as information dispatcher in both directions. On the one hand, it gathers request traces from the instrumented processes and forwards them either to the viewer GUI or to a data recorder that stores the information to files. This tool is the only one of the evaluated ones that provides some sort of logging support. On the other hand, it forwards configuration requests from the viewer GUI to the corresponding process. The monitoring configuration of processes is conveniently created by a GUI that connects to the CORBA interface repository. The user can select which parameter and return values of which methods of which object types should be monitored. Moreover, the user can specify the monitoring points i.e. *client sending*, *server receiving*, *server sending* or *client receiving*. The configuration is saved to files and can be assigned to instrumented processes at runtime.

The tool generates request reports including the parameter and return values as well as reports for CORBA exceptions. Its primary benefit according to management capabilities is therefore fault management. The user gets informed about requests and exceptions and may for instance trace a failure back to a wrong parameter value in a request.

The tool comprises a parser for the log files. The parser computes request latency measures from the timestamps of the request reports and thus provides some fundamental performance measures. Moreover, as each request report contains the issuer and the target process, it allows detecting dependency relationships.

Object/Observer does not support accounting and security management.

The tool showed some overhead. The major reason is that Object/Observer in contrast to the other tools has to parse the message buffers completely in order to retrieve and reconstruct the parameter and return values of the requests.

The instrumentation is fairly small and increases a process' memory footprint by about 600 kilobytes. The processes require typically 7 MB of memory, all processes and GUI tools consume up to 16 MB of memory.

Similarly to the other tools comprises the instrumentation procedure comprises few lines of code, consisting merely of the instantiation of two C++ classes.

The monitoring can be configured at the instrumented processes themselves at a fine-grained level.

The tool does not provide support for any management standard. The OMG IDL interfaces of the components are available and their usage is well documented. Object/Observer is therefore open to user specific changes.

The three GUIs, the viewer tool, the file parser, and the setup editor are well-designed and easy to use. Especially the viewer tool proved its robustness during overhead measurement, because it managed to display about 4000 request reports at a rate of tens per second. The documentation is good and describes all features well. The usage of Object/Observer is therefore well documented and convenient.

## 3.3 Discussion

The evaluation of the three tools shows common approaches and solutions.

The tools provide management on a process basis. Each tool has equipped the instrumentation with an OMG IDL interface to make it accessible to the management application. Subsequently, each tool has to provide some means to find these CORBA objects. Crucial to this service is that only processes that are effectively running are accessed. This is of increased importance as Orbix automatically launches a registered server if it is not active at a client's request [6]. The instrumentation therefore registers and deregisters itself at a common service: a certain server process in case of Object/Observer, via an event channel at Corba Assistant and via the naming service at Orbix Manager. The latter two approaches are far more flexible solutions, especially the use of the naming service is often used in the context of CORBA-based implementations of management standards [13, 16].

Instrumentation procedure is common to all tools and requires to instantiate a certain C++ class. This instrumentation class is based on the *interceptor* concept that is new to the 2.2 release of the CORBA specification [11]. An interceptor is called by the ORB and enables to interfere a CORBA request unaware to the application itself. Orbix does not yet conform to this specification release but already provides a comparable functionality called *Filter* and *Transformer* (see [21] for further discussion).

The architecture of the Orbix Manager most adheres to common systems management standards. Managed objects are inquired on demand whereas the managed objects emit notifications about certain events when they occur. The filtering mechanism is crucial in order to reduce overhead and to avoid distracting information.

The logging mechanism of Object/Observer is a fundamental service that is missing from the other tools. Orbix Manager stores properties per server name and collects notifications but this information gets lost with the termination of the GUI.

However, missing features and shortcomings described in this paper may have already been fixed in subsequent releases of these tools.

## 4 Conclusion

The evaluated CORBA management tools focus on fault and configuration management. The manageable units are commonly CORBA processes. The instrumentation requires the source code of an application but is done with few lines of code. Each tool accesses the instrumentation through a CORBA object interface whereas only Object/Observer enables to control instrumentation. The Object/Observer's main benefit is fault management and is the only tool that provides detailed request reports including parameter values. Moreover, it is the only tool that supports persistent storage of management configuration and collected information. Orbix Manager and Corba Assistant strongly adhere to systems management principles of which Orbix Manager is the more advanced tool. It provides extensive process related information and is equipped with a exception notification facility. Moreover, it is the only tool that is integrated into a management standard. Nevertheless does Corba Assistant outstrip it in performance management support. Furthermore, Corba Assistant is the only tool that extends the management view to CORBA objects and thus provides more detailed measures.

The tools provide valuable support for management of CORBA-based applications. Orbix Manager further indicates the advent of systems management support by ORB vendors, others are expected to follow. Considerations on revising the OMA [14] indicate a growing awareness at the OMG itself. It is thus likely that system management interfaces will be integrated into the CORBA specification.

## 5 Acknowledgements

We thank Siemens PSE Austria for the support of this study, IONA Technologies for providing us the fundamental Orbix MT releases for Solaris and the Orbix Manager, Black & White Software and the Fraunhofer-IITB for the possibility of evaluating their products.

## 6 References

[1] Hajo Brunne, "Principle Design Patterns for Manageable Object Request Brokers". In *Proceedings of the International Workshop on CORBA-Management at the OMG TC Meeting at Dublin*, 22–26 September 1997, Dublin, Ireland.

[2] G. Coulouris, J. Dollimore, T. Kindberg, *Distributed Systems: Concepts and Design*, Addison-Wesley, Wokingham, England, 1994.

[3] M. Debusmann, R. Kröger, C. Weyer, "Towards an Automated Management of Distributed Applications". In *Proceedings of the IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)*, BTU Cottbus, 30 September - 2 October 1997.

[4] R. Friedrich, S: Sunders, G. Zaidenweber, D. Bachmann, S. Blumson, *Standardized Performance Instrumentation and Interface Specification for Monitoring DCE-Based Applications*, OSF DCE-RFC 33.0, July 1995.

[5] A. Gokhale, D. C. Schmidt, "Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks". In *Proceedings of the 17th International Conference on Distributed Systems (ICDCS 97)*, 27-30 May 1997, Baltimore, USA.

[6] Iona Technologies PLC, *Orbix Programmer's Guide*, October 1997.

[7] M. Ji, E. W. Felten, K. Li, "Performance Measurements for Multithreaded Programs". In *Proceedings of ACM SIGMETRICS/IFIP WG7.3 SIGMETRICS/PERFORMANCE '98*, 22-26 June 1998, Madison, Wisconsin, pp. 161-170.

[8] A. Leinwand, K. Fang, *Network Management: A Practical Perspective*, Addison-Wesley, Reading, Massachusetts, 1993.

[9] Dennis Longley, Michael Shain, *Data & Computer Security: Dictionary of standards, concepts and terms*, Macmillan Publishers, UK, 1989.

[10] Microsoft Corporation, *Windows Developer Journal*, February 1996.

[11] Object Management Group, *The Common Object Request Broker Architecture and Specification*, Revision 2.2, February 1998. OMG Document Number: formal/98-02-01.

[12] Object Management Group, *CORBA services: Common Object Services Specification*, November 1997. OMG Document Number: formal/97-12-02.

[13] Object Management Group, *CORBA-Based Telecommunication Network Management System*, OMG White Paper, May 1996.

OMG Document Number: telecom/96-07-01.

[14] Object Management Group, *Reference Model Extension Green Paper*, April 1998. OMG Document Number: ormsc/98-04-01.

[15] The Open Group, *Systems Management: Common Management Facilities*, CAE Specification, October 1997. The Open Group Document Number: C423.

[16] J. Pavón, J. Tomás, Y. Bardout, L.-H. Hauw, "CORBA for Network and Service Management in the TINA Framework", *IEEE Communications Magazine*, vol. 36, no. 3, March 1998, pp. 72-79.

[17] Jeffrey Richter, *Advanced Windows*, Microsoft Press, Redmond, 1995.

[18] Marshal T. Rose, *The Simple Book: An Introduction to Internet Management*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[19] R. M. Soley, C. M. Stone, *Object Management Architecture Guide*, Revision 3, Object Management Group, June 1995. OMG Document Number: ab/97-05-05.

[20] William Stallings, *SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards*. Addison-Wesley, Reading, Massachusetts, 1993.

[21] Bernfried Widmer, *Management of CORBA-Based Distributed Object Systems*, Master's Thesis, Technical University of Vienna, November 1998.

# Managing Shared Business-Objects –

*Position Paper*

Chris Salzmann*
Institute for Informatics
Munich University of Technology
80290 München, Germany

*salzmann@in.tum.de*

February 12, 1999

**Abstract**

*Modern component architectures extend the range of reuse from the data-model level towards the level of functionality. In modern distributed object architectures different clients share a set of so called business objects for representing encapsulated functionality or certain entities. One of the advantages of this approach is, that several applications can share one instance of such a business object. The advantages of sharing resources are well known: higher reliability, compact code and lower costs, just to mention a few. However, in large distributed systems, that share objects instances the effort for managing and configuring those systems increase rapidly.*
*In this position paper we give the benefits of that approach and we sketch out what facilities a systems has to provide to maintain and manage those shared objects.*

***Keywords:*** Software Engineering, Administration of Distributed Objects, Configuration Management, Distributed Systems, Business Objects and Software Architecture
***Workshop Goals:*** Contact to other people working in the area of managing distributed object systems as well as sharing experiences.

## 1 Introduction

In large distributed system the level of reuse is raised from the classical data-model reuse (client/server) to a partly reuse of application logic, so called *business objects* [EE98, HMS99]. The purpose seems obvious: higher reuse, therefore lower costs and shorter product cycles. However, a further remarkable aspect of independent components, that represent a reusable unit of application logic, is the opportunity of *sharing* that component-instance between several software-programs.

Although one can argue, that sharing is just another kind of reusing, the advantages of sharing the same instance of a component (i.e. the object) does not lie in the benefits mentioned above (i.e. lower costs, shorter product cycle etc), but in more efficiency of maintaining and configuring the software system (so is the functionality encapsulated in one instance and only this has to be modified). Modern enterprise-wide application systems that are coming up these days like IBM's SanFrancisco [IBM, HMPS98] framework or the CORBA ORB *ComponentBroker* [IBM98] have a special emphasis on this maintenance and configuration aspect by offering high flexibility in reconfiguring a business application in its framework. Together with BMW we were evaluating the actual approaches of business object frameworks and realized, the the issue of maintenance of shared, distributed object systems gets new weight [HMPS98].

It is the goal of this position paper to list some of the advantages of shared business objects and discuss some of the difficulties that a system bring with for managing such shared object-architectures.

## 2 Three Tier Architecture

In this section we want to start with a brief introduction of the scenario and give some definitions of terms we use in this position paper.

*This work was supported in part by the FORSOFT consortium of the Bayerische Forschungsstiftung and the BMW AG.

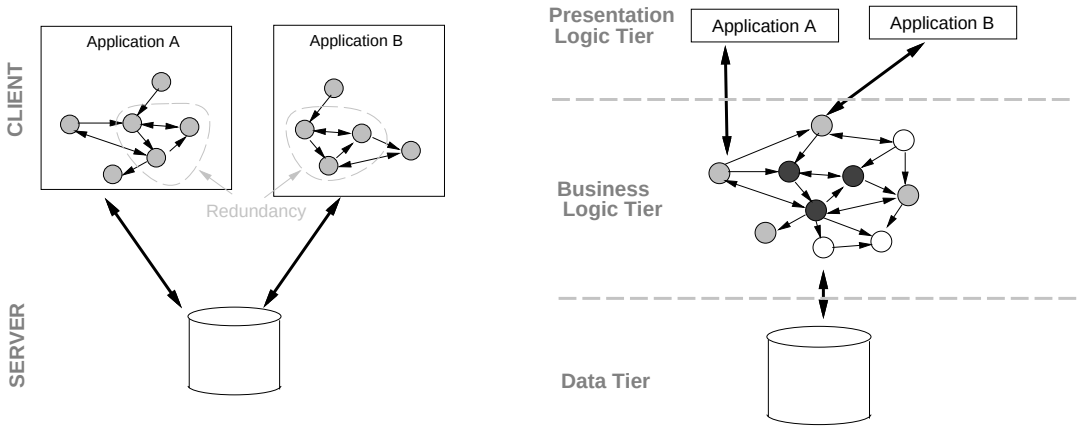


Figure 1: Client/server vs. 3T architecture: raising the level of reuse

The classical client/server architecture (C/S-architecture) separated the data-model from the presentation and eventually the applicational logic. The result was an (eventually) fat client, that shared its separated data model with other clients (see figure 1). So what C/S-architecture brought in advantage to the older monolithic architecture was *reuse on the data-model layer*.

Nowadays, new architectures are gaining ground in the commercial wold: so called *"three-tier-architectures"* (3T) [BMW98] claim to raise the level of reuse and lower the costs of large scale distributed application systems. The 3T-client exists only of presentation-logic and (eventually) some application logic that is that specific on the needs of the client that it's not reusable. Other application logic is encapsulated in special, sharable objects, that reside on the middle layer – so called *business objects*. The data-model stays the same as in the C/S-architecture. So the 3T-architecture *raises* the level of *reuse* form the data-model layer to the *applicational logic* layer. The shared application logic on the middle layer we call *business logic* (see figure 1).

### 2.1 Shared Business Objects

The main advantage of encapsulating application logic in independent objects is the finer granularity and functional grouping of reusable code. Beside the reuse of code in succeeding versions of a software system the mutual sharing of software particles is another aspect of reuse. In figure 1 the classical C/S-model is mapped to a 3T-architecture. Where in the C/S case the two applications $A$ and $B$ had a certain amount of redundancy, the conjoint using of application logic resources lowers the redundant code. In our example the conjoint used objects are marked in a dark gray.

This conjoint using of given resources brings up a strong dependency between applications: was in the C/S case application $A$ only dependent from $B$ via the data model, we now get additional dependency via the three shared business objects. An upgrade or modification caused by one of the applications would force all the dependent ones to shut down for the time of modification. Therefore – to keep the advantage of commonly used applicational logic – a more flexible management and configuration facility is needed.

## 3 Configuration of Shared Business Objects

As mentioned above, the configuration, meaning the restructuring and modification of an existing distributed application (adding/dropping objects and adding dropping links between them) gains more importance the larger the systems get. We are first explaining how modern business frameworks already ease the configuration by static structures, following with an explanation why this is not enough and why we need more flexible *dynamic* configuration for those systems.

### 3.1 Static Configuration

By *static* configuration we mean the modifications of the structure and architecture that take place on the source code level. For this the complete system must be – obviously – shut down, must be restructured, recompiled, relinked and booted. For a system that is highly decentralized and linked, this way is inappropriate, since for changing one shared object, all dependent (i.e. this resource using) applications or objects must be shut down.

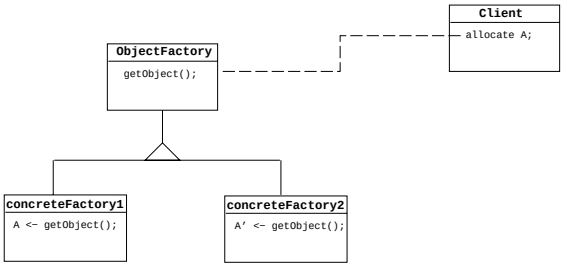


Figure 2: Semi-Dynamic Configuration: Instantiation Factories

## 3.2 Semi-Dynamic Configuration

Semi-dynamic configuration management is already realized as a promising target and systems like IBM's SanFrancisco framework support the system manager by using mechanisms like instantiation factories. Factories – a pattern, well known from [G+94] – encapsulate locally the instantiation and allocation of objects (see figure 2). So for the case, that instead of an object $A$ an object $A'$ should be instantiated every time an object $A$ is allocated in the source code, the system manager needs only to change one line of code (in the factory object) instead of tracking all allocation occurances in the source. The change can even take place at runtime, if the system supports dynamic linking facilities. This saves a lot of effort and raises the flexibility. SanFrancisco uses almost for every business object factory structures to ease the configuration.

However, semi-dynamic configuration changes only the instance, allocated in the *future*. The already existing instances remain the same and must be handled separately.

## 3.3 Dynamic Configuration

The advantage of shared business objects is easy to see: higher reuse, higher semantical grouping and concentration as well as lesser code and costs. However, with more sharing and more efficient use of components there come also higher dependencies. If an little shared component is only responsible for a small amount of applications, an efficient shared one is responsible for a large amount of applications (just think of a component that represents the task "book order" in a company). Therefore it also may often be the case, that a component has to be extended or modified for one of its applications. By raising the amount of shared resources and therefore lowering the redundancy, we are also raising the amount of dependencies between applications. If a business object is shared by a large amount of applications (see figure 3 a change of the object, caused by one of the applications, needs all the depending applications to be shut down for reconfiguration. So we get an *indirect dependency* between all applications that are linked via a shared business object.

It is obvious, that the benefit of the reuse would be worthless, if for every modification of a component that is needed by one of its applications all applications have to get shut down. Therefore *dynamic* configuration management of those shared components is needed, that allows to modify the structure of the system during runtime and therefore lower the stagnation time of the system to a minimum. So with dynamic management only the dependent instances (or applications) have to be transformed into a special state, that allows modification. This state depends on the category of dependency (what is related to the changed object and in what way) and the category of consequences (what has to be done to transform the dependent instances into a consistent state).

We started to elaborate the principal problems and issues of dynamic configuration of distributed systems on an abstract, formal basis [Sal99] and are going to extend it to different application domains.

The complexity of the modification process depends on several parameters: the *target* of change (gets a component changed or a connection), the *source* of change (is the stimulus caused by the state of the system or by the external environment), the *time* of modification ( design time, linking time, runtime), the *constraints* of the modification (a certain state of the system, an ring structure of the components etc.) and last but not least the *nature* of the component. So different levels of complexity as well as different types of change and different types of dependency will lead to different categories of consequences for the system. It is essential , for example, to know on which processes of the system (i.e. the usecases or the business processes) a change of structure or component might have impact. So if one reconfigures a business object $o$ for example, we want to know in advance which 'backbone-processes' and which functionality is affected in which way.

To develop a practical usable shared object management we have to watch the following issues:

- Define the categories of dependencies (i.e. direct, indirect etc).

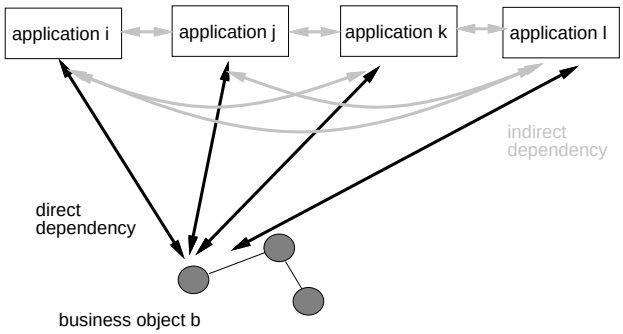


Figure 3: Raising the Level of Dependency by Sharing Objects

- Define the categories of consequences (i.e. change the interface, change the data-model etc.).
- Define the relations to usecases and functionality.
- Determine which dependencies causes what consequence.
- Map it to existing middleware.

There is already a large amount of research done in the area of reconfiguration - static [Ins84] and even some constrained dynamic [MDEK95, Med96]. However, in this special context of business objects in wide area systems and mission critical applications (one might think of the logistics systems of large companies like BMW) a close specification to the requirements is needed.

# 4 Consequences

The previous section illustrated, that modern object-sharing architectures offer a wide variety of advantages but need a bigger emphasis on dynamic behavior. Especially the management of the configuration and versioning brings up some new questions. A range of open problems and consequences keeps unsolved and asks for discussion. Some of the fundamental needs are:

- an infrastructure that supports dynamic configuration and management of shared objects.
- support of the dependency management (i.e. disbanding of dependencies) in the object architecture.
- tool support to design and manipulate those structures using the offered management facilities.
- tool support that illustrates the relations between change and usecases/functionality.
- a methodology that helps identifying and designing shared components in a software field.

We think that research results in the area of configuration and management of shared resources must be combined and adapted for the concrete needs of distributed object architectures to bring up satisfying results.

# 5 Conclusion and Position

With the ongoing net-centralization of IT-systems (i.e. inter- and intra-net applications) the structure of software gets by far more decentralized and distributed as we know it from classical client/sever structures. Those systems distinguish themself by a highly dynamic behavior in their structure and topology.

We think that a methodology for developing such systems with high-level dynamics is needed, as well as a framework for managing and maintaining such systems. Both have to emphasize the aspect of dependency and dynamic configuration of the system. We are therefore working on an abstract system model, based on [BDD+92] to describe the scenarios and dependencies, outlined here. This shall lateron serve as a foundation for a methodology and a framework to design and manage such highly dynamic systems.

**Acknowledgments** I want to thank my colleagues, especially Sascha Molterer and Wiebe Hordijk for fruitful discussions.

# References


[BDD+92] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The design of distributed systems - an introduction to focus. Technical Report TUM-I9202, Technische Univerität München, 1992.

[BMW98] BMW AG. BMW Architecture Blueprint, 1998.

[EE98] Wolfgang Emmerich and Ernst Ellmer. Business objects: The next step in component technology ? In *CBISE 98*, Pisa, Italy, 1998.

[G+94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[HMPS98] Wiebe Hordijk, Sascha Molterer, Barbara Paech, and Chris Salzmann. Working with business objects - a case study. In D. Patel, J. Sutherland, and J. Miller, editors, *Business Objec Design and Implementation II*, Heidelberg, 1998. Springer Verlag.

[HMS99] Wiebe Hordijk, Sascha Molterer, and Chris Salzmann. On the reuse benefit of business objects. Technical Report Not yet published, Munich University of Technology, 1999.

[IBM] IBM. IBM SanFrancisco Extension Guide.

[IBM98] IBM. The component broker connector overview redbook, 1998.

[Ins84] British Standarts Institution. *Configuration management of computer-based systems.* BSI, 1984.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC 95), Sitges, Spain, September 1995*, 1995.

[Med96] Neno Medvidovic. A classification and comparison framework for software architecture describtion languages. Technical Report UCI-ICS-97-02, University of California, Irvine, 1996.

[Sal99] Chris Salzmann. An abstract model for dynamics in wide area computing. working paper, 1999.

# Position paper

# Encapsulation of protocols and services in medium components to build distributed applications

Antoine Beugnard, Robert Ogor
ENST Bretagne[1]
BP 832, 29285 Brest Cedex
{Antoine.Beugnard,Robert.Ogor}@enst-bretagne.fr

## Introduction

Distributed applications are usually built using two main strategies. The first one requires the knowledge of mechanisms dedicated to communication and protocols usually using explicit message passing. The standardization of protocols plays a fundamental role, but the their proliferation requires wide knowledge and skills on the part of the designer (software architect). The second strategy tries to abstract totally the communication with the aim of building distributed applications, as "easily" as non-distributed applications. This usually leads to the development of middleware such as Corba [2] or Linda [7] that attempts to reduce the communication to a single and abstract way of communicating. Sometimes, this single communication means leads to rebuilding protocols over it and can be considered as constraining. We propose something intermediate.

Since the component architecture of software seems to be becoming a promising way of building classical applications, we want to consider communication as a component. These components, called mediums, encapsulate various protocols and communication paradigms. A medium component is intermediate in the sense that it offers a single architectural abstraction, the component, but numerous protocols at different levels. We can imagine a Corba medium to make distributed objects communicate, but also an asynchronous point-to-point communication between processes, or a component that ensures the synchronous broadcast of messages among a set of objects.

As for classical component-based applications, a medium-based distributed application is developed in two phases. The first one consists in assembling (with a visual composer tool) a medium with classical components. Once the application is described, the second phase generates the application, in our case, all the parts of the distributed application.

In the first section we describe the overall architecture and the proposed life cycle of medium-based applications. Then, a very simple distributed application illustrates the concepts of communication components (medium). We conclude with a comparison with other approaches such as Corba [2] (for middleware strategy), BAST [4, 5] (for the OSI-protocol layer strategy) and Connectors [6] (for a centralized equivalent of mediums).

## Architecture and development life-cycle

### Medium definition

We propose to use special components, called "mediums", as a means to define and capitalize various communication protocols and distributed services required by a distributed application. Considering a communication means as a component enables various protocols to be used and handled easily by non-specialists of distributed application architectures. Like a classical component, a medium offers a unique and uniform interface that, thanks to

[1] This work is partially granted by "la région Bretagne".

introspection, simplifies its manipulation through visual assembling tools. A medium offers the following services:

1. An interface to this communication means, (protocol entry points)
2. Communication services, (protocol implementation)
3. Dedicated services such as configuration, quality of service, etc. (protocol configuration)

For instance, a consensus protocol could simply be encapsulated in a medium as shown in figure 1. The entry points are `propose()` for a component to propose a value, and `decide()` for the medium to warn the components about the decision[2].

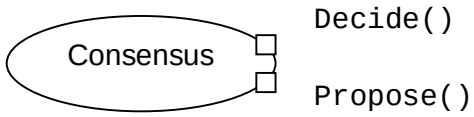


Figure 1: An external view of a medium

An application is built with many standard components (ActiveX, Beans). Usually (in a centralized application), these components are connected through local adapters that are generated once the assembly is finished. The connection is often limited to a point-to-point method call. Mediums are a generalization and a *reification* of such connections. To build a distributed application, a designer would have access to a set of mediums that could be used to inter-connect the standard components according to the designer's communication needs. Here is a list of potentially useful mediums: channel (point-to-point asynchronous), broadcast (asynchronous), causal order broadcast (asynchronous with delivery order guarantee), consensus, failure detection, data flow broadcasting (with QoS), voting protocol, blackboard, distributed shared memory, bus, etc…

To summarize, the life cycle of mediums consists of the following two steps:

1. Use by an application architect to assemble the components
2. Automatic application generation from the "map" elaborated by the architect

The second step is beyond the scope of this position paper. We will just give an idea of the internal structure of mediums and concentrate on the architect's point of view. The step consisting in the design of the medium is also disregarded; it is tightly related to the second step where the internal structure of mediums enables the application generation.

## Mediums and component assembling

The distributed application architect has many components at his disposal. To build the application he has only to connect usual components through mediums that are adapted to the communication needed. In a simple case, a point-to-point medium would replace the standard local connector, but a medium could also offer a service of synchronized and ordered multicast of messages among a set of receivers or act as a software bus. Figure 2 illustrates the connection of 5 standard components by way of two different mediums.

Mediums do not compel the architect to adopt a single way of communicating, but allow him/her to use different protocols *at the same abstraction level*, depending on the actual needs. From the protocol designer's point of view, mediums are a way to capitalize and offer a unified vision of the numerous ways of communicating.

Being at different abstraction levels may appear confusing. But, in the real world do not we use very different ways of communicating, from radio or TV, optical broadcasting, telephony, electronic mail, news, newspapers, etc? The important point is to use a common interface: our senses. In the case of mediums, we would like to propose a single abstraction: a communication component. Thus, building a distributed system results in assembling components according to our needs.

[2] This is a simplified consensus protocol.

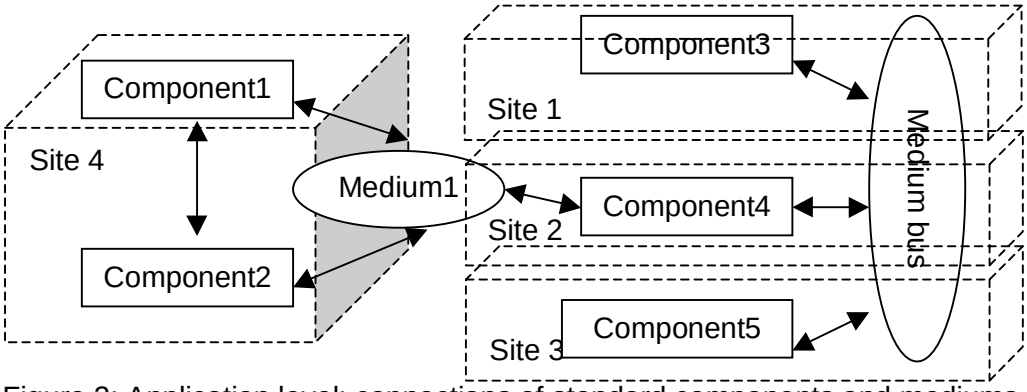

Figure 2: Application level: connections of standard components and mediums

## Overview of the internal medium architecture

The internal architecture of mediums resembles usual protocol stacks. We prefer to use the name "chain" instead of "stack", because the implementation is more like a chain. In fact, the medium hides at least as many chains as there are components connected to it. The extremities of each chain are an object interface and a network connection. Between them are implemented the algorithms and services delivered by the medium.

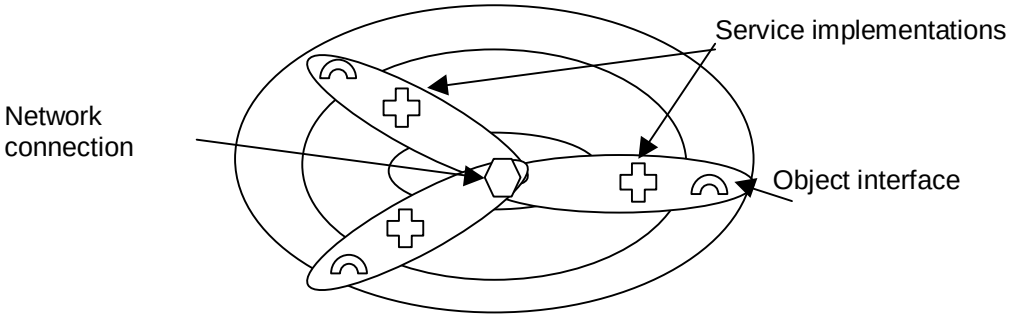

Figure 3: Internal architecture of a medium

The current internal architecture is described in [1]. It is under-optimized. We are focusing on the architectural aspects of mediums not on performances. We can imagine an optimized version of mediums with pre-compiled chains.

## Current medium implementation

To evaluate the concept of mediums, we have already implemented two very different communication means that could be considered to be at different abstraction level. The first one is a simple asynchronous point-to-point channel; the second is a distributed shared memory based on the Linda model. We intend to develop a medium for synchronous message dispatching, a medium with QoS management for multimedia data transfer, etc.
For simplicity and portability reasons the prototype is implemented in Java over a classical TCP/IP protocol. This results in a framework of classes named Covadis [1]. We are currently trying to increase the variety of mediums and underlying protocols thus testing the adaptability of the framework.

## Strategies for the generation of mediums

The distributed application having been assembled in the first stage, the second phase consists in generating the codes for all the different sites. This generation could rely on many different variants. For instance the same abstract communication means called a shared memory can be implemented in at least two ways: a centralized memory on one unique site

and access protocols on others, or a distributed memory on all sites including the access protocols. The generation variations could be related to the algorithmic or to the supporting underlying protocol (IP, ATM, and why not Corba). A medium is a component, and as such, owns certain properties, including an implementation strategy used to choose the variants[3].
Mediums are handled by two categories of developer: distributed applications architects and medium designers. This paper concentrates on the former. The internal architecture and the medium designer constraints are detailed in another document [1]. We can simply say that, from now on, the medium architecture does not support any "composition" operator that could create a new medium from the assembly of others; a medium has a flat internal structure.

# A simple application

In order to illustrate the use of mediums in the development of a distributed application, we can imagine having access to pre-defined mediums such as:

1. A video-stream broadcast (one-to-many) medium using the RTP protocol for instance
2. A voting medium that collects yes/no answers to a question and produces an accepted/refused output

We can imagine an interactive film being broadcast and from time to time the audience being asked a question to know whether they want such or such an alternative. Then the film continues according to the poll results.

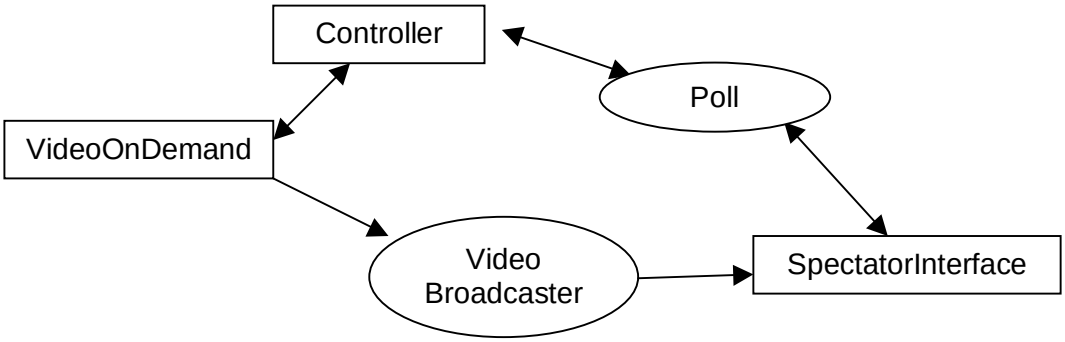


Figure 2: An interactive film application

This architecture requires some comments. Firstly, it seems to have a single spectator. In fact, some medium could be connected to an a priori unknown number of components. Secondly, the connection between the VOD and the Controller seems local. This could be the case if the controller and the VOD were on the same site, but, if needed, a more sophisticated medium could be used. Special attention is required at the interface level. Mediums, like normal components, need to have instrospection facilities to be able to list the services it offers to the application architect. For instance, the Poll medium offers the following "standard" interface:

in Propose(String question) : boolean on the Controller side

out Proposed(String question), in Accept() and in Refuse() on the Voter side

As with classical components, the visual assembler tool should be able to generate some glue to adapt the medium interface to that of its client's. But this glue is local and depends only on the introspection abilities of components.

[3] Classical components could use a generator strategy if implementation variants exist.

# Comparison with other approaches

What are the current classical approaches to distributed software development? The main trend relies on the use of middleware such as the general purpose CORBA, DCOM or the more specific ibus [8], javaspace. But, the good old method of using layers of protocols (socket, alternate-bit, etc) is still being used. To organize the jungle of protocols BAST [4, 5] proposes a classification - in the sense of an object classification - that improves the re-usability and composition of protocols. We compare the mediums to these approaches. Then, we describe another work related to the reification of the communication, the connectors [6] currently used for centralized applications but whose author intends to adapt to distributed ones.

## CORBA and related software bus approaches

Making distribution a main issue requires interoperability. The OMG defined and adopted the standard CORBA [2] architecture (Figure 4) as a very pragmatic and operational tool to guarantee the interaction of different programs running on various OS, written in various languages, and even for pre-existing applications. Once this pre-requisite is available, the OMG improved and is continuing to improve the "software bus" by the addition of many services.

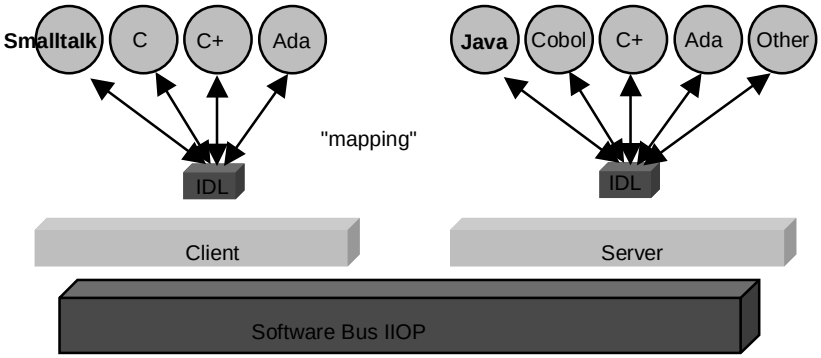


Figure 4: CORBA, interoperability through IIOP

But the level of communication protocol is unique; an object sends a request to another one. Objects need references to communicate. This basic protocol allows more abstract communication means to be built, but these have to be implemented, and are not really encapsulated. So, except for CORBA extensions of the IDL, we do not see how the current point-to-point communication protocol could be extended (for instance in the stubs and skeletons) in a broadcast protocol.
To avoid the need for object references, architectures such as buses were implemented to allow anonymous objects to join the bus and communicate with other objects connected on it. This simple and abstract principle offers point-to-point communication when references are known, and when it is not the case, broadcasting is used. Two drawbacks emerge:

1. The component should be adapted to the bus, and therefore should be specifically developed or adapted.
2. This generality could drastically reduce performances

The bus approach shows the need for several communication means and protocols. This is what the mediums are offering. Behind a single, abstract and easy-to-use component the medium developer can deliver various communication strategies including CORBA-like or bus-like ones. And although performances were not our goal, the use of different mediums for different communication means would certainly allow a kind of specialization and optimization. We are currently working on a medium implementing a Corba bus. One of the

main issues is to encapsulate the instrospection capabilities of Corba (Repositories) into a medium.

## BAST

The BAST framework [4, 5] is a very interesting classification of protocols. Special attention was given to the two ways of using protocols according to the user's point of view: the protocol designer and the application architect (the protocol user).
This work leads to the classification of figure 5.

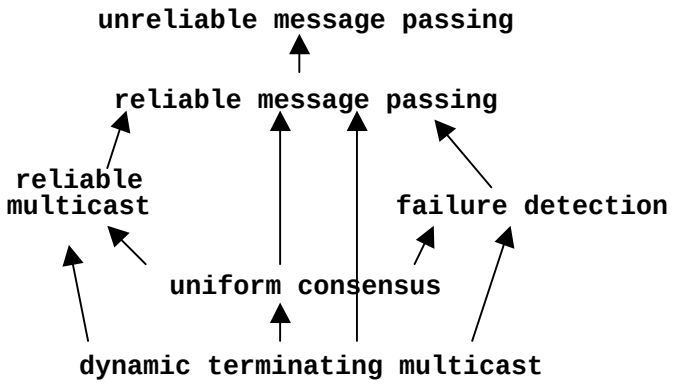


Figure 5: The BAST Classification of protocols

As shown by the protocol hierarchy of figure 5, BAST offers a very good way of re-using protocols of the lower level (the topmost) to derive protocols of a higher level. Relations between protocols are very efficiently pointed out. The protocol use by an architect remains difficult, because of the lack of a single way of interfacing objets to the framework.

## Connectors

One great challenge using objets is their reusability. A trade-off between generality and usefulness must be found. The bigger an object is, the more useful, but the less general it is. As shown by the long experience of Smalltalk, a powerful system of objects must generate a lot of small, reusable objects in order to build a large hierarchy of objects. Objects are very often a mix of "pure" objects and application dedicated glue, integrated in the normally protected and consistent syntactic structure called the class.

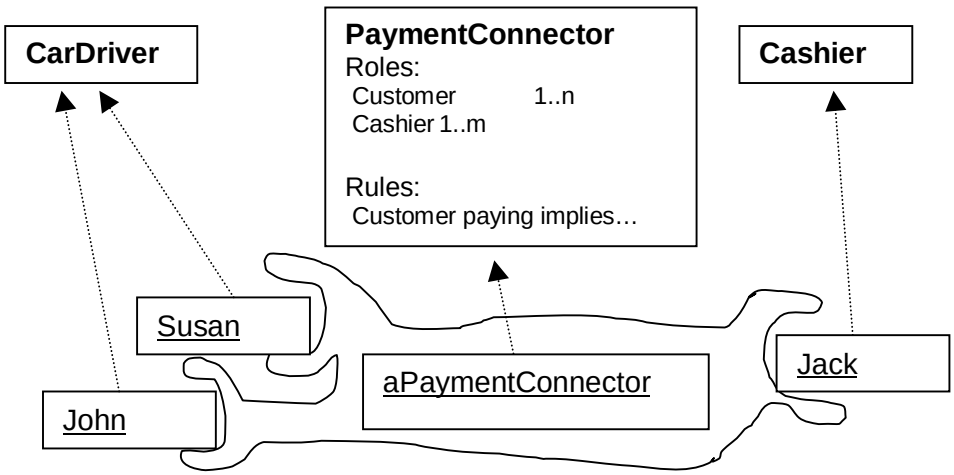


Figure 6: A connector example from [6]

Manuel Günter proposes in [6] to separate these two notions and to consider objects as primary entities with no (or reduced) communicating means. Then, he defines an application as the grouping of such primary objects with other objects, connectors, acting as glue for the application, linking the objects together. Figure 6 illustrates these two kinds of objects in a very simple application. The small, primary entities so defined are reusable directly in another context, by simply changing the connectors.
Connectors are high level objects that are able to observe and trigger events in the objects connected to them. The implementation uses the meta-programmation facilities of Smalltalk to achieve this. Connectors are generic objects programmable through a connecting language that describes the ways objects interact.
In a way, connectors and mediums come from the same idea: communication reification, the former in a centralized context, with a language-defining-the-protocol (the rules, see figure 6) specification of communicating components, the latter in a distributed context with built-in protocols.

# Conclusion

Reification is one of the most powerful idioms for structuring and reusing parts of complex systems. Various design patterns [3] rely on reification (Strategy, State, Factories, …). We would like to check this new reification attempt with mediums against true applications and case studies.
Connectors [6] and mediums [1] are similar in essence. They try to encapsulate communication to make objects connected to it more reusable, putting communication issues into special objects. The communication being encapsulated makes it simple to use through the interface of a classical component. Mediums are components, but the way they are used - to interconnect other components - and their internal architecture make them specific enough to be considered as special software objects.
Up until now, we have implemented various levels of protocols into mediums to illustrate the wide spectrum of potential encapsulated protocols. It seems feasible to hide Corba [2] or even connectors in special mediums in order to show the generality of this approach. The goal is to build a wide range of mediums so that distributed application architects can choose the communication means ideally suited to their requirements.

## Bibliography


[1] Eric Carriou*, Covadis : un framework pour la conception visuelle d'applications distribuées*, rapport de stage de fin d'étude, ENST Bretagne, France, octobre 1998
[2] OMG, Corba, http://www.omg.org
[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley, 1995.
[4] Benoît Garbinato*. Protocol Objects & Patterns for Structuring Reliable Distributed Systems*. PhD thesis, Swiss Federal Institute of Technology, Lausanne (EPFL), May 1998. http://lsewww.epfl.ch/bast/
[5] Benoît Garbinato, Rachid Guerraoui. Flexible Protocol Composition in Bast. In *Proceedings of the 18th International Conference on Distributed Computing Systems* (ICDCS-18), Amsterdam (The Netherland), May 1998.
[6] Manuel Günter, *Explicit Connectors for Coordination of Active Objects*, Master's Thesis, University of Berne (1998).
[7] A. Rowstron and A. Wood, *An efficient Distributed Tuple Space Implementation for Networks of Heterogeneous Workstations*, 1996, http://www.cs.york.ac.uk/linda/new_linda.html
[8] M. Silvano, *IBus - The Java Intranet Software Bus*, 1997, http://www.olsen.ch/export/proj/ibus

# Supporting Reliable Evolution of Distributed Objects

**Jonathan E. Cook** **Jeffrey A. Dage**
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
{jcook,jdage}@cs.nmsu.edu

**Abstract**

*Distributed object systems offer a foundation for systems to be highly malleable and configurable, even after deployment. While this malleability offers many benefits and opportunities for creating novel systems, it also becomes a potential source of problems. This is because, unfortunately, new versions of objects can introduce new errors and break existing, depended-upon behavior.*

*We believe that for this move towards distributed, component-based systems to* not *have a negative impact on system reliability, the middleware frameworks must allow and support the composition, manipulation, and execution of multiple versions of components. Doing so will ensure that the move towards distributed, component-based software systems does not lessen, but rather provides opportunities to enhance, the reliability that software will achieve through the next century.*

## Introduction

Recently, system construction has moved away from monolithic systems and towards more component-based architectures, with distributed object frameworks being an example of this move. These frameworks provide a separation between the components that make up a system, where individual components can be replaced relatively independent of one another. This ability means that systems, even after they are deployed, are now more malleable than ever. Much recent work has revolved around the basic support necessary for enabling this malleability [4, 5, 6, 8, 12].

On one hand, malleability is good, since the system can be tailored to a specific task, or a failing piece can be selectively upgraded. Unfortunately, a component that has been modified to fix one defect may inadvertently break some other existing functionality, or the modified system composition might violate assumptions made by individual versions of the components. Indeed, a recently held workshop on dependably upgrading critical systems [3] noted many of the fundamental reliability concerns that still need to be addressed in this field. These concerns are still open, since the frameworks have thus far been focussed on basic enabling mechanisms, and not on support for ensuring system reliability.

A strong argument has already been made that the architectural description of a system should take into account the versions of the components [11]. This aids in ensuring that valid compositions of components are made during deployment, and that the engineers make explicit the version dependencies amongst components.

We take a different tack and focus on the situation where versions of components are expected to be "interchangeable". That is, a subsequent version of a component is created, perhaps to fix a bug or add an

This work was supported in part by the National Science Foundation under grant CCR-9804067 and the Department of Education under grant P200A70303-97. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

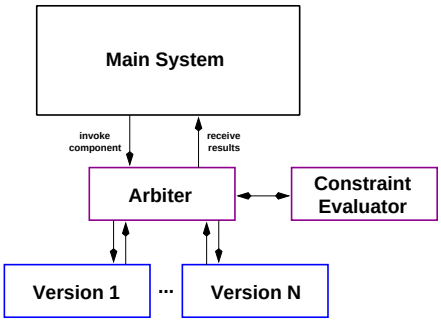


Figure 1: The HERCULES Framework.

enhancement, but where this new version is intended to replace the old version. It is in this situation that reliability can be compromised by a faulty new version breaking existing functionality that was depended upon by the users.

If, however, one was able to keep both versions active in the system, and knew what the intended differences between them were, system integrity could be maintained by selectively using one version or the other. Middleware, such as distributed object systems, provides an excellent opportunity to build exactly that type of support for system evolution. Since components can be swapped in and out during run-time, there is no reason to simply ignore the version history of a component when it comes time to upgrade it.

Indeed, it is our position that support for the composition, manipulation, and execution of multiple versions of components is imperative if we are to enhance the reliability of component-based systems through the next century. In [2] , we present a framework, HERCULES, that is a first step in this direction. In the rest of this paper, we briefly present this framework and outline how it can be utilized in the evolution and testing of distributed object systems.

## The HERCULES Framework

The HERCULES framework, shown in Figure 1, is focussed on managing versions of a component throughout the component upgrading process. Thus, it is focussed on supporting the evolution of the system.

Between the external system that uses a component and the component versions, we place an *Arbiter* that acts to present to the system the image of a single component. The Arbiter invokes each of the component versions when the system requests it, and sends the selected result back to the system. The Arbiter also contains component management facilities. The middleware frameworks that implement distributed object systems are natural places to implement an Arbiter.

To select a result, the Arbiter uses a *Constraint Evaluator* (CE), providing the invocation parameters and component state information to the CE. The CE evaluates the formal specifications of each version's addressed domain, and decides which version of the component will produce the correct result. The Arbiter then selects this result to send back to the external system, and logs statistics on which versions produced this same (presumably correct) result.

The constraints on each version indicate the domain(s) that it specifically addresses. When a new version of a component is created, it is meant to address specific conditions (a domain) where the existing version(s) failed. By specifying these conditions formally using a constraint expression, we can detect at run-time when

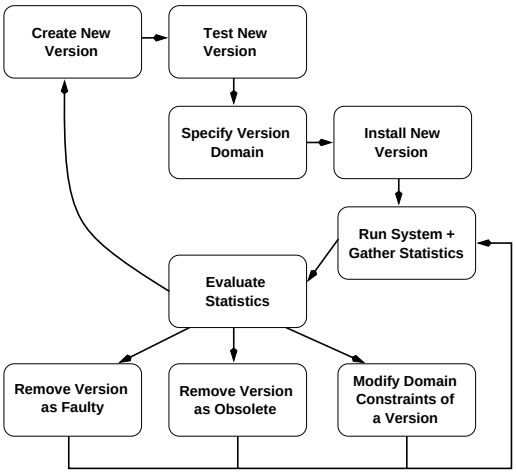


Figure 2: The Component Upgrading Process.

those conditions occur. When they do, the new version of the component is assumed to be producing the correct answer rather than the existing version, and the result given to the external system is the result from the new version. If for some invocation, however, the conditions that the new component specifically addressed are not true, and the new version produces a result different than the existing version, then it is assumed that the new version has broken some correct behavior of the existing version, and the result given to the external system is that produced by the existing version, not the new version.

In this manner, system reliability is increased throughout the upgrade process. Note that we are not evaluating the *correctness* of the result of the component with regard to some specification. We are only deciding *which* version of the component is assumed to be correct, based on the domain it is meant to address. This version is called the *authoritative* version for that particular invocation. Any other version that produces that same result as the authoritative version is also correct, for that invocation.[1] By logging the number of times each version produces a correct result, we can at some point decide that the new version is reliable, and then take the old version off-line.

## The Component Upgrading Process

Distributed object systems, and other component frameworks, have enabled the malleability of deployed systems, but so far they have not explicitly supported it. Without some level of support, systems will be (and are now) actually less robust than previous monolithic systems. It is our position that middleware must define and embrace a process for evolving a deployed system, and that by enabling version management, this process can enhance system reliability throughout evolution.

Figure 2 shows the overall process of component upgrading that we envision. A new version of the component is created and tested, and given a specific domain. This domain may be specified before the version is constructed, as part of the change order specification, but we place it after testing because the

[1] Deciding when results are the same is not always a trivial equality comparison; some data types and problem domains need more expressive tests for deciding when two values are equal (e.g., floating point numbers).

domain may change after testing, if it is decided that the version as constructed operates on a different domain than was originally specified. Initial techniques we have used to specify constraints are similar to language-based self-checking, or assertion, methods [1, 7, 9, 10].

The new version is then installed into the running system, and the system runs and gathers statistics on all running versions. At chosen points in time, an engineer will inspect the statistics for all versions of a component, and may choose one of four different actions (besides doing nothing):

- An existing version is removed as faulty.
- An existing version is removed as obsolete (the statistics show that its descendants correctly implement all of its own behavior).
- The domain constraints of a version are modified, because it is decided that it is operating correctly over a larger domain or that it is failing in some part of its own domain.
- A new version of the component is created and installed.

With this process, the system reliability is increased throughout the evolution of its components, and the engineer takes an active role in maintaining the configuration of the system while it is running. An important note is that this framework and process allow for future capabilities in automation. Some of the statistics evaluation can be moved onto the running system, and by using rule/action specifications, automated decisions can be made about when a component is faulty enough to remove or when it is obsolete, and actions can be specified that do not require human intervention.

## Performance Issues

With such a framework as ours, performance concerns naturally arise. Rather than having direct execution of a single component version, we have introduced an Arbiter, a Constraint Evaluator, and the execution of multiple versions of a component. These additions certainly change the expected performance of the system. In a distributed object system, where network delays mandate a well-designed partitioning of the system, these extra delays may not be as much a concern as for say, a single-processor embedded system.

However, we have not overlooked this issue, and our framework allows for taking advantage of multiple processors to execute multiple versions, for delaying execution of non-authoritative versions, and for limiting the number of versions executed, due to time or space constraints.

## Supporting Testing

While we have presented this framework as supporting the evolution of malleable component-based systems, it also applies to certain activities in the testing phase. Indeed, this framework might be most beneficial during the alpha and beta test phases, where a system might undergo many rapid changes which would ideally be deployed to the testers as soon as they are available. The HERCULES framework would better isolate the multiple changes to different components, and would provide better test results because the overall system would be isolated from incidental change mistakes, and because the statistics that are logged regarding each component version would give clear indication as to the reliability of specific versions.

Alternatively, for a system where many variants are delivered, our methods may ease the burden of testing many different compositions of the system, and by allowing multiple versions of components to run during testing, feedback as to which configurations will be reliable is obtained.

## Conclusion

Distributed object systems are only an example of the type of malleable, component-based systems that are here now and will be even more so in the future. So far, these frameworks have merely focussed on enabling malleability, but not supporting it in the sense of providing mechanisms to ensure that the overall system remains reliable.

It is our position that for these frameworks to do so means that they must embrace the management and manipulation of multiple versions of the components. It is not enough to simply allow a new version to be deployed, nor even to be able to rollback to a previous version. The engineering knowledge of how the versions differ can be used to manage multiple versions simultaneously, and to increase overall system reliability.

We have summarized HERCULES, the framework representing our preliminary investigation into reliable component upgrading. In this work we demonstrated the basic foundations of specifying version domains, using an arbiter to select a version at the time of invocation, and gathering statistics to understand version reliability. This investigation has shown that reasoning about component versions enhances the knowledge that an engineer can bring to bear during system maintenance and change, and that it appears feasible to extend the use of version information to run-time issues such as component upgrading.

We believe that there is great potential for new techniques such as these to provide users with highly reliable, yet still dynamic, systems.

## References


[1] J. Cook. Assertions for the Tcl Language. In *Proc. 5th Annual Tcl/Tk Workshop '97*, pages 73–80. Usenix, July 1997.

[2] J. Cook and J. Dage. Highly Reliable Upgrading of Components. In *Proceedings of the 1999 International Conference on Software Engineering*, May 1999. To appear.

[3] D. Gluch and C.B. Weinstock, eds. Workshop on the State of the Practice in Dependably Upgrading Critical Systems. Technical Report CMU/SEI-97-SR-014, Software Engineering Institute, Aug. 1997.

[4] K. Goudarzi and J. Kramer. Maintaining Node Consistency in the Face of Dynamic Change. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 62–69. IEEE Computer Society Press, May 1996.

[5] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 15(11):1293–1306, Nov. 1990.

[6] M. Little and S. Shrivastava. Using Application Specific Knowledge for Configuring Object Replicas. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 169–176. IEEE Computer Society Press, May 1996.

[7] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 1988.

[8] N. Rodriguez, R. Ierusalimschy, and R. Cerqueira. Dynamic Configuration with CORBA Components. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 27–34. IEEE Computer Society Press, May 1998.

[9] D. Rosenblum. Automated Monitoring of Component Integrity in Distributed Object Systems. In *Advanced Topics Workshop of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*. USENIX Association, June 1997.

[10] D. S. Rosenblum. A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering*, 1995.

[11] A. van der Hoek, D. Heimbigner, and A. Wolf. Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage a Trois. Technical Report CU-CS-849-98, University of Colorado, Jan. 1998.

[12] I. Warren and I. Sommerville. A Model for Dynamic Configuration which Preserves Application Integrity. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 81–88. IEEE Computer Society Press, May 1996.

# Progressive Implementation of Distributed Java Applications

Paulo Borba* Saulo Araújo Hednilson Bezerra
Marconi Lima Sérgio Soares
Departamento de Informática
Universidade Federal de Pernambuco

## 1 Introduction

In this position paper we overview on-going research work aimed at defining, formalizing, and validating a method for the systematic implementation of distributed object-oriented applications. In particular, this method supports a progressive approach for object-oriented implementation, where distribution, concurrency, and persistence aspects are not initially considered in the implementation process, but are gradually introduced, preserving the application's functional requirements and semantics.

By initially abstracting from those subtle aspects, engineers can, for example, quickly develop and test a local, sequential, and non-persistent application prototype useful for capturing and validating user requirements. As requirements become well understood and more stable, that prototype can be used to derive a structured and functionally complete prototype, which is then progressively transformed into the final distributed, concurrent, and persistent version of the application, by carefully dealing with the aspects mentioned above, one at a time.

In this way we can significantly reduce the impact caused by requirements changes during development, since most changes will likely occur before the functionally complete prototype is transformed into the final version of the application, which is larger and much more complex than the prototypes. Furthermore, the progressive approach naturally helps to tame the complexity inherent to distributed systems, by supporting the gradual testing of the various intermediate versions of the application. For example, problems in the business logic layer can be isolated from problems in the persistence and communication layers.

Of course, a central assumption to our method—the Progressive Implementation Method (Pim)—is that it is possible to initially abstract from distribution and concurrency aspects. In fact, Pim considers that there are differences between the implementation of local and distributed objects, but that local objects can be gradually transformed into distributed ones. However, this might not be possible when distribution or concurrency is inherent to the application objects. Indeed, we believe that our method is not useful for all kinds of distributed applications, but it has been proven specially useful for the implementation of distributed information systems, which are usually distributed and concurrent for performance and fault tolerance reasons only.

As Pim is just an implementation or coding method, it should be integrated to design and testing methods in order to be used in practice. However, Pim relies on the use of specific architectural and design patterns for structuring object-oriented applications, so that it imposes constraints on design methods chosen for integration. Fortunately, those constraints

*Supported in part by CNPq, grant 521994/96–9. WWW: `http://www.di.ufpe.br/~phmb`. Electronic mail: `phmb@di.ufpe.br`. Telephone: +55 81 271 8430, extension 3323. Fax: +55 81 271 8438.

basically correspond to the use of simple or well known patterns useful for supporting the progressive introduction of distribution, concurrency, and persistence aspects.

This paper is organized as follows. We first explain how we formalize and justify Pim's activities and key tasks[1]. We then discuss in separate sections the introduction of distribution and concurrency aspects, in that order. Persistence aspects are not considered here for scope and space reasons. Examples are used to illustrate several aspects of Pim, including some refinement laws and the constraints imposed on design methods. The current version of Pim is specific to Java [4] and RMI [7], so that we use them throughout the paper. At the end we discuss the current status of this work and its limitations.

## 2 Laws of Progressive Implementation

Most tasks of Pim are just informally described and illustrated by examples. However, key tasks are formalized by semantic preserving refinement laws, which basically indicate that a class, a command, or a program can be safely replaced by another. Those laws are essential for precisely determining subtle constraints and code modifications associated to a given task. Moreover, the laws justify the soundness of Pim, assuring that the final version of a given application preserves the semantics of its functionally complete prototype, as long as database and distribution services work properly; for example, database and network connections are not indefinitely down.

As in [2], here we represent and formalize the laws by using class operators for building classes from smaller pieces, as if we were defining an algebra for classes. For example, the "$\langle \bullet \quad \bullet \rangle$" operator constructs classes so that

$$\langle A \bullet M \bullet I \rangle$$

denotes the class formed by the private attributes[2] in $A$, the methods in $M$, and the initializers (constructors) in $I$. The "$\otimes$" operator puts attributes together, so the expression

$$\langle A \otimes B \bullet M \bullet I \rangle$$

denotes the class formed by the attributes in $A$ and in $B$, besides the methods and initializers[3] respectively in $M$ and $I$.

Using those operators, we can concisely formalize refinement laws. For example, consider methods $m$ and $m'$ having the same signature. Then, provided that $m$ is refined by $m'$, denoted $m \sqsubseteq m'$, we have that

$$\langle A \bullet M \oplus m \bullet I \rangle \sqsubseteq \langle A \bullet M \oplus m' \bullet I \rangle$$

This law basically indicates that method refinement implies class refinement. The same notation is used to formalize the laws of Pim, precisely indicating how the classes of application prototypes should be progressively transformed to consider distribution, concurrency, and persistent aspects.

## 3 Introducing Distribution

One of the constraints that Pim imposes on design methods is the use of the Facade design pattern [3]—which provides a unified interface for all services of a subsystem—in order to structure applications. Hence Pim assumes that the functionally complete and local prototype has *business facade classes* such as the following:

[1] For simplicity, we consider that a method just consists of activities, which are described by a group of tasks.
[2] Hereafter we consider that classes have only private attributes; so we omit the word "private" for brevity.
[3] Usually called "constructors" in Java.

```
class MyBusinessImplementation implements MyBusiness {
   private CustomerFile customers;
   private ProductList products;
   ⋮
   void addCustomer(Customer customer) {
      customers.add(customer);
   }
   void setPrice(ProductCode code, double price) {
      products.setPrice(code,price);
   }
}
```

where the classes `CustomerFile` and `ProductList` provide services for the insertion, updating, querying, and deletion of customer and product records[4].

The objects of business facade classes are precisely the ones that should be distributed or available for remote access. So Pim also assumes that the functionality of those objects is abstracted by corresponding *business facade interfaces*:

```
interface MyBusiness {
   void addCustomer(Customer customer) throws CommunicationException;
   void setPrice(ProductCode code, double price)
      throws CommunicationException;
   ⋮
}
```

which explicitly indicate that subsystem services might not be available due to problems in the communication or distribution infrastructure and basic services. This is necessary because business facade interfaces abstractly represents the services of a subsystem, which might be local or remote to the clients of those services.

Fortunately, in order to transform the local prototype into a distributed one, we do not need to change the business facade classes and interfaces. Pim simply suggests the use of object adapters [3] for them. The adapters basically encapsulate the RMI code that is necessary for allowing the distributed or remote access of business facade objects. In this way, the business logic layer becomes totally independent from the RMI communication layer, so that changes in the latter layer do not impact the former layer.

There are two kinds of adapters: *source adapters* and *target adapters*. Roughly, the latter wrap business facade objects in the places where they are located, and the former represent those objects in remote locations. In a typical client-server system, user interface objects would request the services of a source adapter located in the client machine. The source adapter would then request the services of a corresponding target adapter located in the server machine. Finally, the target adapter would request the services of a business facade object also located in the server side.

### 3.1 Target Adapters

A target adapter contains a reference to a `MyBusiness` object—likely a business facade object—and methods that simply invoke corresponding methods on that object. For example, the following is a target adapter classs for any class that implements `MyBusiness`:

[4]For brevity, we omit the "`public`" qualifier from type and method declarations. We also assume that the illustrated methods raise no exceptions.

```
class MyBusinessTargetRMIAdapterImplementation
   extends UnicastRemoteObject implements MyBusinessTargetRMIAdapter {
   private MyBusiness myBusiness;
   ⋮
   void addCustomer(Customer customer)
      throws CommunicationException, RemoteException {
      myBusiness.addCustomer(customer);
   }
}
```

where the code inherited from the RMI class `UnicastRemoteObject` allows target adapters to be remotely accessed, and `RemoteException` is raised by RMI to notify communication or configuration problems during the call of a remote method [7].

As target adapters shall be used as remote objects, we must also have an interface with the same signature as the target adapter class:

```
interface MyBusinessTargetRMIAdapter extends Remote {
   void addCustomer(Customer customer)
      throws CommunicationException, RemoteException;
   ⋮
}
```

where `Remote` is an RMI interface used to identify remote object types; in particular, `Remote` is indirectly implemented by `UnicastRemoteObject` [7].

Note that the methods of target adapters may indirectly raise `CommunicationException` because they invoke methods of `MyBusiness` objects, instead of a particular implementation of that interface. This gives an extra flexibility, as discussed later.

### 3.2 Source Adapters

A source adapter contains a reference to a target adapter and basically calls the methods of the second, catching RMI specific exceptions and replacing them by `CommunicationException`. For example, observe the following source adapter class for any class that implements the interface `MyBusiness`:

```
class MyBusinessSourceRMIAdapter implements MyBusiness {
   private MyBusinessTargetRMIAdapter myBusiness;
   ⋮
   void addCustomer(Customer customer) throws CommunicationException {
     try {
        myBusiness.addCustomer(customer);
     } catch (RemoteException exception) {
        throw new CommunicationException(exception.getMessage());
     }
   }
}
```

This class implements `MyBusiness` so that source adapters effectively represent the services provided by their corresponding business facade objects. For instance, user interface classes typically declare variables of type `MyBusiness`. In the functionally complete and local prototype, those variables are instantiated with `MyBusinessImplementation` objects. In the distributed prototype, those variables are instantiated with source adapters.

In fact, note that a target adapter might refer either to a business facade object or to a source adapter. This gives us extra flexibility in such a way that a source adapter $sa_1$ might refer to a target adapter $ta_1$ that refers to a source adapter $sa_2$ and so on, until we finally have that $ta_n$ refers to a business facade object. In this way we can easily support various configurations for a distributed system.

### 3.3 Distribution Tasks

In summary, Pim's tasks for transforming a local prototype into a distributed prototype include the following: create specific source and target adapter classes; replace, where appropriate, instances of business facade classes for source adapters; configure and execute target adapters as server processes; properly handle communication exceptions raised by source adapters. Furthermore, as RMI remote method calls actually copy argument and result objects, we must declare the parameter types and result types of remote methods as subtypes of `Serializable`, a Java interface that guarantees class serializability. In addition, we must insert update operations in order to guarantee that updates to object copies are reflected in the original objects. For example, clients of `MyBusiness` typically write code such as the following:

```
Customer customer = myBusiness.fetchCustomer(customerId);
customer.setName(...);
customer.setAddress(...);
```

But, if `myBusiness` holds a reference to an RMI source adapter, `customer` will not hold a reference to an original customer. So, in order to reflect the changes in the original customer, we should add the following method call at the end of that code:

```
myBusiness.updateCustomer(customer);
```

where the `MyBusiness` method `updateCustomer` is responsible for fetching and updating the original customer with the attribute values of its argument.

## 4 Dealing with Concurrency

After performing the tasks presented in the previous section, the business facade objects are ready to be distributed, but are not ready for concurrent access, which likely comes as a consequence of distribution. In fact, the concurrent access to the methods of business facade objects might generate a vast range of undesirable interferences, since the distributed prototype code uses no synchronization mechanisms.

In order to avoid undesirable interferences and preserve the semantics of the original functionally complete prototype, Pim suggests tasks for bridging the gap between Java applications developed to be used in sequential and concurrent environments [1]. Those tasks are based on practical guidelines for safely introducing, moving, and removing synchronization mechanisms without leading to deadlock, livelock, or to undesirable interference. Such guidelines are, for example, derived from common design patterns for structuring concurrent object-oriented programs [6].

Pim's synchronization laws (see Section 2) formalize those guidelines. Each law typically helps to correctly increase either liveness or safety. For example, a useful law for increasing liveness establishes that we can remove the Java `synchronized` qualifier[5] from a method as long as some constraints are satisfied:

[5] The methods qualified as `synchronized` cannot be executed while there is another `synchronized` method being executed in the same object [4].

$$\langle A \otimes b \bullet M \oplus \texttt{synchronized}(m) \bullet I \rangle \sqsubseteq \langle A \otimes b \bullet M \oplus m \bullet I \rangle$$

provided that

- *body*($m$) is in the form "`b.n(`$\overline{\texttt{e}}$`)`", where *name*($b$) is `b`, `n` is any method name, and $\overline{\texttt{e}}$ is an arbitrary list of expressions formed only by variables and constants[6]; and
- *type*($b$) is a fully synchronized class—its methods are all synchronized, and it has no public instance variables [6].

Roughly, this law establishes that it is sound to remove synchronization from a method $m$ that simply invokes another method that is accessed in a serial way. The justification is that $m$ will be serially accessed anyway.

Pim's tasks for transforming a sequential prototype into a prototype adapted to be used in a concurrent environment are roughly the following: using synchronization laws, introduce `synchronized` qualifiers to the methods of business facade classes, and also to the methods of *business collection classes* such as `CustomerFile`; whenever possible, remove the `synchronized` qualifiers from the methods of business facade classes, using the law illustrated in this section, for example; apply other synchronization laws to further improve liveness.

By using Pim, the business facade class presented in the previous section would first receive `synchronized` qualifiers:

```
class MyBusinessImplementation {
   private CustomerFile customers;
   private ProductList products;
   ⋮
   synchronized void addCustomer(Customer customer) {
      customers.add(customer);
   }
   synchronized void setPrice(ProductCode code, double price) {
      products.setPrice(code,price);
   }
}
```

which can be safely introduced without leading to deadlock because this facade class satisfies some constraints. In fact, we can guarantee that the objects of the class above are safe; they have the same observational [8, 9] behaviour no matter if executed in a sequential or concurrent environment. Moreover, this behaviour is equivalent to the behaviour of the original business facade objects (defined on Section 3) when they are executed in a sequential environment.

After achieving safety, we could benefit from the synchronization law presented in this section for optimizing `MyBusinessImplementation`. The `synchronized` qualifiers would be safely removed, assuming that the classes `CustomerFile` and `ProductList` serialize the insertion, updating, querying, and deletion of customer and product records.

## 5 Conclusion

We have given an overview of the Pim method for the systematic implementation of distributed object-oriented applications. Pim supports the progressive transformation of an structured and functionally complete application prototype into a distributed, concurrent, and persistent

[6] We assume that method bodies are well typed, so that we do not need to impose further constraints on `b`, `n`, and $\overline{\texttt{e}}$.

version of the application. We have concentrated here on distribution and concurrency aspects because of scope and space reasons. Persistence aspects are considered elsewhere [10].

Pim's progressive approach can significantly reduce the impact caused by requirements changes during development, since most changes likely occur before distribution, concurrency, and persistence aspects are introduced to the application. Furthermore, Pim naturally helps to tame the complexity inherent to distributed systems, by supporting the gradual testing of the various intermediate versions of the application. In this way, we can, for example, isolate problems in the business logic layer from problems in the persistence and communication layers.

Pim is not useful for any kind of application. It is only useful when concurrency and distribution are not inherent to the application, but are necessary for performance and fault tolerance reasons only. For example, Pim has been quite useful for the implementation of distributed information systems in different domains. In particular, Pim has been partially used to implement the following applications: a customer management system for a telecommunications company; a retail store system; and a mobile dairy sales management system. From our limited practical experience so far, managers and engineers are initially skeptical about the method, mainly because it implies rework on some classes. However, after using Pim they usually get convinced of its benefits.

As Pim is just an implementation or coding method, it should be carefully integrated to design and testing methods in order to be used in practice. In the experiences reported above, Pim has been integrated either to the Rational's Objectory process [5] or to *ad hoc* processes. Also, besides fully describing the method, we should provide tool support (via component and IDE wizards) for Pim, since most tasks are tedious and can be largely automated. The current version of Pim is specific to Java and RMI, but we intend to generalize it so that we can support other middleware technologies such as CORBA and DCOM.

## References


[1] Paulo Borba. Systematic development of concurrent object-oriented programs. In *US–Brazil Joint Workshops on the Formal Foundations of Software Systems*, volume 14 of *Electronic Notes in Theoretical Computer Science*. Rio de Janeiro, Brazil, 5-9th May, 1997, and New Orleans, USA, 13-16th November, 1997.

[2] Paulo Borba. Where are the laws of object-oriented programming? In *I Brazilian Workshop on Formal Methods*, pages 59–70, Porto Alegre, Brazil, 19th–21st October 1998.

[3] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[4] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[5] Ivar Jacobson. *The Unified Software Development Process.* Addison-Wesley, 1999.

[6] Doug Lea. *Concurrent Programming in Java.* Addison-Wesley, 1997.

[7] Sun Microsystems. *Remote Method Invocation Specification*, 1998.

[8] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[9] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[10] Euricélia Viana and Paulo Borba. Integrando Java com bancos de dados relacionais. In *III Brazilian Symposium on Programming Languages, submitted*, 1999.

# From Distributed Object Features to Architectural Styles

Bastiaan Schönhage[1,2] and Anton Eliëns[1,3]

1 - Vrije Universiteit
Department of Mathematics and Computer Science
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
email: bastiaan, eliens@cs.vu.nl

2 - ASZ Research & Development
P.O. Box 8300, 1005 CA Amsterdam, The Netherlands

3 - CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

**Abstract**

Distributed object oriented software is gaining interest. Nevertheless, developing distributed systems is inherently more difficult than creating single-machine applications. Middleware solutions and distributed object frameworks are intended to help developers in building distributed software. Deciding on which distributed technology to deploy, however, is not straightforward.

In this paper, we will introduce three architectural styles to classify distributed OO software. The styles differ in the dimensions of data location and object migration. In addition to a description of the styles we will give examples, discuss guidelines for their usage, and indicate which technologies can be deployed to implement the styles. Thus, by classifying and cataloging distributed OO software in architectural styles and providing guidelines for their usage, we are trying to give some guidance for the development of distributed software.

## 1 Introduction

Software engineers, both academic and in business, are increasingly interested in Software Architectures. The software architecture of a program or computing system is *the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them* [1]. An important aspect of software architectures is the fact that they force us to think about the quality requirements of a system in addition to the functionality. Moreover, software architectures allow for an evaluation of software systems early in the development cycle.

Architecture-based development promises, amongst others, high-level reuse and separation of concerns. But what we are mainly interested in here is that, while traditional approaches are primarily concerned with functionality, software architectures are concerned with the interaction and communication of components [3].

Whereas software architectures describe or prescribe one particular system, Shaw and Clements [8] and Shaw and Garlan [9] classify groups of software architectures in architectural styles. Architectural styles are *descriptions of component types and a pattern of their runtime control and/or data*

*transfer* [1]. Architectural styles are often, as in our case, based on practical experience. By making design decisions and considerations explicit by means of styles and guidelines, we are able to transfer this important knowledge to other similar software development projects.

In Section 2, we will start with an object-feature-space to classify objects in distributed OO software. However, software architects are not so much interested in object features *per se* but more in the behavior of the system as a collection of collaborating objects. Therefore, Section 3 presents three architectural styles. The **distributed objects**, **dynamically downloaded classes** and **mobile objects** architectural styles are all based on distributed OO technology but differ in the dimensions of data location and object migration. Section 4 illustrates the application of architectural styles in practice. It shows how the styles are deployed in a visualization application supporting collaborative decision making. As a conclusion, we will evaluate the architectural styles and provide some rules-of-thumb for deploying them in Section 5.

# 2 Distributed Object Feature-space

In the following classification we will examine some features of objects that comprise a software system. This is not intended as a classification of distributed software architectures. Instead, we will look at properties of single objects and illustrate which features are supported by different technologies. The discussed feature-space will later-on be used as the basis for the architectural styles.

When objects are **remotely callable** it means that methods of the objects can be invoked remotely by objects residing on other machines. By **downloadable** we mean that object-code (classes) can be dynamically downloaded to clients where the objects are instantiated. When components are **mobile**, objects migrate taking functionality, data and status along. Thus, mobility implies that the object (instance) is moved *as-is* whereas downloadable objects are newly created objects based on classes available at a server.

An object is **inter-operable** if it can be employed by other objects written in different languages, running on a multitude of platforms. Hence, inter-operability is the combination of language, operating system, and network independence.

When an object contains **meta-information**, other objects can retrieve information about the object. We distinguish two types of meta-information: weak and strong. When an object contains weak meta-information, it contains syntactical information about its methods and attributes. Strong meta-information is a semantical description of the methods and attributes that are part of an object. This semantical description can be specified in a formal specification language or informally in text.

Table 1 shows the classification of some (distributed) object technologies according to the above mentioned features. The illustrated technologies are: OMG's CORBA [10], an agent ORB (such as Mole3.0 [2] and Voyager [5]), Microsoft's DCOM and ActiveX [4], and Java with RMI (Remote Method Invocation) [11] and JavaBeans.

Although Table 1 is illustrative to make commonalities between object technologies clear, it is limited to features of single objects. The classification is useful to characterize the components of a style: the architectural styles discussed in the remainder of the paper are partly determined by a subspace of the object-feature space given above.

# 3 Distributed OO Architectural Styles

An interesting question arises: how to get from technology and object features described in the previous section to architectural styles? One answer is by experimenting with the influence of technology and (quality) requirements on software architectures. Based on our experience we

| | remotely callable | downloadable | mobile | inter-operable | meta-information weak | meta-information strong |
|---|---|---|---|---|---|---|
| **CORBA** | + | – | – | + | + | – |
| **Agent ORB** | + | – | + | – | + | – |
| **DCOM** | + | – | – | + | + | – |
| **ActiveX** | + | + | – | + | + | – |
| **Java RMI** | + | + | – | – | + | – |
| **JavaBeans** | – | + | – | – | + | + |

Table 1: A classification of distributed OO technology

have used the features remotely callable, downloadable and mobile as the determinants for architectural styles. We have chosen these features because they describe three distinct ways to make objects collaborate in a distributed environment. And, as stated in the introduction, we are in this paper mainly interested in the interaction and communication, in short collaboration, of components.

The first style, the **distributed object architectural style** comprises software architectures which consist of software components providing services to client applications or other service components. Each object is located at a single, fixed place. Objects on different machines are being connected by an Object Request Broker (ORB) that abstracts from the used network and programming language.

The constituent objects are remotely callable. Because distributed objects only expose their interface via an ORB they are inter-operable. Example technologies supporting this architectural style are CORBA and DCOM.

The second architectural style is the **dynamically downloaded classes style**. Here classes are downloaded to and run on client machines, as illustrated in Figure 1(a). The dissimilarity between distributed objects and dynamically downloaded classes architectures is that while functionality is fixed at one place in the former, it is transported when needed in the latter. Objects, which are running on client-machines, are instantiated from classes which are dynamically downloaded from a server. This implies that client applications are always using the latest version of the available classes, alleviating the job of maintaining a large number of client applications.

In terms of the object-feature space of Section 2, the objects in this style are dynamically downloadable. Additionally, they have to contain (preferably strong) meta-information because the client application has to know how to use the newly downloaded object. Example technologies supporting this style are Java applets, JavaBeans and ActiveX.

A third style is the **mobile objects architectural style**. Mobile objects migrate from host to host, taking both functionality and data while they move, as illustrated in Figure 1(b). Consequently, mobile objects communicate with local objects at the host they currently reside on. This means that mobile objects are a perfect means to implement agents, which wander through a network while collecting information, negotiating with other agents and reporting back to the user who launched the agent.

Mobile objects are, obviously, mobile and contain weak (or even strong) meta-information. Additionally, mobile objects are often remotely callable, e.g. to invite an agent to your machine. Technologies supporting the mobile objects architectural style are agent ORBs such as Voyager and Mole.

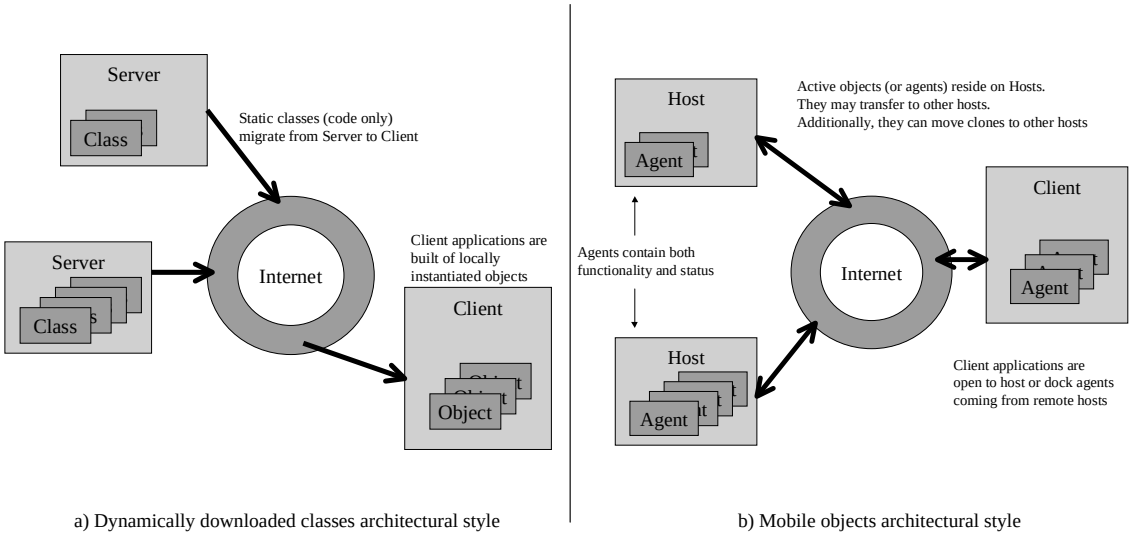


Figure 1: Two architectural styles for distributed OO software

# 4 Application of Architectural Styles in Practice

To illustrate the architectural styles described above, we will now show how the styles are deployed in a practical setting. The application under consideration is an information visualization system. The **distributed visualization architecture** (DIVA) is being used at the Gak (social security provider in the Netherlands) to experiment with business visualization to support decision making [7, 6].

This paper focuses on one particular aspect of DIVA to illustrate the styles, namely, extending the functionality of the system. The system's goal is to support multiple users in visualizing shared information. During a visualization session, users can come up with new visualization primitives which show the information from a different perspective. For example, one of the users discovers an elegant way to display relevant information that otherwise remains hidden. This new perspective must then be shared with other users to support collaboration of the participants. The remainder of this section shows how each style solves the 'problem' of adding functionality (a new visualization) to the system.

**Distributed objects architectural style** In case our system is based on the distributed objects architectural style, the only way people can add functionality is by adding a new distributed object to the runtime system. Figure 2(a) illustrates the process of adding a new visualization perspective. On the left we see a user who has created a new visualization which is made available by means of a distributed object; on the right hand side is one of the users who wants to take a look at the new visualization perspective. At the bottom of the image we see the shared information which is updated periodically.

Once somebody announces that a new visualization primitive is available, other users can connect to the distributed object and request for a visualization of information (1). Consequently, the object retrieves the information from the shared information server and creates a new visualization (2). Finally, the resulting visualization is sent to the requesting client (3). Whenever the information is updated, visualizations will be updated by the distributed object and transmitted to all users (`update`).

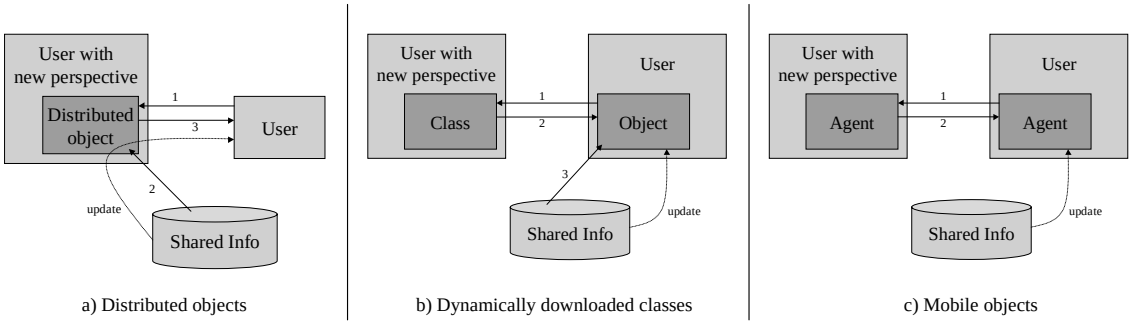


Figure 2: Architectural styles in Diva

**Dynamically downloaded classes architectural style** When we base the visualization application on an architecture that conforms to the dynamically downloaded classes architectural style, users who have created the new visualization perspective provide a class that can be downloaded to other users' machines. As Figure 2(b) illustrates, a user connects to the server that contains the class of the new perspective (1). This server can be the user's machine, or a shared server to which the class has been uploaded. After that, the class is downloaded to the client's machine and instantiated as a new visualization object in the local visualization application (2). Finally, the information is retrieved from the shared information server and accordingly visualized (3). Because each user contains the functionality to present information at her own machine, updates have to be sent to all client machines.

**Mobile objects architectural style** The third architectural style, based on mobile objects, is in some respect similar to the dynamically downloaded classes architectural style. In both cases functionality —*how* we visualize information— is downloaded from a server onto the client machine. However, a mobile object keeps its current status. In Figure 2(c), we see that a user requests an agent from the user with the new perspective (1). Consequently, the display agent clones itself and moves the clone to the requesting user's machine (2). The clone, which contains all the knowledge the originating agent has, does now have to contact the shared information server to show the visualization on the user's display. Updates of the shared information have to be sent directly to all clones of the original display agent.

# 5 Evaluation and Discussion

The goal of describing and classifying the distributed objects, dynamically downloaded classes and mobile objects architectural styles in the previous sections is to provide a basis for deciding on the right type of architecture when creating a distributed object oriented system. In this section we will examine features of the discussed styles and, additionally, give some rules of thumb for choosing which style to deploy in particular cases.

### Feature-based classification

Table 2 contains a feature-based classification of the architectural styles discussed. The **constituent parts** describe the primary building blocks of architectures: components and connectors. The **location issues** determine where objects are created and where they are located during their lifetime. The **functionality issues** determine whether the functionality of the overall system or parts of the system is fixed or extensible.

| | Constituent parts | | Location issues | | Functionality issues | | |
|---|---|---|---|---|---|---|---|
| Style | Components | Connectors | Place of object creation | Object location during lifetime | Funct. of system | Client application | Server application |
| **Distributed objects** | object | ORB | server | server | extensible | fixed | extensible |
| **Dynamically dowloaded classes** | object | various | client (class at server) | client | extensible | extensible | fixed |
| **Mobile objects** | object/agent | procedure call | any | any | extensible | extensible | extensible |

Table 2: Feature classification

The **location issues** are the main discriminators of the architectural styles discussed. The distributed objects architectural style can add new objects at the server side of the system, where objects are staying for the remainder of their lifetime. Dynamically downloaded classes architectures are complementary: they create fixed-place objects at the client side of the distributed system. Architectures based on mobile objects are more flexible in this aspect because objects can be created anywhere and, additionally, can be migrated through the complete system.

The **functionality issues** state that software based on any of the architectural styles is extensible at run-time: new functionality can be added without recompiling or restarting the system. The distinction between the styles, however, is the location of this extensibility. Systems based on distributed objects architectures extend the functionality of systems at the server side, while dynamically downloaded architectures achieve this by extending the client components. Because in the mobile objects architectural style objects can migrate to other hosts both client and server parts are extensible.

### Rules of thumb

Distributed objects are often regarded as the object oriented variant of client–server. However, distributed objects are more than that: distributed objects are namely both client and server at once. Rules 1 and 2 illustrate when it is useful to deploy the distributed objects architectural style. Because inter-operability is a key feature of distributed objects, this style allows the wrapping of dedicated hardware and legacy software into a heterogeneous distributed system (1). Additionally, distributed objects only expose the interface and do not give away the implementation. This allows for its usage in systems where software is not allowed to 'leave' a server because of strategic or security reasons (2).

| Nr | When to use | Style |
|---|---|---|
| **1** | Dedicated hardware or legacy code | Distributed objects |
| **2** | Strategic or even secret code (you do not trust to give away) | Distributed objects |
| **3** | Lots of users expected, resulting in overloaded servers | Dyn. downloaded classes |
| **4** | Often new versions of software (maintainability) | Dyn. downloaded classes |
| **5** | A lot of communication and / or negotiation between the components | Mobile objects |

Table 3: Rules of thumb

When a large amount of clients is expected to be running applications on a single server, the server can easily become overloaded —imagine what would happen when all Java applets would

be running on the server instead of on the client's machine. In this case moving the processing to the client, by deploying dynamically downloaded classes, is the natural solution (3). Additionally, when (parts of) applications are updated often, for example because of changing legislation, architectures based on dynamically downloaded classes are much easier to keep up to date. Clients are automatically using the latest version of the available software (4).

Rules 3 and 4 for dynamically downloaded classes also hold for the mobile objects architectural style, because in this style functionality is downloaded to the client's machine too. However, when multiple objects have to negotiate, for example to vote on something, the amount of communication between objects can be very high whereas the result is only a small answer: 'yes' or 'no.' When the communication can be done locally by moving all mobile objects to the same host, the performance of the system can be improved dramatically (5).

# 6 Conclusion

This paper started with the presentation of an object-feature-space to classify (distributed) object technologies. The features, such as mobility and inter-operability, appeared to be useful to characterize properties of single objects. Additionally, they determine to a large extend the architectural styles in this paper.

Based on our experiences in developing (distributed) OO architectures we have presented three architectural styles for distributed object oriented systems: the **distributed objects, dynamically downloaded classes** and **mobile objects architectural style**. The discriminating feature of the styles is whether data and functionality are fixed at one place or migrating through the distributed environment. Not surprisingly, we discovered that no best style existed. However, particular styles are better suited to meet specific requirements than others. The discussed 'rules of thumb' are a first direction into deciding which style to deploy in particular situations.

# References


[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI series in Software Engineering. Addison-Wesley Publishing Company, 1998.

[2] J. Baumann, F. Hohl, K. Rothermel, M. Schwehm, and M. Strasser. Mole 3.0: A Middleware for Java-based Mobile Software Agents. In *Proceedings of Middleware'98*, pages 355–370, 1998.

[3] Paul C. Clements. Coming Attractions in Sofware Architectures. Technical report, Software Engineering Institute, 1996.

[4] Microsoft. DCOM Architecture - white paper, 1998.

[5] ObjectSpace. *Voyager Core Technology 2.0 User Guide*, 1998.

[6] S.P.C. Schönhage, P.P. Bakker, and A. Eliëns. So Many Users — So Many Perspectives. In *Proceedings of "Designing effective and usable multimedia systems", 9-10 September 1998, Fraunhofer Institute IAO, Stuttgart, Germany*. IFIP, 1998.

[7] S.P.C. Schönhage and A. Eliëns. A flexible architecture for user-adaptable visualization. In David S. Ebert and Charles K. Nicholas, editors, *Workshop on New Paradigms in Information Visualization and Manipulation '97, Conference on Information and Knowledge Management, 10 - 14 November 1997, Las Vegas, USA*. ACM Press, 1997.

[8] Mary Shaw and Paul Clements. A Field Guide to Boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of COMPSAC, Washington, D.C.*, 1997.

[9] Mary Shaw and David Garlan. *Software Architecture: perspectives on an emerging discipline*. Prentice Hall, 1996.

[10] J. Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, 1996.

[11] Sunsoft. Java Remote Method Invocation - white paper, 1998.

# Towards Dynamic Semantic-Directed Configuration Management

**Michael Goedicke and Torsten Meyer**
Specification of Softwaresystems
Department of Mathemantics and Computer Science
University of Essen
D - 45117 Essen, Germany
+49 209 183 - 3481 / 2168
{goedicke,tmeyer}@informatik.uni-essen.de

**ABSTRACT**
A major challenge in designing contemporary distributed systems based on interacting objects is dynamic accommodation of evolutionary change by adding or removing objects during runtime. This implies that semantic information about the constituent objects has to be available in order to check the correctness of adding a new or changed object. In CORBA, for example, this kind of information is simply represented as a list of keywords. In contrast to this simple way we present an approach to achieve semantics directed configuration of object-based software systems in such a way that objects organize themselves on the basis of a richer representation of their semantics concepts.

## 1 INTRODUCTION AND RELATED WORK

Building and evolving complex distributed software systems requires careful consideration of managing change. While in deploying modern distributed systems on platforms like CORBA many low level interoperability problems are going to be solved soon, the problem of architectural mismatch [6] still persists. As discussed in [6] this problem can be overcome by stating explicitly all assumptions regarding the required context for using a specific (new or changed) component. One approach is to choose a suitable representation for defining a wide range of architectural properties of a given system at design time. Various approaches like ACME [5], ΠΙ [9], or UniCon [17, 1] are proposing such schemes helping the designer to state desired architectural properties during design time.

However, in the context of distributed systems deployed on a large scale, e.g. over the Web/Internet, the architectural mismatch problem reemerges in another variant. In such a case it is even more difficult as discussed above since a system has to be extended or changed during runtime. In addition, in many cases it is not possible to change or even look into the operational details of the component infrastructure on which the change or extension has to take place. In order to enable a correct and robust change management in such a setting semantic information regarding the services of software component is essential [8]. This allows to check to which degree a request to include a new / changed component can be accommodated by a given set of components.

In this contribution we present an approach for dynamic configuration of distributed systems based on semantic properties of its constituent components. Our approach uses a prominent model for dynamic change developed at Imperial College, provides it with a formal foundation based on distributed graph transformation (a well-known graph rewrite approach developed at the Technical University of Berlin) and extends it towards semantics-directed system configuration.

In the following we give a brief account from where we draw our basic techniques and other related work. In chapter 2 we will emphasize the application development, using a CORBA environment as a foundation and introducing semantics descriptions as distributed graphs. Then in chapter 3 we will investigate dynamic configuration management based on semantic-directed component interactions. Finally, we will present the structure of a negotiation protocol.

We have presented an approach to realize configurable distributed systems based on semantic-directed component interaction with the help of graph transformation techniques in [8]. This work investigates in detail the basic functionalities of checking and changing semantics descriptions and can be seen as a prelude to the actual paper which emphasizes the negotiation aspect.

Prominent work related to configurable distributed systems is done at the Imperial College, London. In [12] J. Kramer and J. Magee propose a basic model for dynamic change management which separates structural issues from application specific ones, thus distinguishing change rules at the architecture configuration level and change actions at the component level. An example for this dynamic change model is given in [11]. In [13] self-configuring architectures are investigated and in [4, 14] a graphical user interface and a notation for component configurations based on the architecture description language REGIS/DARWIN [15] are presented.

We use the basic change model [12] as a foundation for our approach. We extend it towards semantic-directed system configuration and provide it with an underlying formal representation based on distributed graph transformation. In [18] we have compared the realization of the change model using DARWIN and distributed graph transformation in the light of the example introduced in [11] and presented the benefits our approach provides.

The concepts as well as the formal definition of the graph transformation approach used here - distributed graph transformation - are introduced in [19]. It is based on algebraic graph transformation [2].

## 2 APPLICATION SPECIFICATION

In this chapter we will sketch how the architecture of distributed software systems can de designed such that interaction with dynamic configuration management (presented later in this contribution) is enabled and configurable distributed systems can be realized.

### CORBA

Our approach is based on the Object Management Group's standard CORBA [16, 22]. CORBA defines a higher level infrastructure which enables distributed computing using object-oriented technology. Integration of heterogeneous systems is defined by the Object Management Architecture OMA, the industry standard reference model for object technology. The aim of this paper is to sketch how dynamic configuration of CORBA objects based on semantic properties can be realized on top of the CORBA Dynamic Invocation Interface (DII). While basics wrt. the DII can be found in [16, 22] and first ideas how the DII can be used within this context are presented in [8], this paper focuses on the negotiation aspect. Thus a software component diagram comprising an export section (services that are made public by OMG IDL), an import section (used services which are provided by other objects registered to a CORBA environment), and a body section (the object implementation) shall suffice to abstract from further CORBA details (cf. figure 4).

### Component Semantics

For representing semantic properties we propose to connect a graph to each component. For checking semantic conditions of a server component before a service may be executed we propose to match a graph representing semantic requirements with the server component's semantics graph. Finally, for dynamic evolution of semantic graphs we use graph transformation techniques. These choices are justified by the following reasons:

- for realizing dynamically configurable systems representation of component semantics by text-based means such as keyword lists or simple name/value pairs does not suffice,
- graphs combine intuitive usability and formal basis,
- compliance tests and dynamic changes can be represented well by graph rewrite rules,
- power and parameterizability of the used rewrite approach (distributed graph transformation, see below).

In the following we will use a specialized graph transformation approach developed at the University of Berlin which allows to handle distribution and modularization aspects: *distributed graph transformation* [19, 20].

Distributed the graph transformation is graph transformation at two abstraction levels - the *network* and the *local* level. The network level contains the description of a system's network topology by a network graph and its dynamic reconfiguration during runtime by network rules. The combination of the network graph structure and the local object structures is specified by distributed graphs in order to describe distributed object structures. A *distributed graph* consists of a *network graph* where each network node is equipped with a *local graph* describing a local system state. Each network edge is equipped with a relation on local graphs defining used data and object structures in that way. Each node in a distributed graph may typed and attributed by further data types (cf. [8, 20] for more details to attribution of graphs and distributed graphs).

Communication between local systems takes place via export and import interfaces. In *export interfaces*, local systems present data and objects accessible for other local systems and *import interfaces* contain data and objects from remote export interfaces. Network nodes being the source of a network edge may be interpreted as interfaces. Figure 1 shows an example of a distributed graph where import and export graphs are distinguished by different gray shades of local graph elements. In all examples use relations are unambiguously presented by obeying compatibility with labels.

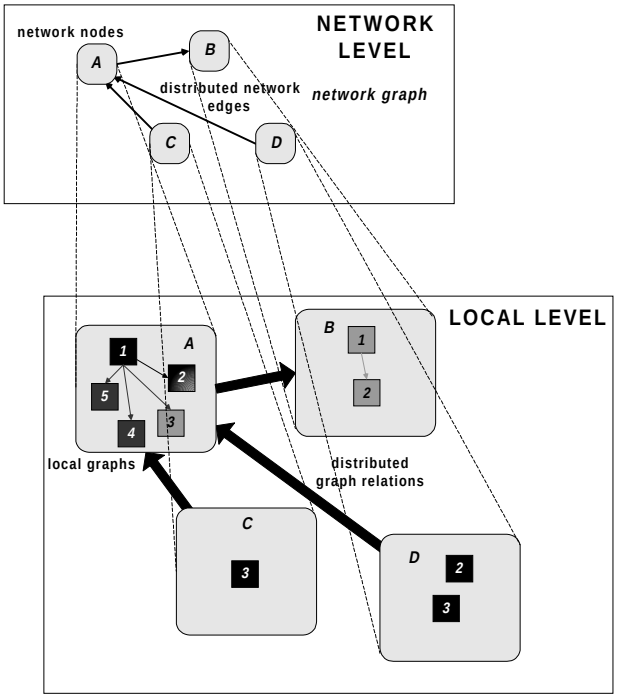


**Figure 1 This example of a distributed graph shows a local view of the system *A*. In this and in the following distributed graphs body elements of local graphs are depicted in *medium gray*, exported graph elements are depicted in *light gray*, imported graph elements are depicted in *black*, and common parameters graph elements (i.e. graph elements which are imported and exported) are depicted in mixed *light gray* / *black*. Relations between local graph elements are not explicitly depicted, we assume a mapping between graph elements with identical labels.**

While graphs are a suitable means for representing semantic properties of software components, checks and changes to semantic graphs can be realized by *graph rewrite rules*. They are used to transform graphs in a rule based manner: *if* a certain structure exists *then* it may be transformed into a new one. At an abstract level a graph rewrite rule consists of a left hand side rule graph L and a right hand side rule graph R. Within a *graph rewrite step* the occurrence of L in a working graph is replaced with R. A detailed description of this graph rewrite approach called double pushout can be found in [2, 8].

In order to realize a graph rewrite step we also need a notion

of graph matching. Graph matching is essential for checking semantic properties represented by graphs. A graph match from a left hand side rule graph into a working graph is a structure preserving mapping between these graphs. It is checked whether nodes and edges, respectively, match by comparing their labels. A formal introduction to graph matches and morphisms can be found in [8, 2].
In the next section we will present a sample semantics description of a software component.

**Example**
Now let us consider a WWW search engine for software components within the Internet. The information involved with a service offer is not only text or text-based HTML code, but more realistically it is comprised of a variety of different and complex knowledge representations - from diagrams, audio and video data to even Java appletts. Thus representation of component semantics by text-based, pseudo-semantic means like simple keyword lists or name/ value pairs (as realized within the CORBA trader) does not suffice.
In the following example we will search for a software component implementing a diagram editor. First the syntactic description of our desired component is obvious: we are looking for operations for adding and deleting bubbles and arrows as well as an operation for printing the diagram.
Since the search result for this generic type of diagram editor would be very large and unstructured, we introduce an actual requirement to the diagram editor object's semantics: the desired kind of diagrams have to be Petri-Nets. This semantic property of a server object can be represented by a graph easily but nearly impossible with textual means. As an example for a class of graphs especially suitable to represent the semantics of a software component we will consider so-called Component Model - graphs. They are defined formally in [10] and are used for describing semantic properties of specification and programming languages. Figure 2 shows an exemplary distributed CM-graph for a diagram editor component. A client application may check this Petri-Net requirement via a graph matching before invoking a request to a diagram editor component.
In the next section we will sketch how the interaction with the configuration management can be realized on top of such an application.

## 3 SEMANTICS NEGOTIATION AND DYNAMIC CONFIGURATION MANAGEMENT

First we will sketch how a component can be found with the help of semantic checks. Then we will discuss how this component can be integrated into a system dynamically.

**Semantic Checks and Change Actions**
As described above we use a special class a graphs - CM-graphs with graph rewrite rules in their body subgraphs - to represent semantic properties of components. Then semantic requirements can be modeled by the left hand side rule graph L of a semantic check rule. The semantic condition represented by L can either check for desired semantic properties (semantic correctness check) or for semantics which are to be avoided (semantic inconsistency check).
If a graph match from the left hand side rule graph L into a source graph representing a components's actual semantics exists, the graph rewrite rule can be applied to the source graph. By replacing the subgraph of the source graph corresponding to L by the right hand side rule graph R, a change in the semantics graph can be achieved. Also a correctness check representing only a consistency/ inconsistency condition and no change action can be described by a graph rewrite rule. For these graph rewrite rules L = R holds; the graph matching from L into the source graph checks for a special semantic state in the source graph and no replace action is performed.

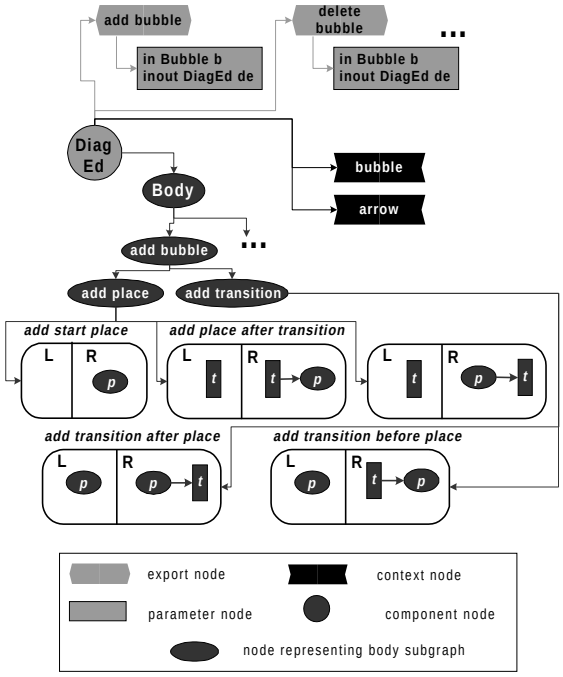


**Figure 2 This example of a distributed CM-graph defines Petri-Net semantics of a diagram editor component. For representation of its interface the CM-graph contains export and parameter nodes for offered services as well as context nodes for used (import) components. Edges only serve as connections between nodes and are not attributed. The export, parameter and import nodes are included in the distributed graph's export and import graph, respectively. In the body subgraph it is shown that bubbles are either places or transitions, thus the operation for adding bubbles is splitted into an operation for adding places and one for adding transitions. The semantics of these operations are given by graph rewrite rules which are part of the body subgraph.**

An important part of the approach is the strategy how semantic properties of distributed components are negotiated, i.e. how the various results from semantics checks are achieved and how they are exploited. Basically there exists a wide range of possible protocols. On the one end negotiation could be done by "fixing" apparent inconsistencies when an object reference is established. On the other end a full check of semantic properties beforehand, i.e. answering the question what are the semantic properties of a service in principle, is also desirable in some circumstances.
Wrt. semantic checks we will distinguish between 'hard' *correctness checks* and 'soft' *compliance checks*.
Within a *correctness check* the client components's semantic requirements have to be satisfied by a potential server component exactly. Thus the existence of an exact graph

match is required. If no graph match exists from a client's left hand side rule graph into a potential server component's semantics graph, the server component may fulfil syntactic requirements but does not satisfy the semantic ones.
While correctness checks demand that the server component semantics description match the client's requirements exactly, *compliance checks* represent an approach more realistic in open and evolving systems. Compliance checks identify common parts of the client's semantic requirements and the server's semantic properties and allow to use server components which match not exactly while trying to manage or tolerate inconsistencies. A compliance check especially relevant within this context is *partial correspondence*, where a match from a subgraph of the client's left hand side rule graph into the server's semantics graph exists. This means that the server component satisfies only a part of the client's semantic requirements. One way to handle this inconsistency is that the usage of the server object is permitted and the client is informed that it has to find additional server objects realizing the missing functionality. On the other hand, the server component may act autonomously and recursively become a client for searching the missing bits of functionality.
A simple protocol for negotiating semantic properties of components at an abstract level is sketched in figure 3. Especially partial semantic compliance is important within this context.

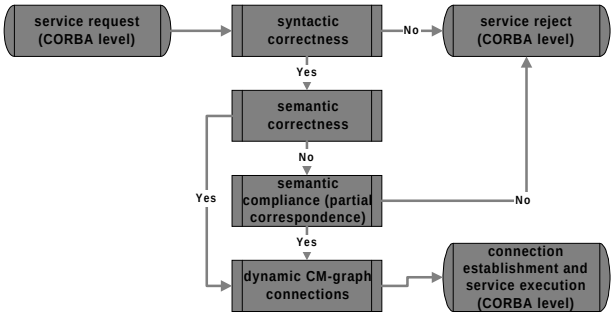


**Figure 3 This diagram shows a general negotiation protocol. The starting point is issuing a service request at the CORBA level. First the syntactic description of the requested service is checked. This can be done either at the CORBA level or at the CM-graph level by inspecting the export and import graphs of a potential server CM-graph. The assumption for semantic checks is that the syntactic check has been passed. If the server component does not meet the requirements for semantic correctness, it is checked for semantic compliance.**

### Semantics Insertion/Deletion and Dynamic Connections/ Unbindings

After a semantically corresponding component has been found, the new component has to be integrated into the system dynamically and relations between its semantics graph and the graphs of the other system components have to be established. In [18] we have shown how distributed graph transformation can be used as an underlying formalism of the original change model. We have sketched how the application can be specified by local graphs and local graph rewrite rules, how the change management's functionalities of dynamic component insertion and deletion can be realized by graph transformation at the network level, and finally how the interaction between both application and change management can be modeled by distributed graph transformation. This paper, however, has a different intention: it is not about specifying the entire application by graph transformation, but it assumes a CORBA architecture in terms of distributable components and component connections as a foundation. Distributed graph transformation is used here for describing and checking semantic properties of components. The configuration management activities of dynamic insertion and deletion do not refer to the components themselves, but to the semantic descriptions of a component. Thus distributed graphs representing component semantics are not isolated entities, but also the dependencies between semantics descriptions are modeled.

### EXAMPLE

Figure 4 sketches the negotiation protocol steps wrt. the diagram editor example:

1. Step 1 begins with an unactualized import at the CORBA level: a client component searches for services realizing the diagram editor functionality as well as simulation functionality (i.e. representation of firing transitions).
2. Then the graph rewrite rule depicted within step 2 checks for syntactic correctness, this is done via export CM-graphs of potential server components.
3. Step 3 shows a graph rewrite rule checking for semantic correctness, its left hand side rule graph contains descriptions of the Petri-Net and the simulator requirements as introduced above.
4. If no component can be found which meets the entire requirements, the graph rewrite rule depicted within step 3 is splitted into graph rewrite rules for every operation within step 4.
5. Assuming the existence of components *DiagEd* and *PNsim* the rules for adding places and transitions are provided by *DiagEd* while the rules for firing transitions and accessing tokens are provided by *PNsim*. The connected CM-graph structure is sketched within step 5.
6. The final establishment of the connection at the CORBA level is depicted within step 6.

## 4 CONCLUSION AND FURTHER WORK

In this contribution we showed an approach to represent semantic information of a component within a distributed software system (based on CORBA). Since this kind of information is expected to be available with such components during their entire lifetime, search for a partially specified component and automatic (re-)configuration of a component configuration based on such semantic information is enabled.
Our approach is based on a graph representation of semantic component properties and graph transformations are used to describe the check for compatibility between a request for a specific component and the available components. We consider various levels of correspondences and describe the way a given level can be established using a special protocol. In our example here we used a representation of functionality - a diagram editor which is extended stepwise by additional means to represent and simulate Petri Nets. Although this kind of semantic information is essential it is only a part of the semantic properties we would like to

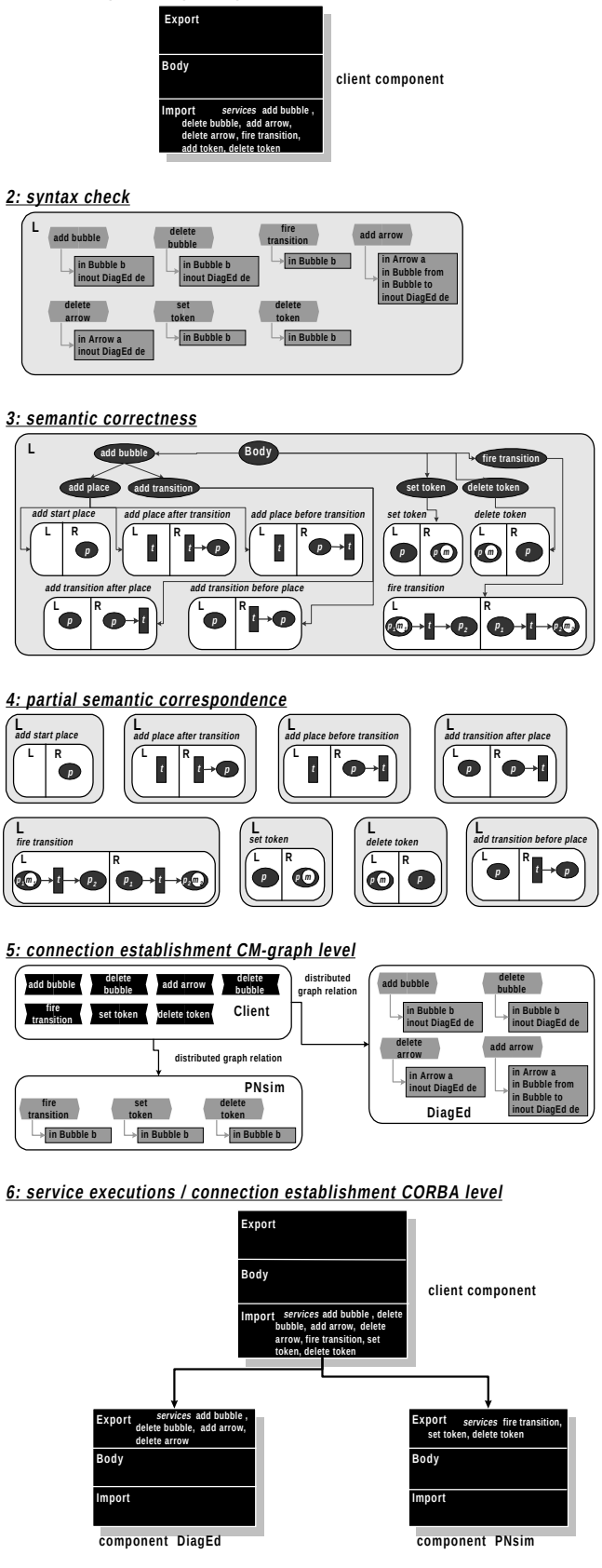


**Figure 4 A sample negotiation protocol.**

represent. Other kinds of important properties are e.g. safety and quantifiable performance. In the area of performance we have established concepts and methods [7] to assess and influence the design process of an software architecture under evolution. These findings will be included in the ongoing work. The challenge which lies in here is to consider global system properties e.g. performance and the modular structure of components of our approach presented here.

In order to execute the various checks in form of graph transformations, we currently implement with the group of Hartmut Ehrig (Technical University of Berlin) a distributed graph transformation system [21, 9] which will help to make the approach presented here more practically.

**ACKNOWLEDGEMENTS**

The original change model on which our approach is based was developed by J. Kramer and J. Magee. Their inspiration and support is gratefully acknowledged. We would like to thank B. Sucrow and G. Taentzer for their helpful inspirations regarding graph transformation and distributed graph transformation.

**REFERENCES**

1. Allen, R. and Garlan, D. Formalizing Architectural Connection. *Proc. of the 16th International Conference on Software Engineering* (Sorrento, Italy, 1994).
2. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M. Algebraic Approaches to Graph Transformation. *In Rozenberg, G. (ed.), Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1 Foundations* (1997), World Scientific, Singapore, 163 - 245.
3. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *Int. Journal of Software Engineering and Knowledge Engineering vol.2/1* (1922), 31-57.
4. Fossa, H. and Sloman, M. Implementing Interactive Configuration Management for Distributed Systems. *Proc. 3rd International Conference on Configurable Distributed Systems* (Annapolis, U.S.A, 1996), IEEE Computer Society Press, 44-51.
5. Garlan, D., Monroe, R. T., Wile, D. ACME: An Architectural Description Interchange Language. *Proc. CASCON* (1997).
6. Garlan, D., Allen, R., and Ockerbloom, J. Architectural Mismatch -or- Why it's hard to build systems out of existing parts. *Proc. 17th International Conference on Software Engineering* (Seattle, USA, 1995).
7. Goedicke, M., Meyer, T., and Piwetz, C. On Detecting and Handling Inconsistencies in Integrating Software Architecture Design and Performance Evaluation. *Proc. 13th IEEE International Conference on Automated Software Engineering* (Honolulu, USA, 1998), IEEE Computer Society Press.
8. Goedicke, M. and Meyer, T. Dynamic Semantics Negotiation in Distributed and Evlving CORBA Systems:

Towards Semantics-Directed System Configuration. *Proc. 4th International Conference on Configurable Distributed Systems* (Annapolis, U.S.A, 1998), IEEE Computer Society Press.

9. Goedicke, M. and Meyer, T. WWW-based Software Architecture Design Support for cooperative Representation and Checking. *Proc. 3rd International Software Architecture Workshop* (Orlando, USA, 1998), ACM Press.
10. Goedicke, M. On the Structure of Software Description Languages: A Component Oriented View. *Habil. Thesis (*1993), Dept. of Computer Science, University of Dortmund, Germany.
11. Kramer, J. and Magee, J. Analysing Dynamic Change in Software Architectures: A Case Study. *Proc. 4th International Conference on Configurable Distributed Systems* (Annapolis, U.S.A, 1998), IEEE Computer Society Press.
12. Kramer, J. and Magee, J. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering SE-16 11* (1990), 1293-1306.
13. Magee, J. and Kramer, J. Self Organizing Software Architectures. *Proc. SIGSOFT Workshops* (1996), ACM Press, 35-38.
14. Magee, J. and Kramer, J. Dynamic Structure in Software Architectures. *Proc. 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (San Francisco, U.S.A., 1996), ACM Press.
15. Magee, J., Dulay, N., and Kramer, J. Regis: A Constructive Development Environment for Distributed Programs" *IEE/IOP/BCS Distributed Systems Engineering Journal 1/5* (1994), 304-312.
16. Object Management Group, Inc., "The Common Object Request Broker: Architecture and Specification", revision 2.1, 1997.
17. Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., and Zelesnik, G. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering Vol.21 No.4* (1995).
18. Taentzer, G., Goedicke, M., and Meyer, T. Dynamic Change Management by Distributed Graph Transformation: Towards Configurable Distributed Systems. *Accepted for 6th International Workshop on Theory and Application of Graph Transformations* (Paderborn, Germany, 1998).
19. Taentzer, G. Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems. *PhD thesis* (1996), Dept. of Computer Science, Technical University of Berlin, Shaker.
20. Taentzer, G., Fischer, I., , Koch, M., and Volle, V., "Distributed Graph Transformation with Application to Visual Design of Distributed Systems", in Rozenberg, G. (ed.), *Graph Grammar Handbook 3: Concurrency & Distribution*, World Scientific,1999.
21. Taentzer, G., Ermel, C., and Rudolf, C. AGG-Approach: Language and Tool Environment. *To appear as article in Rozenberg, G. (ed.), Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2 Specification and Programming* (1998), World Scientific, Singapore.
22. Vinoski, S., "CORBA: Integrating Diverse Applications Within Distributed Heterogenous Environments", *IEEE Communications Magazine (Feb. 1997)*, IEEE Computer Society Press, 1997.

# Software Engineering of a Distributed Object Architecture for Federated Client/Server Systems

H. Gomaa
Department of Information and Software Engineering
George Mason University
Fairfax, VA 22030-4444
+1 (703) 993 1652
hgomaa@isse.gmu.edu

G.A. Farrukh
Booz Allen and Hamilton Inc.
1725 Jefferson Davis Hwy, # 1100
Arlington, VA 22202
+1 (703) 872 4145
gafarrukh@hotmail.com

## 1. Introduction

This paper describes the software engineering of a software architecture, composed of distributed objects, for a federation of client/server software systems. The architecture is specified in the Darwin architecture description language (ADL) [10,11] and implemented on the Regis distributed environment [12]. The architecture is composed of reusable domain specific black box architecture patterns [1,7] and extensible domain specific white box architecture patterns. The implementation of this architecture is a framework [8], which can evolve and be configured to allow new clients or servers to join the federation.

## 2. Federation of Client/Server Systems

Many information systems take the form of client/server applications, e.g., banking systems. Frequently, legacy information systems in the same application domain need to cooperate in order to provide an enhanced service to their customers. Rather than individual client/server systems having to wrestle with interfacing to each other, all information systems that are interested in cooperating together form a federation of client/server systems, which provides an agreed set of services. For each federation service, there will be a corresponding federation transaction type to support it, usually consisting of a client request and server response. Federation transactions are sent using a standard federation protocol for transmitting this information. It is assumed that individual client and server members of the federation have their own local transaction format, which in the general case is different from the federation format. A federation service is therefore provided to translate local transactions to/from federation transactions.

A Federation Interface Manager (FIM) is a software subsystem that mediates each member information system's interface to the federation. The goal is to allow each information system's clients and server to be integrated into the federation with the minimum amount of software modification. There are logically two kinds of FIMs, client FIMs and server FIMs. As new

members of the federation can be added after the federation is operational, it is also necessary to decouple clients from servers through the use of object brokerage services. A high level view of the federation is shown in Fig. 1.

The goal is to have FIMs that are reusable by each member of the federation. However, each member of the federation is likely to need a different variation of the FIM, since each FIM will need to reflect the characteristics of the individual client or server that it is interfacing to, as well as the characteristics of the federation, which are common. Thus a FIM has some aspects that are common to all members of the federation, and other aspects that are specific to each individual member of the federation, such as translation from the individual member's local transaction format to federation transaction format. In addition, federation services can be split into those that are domain independent, e.g., object brokerage services, and those that are domain specific, e.g., banking services in an electronic commerce domain.

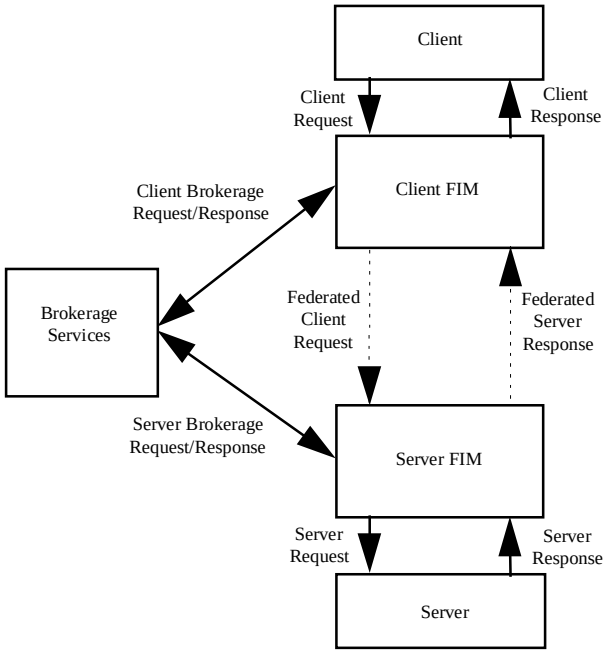


Figure 1: Federation of Client/Server Systems

## 3. Architecture of Federation Interface Manager

The approach described in this paper uses the Darwin Architecture Description Language (ADL) [10, 11] to specify the architecture of the federation of client/server systems. In this paper, distributed components types and architecture patterns are specified using Darwin, which is part of the Regis configuration environment [12,18] for parallel and distributed programming. The Regis environment uses the Darwin ADL for the external specification of each component type, while the internals of component types are implemented in C++.

Darwin components are mapped to corresponding implementation components in the Regis environment. These components can be instantiated at different nodes within the distributed environment. The instantiation can either be static or dynamic.

The Regis environment supports a variety of ways to bind components including first party binding, third party binding, and dynamic invocation binding. Also it supports safe unbinding of components. These services are critical to the design of dynamic distributed systems and are used in the research.

The Regis environment also supports a variety of synchronous and asynchronous communication mechanisms. One-way messages are sent and received by *port* interfaces, messages with reply are handled by *entry* interfaces, and broadcasts are handled by *event* interfaces.

## 4. Overview of Distributed Object Architecture

The FIM architecture operates at three different levels of reuse: federation, domain, and application. At the Federation Level, the architecture is most abstract and can be used for any application domain. The overall structure of client FIMs, server FIMs, and object brokerage services, is defined (Fig. 1). The architecture is defined in terms of the interfaces and interconnections of the component types, and is specified in the Darwin ADL. The implementation of the component types providing federation level services is also domain independent.

At the Domain Level, domain specific functionality is defined. In particular, domain level transactions are defined, e.g., for electronic commerce, which define the type of service provided for the domain, transaction requests containing service requests and parameters, and transaction responses, are defined.

At the Application Level, functionality for individual applications (clients and servers) is added. For example, for an ATM client, its transaction types and translation mechanisms are defined. At this level, individual clients and servers extend and instantiate their respective FIMs and become members of the federation.

The FIM architecture is a reusable and extensible software architecture for a family of systems, also referred to as a product line architecture. The FIM architecture along with its C++ implementation represents a framework [8], which is composed of three architecture patterns (defined in the Darwin ADL) and component types implemented in C++. An architecture pattern [7] is defined in this paper to be a set of interconnected components specified in an ADL in terms of their interfaces and interconnections. Each architecture pattern is the realization of a domain feature [7]. The patterns are micro-architectures for the Client FIM, Server FIM, and object brokerage services. The object brokerage services form a black-box architecture pattern, which is reused without adaptation. There is one instance of this pattern for a given federation. The Client FIM and Server FIM are white-box architecture patterns, which need to be extended before they can be used. There are several variant implementations of the white-box architecture patterns in a given federation.

## 5. Description of Federation Interface Manager

The overall architecture of the federation interface manager consists of three major subsystems: client FIM, server FIM, and object brokerage services. The FIM architecture allows clients and servers to join a federation and to interface with each other. In addition, the FIM architecture can be extended by adding and removing clients and servers after the federation is in operation.

### 5.1 FIM Object Brokerage Services

The FIM architecture object brokerage services are used to decouple clients and servers so that clients and servers can interact with each other and that new clients and servers can join (or leave) at any time. For this purpose, the Federation Registration Server is used. Server FIMs register with Federation Registration Server for that federation and Client FIMs can query it for services. The Federation Registration Server is responsible for maintaining a database of active servers and their addresses. There is one instance of the Federation Registration Server per federation, which is referred to as the Brokerage Services component in Fig. 1.

The Federation Registration Server component is responsible for maintaining a database for all servers in the federation. The database contains the server ids and addresses of servers. In every federation, there must be exactly one instance of the Federation Registration Server component type. Both the Client FIM and the Server FIM communicate with Federation Registration Service component via Regis entries [12]. The Darwin component for the Federation Registration Server is shown below.

```
component Federation_Registration_Server
{
        provide         FRS_Request <entry FrsClass, FrsClass>
}
```

The above component has one incoming interface, *FRS_Request*. This interfaces is of *entry* type, which is used for synchronous message communication with reply. This interface is used by both client FIMs and server FIMs. A server FIM uses it to register and de-register its services with the Federation Server Router, while a client FIM uses it to request a service from a specific server FIM.

### 5.2 Client FIM

A Client FIM is responsible for translating client transactions into federation format and routing them to the server FIM. It is also responsible for receiving server responses from a Server FIM, translating them back to the client format, and sending them to its client. One instance of the Client FIM is needed for every client in a federation. The software architecture diagram for the Client FIM is shown in Fig. 2, using a graphical representation of the Darwin ADL. The dotted lines show run-time binding via Regis components [12] using the Federation Registration Server (FRS). It is important to note that each Client FIM can be instantiated at different nodes within the distributed federation of client/server systems. The Client FIM components are briefly described as follows:

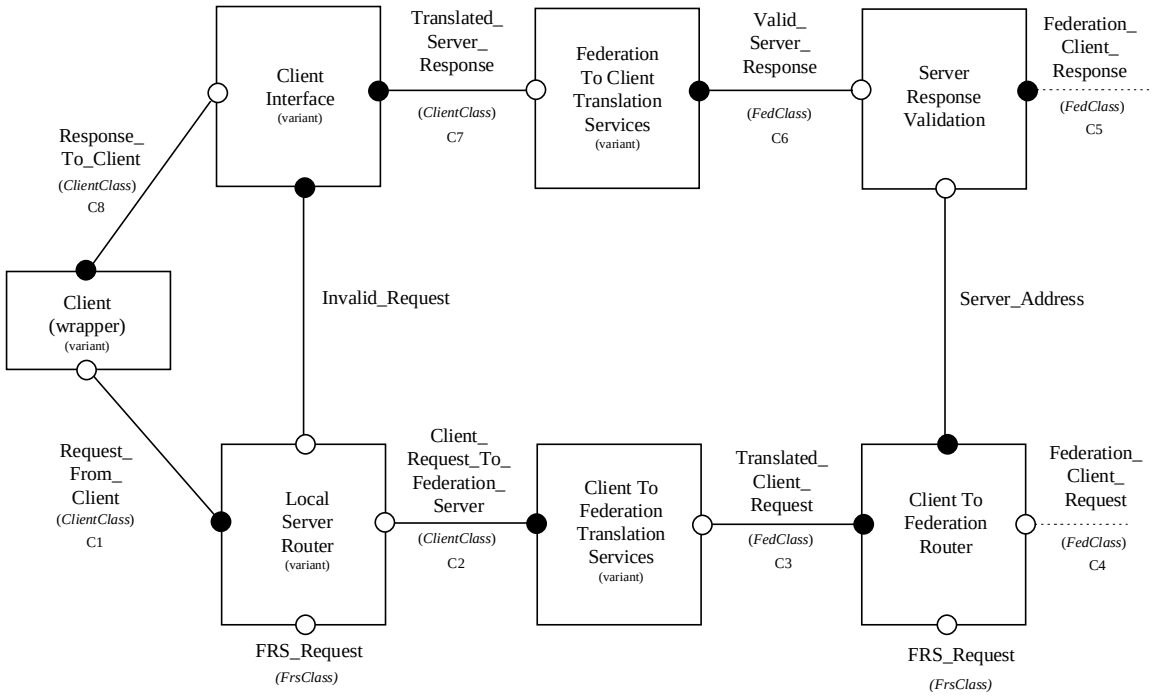


Figure 2: Software Architecture Diagram for Client FIM

The Client component is a client specific wrapper for an actual client and is responsible for sending client requests to the Local Server Router in client format (ClientClass).

The Local Server Router is a variant component responsible for routing valid client requests to appropriate servers and rejecting invalid requests. It queries the Federation Registration Server for the address of the server and forwards valid transactions. The Local Server Router component is shown below using the Darwin language syntax.

```
component Local_Server_Router
{
        provide         Request_From_Client <port ClientClass>;

        require         FRS_Request <entry FrsClass, FrsClass>;
        require         Invalid_Request <port ClientClass>;
        require         Client_Request_To_Federation_Server <port ClientClass>;
}
```

This component has one provide interface and three require interfaces. The provide interface, *Request_From_Client*, is used to receive client requests from client component as shown in Fig. 2. FRS_Request interface is used to send messages to the Federation Registration Server component and check if the server FIM to which this message is sent actually exists within the distributed federation. If the request is found to be invalid, this component sends a message to the Client Interface component via the *Invalid_Request* port (Fig. 3). If the request is send to be valid, it is forwarded to the Client To Federation Translation Services component via the *Client_Request_To_Federation_Server* port for further processing and transmission to the server FIM. All insterfaces in the Local Server Router component are bound statically. This task is

performed when component instances of the Client FIM are put together. Examples of these bindings are shown below using the Darwin language followed by the

```
bind
        Cli.Request_From_Client -- LSR.Request_From_Client;
        LSR.FRS_Request -- FRS.FRS_Request;
        LSR.Invalid_Request -- CI.Invalid_Request;
        LSR.Client_Request_To_Federation_Server -- CTFTS.Client_Request_To_Federation_Server;
```

In the above code example, *Cli* is an instance of Client component, *LSR* is an instance of Local Server Router, *FRS* is an instance of Federation Registration Server component, *CI* is an instance of Client Interface component, and *CTFTS* is an instance of Client To Federation Translation Services component. These statements show how instances of the Local Server Router component are bound with instances of other components in a Client FIM. These instances must be bound to communicate with each other.

The Client To Federation Translation Services component is responsible for translating client requests from client format to federation format (FedClass). An abstract class, ClientFederationTranslator class, is defined, which is later specialized to address the translation needs of a given client.

The Client To Federation Router is a domain independent component, as it deals entirely with transactions in federation format, which it sends to the server FIM. It queries the Federation Registration Server for the address of the server and sends the transaction to the server. Fig. 2 shows that instances of Client To Federation Router component send client requests to server FIMs. The binding for this communication is not done statically. This binding is done at run-time with information obtained from the Federation Registration Server. This is an important element in the design of federation interface manager, which allows communication with potentially unlimited number of servers. This de-coupling approach between clients and servers is a key factor allowing dynamic addition of clients and servers to the federation. Similarly, clients and servers can leave a federation after fulfilling existing requests and denying any new requests.

The Server Response Validation component is also domain independent. It receives responses from Server FIMs and passes them on for translation.

The Federation To Client Translation Services component is responsible for translating federation format responses into client format responses.

Client Interface. This is a client specific component, which is responsible for passing server responses to clients.

### 5.3 Server FIM

A Server FIM receives client requests from a Client FIM, translates them to server format from federation format, and sends them to its server. It is also responsible for translating the local format of a response from the server to federation format and route it to the Client FIM. Once

instance of the Server FIM is needed for every server in a federation. The software architecture diagram for Server FIM is shown in Fig. 3. The architecture of the Server FIM is similar to that of the Client FIM and the Server FIM components are briefly described as follows:

a) Client Request Validation. This is a domain independent component type and is responsible for registering its address with the Federation Registration Server component so it can receive clients requests. An instance of this component type receives a client request, as FedClass message, and passes it on to the Federation To Server Translation Services component in federation format. As an example, the Client Request Validation component is shown below in Darwin:

```
component Client_Request_Validation
{
        provide         Federation_Client_Request <port FedClass>;

        require         FRS_Request <port FrsClass>;
        require         Client_Address <port PortAddress>;
        require         Valid_Client_Request <port FedClass>;
}
```

This component has one provide interface, *Federation_Client_Request*, and three require interfaces, *FRS_Request*, *Client_Address*, and *Valid_Client_Request*. Fig. 3 graphically depicts the relationship of each of these interfaces. The most interesting interface in this component is the Federation_Client_Request interface, which is used to received client request from client FIMs. This interface is bound to the sending client FIM's require interface dynamically. This is depicted with dotted lines in Fig. 3.

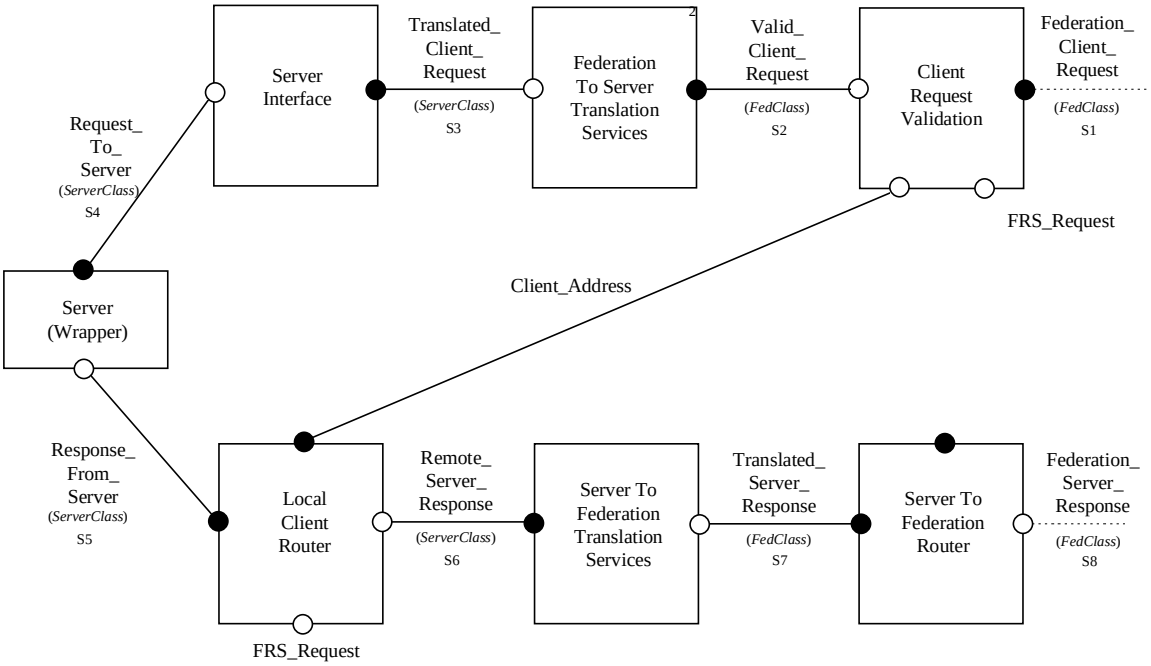


Figure 3: Software Architecture Diagram for Server FIM

b) Federation To Server Translation Services. This is a variant component type, and is responsible for translating a federation format transaction into a server format transaction. An instance of the Federation To Server Translation Services component receives a client request, as FedClass message, and translates it into a client request of ServerClass message type. The ServerClass type defines the server transaction format. This component uses translator classes for performing actual format translation.

c) Server Interface. This component type is variant. Instances of this component type deal only with ServerClass message type. Server Interface component receives client requests from Federation To Server Translation Services and passes them on to the Server component.

d) Server. The Server component type is a wrapper for the actual server. The server component receives client requests from the Server Interface component. It processes those requests and sends a response to the Local Client Router component.

e) Local Client Router. This component type is variant. An instance of the Local Client Router component type receives server responses from Server and passes them on to the Server To Federation Translation Services component.

f) Server To Federation Translation Services. This component type is also variant and is responsible for translating server responses from server format to federation format. An instance of the Server To Federation Translation Services component type translates server responses from server format (ServerClass message) to federation format (FedClass message). The translated responses are sent to the Server To Federation Router component.

g) Server To Federation Router. This component type is responsible for sending a server response to client FIM. An instance of the Server To Federation Router component type receives server responses in federation format (FedClass) and it sends them to Client FIM.

## 6. Comparison with Object Broker Approaches

There are a number of similarities between the approach described in the paper and object broker approaches such as CORBA [17]. In particular, object brokers allow client objects to transparently make requests and receive responses from server objects located locally or remotely. The client is unaware of the mechanisms used to communicate with, activate, access or store information at the server objects. In the federated architecture described in this paper, the brokerage services shown in Fig. 1, and described in Section 5.1, provide a similar service to a CORBA object broker. The four domain independent components, which deal with routing of the federation transactions, provide services equivalent to those in the object broker infrastructure. These are the two client side components, Client To Federation Router and Server Response Validation, of the Client FIM (Fig. 2) and the two server side components, Server To Federation Router and Client Request Validation, of the Server FIM (Fig. 3).

Instead of using the CORBA Interface Definition Language (IDL), the federated architecture approach uses the Darwin Architecture Description Language (ADL) [10,11] for specifying

component interfaces separately from their implementation. A CORBA solution provides the added advantage of being heterogeneous in both target platform and target language. The Regis distributed configuration environment [12] described in this paper is target platform independent but not target language independent as the target language has to be C++. The approach in this paper uses a dynamic binding approach, similar to CORBA's dynamic invocation interface, as the specific server component that a client communicates with depends on the particular client transaction and is thus determined at run time. Darwin also provides additional capabilities for dynamic architectures [18], which were not used in this research.

The other parts of the federated architecture are federation specific, domain specific or application specific, and thus would use the object broker services and not be part of them. For example, the client/federation and federation/server translation services, domain specific transactions, etc. are all specific to the federation architecture. Thus, the federated architecture described in this paper could be built on top of a CORBA infrastructure, with the Client and Server FIMs using the object brokerage services provided by CORBA.

## 7. Conclusions

This paper has described the software engineering of a software architecture, composed of distributed objects, for a federation of client/server software systems. The architecture is specified in the Darwin ADL and implemented on the Regis distributed environment. The architecture is composed of reusable domain specific black box architecture patterns and extensible domain specific white box architecture patterns. The architecture is extensible, allowing it to evolve after it has been deployed. As part of our current research into reusable and extensible software architectures, we are developing an object-oriented UML [15] based analysis and design method for distributed product line architectures composed of use cases [16] mapped to distributed architecture patterns.

## 8. Acknowledgements

The authors gratefully acknowledge several valuable discussions with J. Kramer and J. Magee on using Regis for this research. This research was supported in part by NASA Goddard Space Flight Center, the Virginia Center of Innovative Technology, and DARPA.

## 9. References

[1] Gamma E., Helm, R., Johnson R., and Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.
[2] Gomaa H., "A Reuse-oriented Approach for Structuring and Configuring Distributed Applications", Software Engineering Journal, March 1993.
[3] Gomaa H., "Reusable Software Requirements and Architectures for Families of Systems", Journal of Systems and Software, April 1995.
[4] Gomaa H., et. al. "A Knowledge-Based Software Engineering Environment for Reusable Software Requirements and Architectures," J. Automated Software Engineering, Vol. 3, Nos. 3/4, August 1996.

[5] Gomaa H. and Farrukh, G.A., “An Approach for Configuring Distributed Applications from Reusable Architectures, Proc. IEEE International Conference on Engineering of Complex Computer Systems, Montreal, Canada, Oct. 1996, pp. 442-449.
[6] Gomaa H. and Farrukh, G.A., “Automated Configuration of Distributed Applications from Reusable Software Architectures”, Proceedings IEEE International Conference on Automated Software Engineering, Lake Tahoe, November 1997.
[7] Gomaa H. and Farrukh, G.A., “Composition of Software Architectures from Reusable Architecture Patterns”, Proc. IEEE Internatl. Software Architecture Workshop, Orlando, October 1998.
[8] Johnson, R.E., “Frameworks = (Components+Patterns)”, CACM, Vol. 40, No. 10, October 1997, pp. 39-42.
[9] Kang K.C. et. al., "Feature-Oriented Domain Analysis", Technical Report No. CMU/SEI-90-TR-21,
Software Engineering Institute, November 1990.
[10] Kramer,J., Magee, J., Sloman, M., and Dulay, N., "Configuring Object-based Distributed Programs in REX", Software Engineering Journal, March 1992.
[11] Magee, J., Dulay, N., and Kramer, J., “Structuring parallel and distributed programs,” Software Engineering Journal, March 1993, pp. 73-82.
[12] Magee, J., Dulay, N., and Kramer, J., "Regis: A Constructive Development Environment for Parallel and Distributed Programs", J. Distributed Systems Engineering, 1994, pp. 304-312.
[13] Parnas, D., "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, March 1979.
[14] Shaw, M. and Garlan, D., Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.
[15] Fowler, M. and Scott, K., "UML Distilled", Addison Wesley, Reading MA, 1997.
[16] Jacobson, I., Object-Oriented Software Engineering, Addison Wesley, Reading, MA, 1992.
[17] Mowbray T. and W. Ruh, “Insider CORBA – Distributed Object Standards and Applications”, Addison Wesley, Reading MA, 1997.
[18] Magee, J. and Kramer, J., “Dynamic Structure in Software Architecture”, Proceedings ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM Software Engineering Notes, Vol. 21, No. 6, Nov. 1996.

# Challenges for Distributed Event Services: Scalability vs. Expressiveness

Antonio Carzaniga† David S. Rosenblum‡ Alexander L. Wolf†

†Dept. of Computer Science
University of Colorado
Boulder, CO 80309, USA
{carzanig,alw}@cs.colorado.edu

‡Dept. of Info. and Computer Science
University of California
Irvine, CA 92697-3425, USA
dsr@ics.uci.edu

## 1 Introduction

The event-based style is a very promising approach for the development and integration of distributed objects. An *event notification service* (or *event service*) is the glue that ties together distributed components in an event-based architecture. An event service implements what is commonly known as the publish/subscribe protocol: components publish *events* to inform other components of a change in their internal state or to request services from other components; the event service registers the interest of components expressed by means of subscriptions and consequently dispatches event notifications. In practice, the event service mediates and facilitates the interaction among applications by filtering, aggregating, and delivering events on their behalf. Because of this decoupling, an event service is particularly suitable for supporting heterogeneous distributed objects.

The functionality of an event service is characterized by two conflicting requirements: *scalability* and *expressiveness*. Scalability means that the service must be available over a wide-area network populated by numerous components each one producing and consuming many events. Expressiveness demands a rich *subscription language* that gives applications a flexible and fine-grained selection mechanism to describe precisely those events or combinations of events in which they are interested.

This tension between scalability and expressiveness is evident in all the recently proposed technologies. The ones that provide an event service facility (e.g., the CORBA Event Service [4], the Java™ Distributed Event Specification [8], iBus [7], JEDI [2], Keryx [10], Elvin [6], and TIBCO's TIB/Rendezvous™ [9]), as well as other, more mature technologies not explicitly targeted at this problem domain (e.g., the USENET news infrastructure and IP multicast), represent potential or partial solutions. One problem with some of these technologies (CORBA, Java Events, iBus, USENET News, and IP multicast) is that they offer only a limited selection capability, typically based on a predefined set of "channels" or equivalent multicast addresses, that greatly reduces their potential use as a generic event service. On the other hand, the systems that offer a better data model and better event filtering adopt either a classical centralized architecture (Elvin) or a simple extension of the centralized architecture in which the distributed components are connected in a hierarchical structure (JEDI, Keryx and TIB/Rendezvous). While this latter approach is relatively simple and effective in many cases, we argue that it has some fundamental shortcomings when scaling up to wide-area networks. In particular, it introduces unnecessary message traffic, it overloads higher nodes in the hierarchy, and it has a single point of failure in every node.

We believe that the successful integration of distributed objects by means of events depends on both the scalability and the expressiveness of the event service. Here we propose a research approach to this problem that we pursued with our SIENA project [1, 5]. In particular, we focus on how to realize scalable true content-based routing of events over a distributed event service with a generic topology.

# 2 Conceptual model for a Distributed Event Service

## 2.1 Event Model and Subscription Language

Events are represented by a data structure that we call a *notification*. The data model or the encoding schema of notifications is what we call an *event notification model* or simply *event model*. The event model defines what information can be communicated by means of events, or at least how that information must be encoded. Most of the existing event-based systems adopt a record-like structure for notifications, while others allow more sophisticated modeling by exploiting features akin to an object-oriented language.

Tightly related to the event model is the *subscription language* that defines the form of the selection expressions submitted with subscriptions. Two aspects of the subscription language are crucial to the expressiveness of an event service:

- the *scope* of the selection predicates: the part of the event model that is visible within subscription expressions. In some cases, events have an articulated structure that allows the encoding of much information, but only a limited and / or simple part of that structure can be used as a selection criteria in subscriptions.
- the *expressiveness* of the selection predicates: determines the sophistication of subscriptions. In practice, a subscription language is expressive if it has various basic selection predicates and the ability to combine predicates for the selection of one single event at a time as well as for grouping events into higher-level abstractions.

In terms of *scope*, most existing technologies limit the selection to a single well known element of a notification usually called a "channel" or "subject". A few systems allow filtering based on the content of the whole notification. In terms of *expressiveness*, the simplest models allow a single equality test (channel), while the most sophisticated ones allow for other predicates and conjunctions of predicates.

| | | *scope of subscriptions* | |
|---|---|---|---|
| | | *one field (not structured)* | *multiple fields (structured)* |
| *expressiveness* | *simple equality* | channel-based | simple content-based |
| | *other predicates and expressions* | subject-based | content-based |
| | *multiple events* | subject-based + patterns | content-based + patterns |

Table 1: Classes of Subscription Languages.

Table 1 gives a classification of subscription languages. Note that the difference between "content-based" and "subject-based" is that a channel allows only a straight equality test (e.g., *channel* $= X$) whereas the subject subsumes richer predicates like wild-card string matching (e.g., *subject*="A$*$B"). In both cases, the filter applies to one single (unstructured) element.

## 2.2 Architecture of an event service

Usually, an event service is realized with one or more components called *event servers* (or brokers or dispatchers). The implementation of an event server can be anything from a library to an operating system service to a separate process on the same host or on a remote host. At this point we are not interested in distinguishing these cases. The architecture of an event service is determined by the number of servers, by the topology of connections among them, and by the kind of server-to-server communication protocol. By "communication protocol", we refer to the type and amount of information that event servers exchange. This protocol is obviously implemented on top of some communication mechanism that could range from

shared memory to application-level network protocols such as SMTP or HTTP. At this level, standard encoding and/or tunneling techniques can be used, so we do not discuss the details here.

Most existing technologies that have a distributed architecture adopt a hierarchical topology to connect their servers. In this topology every server may be connected as a common client to a "master" server. The protocol that connects two servers is thus the same one that connects clients and servers. So, except for notifications, which can flow from servers to clients and from servers to other lower-level servers, any other information may flow upward in the hierarchy.

Other technologies, such as IP multicast, have an underlying peer-to-peer network with a generic topology. In this architecture, two connected routers (event servers) exchange routing information (subscriptions) and data (notifications) as peers in both directions.

### 2.3 Classification Framework

We can use the subscription language, which determines expressiveness, and the architecture, which influences scalability, as our classification metrics. Values for the subscription language are: "channel", "subject", "content", and "content + patterns". For the architecture, we have the values "centralized", "hierarchical", and "generic peer-to-peer". Table 2 positions several technologies that are related to event-based infrastructures, including our system SIENA, with respect to these two metrics.

| | | *architecture* | | |
|---|---|---|---|---|
| | | *centralized* | *hierarchical* | *generic peer-to-peer* |
| *subscription language* | *channel* | CORBA, Java Field | CORBA, Java | IP multicast, iBus |
| | *subject* | ToolTalk, | NNTP, JEDI, TIBCO | |
| | *content* | Elvin | Keryx | |
| | *content+patterns* | Yeast, GEM, active database | | SIENA |

Table 2: Classification of Event-Based Infrastructures.

## 3 SIENA: Multicast Routing Revisited

### 3.1 Nature of the Event Service: A Routing Problem

In IP multicast [3], a datagram may be addressed to a *host group*—a "virtual" address that refers to a set of "physical" addresses. Hosts can send a datagram with the usual *IP_send* primitive. Hosts can also join (or leave) a group at any time using the special control primitive *JoinHostGroup* (or *LeaveHostGroup*). The job of multicast-enabled routers is to forward every incoming datagram to one or more of their neighbor routers according to (1) destination (and source) address of the datagram and (2) the group membership information, i.e., whether or not a group has members in one of the attached networks. In IP multicast, a special group membership protocol disseminates group membership information among routers.

It is quite evident that, in a distributed event service, the task of an event dispatcher is substantially equivalent to the one of a multicast router. Subscribing corresponds to joining a group and sending a datagram corresponds to publishing an event. Notice how in a channel-based event service these operations are exactly isomorphic. Depending on the type of event service, however, there might be some fundamental differences.

## 3.2 A Fundamental Difference: Content-Based Addressing

In order to understand the new challenges of an event service, we must examine the routing problem in a bit more detail. In very simplistic terms, routing a datagram $D$ means computing the function *next-hops* $= r(\textit{destination}_D, \textit{routing-info})$. Similarly, managing the routing information in response to a control request $C$ (e.g., a *JoinHostGroup*) is done by updating the routing table $\textit{routing-info}' = c(\textit{group}_C, \textit{host}_C, \textit{routing-info})$, possibly forwarding that information to other neighbor routers.

In the case of IP multicast, *routing-info* can be as simple as a table that associates *next-hops* (interfaces) to group addresses. So, $r(\textit{destination}_D, \textit{routing-info})$ is simply a table lookup $\textit{routing-info}(\textit{destination}_D)$. Group membership maintenance is also easy because $\textit{group}_C$ is a key in the *routing-info* table, so when a host joins a group $\textit{group}_C$, either $\textit{group}_C$ is the routing table or it is not. In this latter case, the router propagates the new membership information, while in the first case the propagation is stopped.

This simplification is possible thanks to the fact that, in IP multicast as well as in a channel-based event service, there is a one-to-one mapping (in fact, the identity function) between destination addresses ($\textit{destination}_D$) and group addresses ($\textit{group}_C$). In other words, a datagram / event is explicitly addressed to one specific group / channel.

Content-based addressing is rather different. The correspondence between the "destination address" of a notification, which is in fact its entire content, and a "group address", determined by a subscription, is not as simple to compute and, more importantly, it is not a one-to-one relation, since a notification might well match more than one subscription and vice-versa. Similarly, when propagating new subscriptions (membership information) in content-based addressing, we can no longer rely on the fact that a subscription is a *key* in the subscriptions table since two different subscriptions might define partially overlapping sets of notifications. In this case, it is crucial to be able to compare the new subscription against the old ones to see if there exist one that *covers* the new one completely so that the new one will not need to be propagated.

## 3.3 The SIENA Event Service

In SIENA we combine a content-based subscription language with a distributed realization based on a generic topology of servers.

The event model is a record-like structure consisting of a set of named attributes, similar to a `struct` in C. A *simple* subscription is a conjunction of filters, each one specifying a condition for an attribute. For example, $\textit{stock} = \text{"}\textit{DIS}\text{"}, \textit{gain} > 10, \textit{gain} < 20$ would select all the events having an attribute named "stock" whose value is "DIS" and an attribute named "gain" whose value is between 10 and 20. SIENA is also capable of observing *compound* events (or *patterns*), i.e., sequences of events. A compound subscription is simply an expression whose elementary terms are simple subscriptions.

SIENA extends the well-known publish / subscribe protocol by introducing another primitive called *advertise*. An *advertisement* is a meta-publication in the sense that it announces the classes of events that an object intends to publish. Advertisements are the dual of subscriptions in that subscriptions declare the intention of receiving notifications, and thus they define the "destination address" of notifications in the routing tables, while advertisements define the "source address" of notifications. Advertisements do not just serve to make the interface complete and symmetrical. The information provided by advertisements can be used to disseminate routing directions more efficiently, thus making the event service more scalable.

## 3.4 Content-Based Routing in SIENA

The routing of notifications in SIENA is based on a generalization of the correspondence between virtual addresses as defined by notifications, subscriptions, and advertisements. We call these correspondences *covering relations*.

A subscription covers a notification when its filter condition is satisfied by the notification. This relation in SIENA embodies the semantics of subscriptions described above and thus involves the evaluation of a conjunction of simple predicates. The covering relation between subscriptions and notifications is used in the routing function. In particular, notifications are forwarded along the paths put in place by subscriptions.

A subscription $x$ covers another subscription $y$ when every notification that is covered by $y$ is also covered by $x$. This relation is used in propagating subscriptions to set up the appropriate routing infor-

mation. When a server receives a new subscription $y$ it looks for a previously registered subscription $x$ that covers $y$. If such a subscription does not exist, the server propagates an equivalent subscription to its neighbors thereby setting up a forwarding path for future notifications. The covering relation between subscriptions is sensibly more complex than the covering between subscriptions and notifications since it includes a universal quantifier over the set of notifications. However, because SIENA allows only a fixed set of "well-behaved" operators (including the usual relational operators such as $=$, $<$, and $\leq$, plus some simple wild-card string match operations), it is still quite efficient to compute.

Similar covering relations exist between advertisements and subscriptions, as well as between different advertisements. These relations can be used as the basis for a dual routing strategy that floods the network with advertisements and forwards subscriptions only along the paths set up by advertisements. Propagating advertisements is also necessary to realize a distributed observation of patterns of events.

### 3.5 Trading Expressiveness for Scalability

As we have seen, the covering relations play a fundamental role in the observation and dispatching of notifications. Many optimization strategies that can be applied to the dispatching algorithms rely on the covering relations as well [1]. Note that since they are an essential part of any basic routing operation, their relevance goes beyond the implementation of SIENA and extends to any event service.

For the sake of scalability it is therefore necessary that these relations be efficient to compute. On the other hand, the features of the subscription language heavily affect the complexity of the covering relations. As we have seen, in IP multicast they are reduced to equality tests between 32-bit numbers, while in SIENA they entail the evaluation of simple predicates and simple first-order logic expressions.

It is easy to show that adding only a little more expressive power to the subscription language makes some of the relations not computable at all. For example, if we allowed user-defined operators in subscriptions, we would still be able to match notifications against subscriptions, but we would lose the ability to reason about the implications among subscriptions. Thus, we would not be able to verify the covering between subscriptions and, as a consequence, we would be forced to broadcast every new subscription.

## 4 Conclusions

We envision a wide-area event service as an effective platform for the integration of distributed heterogeneous objects. However, in the realization of such an infrastructure we see two major conflicting challenges, namely scalability and expressiveness. The fact that these two are conflicting features is shown by a pattern in current event-based technologies: some of them offer rich selection mechanisms, but with a centralized architecture, while others adopt a more scalable distributed architecture, but they give scarce accuracy in filtering events. We know of no event service besides SIENA that features a scalable architecture *and* a fine-grained selection and aggregation mechanism.

By analyzing the functionality of an event service and by comparing it to the well-known problem of routing, we found that the trade off between scalability and expressiveness is not really specific to any implementation, but rather it is intrinsic to the problem domain. In this paper we also sketched some design solutions that we adopted for SIENA. In particular, we have formulated an event service that combines an expressive API with a generic distributed architecture. The dispatching algorithms that implement the SIENA event service are based on the generalization provided by our analysis.

## Acknowledgments

We would like to thank Gianpaolo Cugola, Elisabetta Di Nitto, Alfonso Fuggetta, Richard Hall, Dennis Heimbigner, and André van der Hoek for their considerable contributions in discussing and shaping many of the ideas presented in this paper.

# References


[1] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, Dec. 1998.

[2] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, Kyoto, Japan, Apr. 1998.

[3] S. E. Deering. *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University, Dec. 1991.

[4] Object Management Group. CORBAservices: Common object service specification. Technical report, Object Management Group, July 1998.

[5] D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. In *Proceedings of the Sixth European Software Engineering Conference*, Zurich, Switzerland, Sept. 1997. Springer–Verlag.

[6] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, Brisbane, Queensland, Australia, Sept. 3–5 1997.

[7] SoftWired AG, Zurich, Switzerland. *iBus Programmer's Manual*, Nov. 1998. http://www.softwired.ch/ibus.htm.

[8] Sun Microsystems, Inc., Mountain View CA, U.S.A. *Java Distributed Event Specification*, 1998.

[9] TIBCO Inc. Rendezvous information bus. http://www.rv.tibco.com/rvwhitepaper.html, 1996.

[10] M. Wray and R. Hawkes. Distributed virtual environments and VRML: an event-based architecture. In *Proceedings of the Seventh International WWW Conference (WWW7)*, Brisbane, Australia, 1998.

# On the Role of Style in Selecting Middleware and Underwear

Elisabetta Di Nitto
CEFRIEL – Politecnico di Milano
Via Fucini, 2
20133 Milano, Italy
dinitto@elet.polimi.it

David S. Rosenblum
University of California, Irvine
Dept. of Information & Computer Science
Irvine, CA 92697-3425 USA
dsr@ics.uci.edu

**Abstract**

Middleware infrastructures are becoming a pervasive part of many distributed software systems. Wileden and Kaplan argue that middleware, like underwear, should not be the center of attention but should instead be kept hidden from public view, and it should never constrain or dictate what is publicly visible. These are admirable goals, yet the architects of distributed software systems must nevertheless recognize and account for the intimate relationship between middleware and the systems that use them. In particular, it is useful to view middleware infrastructures as inducing *architectural styles*, in the sense that they embody structural and behavioral constraints imposed on the systems that use them. Defining these styles and identifying the important relationships between them will allow architects to exploit the styles in a way that helps them defer as long as possible those architectural decisions that limit middleware choices, and to develop architectures that can accommodate the widest range of middleware. Otherwise, unattractive middleware choices may creep up on an architect in an annoying way.

# 1 Introduction

Software developers are beginning to make extensive use of *middleware infrastructures* to facilitate component interoperability in large-scale distributed systems. There is an increasingly large number of middleware infrastructures from which to choose, including systems based on middleware standards such as CORBA [6] and Enterprise JavaBeans [7]; proprietary commercial middleware products such as TIBCO's TIB®/Rendezvous™ publish/subscribe software and Talarian's SmartSockets® middleware; and research systems such as JEDI [3] and SIENA [2]. The fact that middleware plays a key role in facilitating interoperability means that it ends up being a critical and pervasive element of any system it supports, fundamentally affecting the architecture, implementation and evolution of the system.

Wileden and Kaplan recently argued that middleware should be treated like underwear [8]. In brief, they said that middleware, like underwear, should not be the center of attention but should instead be kept hidden from public view, and it should never constrain or dictate what is publicly visible. We agree that this is an admirable ideal to strive for, yet as a practical matter one cannot deny the existence of an intimate relationship between middleware and the systems that use them. Middleware, like underwear, must be well matched to the things that clothe it. A particular middleware, like a particular item of underwear, has certain properties, imposes certain

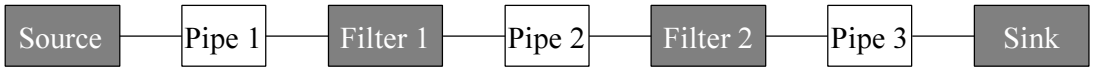


**Figure 1. A pipe and filter architecture.**

constraints, and achieves certain effects that make it well-suited for some situations and ill-suited for others. Indeed, Wileden and Kaplan note that

> *[N]o one style can be expected to meet all needs. Just as football uniforms and tuxedos are best worn with different styles of underwear, different software applications have different middleware needs [8].*

We note further that while the football player must make his or her underwear choices before suiting up, middleware is often not the first item selected to outfit a software system. Other application elements and properties may be decided upon first, with the choice of middleware postponed to the final stages of implementation development. There are numerous sound engineering reasons for postponing middleware decisions, such as maintaining a high level of abstraction and a separation of concerns in early stages of design. But there is no guarantee that an arbitrarily chosen middleware will be tailor-made for the system and will be slipped on with ease. Even if the middleware is selected at the outset of a project (perhaps for economic reasons, or to maintain compatibility with an existing system), the architecture must develop in a way that allows it to tastefully accommodate the middleware.

There arises then a tension between the architectural properties and constraints of an application under development and the properties and constraints of the middleware that will be selected for the application. In other words, a middleware induces a particular *architectural style* in the systems that use it, imposing a variety of structural constraints on the configuration of system components and behavioral constraints on the interactions that take place through the middleware. As more and more decisions are *explicitly* made about the architecture of a system, more and more choices are *implicitly* made about the middleware that will be selected.

We have begun studying the notion of middleware-induced architectural styles and the ways in which they can be defined and exploited in distributed system development [4]. In this paper we discuss some of the issues we are addressing in our work.

# 2 An Example: A Pipe-and-Filter Application

The following example shows how the design of an architecture can conflict with the selection of a specific middleware. The architecture that we consider is shown in Figure 1 and is an instantiation of the pipe-and-filter style as it is defined in [1]. The first component of the architecture, the Source, generates some data and sends them to Filter 1. Pipe 1 is in charge of managing the communication between the two. Both Filter 1 and Filter 2 receive data from the component on their left, perform some computation and produce new data that is forwarded to the component on the right through the connecting pipe. Finally, the Sink consumes the data received from Filter 2. There are several variations of the pipe and filter style that concern the communication mode between components. In particular, components can produce output data incrementally or all at once. Moreover, the communication can proceed according to a *pull* model, in which the producer always initiates communication, or a *push* model, in which the consumer always initiates it.

To demonstrate that the choice of the middleware infrastructure is not independent of the architecture of the system to be implemented, let us compare the implementation we would obtain if we used CORBA as middleware to the implementation we would obtain if we used JEDI. CORBA provides the mechanisms to support point to point communication through remote method invocation. Its main component, the ORB, allows the elements of a system to abstract from the physical location of their counterparts. JEDI is an event-based middleware developed according to the publish/subscribe approach. Components can publish events, and they can subscribe for patterns of events and receive all the events that match their patterns, independently of the source that has published them. An event dispatcher is in charge of managing the dispatching of published events.

By using CORBA, the architecture in Figure 1 is implemented by defining as CORBA objects the Source, the two filters, and the Sink. The pipes are simply implemented as remote method invocations between objects. If we use a push communication model between components, the Source acts as the initiator of a computation by invoking a method PushData provided by Filter 1. In turn, Filter 1 invokes the method PushData provided by Filter 2, which, finally, invokes the method PushData provided by the Sink. In this case, therefore, Filter 1, Filter 2, and Sink act as CORBA servers, and they define an IDL interface containing the method PushData.

If components communicate according to a pull model, the structure of the system is fairly similar. The difference is that the CORBA servers in this case are the Source and the two Filters while the Sink acts as a client.[1] The IDL interface provided by the CORBA servers consists of the method PullData.

Notice that since in both cases all the components that act as CORBA servers share the same interface, the system can be easily reconfigured. The actual attachments between components can be defined when the system is started by passing as a parameter to each component the name of the next component.

Let us try to implement the same architecture using JEDI. As in the CORBA case, the two filters, the Source, and the Sink are implemented as JEDI components. The JEDI event dispatcher acts as a pipe between components. The data transmitted between two components are encapsulated in events, and each component must subscribe to the events carrying the data in which it is interested. Figure 2 depicts a UML collaboration diagram representing an interaction scenario between Filter 1 and Filter 2. As shown in the figure, the choice of JEDI as a middleware infrastructure completely transforms the topology of the architecture of Figure 1. Indeed, if we look at the details of the way the communication is managed, the difference becomes even more evident. In this scenario we assume a pull model of communication, and we focus on the interaction that occurs between the two filters when Filter 2 decides to pull data from Filter 1. Filter 2 simulates a data request by generating the event ReadyToGetData and by subscribing to the event Data that will carry the data produced by Filter 1 (see Figure 2). However, Filter 1 must subscribe to the event ReadyToGetData before Filter 2 generates it. Since the event-based communication is one way, the relationship between events ReadyToGetData and Data is not managed explicitly by the infrastructure, but instead must be managed directly by the components. This places many additional requirements and constraints on the components that were not reflected in the initial choice of pipe-and-filter for the system's architectural style.

In summary, the implementation of the pipe-and-filter architecture in CORBA is straightforward, since CORBA provides direct support for the characteristics of point-to-point

[1] In both the push case and the pull case, Filter 1 and Filter 2 act as both client and server.

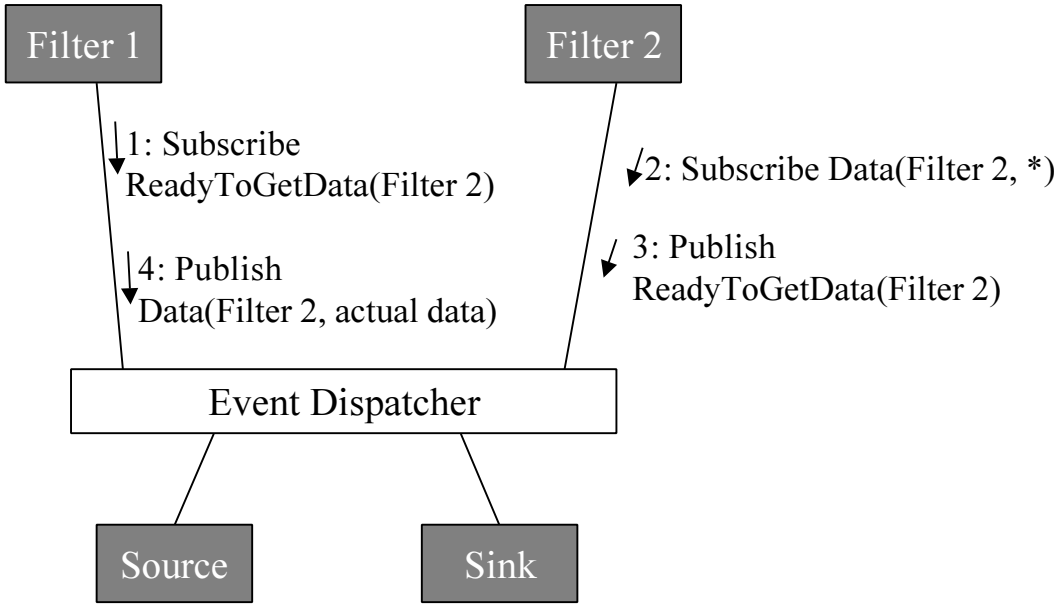


**Figure 2. Implementation of the pipe and filter architecture using JEDI.**

and synchronous communication underlying the style itself. If more complex pipes are needed (e.g., buffered pipes), they would also be defined as CORBA objects and inserted into the chain of inter-object method invocations. The implementation in JEDI, however, requires the architecture to be modified to the point where it no longer resembles a pipe-and-filter architecture. Indeed, the components interact only with the Event Dispatcher, and they must carefully manage synchronization issues in order to simulate the effect of pipe-and-filter style interaction.

In other words, CORBA induces an architectural style that is compatible with the explicitly chosen pipe-and-filter style, while JEDI induces an architectural style that is incompatible (or difficult to reconcile with) the pipe-and-filter style.

# 3 Defining and Exploiting Middleware-Induced Styles

As shown in the previous section, because of the pervasiveness of middleware, every *explicit* decision made in developing the architecture of a system potentially introduces, at least in the case we considered, an *implicit* decision about the middleware that will be used to implement the architecture. As the structural and behavioral aspects of the architecture become defined, the range of middleware choices becomes increasingly limited. How then should the effects of such decisions be made known to the architect? We believe that decisions must be guided by knowledge about the different architectural styles induced by different middleware:

- Middleware-induced styles should be *explicitly defined* and made available to the community of software designers. These styles, in fact, would provide a valuable help in the definition of the architecture of a system, so that the architecture can be easily implemented with a selected middleware infrastructure.

- Middleware-induced styles should also be related to the middleware-independent architectural styles that have been defined so far in the software architecture community [1]. We envisage the definition of a *style map*, where one or several style specialization hierarchies are defined. For instance, one specialization hierarchy could be rooted by a middleware-independent event-based style and could contain all the styles induced by specific event-based middlewares. Definition of an architecture would begin with the style at the root of one of these hierarchies. As architectural decisions are made, the architect would gradually transit the hierarchy toward the leaves. An important design guideline would be to defer making transitions in the hierarchy as long as possible, and to remain as close to the root of the hierarchy as possible, so as to offer the widest possible choice at the time the middleware is to be selected. With such hierarchies, one could determine how significant a design decision is in terms of the effect it has on limiting middleware choices. For instance, it might turn out that selecting between synchronous and asynchronous interaction places greater limitations on middleware choices than selecting between topologies that separate components structurally and those that separate components through data subtyping.

- The styles and style map should be formally described in an *architecture description language* (ADL) [5]. ADLs are being conceived as languages for specifying architectures and architectural styles. The formal definition of middleware-induced styles in a suitable ADL could provide substantial advantages to the architect. In particular, the architect could exploit the features of the ADL to define an architectural model as an instance of a particular style. The architect could then formally check the model for consistency with the properties of the style as the model is refined toward an implementation.

  In a recent paper we evaluated a number of architecture description languages (ADLs) on their ability to support the definition and exploitation of middleware-induced architectural styles [4]. We found that no one ADL provides all the necessary capabilities and features, and hence new ADLs are needed that can support middleware-based architectural development. Hence, new ADLs are needed to support development of systems according to our vision of middleware-induced styles.

# 4 Conclusion

In this paper we have discussed the pervasive nature of middleware infrastructures and the architectural styles they induce in software systems. We have also touched upon a number of issues we are exploring in our study of middleware-induced styles.

Wileden and Kaplan argued that middleware should never constrain or dictate the architecture of a system, just as underwear should never constrain or dictate the rest of one's clothing. We feel that it is nevertheless useful to provide ways of informing architects about the consequences of their design decisions with respect to middleware choices, without restricting their freedom of movement. In this way it may be possible to avoid having unattractive choices of middleware creep up on an architect in an annoying way.

# Acknowledgments

Authors would like to thank Professor Alfonso Fuggetta for his useful comments and suggestions on the issues discussed in this paper and Fabio Lomazzi and Laura Sfardini who have been working at the implementation of the pipe and filter architecture we have used as example.

This effort was sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021; by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061; and by the National Science Foundation under grant number CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Air Force Office of Scientific Research or the U.S. Government.

# References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Reading, MA: Addison Wesley, 1998.

[2] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, "Design of a Scalable Event Notification Service: Interface and Architecture", Department of Computer Science, University of Colorado at Boulder, Boulder, CO, Technical Report CU-CS-863-98, September 1998.

[3] G. Cugola, E. Di Nitto, and A. Fuggetta, "Exploiting an Event-Based Infrastructure to Develop Complex Distributed Systems", *Proc. 20th International Conference on Software Engineering*, Kyoto, Japan, pp. 261–270, April 1998.

[4] E. Di Nitto and D.S. Rosenblum, "Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures", *Proc. 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.

[5] N. Medvidovic and R.N. Taylor, "A Framework for Classifying and Comparing Architecture Description Languages", *Proc. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, pp. 60–76, September 1997.

[6] J. Siegel, *CORBA Fundamentals and Programming*. New York, NY: Wiley, 1996.

[7] A. Thomas, "Enterprise JavaBeans™ Technology: Server Component Model for the Java™ Platform", Patricia Seybold Group, Boston, MA, white paper prepared for Sun Microsystems, Inc. December 1998.

[8] J.C. Wileden and A. Kaplan, "Middleware as Underwear: Toward a More Mature Approach to Compositional Software Development", Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, CA, January 1998.

# View Programming: Towards a Framework for Decentralized Development and Execution of OO Programs

Hafedh Mili, Ali Mili*, Joumana Dargham, Omar Cherkaoui, and Robert Godin
Département d'Informatique
Université du Québec à Montréal
Case Postale 8888, Station Centre-Ville
Montréal, Québec H3C 3P8, Canada
*Institute for Software Research
1000 Technology Drive, Suite 1000
Fairmont, WV 26554, USA
{Hafedh.Mili@,dargham@larc.info,Omar.Cherkaoui@}uqam.ca
amili@cs.wvu.edu

### *Abstract*

*There has been a lot of interest recently in the problem of building object-oriented applications by somehow combining other application fragments that provide their own overlapping definitions or expectations of the same domain objects. We propose an approach based on the split objects model of prototype languages whereby an application object is represented by a varying set of instances-- called views-- that implement different parts of its domain behavior but that delegate its core functionalities to a* ***core instance****: an object's response to a message depends on the views currently attached to its core instance. Our approach is not purely prototype-based in the sense that core instances and views are members of classes. Further, we recognize that the behavior inherent in views (classes) is often an adaptation of a generic behavior to the domain object at hand, and define* ***viewpoints*** *as parameterized class-like algebraic structures to embody such a generic behavior. In this paper, we first describe view programming from the perspective of the developer. Next, we sketch a semi-formal model of view programming, and describe the steps needed to implement it in a class-based statically typed language, for instance, C++. Third, we look at the challenges and opportunities provided by view programming to support safe, robust, and efficient distributed applications.*

## 1. Introduction

As object-oriented systems scale-up from desktop applications to enterprise-wide information systems, developers are faced with the problem of supporting a myriad of functional areas within the same object model. While the objects manipulated may refer to the same real-world objects, each functional area may have its own data requirements, nomenclature, and classification. Traditionally, this problem has been handled in information modeling by modeling the data required by the functional areas separately, normalizing them, integrating them into a unique *complete* model, and re-deriving the *partial* views needed by the functional areas from that model (see e.g. [Ullman 82]). Notwithstanding the difficulties inherent in programming and manipulating objects through separate data views, this process works best in the context of a centralized and pre-planned development activity. In practice, centralized and pre-planned development are neither

practical, nor always possible, and may not even be desirable.

The concept of views in OOP was first introduced by Shilling and Sweeny [Shilling & Sweeny, 1989] as a *filter* of a global interface of the class, but the views are not separable or separately reusable (see also [Marcailloux et al., 1994]). Harrison and Ossher proposed *subject-oriented programming* as a way to build integrated "multiple view" applications by composing application fragments, called *subjects*, which represent compilable and possibly executable functional slices [Harrison & Ossher, 1993]. In principle, independently developed programs/subjects can be composed a-posteriori, making it possible to decentralize ownership and development of OO applications. In practice, the code of subjects must adhere to specific programming guidelines to make subjects composable [Ossher et al., 1995]. Further, subjects cannot be composed dynamically.

In our approach, an application object consists of a core object, to which we can add and remove functional slices, or *views*, reflecting the changing roles of the object during its lifetime. The set of views "attached" to an object determine the messages to which it can respond, and the way it responds to them. We introduce the concept of **viewpoint** as a generic template that is mapped to domain objects to yield views. *Viewpoints* abstract functional behavior in a domain-independent way, and are developed independently of the classes to which they apply. This supports the decentralized development of integrated OO applications, and removes many of the visibility and ownership dependencies that create development bottlenecks and that reduce the reusability of the resulting applications. Further, to the extent that views embody different functional areas, there is every expectation that the underlying data reside, and/or be owned, in different sites, and the aggregation inherent in view programming appears to provide a reasonable boundary with regard to data distribution. Such a naive scheme would probably be inefficient because of the high traffic between the core object and views, and we show various optimizations that can help reduce the overhead.

In the next section, we present view programming scenarios, and introduce some of the basic structural and behavioral mechanisms. A formal model of viewpoints, views, and *viewable objects* is introduced in section 3. In particular, we discuss a number of issues related to typing, and briefly describe a tool set aimed at supporting view programming in C++. We discuss distribution issues in section 4, and conclude in section 5 by highlighting directions for future research.

# 2. Programming with views

## *2.1 Basics*

Typically, we assume that each object of the application domain supports a set of core functionalities that are made available, directly or indirectly, to all the users of the object, and a set of interfaces that are specific to particular uses, and which may be added or removed during run-time. The interfaces may correspond to different types of users with similar functional interests or to different users with *different* functional interests. We would like client programs to be able to access several functional areas or *views* simultaneously, provided that the views are not mutually exclusive. We would also like to have a consistent and unencumbered protocol to address objects that support several views simultaneously. Existing approaches to view programming do not sup-

port the run-time addition and removal of functional views; all of the views that a user (programmer) might wish to address are «declared»/attached at compile time.

Figure 1 shows an aggregation/delegation-based implementation of this idea. The core object includes two state variables, and supports two operations. The view objects, which point to the core object, may add state (view 1 and view 2), behavior (all three), or simply redirect existing behavior (all three). In this case, upon invoking the operation f() on *view 1*, the request is forwarded to the core object, and *the operation f() is executed in the context of the core object*. The same is true for references to the shared state variables ('a' for *view 2*, and 'b' for *view 3*). Practically, there will be a single copy of such variables, stored in the core object, and read/write requests will be forwarded to the core object. Our approach to sharing state variables is consistent with delegation, but our approach to behavior sharing (methods) is different from the typical delegation or prototype-based approaches where the operation in the object being delegated to is executed in the context of the *delegator* [Malenfant,1995]. In our case, we have a purely *forwarding mechanism.*

## *2.3 Lifecycle behavior*

We look at three aspects of an object's manipulation: 1) object creation, 2) view attachment and removal, and 3) behavior invocation on objects. We will distinguish between two kinds of situations, (i) the case where we use a single view on an object, and (ii) the case where several views are used simultaneously. This distinction is important because we want single-view programming to "reduce" to regular programming, with no overhead for the developer. We illustrate our approach using an example in C++ to identify the issues that will have to be addressed in the context of a statically typed language such as C++.

We consider a merchandising organization that manages a fleet of trucks. The finance department views trucks as assets which are amortizable over a certain period of time. The operations department, which operates merchandise deliveries, views trucks as allocatable time-exclusive resources. It also views them as machinery that requires parts, scheduled maintenance, and incidental repairs. Let **Truck** be the interface of the core object (see Figure 2). We will refer the financial, operations, and maintenance views as **FTruck**, **OTruck**, and **MTruck**, with the C++-like interfaces (see Figure 2 for **FTruck**). These interfaces show one way of implementing behavior forwarding, whereby each view points to the core object (data member _truck in classes FTruck, OTruck, and MTruck). Similarly, the core objects refers to the active views through a collection instance variable called _views.

### 2.3.1 Object creation

The lifetime of an application object is bounded by that of its core instance; the lifetimes of the views are included within those bounds. For the case of single view programming, the developer need only see the definition of the view class, and should be able to create instances of the application object through the view. The following excerpts illustrate what we mean:

```
(0)  #include <ftruck.h>
(1)  FTruck* anFtruck = new FTruck(123);
```

The line (0) includes the file that contains the definition of the view class (**FTruck**). In line (1),

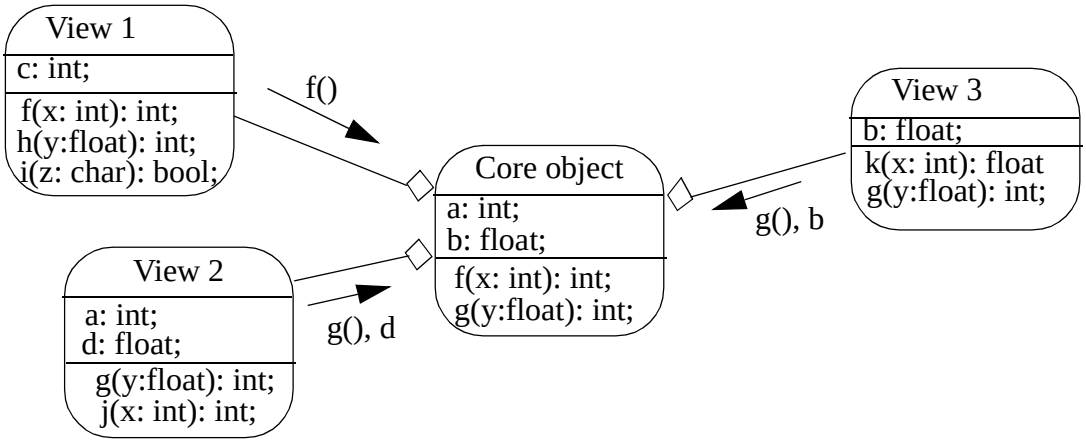


**Figure 1. A model of objects with views.**

we "bring about" the application object with Id 123, and that supports the functionalities of **FTruck**. The idea here is that, behind the scenes, if an instance of **Truck** with Id 123 existed already, either in persistent storage, over the network, or in main memory, then 'anFtruck' is attached to it, and is used to access it functionalities; if no such instance of **Truck** existed, then one is created. This behavior can be obtained if we make sure that all view classes have a one argument constructor that calls some sort of a "core object broker".

For the case of multiple views, developers are aware of both the core object class and of the view classes, and they have to instantiate the core object *explicitly*, and 'construct' view instances for that core instance. This is illustrated in the following code excerpts:

```
(0)   #include <truck.h>
(1)   #include <ftruck.h>
(2)   #include <otruck.h>
(3)   #include <mtruck.h>
(4)   Truck* myTruck = new Truck(id);
(5)   FTruck* myFTruck = new FTruck(myTruck);
```

In this case, we include the core class as well as the view classes. Line (5) shows a way of attaching a view to a core instance using the one argument constructor. View attachment and detachment is discussed below.

### 2.3.2 View attachment and removal

For the case of single view programming, view creation and attachment is indistinguishable from "object creation" since the same operation does both. With several views, view creation is a separate operation. Our intent is to make the behavior embodied in a view available to the core object as long as the view is attached, but also to be able to switch that behavior on and off during the lifetime of the object. Because views may maintain independent state variables, we distinguish between view creation and attachment, on one hand, and view activation on the other, and between view desactivation, on one hand, and view detachment and removal on the other. Within a given application, we might have several requests to create a view of a particular object; a single

view should be created, independently of the number of requests to create such a view, where the first request creates the view, and the subsequent ones return the existing view.

```
class Truck:... {
public:
        ...
        SNumType
                getSerialNumber();

        MakeType getMake();
        Year getModelYear();
        Date getPurchaseDate();
        float getLoad();
        static Truck* getTruck-
                WithId(IdType anId);
protected:
        void setSerialNumber(
                SNumType);
        void setMake(MakeType);
                ...
private:
        IdType _id;
        List<View> _views;
        static Dict<IdType,Truck>
                _registeredTrucks;
                ...
}
// The core class
```

```
class FTruck {
public:
        FTruck(IdType anId);
        ...
        SNumType
                getSerialNumber();
        Date getPurchaseDate();
        float getResidualValue();
        static float
                getAmortPeriod();
        static float
                getAmortRate();
        float getActPurchVal();
protected:
        void setActPurchVal(float);
private:
        Truck* _truck;
        float _actPurchaseValue;
        float _purchaseValue;
        ....
}
// The financial view
```

Figure 2. The core class and financial view of a truck.

### 2.3.3 Behavior invocation

With single view programming, developers see only the interface of the view class, and all the behavior in that interface is available. Behind the scenes, some of the methods that are "callable" from the view are actually delegated to the core object. For example, "getSerialNumber()" is available in the **FTruck** interface, but its implementation forwards to **Truck::**getSerialNumber(). When we have several views, messages are sent primarily to the core object. If one of the views that are currently attached supports the requested behavior, the request is satisfied. Otherwise, the request is denied. In a reflective language such as Smalltalk, this behavior can be accomplished by modifying the dispatching mechanism [Mili & Dargham, 1997]. In a typed and (mostly) statically bound language such as C++, this behavior can be obtained by performing the appropriate compile-time code transformations. Consider the following program excerpts.

```
#include <truck.h>
```

```
#include <ftruck.h>
#include <otruck.h>
#include <mtruck.h>
(1)   Truck* myTruck = new Truck(id);
(2)   FTruck* myFTruck = new FTruck(myTruck);
(3)   OTruck* myOTruck = new MTRuck(myTruck);
(4)   Date t0 = myTruck->getDateNextMaintenance();
(5)   myTruck->releaseOn(t3);
```

In line (4) the programmer invoked a behavior that is available in the MTruck view on the instance of Truck, without referring explicitly to the view instance. The underlying mechanism is a pre-processor that replaces line (4) with the following line:

```
(4')   Date t0 = myTruck->getView('MTruck')->getDateNextMaintenance();
```

because it knows that 'getDateNextMaintenance()' is available in the view class **MTruck** [Mili & Dargham, 1997], but it does *not* know for sure that *at the time* that the call is made, an **MTruck** view is *attached* and *active*[1], and we cannot sort this out at compilation time.

Assume now that the method 'releaseOn(Date)' (see line (5)) is supported by the operations view (**OTruck**) and the maintenance view (**MTruck**). We adopted the approach advocated by Harrison & Ossher [Harrison & Ossher, 1993] which consists of *composing* the various method implementations. Our approach relies on a *universal composition view* which is automatically generated to contain default implementations for the all the multiply defined methods:

```
class __Truck_CompositionView {
public: ...
        void releaseOn(Date t){
                _truck->getView('OTruck')->releaseOn(t);
                _truck->getView('MTruck')->releaseOn(t);}
        ...
```

Developers can edit it to conform it to their intent (see e.g. [Ossher et al., 1995]); the actual mechanics of composition views are slightly more complex [Mili & Dargham, 1997].

By using a code rewriting approach, we strove towards making view programming as natural and transparent as possible. There are, however, some implicit assumptions that developers usually make when dealing with a class hierarchy, that would be violated because of delegation [Mili & Dargham, 1997]. In terms of visibility and access properties, it is useful to think of the relationship between a view and a class as the private subclass relationship in C++[2] where one has to think explicitly of what to export through the subclass relationship. The distinction between the core class hierarchy (**Truck**) and the view class hierarchy (**FTruck**)-- which is transparent to the single view user-- manifests itself when we want to extend the view class **FTruck**: that extension should not break the forwarding mechanism. This leads to a number of more or less easily enforceable guidelines, which compelled us to forbid view extension, but use *viewpoint special-*

1. The actual code substitution is more fault tolerant and allows for a graceful degradation in case no such view is *currently* attached.
2. Intuitively, A is a *private* subclass of B, if the knowledge of the subclass relationship is private to A's methods, i.e. only A's methods can refer to (non-private) data and function members of B

*ization* to the same effect [Mili & Dargham, 1997].

### *2.4 Viewpoints: «horizontal reuse» of functional slices*

We recognize that the functionality provided by a view such as the financial view above may be useful for other kinds of objects/assets and propose to define some sort of a template of a functional view that is parameterized by the elements of the *required interface* of the core object. This template, called *viewpoint*, can then be instantiated for different types of assets, be they trucks or buildings or machines or computers. For example, the _serialNumber attribute is seen as a special case of a general inventory ID, and may be replaced by other domain object specific identifiers. Using *viewpoints*, views may be seen as the mapping between a view and a domain object. The use of viewpoints provides an additional reuse dimension, one for the *developers* of views.

## 3. Implementing viewpoints and views in C++

### *3.1 A framework for viewpoints and views*

A viewpoint is a parameterized type VP[*TH*], where *TH* is a *theory* describing the type of the domain object to which VP may be mapped. We illustrate the syntax through an example:

```
VIEWPOINT FinancialAsset
    REQUIRES CapitalAssetTheory [AgeType -> Year,
                                 PurchaseValueType -> CurrencyType]
    EXTENDS
    void setPurchaseDate(Date d) after
        {setAmortizationPeriod(Year(Date::today() - d)); }
    PROVIDES
    variables
        Year _amortizationPeriod;
        CurrencyType _residualValue;
    operations
        CurrencyType getResidualValue () {return residualValue;}
        void setAmortizationPeriod(Year y){_amortizationPeriod = y;}
                    ...
END VIEWPOINT
```

Figure 3. A viewpoint definition.

The requirements on the (type of) objects to which the viewpoint may be applied are described in the **requires** clause. In this example, we specify such a requirement in terms of a previously defined theory, *CapitalAssetTheory*, whose *sort* or type parameter *AgeType* was bound to the type **Year**, and whose sort *PurchaseValueType* was replaced by the sort *CurrencyType*. The **extends** clause allows us to specify blocs of code that are to be executed by the generated view before (**before**) or after (**after**) the specified core object methods (which must be part of the **requires** interface), in much the same way method wrappers work in CLOS. Other syntactic flavors for the specification of viewpoints have been provided, including the in-line specification of new theo-

ries, or in-line extensions of existing ones [Mili & Dargham, 1997].

A view *V* is generated by instantiating a viewpoint *VP*[*Th*] for a type *T* that satisfies the theory parameter *Th* for a given correspondence Σ, i.e. $T =>_{\Sigma} Th$, and we write $V = Vp [ Th ->_{\Sigma} T ]$. The one-to-one correspondence Σ maps names of *sorts*, *variables*, and *operations* of the *theory* to the corresponding constructs in the type (e.g. class interface). In C++, we may write:

$$V = T \textbf{ as } VP [ s_1 ->\Sigma(s_1),..., v_1 ->\Sigma(v_1),..., op_1 ->\Sigma(op_1),... ];$$

where the "clauses" 'x -> Σ(x)' represent the various substitutions to replace the component constructs of the theory (e.g. sorts or variables) by the corresponding constructs in the type *T.* The reader will notice that the code for the **extends** and **provides** methods of the viewpoint is aggregation-unaware. hence, in addition to the aforementioned substitutions, view generation will transform references to **required** (or **extended**) variables and methods to delegated references[1]. For example, if *f* is a method that is part of the **provides** clause, and if *f* calls the **require**'d method "getPurchaseDate()", then its code:

```
void f(...){...
        Date d = getPurchaseDate();
        ...}
```

will be transformed into:

```
void f(...) {...
        Date d = _truck->getPurchaseDate();
        ...}
```

where _truck is a variable of type **Truck*** that is automatically added to the view[2]. References to **require**'d variables are handled the same way [Mili & Dargham, 1997]. In both cases, we have to make sure that the required variables and methods are somehow accessible to the generated view class[3]. We won't expand further on view generation; the reader is referred to [Mili & Dargham, 1997] for a more complete catalog of transformations and outstanding problems.

## *3.2 Typing issues*

With view programming, there are various hierarchical (symmetric, transitive) relationships between the various constructs with different implications on reuse, behavioral substitutability, and the like. We examined three kinds of relationships:

1. The specialization/subsumption of viewpoints, as a way of incrementally specifying and reusing viewpoints,
2. Subtyping, and more generally, behavioral substitutability of views derived from hierar-

1. This choice is motivated by our desire to repackage existing code where several views are implicitly merged with the core object functionality, into viewpoints (see [Mili & Dargham,1997] and section 4.1). Also, we want our model of viewpoints to be independent of the implementation mechanism-- in this case, aggregation.

2. In reality, there are two possible code transformations, the one shown above, and one to _truck->getCompositionView()->getPurchaseDate(...) so that the view code gets to use whichever version of getPurchaseDate(...) is available to the truck at the time of the invocation, including one (or several) view versions. In effect, using getCompositionView(...) is like opting for dynamic binding. Most approaches to delegation use the latter interpretation. We choose to also support the former mode in case views correspond to different access rights/privileges.

3. In C++, they have to be public members of the core class, or else, the view must be a friend of the core class [Mili & Dargham,1997]

chically related viewpoints, or from views derived from the same viewpoint, but for hierarchically related core classes, and

3. Dynamic subtyping, and more generally, behavioral substitutability of hierarchically related core objects, to which we attach views, possibly generated from hierarchically related viewpoints.

For the purposes of this paper, we will be content to highlight the major issues raised by our framework:

- The multiple specialization of viewpoints raises the issue of combining *before* and *after* methods (our solution: using defaults, that can be overridden),
- Identifying conditions under which the application of two hierarchically related viewpoints to a class yields two hierarchically related (from a typing perspective) view classes,
- Supporting *pure*[1] dynamic typing in a statically typed language in a way that strikes the right balance between flexibility and safety, or at the very least, graceful degradation.

There is a host of other issues discussed in [Mili & Dargham, 1997].

Implementation-wise, we are interested in statically typed languages to be able to perform relatively type-safe code transformations; we chose C++ because it is widely used. Our approach consists of adding non-ambiguous syntactic constructs to the C++ language to define views and viewpoints, and putting programs that use these constructs through a bunch of pre-processors that ultimately, generate standard C++ code. Our tools are being developed with flex and yacc (bison), and we keep discovering unsuspected joys of dealing with C++ semantics.

# 4. View programming and distribution

## *4.1 Issues*

View programming supports the decentralized development of applications that span a variety of functional domains because viewpoints and viewpoint hierarchies can be developed independently from core classes, and the generation of views for a particular class does not require owning the definition of the class. In this section, we are more interested in the distributed implementation of an object with several views. Figure 4 shows a possible distribution scenario involving the same object of Figure 1. we make the distinction between two kinds of “distribution”. First, we have the case where a given site sees (and “believes”) a single view, or a subset of views, and is not aware of the existence of the other views. Second, we have the case where a site knows of all the views, but hosts only one, or a subset thereof. The two situations raise different sets of issues. We assume in this example that site 3 is not aware of the existence of a core object behind the view, or of View 1 and View 2, and the behavior of these should not be available to it, except indirectly as a side effect of methods called on the core object (e.g. through the ***before*** and ***after*** methods). For the case of sites 1 and 2, they know about view 1 and view 2, but don’t know about view 3, and any behavior invoked on the core object should only invoke the methods that are explicitly provided by view 1 and view 2 (or as side effects of such behaviors).

1. this is not just of matter of picking the right implementation for a signature that was known at compile time, in principle, we don’t even know which signatures an object will support at run-time because of the dynamic attachment and removal of views.

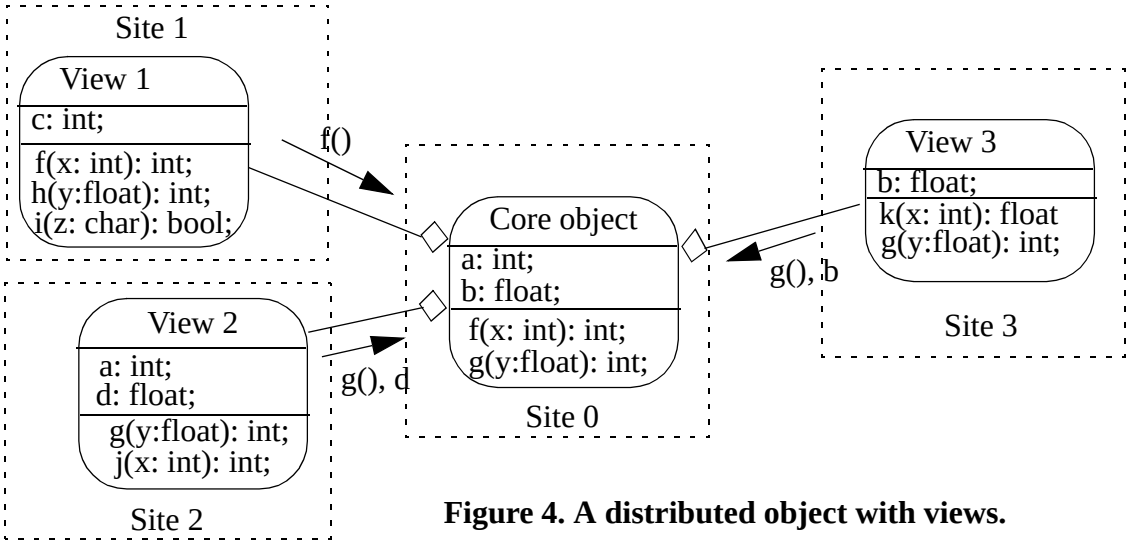


**Figure 4. A distributed object with views.**

## *4.2 A model of distribution*

We address our model of view programming from the perspective of a CORBA/RMI-like model where a single state-holding copy of an object is available over the network whereas different proxies/stubs route requests to that object through ORBs. Figure 5 illustrates such a model. We assume for simplicity that a single ORB manages requests on behalf of all sites. The issues to consider in this model are:

1. where to put the distribution boundaries, and which objects to replicate, if any, and
2. re-evaluating the code transformations implemented to support message forwarding in light of the performance factors of a distributed implementation.

Having a single system-wide copy of any object and stubbing all the objects that are not local to a given site, may have severe performance problems. Consider the code excerpts shown earlier. Assume that the core class **Truck**, and the view classes **OTruck** and **MTruck** reside on sites 0, 1, and 2, respectively, and that we are writing an application on site 1, that uses both views[1]. We assume that the invocation of the constructor of the Truck stub will do the right thing, i.e. either locate a live object with identifier ‘id’, and return a reference to it, or invoke the lifecycle service to create or reanimate such an object from site 0.

```
                #include <truck.h>
                #include <otruck.h>
                #include <mtruck.h>
(1)             Truck* myTruck = new Truck(id);                 // remote
(2)             OTruck* myOTruck = new OTruck(myTruck);         // local & remote
(3)             MTruck* myMTruck = new MTRuck(myTruck);         // remote
(4)             Date t0 = myTruck->getDateNextMaintenance();    // remote
(5)             myTruck->releaseOn(t3);                         // remote & local
```

1. We assume that the client versions of **Truck** and **MTruck** are also called *Truck* and *MTruck*.

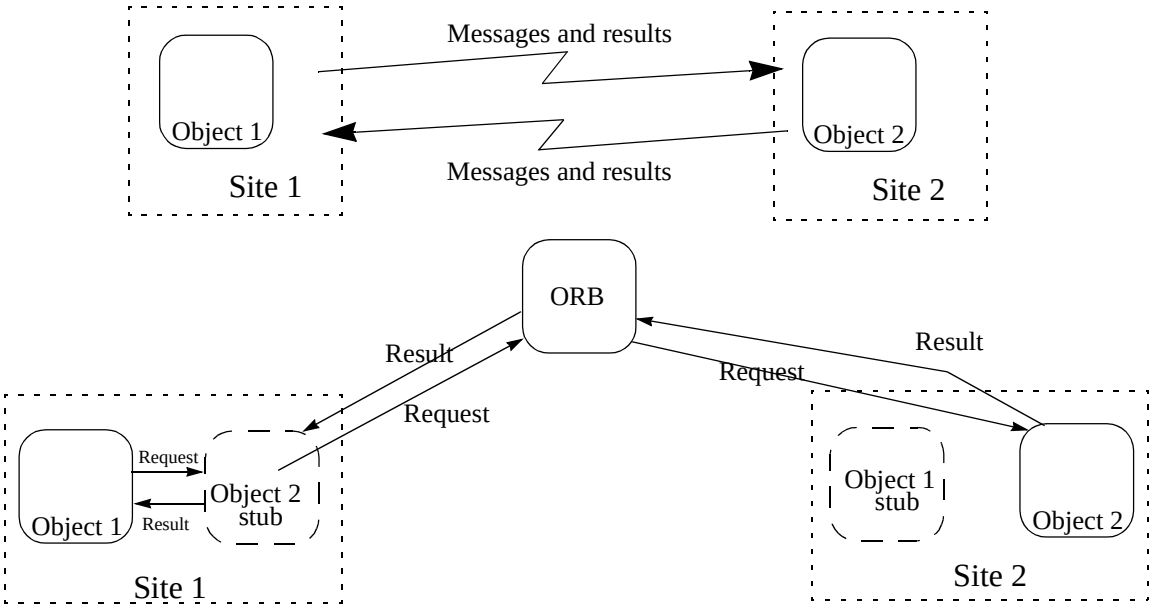


Figure 5. A CORBA-like distribution model.

Line (2) involves both local and remote access: the creation of the view is local but the connection to the core object is remote. Line (3) involves two remote accesses, one to create (or reactivate, or return reference to an existing) OTruck view, from site 1 to site 2, which passes the ORB reference of the core object to the view in site 2, and from site 2 to site 0, to connect the remote view to the remote core object. Line (4), when transformed, becomes:

(4') Date t0 = myTruck->getView('MTruck')->getDateNextMaintenance();

Under this transformation, we need a first remote access to get an ORB reference to the **MTruck** instance (through the "getView('MTruck')" call), and then a remote method invocation of 'getDateNextMaintenance' on that instance. It is clear in this case that the remote call to "getView(...)" is somewhat redundant since instruction (3) has already fetched a reference to that object. A more efficient transformation would yield:

(4") Date t0 = myMTruck->getDateNextMaintenance();

Generally speaking, the view pre-processor can generate variables that will contain references for the views, which will be initialized on the first call, and used thereafter.

Because the method "releaseOn(...)" is provided by two views, line (5) is replaced by:

(5') myTruck->getCompositionView()->releaseOn(t3);

Here again, the code pre-processor could save an ORB reference to the composition view in a local variable to save one remote call. Using the default implementation, i.e.:

```
void __Truck_CompositionView::releaseOn(Date t){
        _truck->getView('OTruck')->releaseOn(t);
        _truck->getView('MTruck')->releaseOn(t);}
```

Using the normal processing mode, the execution of "__Truck_CompositionView::releaseOn(...)" will invoke the following remote calls:

1. One remote call to invoke __Truck_CompositionView::releaseOn(...) from site 1 to site 0,

2. One remote call to invoke OTruck::releaseOn(...) from site 0 *back to* site 1, and
3. One remote call to invoke MTruck::releaseOn(...) from site 0 to site 2.

Because the composition view holds no state, having duplicates carries no overhead, and thus, we can duplicate it in all the sites. This will obviate the need for the first remote call. By explicitly storing pointers to the attached views (rather than going through the core object), the call to OTruck::releaseOn(...) becomes local, and that to MTruck::releaseOn(...) still requires a single remote call. We have thus saved two remote invocations out of three, notwithstanding any remote invocations one of the two methods might make to the core object's methods.

## *5.3 Further optimizations*

A common problem in distributed application design is to identify the recurrent patterns of communication inherent in the application code to help optimize the distribution of data and processing (see e.g. [Purao et al., 1998]). In our case, we assume that data are "owned" by the sites in which they reside, and cannot be moved elsewhere for optimization purposes. Thus, any further optimizations will have to come from duplication. Referring back to the example of Figure 4, we consider the case of the sites S1 and S2, both of which are "aware" of the core object and the views View 1 and View 2. For the sake of simplicity, we ignore the *before* and *after* methods, which create indirect call relations *between* views. Any method of the combined interface will give rise to a call graph with the first call made to the composition view. The level 1 calls will go either to:

- *the core object*: in case the operation is supported by the core object. Subsequent calls will remain local[1] to the core object's site (site 0),
- *a view*: in case the operation is supported by a view. Subsequent calls will involve either local calls within the view, or calls to the core object, with subsequent calls remaining within the core object's site.
- *a combination of the above*, in case the operation is supported by both.

In the end, each call to a method of the combined interface (core object + view 1 + view 2) may ultimately require access to parts of the core object.

We can reduce communication costs by finding a way of limiting the number of hops we need to make across the network for any method call, to one hop. A brute force and not very efficient method consists of duplicating the entire core object everywhere, considering that the duplicates will need to be updated whether the view at that site needs the updated variable or not. A more efficient solution will try to pinpoint exactly those parts of the core object needed by a particular view, and creating copies of those parts near the views that need them. We need to address two questions. First, how to find the slice of a core instance that is needed by a view, and second, how to manage inter-copy consistency in a safe and yet efficient way. Roughly speaking, the answer to the first question is the transitive closure of the **required** interface of the corresponding viewpoint, through the call relationship. Depending on the language, this can be a fairly difficult problem, and the difficulty translates into conservative algorithms which tend to identify (many) more calls than necessary [Grove et al., 1997].

As for the second question, a solution that guarantees that no variable/function member is dupli-

1. This assumes the "secure" generation of view code, in which views call the core object's versions of methods, ignoring any other versions supplied by views; see section 3.

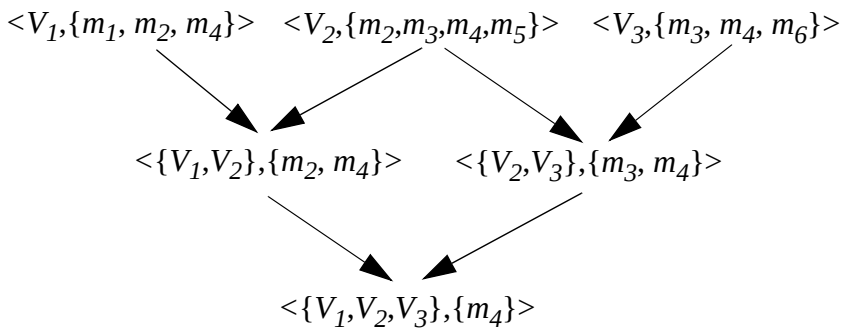


Figure 6-a. Galois lattice of data usage

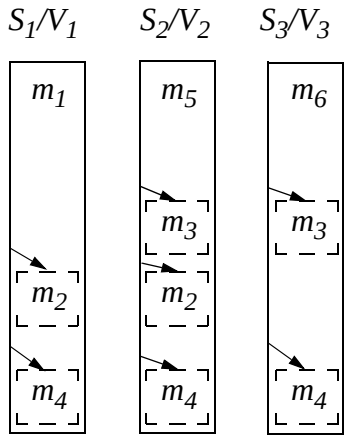


Figure 6-b. Data replication scheme

cated at a site where it is not needed may be constructed by organizing the view induced slices in a *Galois lattice* [Godin et al., 1998] using the "requires" relationship between views and core object members. Consider the following example where we numbered the members of the core class $m_1$through $m_6$ and assume that we have three views $V_1$,$V_2$, and $V_3$ which require access to $\{m_1, m_2, m_4\}$, $\{m_2,m_3,m_4,m_5\}$, and $\{m_3,m_4,m_6\}$, respectively. The resulting Galois lattice is shown in Figure 6-a. Each node consists of a pair of sets, the first being the set of views that share the members included in the second set. The nodes are *maximal* in the sense that for each $<S_V,S_M>$, no view other than those in $S_V$ accesses the members that are in $S_M$, and no member that is not in $S_M$ is accessed by all the views in $S_V$. From this lattice, we can identify the minimal fragment of core class that has to be duplicated across any set of views. Starting from the bottom, we know that $m_4$ is accessed by all three views, and hence that part should be duplicated. Moving up the lattice to node $<\{V_1,V_2\},\{m_2, m_4\}>$, we know that $m_2$ is shared by $V_1$ and $V_2$, and only $V_1$ and $V_2$, and should be duplicated between $V_1$ and $V_2$; we need no longer worry about $m_4$ which was dealt with at the lower levels of the lattice. And so forth. Figure 6-b shows the various slices that reside in each site/with each view.

There are two ways to distribute the slices. One way would follow the scheme in Figure 6-b, and relate the different slices to each other using aggregation: to view 1, the core object looks like an object with data member $m_1$ and two other *objects*-- assuming that objects are the unit of replication-- embodying the data members $m_2$ and $m_4$, respectively. While the impact of the update of any given data member is reduced to a minimum, a view method that updates *several* data members may end up updating far more than *n* duplicates, where *n* is the number of sites. An alternative would duplicate the entire core object in all sites, but would use the scheme of Figure 6-b as a way of targeting those duplicates that must be invalidated; a view whose duplicate of the core object has been invalidated will update the duplicate from one master copy (see e.g. work on *partitioned objects* in [Ben Hassen et al., 1996]).

We could perform a finer analysis based on the access patterns of individual methods of the combined interface of an object. Such an analysis could establish, for a given program, which approach to take, i.e. updating or invalidating selected individual fragments, versus entire objects for all sites. We are in the process of designing experiments that will help us establish whether

such an analysis is generally worthwhile.

## 6. Conclusion

Our work addresses the problem of supporting several functional domains within the same application, by composing at will functional fragments developed by independent third parties. Those same situations that require, or could use, decentralized development of functional domains also require distributed ownership of the functional domain data, and distributed execution of the resulting programs. View programming seems like a perfect fit to the extent that we have resolved most of the issues dealing with the uniqueness of object reference, and the multiple dispatch of methods (i.e. methods supported by several views). There remain a number of issues dealing with optimizing the implementation of distributed view programming which we continue to explore, both theoretically and empirically.

***Acknowledgments:*** *We thank William Harrison and Harold Ossher whose work subject-oriented programming and subject-composition helped us identify (and sometimes solve) some of the issues discussed in this paper. This work was sponsored by Nortel, DEC, IBM, CAE Electronics, Teleglobe, and Machina Sapiens, within the context of the SYNERGIE industry-university initiative (Quebec), by NSERC (Canada), and by YAGO Technologies.*

## References


- [Ben Hassen et al., 1996] Saniya Ben Hassen, Irina Athanasiu, and Henri E. Bal, "A Flexible Operation Execution Model for Shared Distributed Objects," *ACM SIGPLAN Notices*, vol. 31, no. 10, OOPSLA'96 Proceedings, San Jose CA, October 1996, pp. 30-50.
- [Godin et al., 1998] Robert Godin, Hafedh Mili, Guy Mineau, Rokia Missaoui, Amina Arfi, and Thuy-Tien Chau, "Design of Class Hierarchies Based on Concept (Galois) Lattices," *Theory and Practice of Object Systems*, vol 4, No 2, pp. 117-134, 1998.
- [Grove et al., 1997] David Grove, Greg De Fouw, Jeffrey Dean, and Craig Chambers, "Call Graph Construction in Object-Oriented Languages," *ACM SIGPLAN Notices*, vol 32, no 10, OOPSLA'97 Proceedings, Atlanta, GA, October, 1997, pp. 108-109.
- [Harrison & Ossher, 1993] William Harrison and Harold Ossher, "Subject-oriented programming: a critique of pure objects," in *Proceedings of OOPSLA'93*, Washington D.C., Sept. 26-Oct 1, 1993, pp. 411-428.
- [Malenfant, 1995] Jacques Malenfant, "On The Semantic Diversity of Delegation-Based Languages," *Proceedings of OOPSLA'95*, Austin, TX, pp. 215-230.
- [Mili & Dargham, 1997] Hafedh Mili and Joumana Dargham, *View Programming in C++: A co-reference based approach*, rapport technique, département d'informatique, Décembre 1997.
- [Ossher et al., 1995] Harold Ossher, Matthew Kaplan, William Harrison, Alex Katz, and Vincent Kruskal, "Subject-oriented composition rules," in *Proceedings of OOPSLA'95*, Austin, TX, Oct. 15-19, 1995, pp. 235-250.
- [Purao et al., 1998] Sandeep Purao, Hemant Jain, and Derek Nazareth, "Effective Distribution of Object-Oriented Applications," *Communications of the ACM*, vol. 41, no. 8, August 1998, pp. 100-108.
- [Shilling & Sweeny, 1989] John Shilling and Peter Sweeny, "Three Steps to Views," *Proceedings of OOPSLA'89*, New Orleans, LA, pp. 353-361, 1989.
- [Ullman 82] Jeffrey D. Ullman, *Principles of Database Systems*, C S Press, 2nd ed., 1982.
- [Van Hilst & Notkin, 1996] Michael Van Hilst and David Notkin, "Using Role Components to Implement Collaboration-Based Designs," in *Proceedings of OOPSLA'96*, San-Jose, CA, 6-10 October, 1996, pp. 359-369.

# Protocol-Based Runtime Monitoring of Dynamic Distributed Systems
## *Work in Progress*

Andreas Grünbacher and Mehdi Jazayeri
Information Systems Institute, Distributed Systems Group
Technical University of Vienna
{a.gruenbacher, m.jazayeri} @ infosys.tuwien.ac.at

March 24, 1999

**Abstract**

Many systems are built today from components that communicate using standard protocols. Monitoring the communication among these components helps in analyzing and debugging the behavior of such systems. We are trying this approach for a category of Web-based systems.

**Keywords**

Monitoring, distributed debugging, standard protocols.

## 1 Introduction

Reuse has been a popular keyword throughout the software engineering community for many years; yet many enabling technologies such as components, component frameworks and middleware have only recently found widespread acceptance. Another rapidly emerging area of computing has been the World Wide Web. Taken together, these technologies enable an entirely new approach to building software systems. At the core of many such systems is the World Wide Web with its simple client/server architecture.

The combination of Web browsers as clients and server components such as HTTP and database servers yields powerful distributed systems. We call them dynamic distributed systems (DDS).

Due to the complexity that is caused by the distributed nature and heterogeneity of such systems, the development and maintenance of dynamic distributed systems is extremely demanding. This shows up especially when an application is running. Usually, it is anywhere from hard to impossible for the developers to just observe what events occur in the system. Our approach to facilitating DDS development and maintenance is to monitor systems at runtime. This includes the tasks of data collection, analysis and visualization. Usually, the components of a distributed system are modified to collect data for debugging. In a distributed system, a lot of information is already contained in the communication protocols between components. We believe that by analyzing these protocols, enough information about a system can be revealed for debugging at a reasonably high level of abstraction. For protocol-based runtime monitoring (PBRM), the approach described in this paper, we require standard protocols between (at least some of the) components. PBRM is intended to complement, not replace, conventional debugging.

The data obtained from tracing the communication between components will consist of a large set of events, even for small systems. A way to reduce the amount of information presented to the developers is needed, unless they explicitly request the details. Several abstraction techniques have been described in the literature [Basten 93] [Basten 94] [Summers 92]. We are considering to adapt a scheme for hierarchical event abstraction, similar to the one used for the Poet visualization tool[1] [Poet 97].

**Structure of this document.** Section 2 gives an example of a dynamic distributed system that was recently built in our group, discusses the problems encountered, and suggests how runtime monitoring can help us improve this situation. In Section 3 we discuss issues of protocol-based runtime monitoring. Section 4 outlines the architecture of our prototype monitoring system. In Section 5 we describe the current status of this project. Finally, we summarize our key points in Section 6.

## 2 Dynamic Distributed Systems

Dynamic distributed systems consist of components that communicate using different mechanisms in order to achieve the overall goal of the application. These include rather low-level protocols such as HTTP, middleware such as CORBA, COM, RMI, but also more task-specific protocols such as ODBC and JDBC. Depending on the task at hand, communication mechanisms at different abstraction levels are used. In general, higher level protocols support the extraction of more abstract information from the protocol trace.[2]

A good example system is 3DSoftVis[3] [Jazayeri 98], a dynamic distributed system that was recently built by our group. 3DSoftVis deals with visualizing software release histories. It displays software release data as three-dimensional models. The data are stored in a database. The user interface consists of several Web pages, Java dialogs, and a VRML plug-in that handles the actual displaying of the three-dimensional model. All the components on the client machine run inside a Web browser. Figure 1 shows the user interface of this application.

The logical component structure of our example is shown in Figure 2. 3DSoftVis consists of a Web server, a Web browser, and a database server. The Web browser uses a VRML plug-in. The Web server and Web browser talk HTTP to each other. The Web browser and the database server talk JDBC. The VRML plug-in and the Web browser talk EAI.[4]

Figure 3 shows the basic pattern of interaction with 3DSoftVis, as seen by the user. From the Workspace window, the user can display a number of Visualizer windows to view different three-dimensional models.

At the level of detail shown in Figures 2 and 3, the system looks simple. However, describing in full detail the complicated interactions between the Web server, the Web browser, the Java components and VRML is quite challenging, and involves hundreds if not thousands of events. Tracing errors in the application is difficult, because the interactions between the components are not visible.

3DSoftVis is based on coarse-grain reuse. This allowed us to build a complex visualization system in a short time. Had we tried to develop a system like that from scratch, the 3D rendering engine alone would have exceeded our time frame by far.

[1] See `http://www.shoshin.uwaterloo.ca/poet/index.html`.

[2] The task of monitoring a distributed system becomes easier in a restricted framework. Version 2.2 of Corba, for example, allows so-called interceptors to be inserted into the invocation path of CORBA methods. These can conveniently be used to record all method invocations and parameters in a CORBA-based system. One product that is constructed along similar lines is Object/Observer by Black&White Software, `http://www.blackwhite.com`. Our experience shows that most systems don't exclusively use a framework like CORBA, however. Therefore, we are looking at how to support a broader range of systems.

[3] An online demo is available at `http://www.infosys.tuwien.ac.at/~riva/vis/`.

[4] The External Authority Interface (EAI) is an interface between Java and VRML.

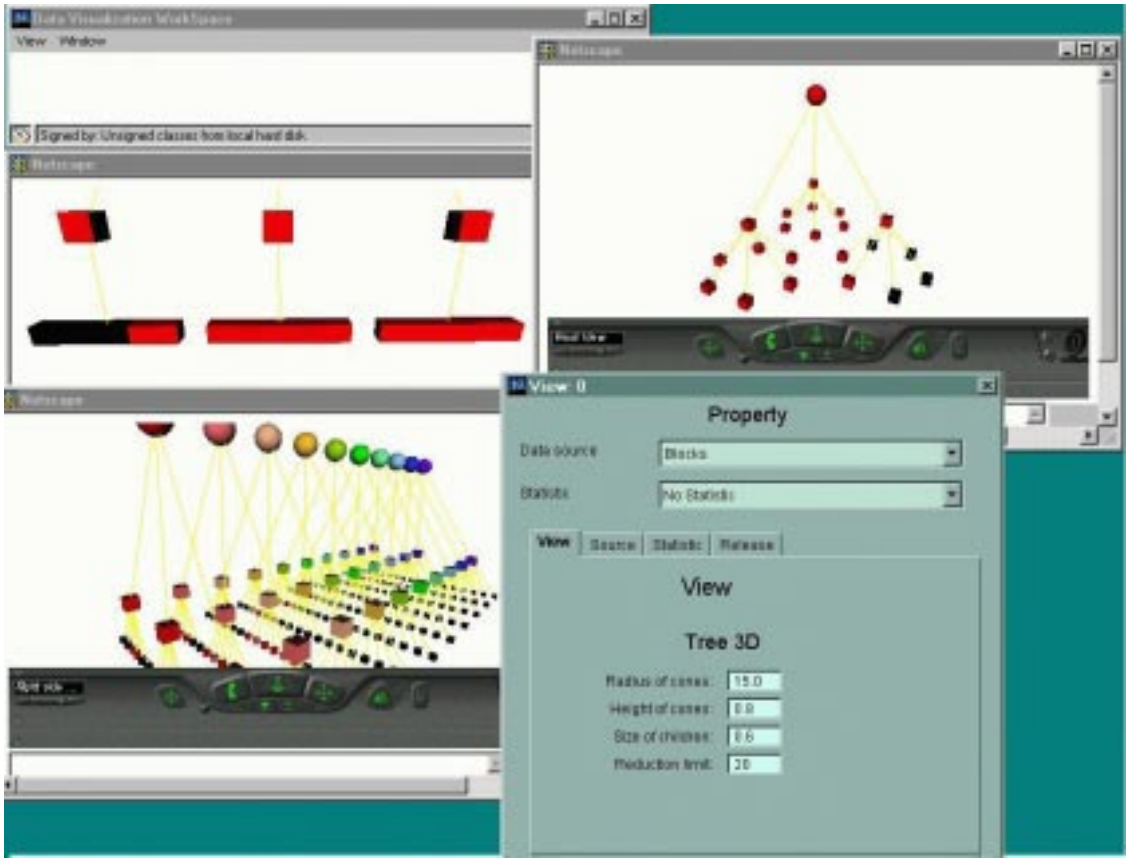


Figure 1: 3DSoftVis: User interface.

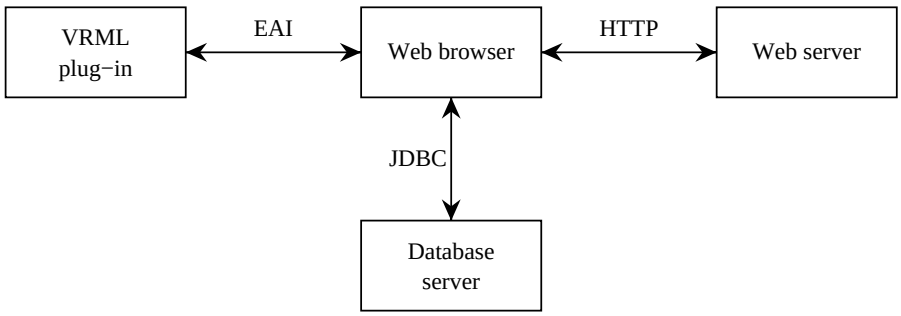


Figure 2: 3DSoftVis: Physical structure.

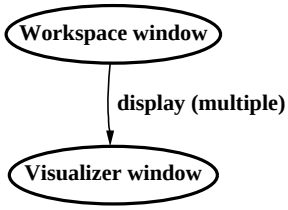


Figure 3: 3DSoftVis: Basic interaction pattern.

PBRM is aimed to help the developers of systems such as 3DSoftVis. Had we been able to monitor our example system, bug tracking would have been a lot easier.

A future monitoring system would take a high-level architectural model of the target system as input and automatically match all actions in the system against that model. This would have the added benefit of always having an up-to-date high-level model of the system. Such an approach is beyond our current aims, however.

# 3 Protocol-Based Runtime Monitoring

Our approach of PBRM is based on protocol event traces. It does not require components to be modified. Most approaches to distributed system debugging discussed in the literature limit the type of systems by either enforcing a common application framework, or by requiring the systems to be augmented by instrumentation code, to make sufficient information available to the debugger [Widmer 98]. In the case of dynamic distributed systems, sufficient information about a system is available to the debugger by examining only the communication between components. In particular, middleware standards such as CORBA, COM, RMI, task-specific protocols such as ODBC and JDBC, but also standard protocols such as HTTP enable us to obtain enough information for in-depth understanding. Furthermore, since dynamic distributed systems often use existing components, modifying these components for the purpose of debugging is not an option.

Unlike traditional debugging environments, we do not consider other types of events such as starting and termination of components. We also do not consider the concurrency model of components.

The user of a monitoring system is generally not only interested in which events occur, but also in the dependencies among them. The precedence relation between events has been formulated by Lamport [Lamport 78]. Using this partial order relation, the events that happen during the execution of a dynamic distributed system can be presented to the user in a meaningful way.

In distributed systems, the precedence relation between events cannot be easily computed from protocol event traces, however. The local clocks of the machines participating in the computation cannot be used as a reference for time-stamping events. This would require perfectly synchronized clocks, a property which is not achievable in practice. A total order between events also cannot be introduced. This would require a central component that sequences events. Central components don't scale with the size of distributed systems, and therefore limit overall performance. In addition, such a centralized component would probably introduce artificial dependencies between events [Fidge 96]. A vector clock algorithm [Fidge 91] is another popular approach to detect causal dependencies between events. While this is a correct and general approach, it would require changing the components as well as the protocols they use. Usually, protocols don't support adding meta-information like timestamp vectors to messages.

Our current approach is to explicitly record those dependencies between events that can be detected at the protocol-understanding level, and to accept that some dependencies are not detected. The advantage of this approach is that it can be implemented efficiently, without the need to change either components or protocols.

Our goal in this project is to investigate the usefulness of protocol-based runtime monitoring. In particular, we would like to answer the following questions:

- Which information is lost in comparison to other debugging approaches?
- Which kind of information can we still obtain? Which information can we reconstruct or guess?
- Which analyses are possible based on the data collected?

- Which problems do low-level protocols such as HTTP pose, and how can these be overcome?

The following three sections describe requirements for PBRM. We break up the monitoring process into the three phases data collection, analysis and visualization.

## 3.1 Data Collection

The monitoring system traces the communication between components.

**Events.** The monitoring system is based on protocol event traces. At the protocol level, a lot of information is available. This includes the type of event (e.g., *Request, Reply*), and associated data. Since it is not known a priori what data the user is interested in, the monitoring system shall allow the user to record arbitrary data in addition to the data required by the monitoring system itself.

**Causal dependencies.** Causal dependencies between events need to be recorded during the data collection phase. Since a vector clock algorithm cannot be used in our scenario, we need to model dependencies explicitly. A unique identifier is generated for each event. Dependent events then refer to the events they depend on by these identifiers.

**Performance.** The amount of data produced in a debug session can get quite large. The data collection processes should therefore provide the option to record only the minimum amount of data necessary for monitoring itself.

**Probe effect.** The modification of system behavior by monitoring is called probe effect [Fidge 96]. Monitoring can hide existing errors, as well as introduce new ones. With our approach, it is impossible to completely avoid influencing the target system.[5] The probe effect can be minimized, however, by keeping the overhead of monitoring small. The data collection processes also need to be implemented carefully. They are not allowed to synchronize on the same resources as the target system, since this would introduce spurious dependencies.

## 3.2 Analysis

From the data collected during the data collection phase, several properties of the monitoring system such as performance, latency, and causal dependencies between events can be analyzed.

Our current focus is on causal dependencies. Additional data in the event stream such as session identifiers and transaction numbers can allow us to reconstruct additional causal dependencies. It might be interresting to note that the causal dependencies we obtain from protocol event traces are *real* causal dependencies, as opposed to *potential* causal dependencies, which we would get from a vector clock algorighm.

Dependency analysis can be performed at two levels. Intra-protocol analysis can be used to group events within a protocol. This is cheaper than inter-protocol analysis, which groups arbitrary events.

Patterns of causally related events are used to hierarchically structure primitive events into abstract events. The notion of convex abstract event sets as introduced in [Black et al. 93] is one promising approach we are considering to implement. We also need an algorithm that detects these event patterns in the event stream.

## 3.3 Visualization

Process-time diagrams (also known as time-space diagrams) are commonly used to visualize distributed systems. The Poet visualization tool [Poet 97] extends process-time diagrams for primitive events to diagrams for primitive as well as abstract events with simple,

[5] Different form physics, influence-free monitoring is possible in software (see [Diaz 94] for an example).

intuitive semantics. We are also considering a textual display with the possibility to expand and collapse abstract events as required, similar to a directory browser.

# 4 Implementation of a PBRM System

Our implementation is based on the introduction of a *proxy* for each communication channel that is to be monitored. The proxies forward messages back and forth between the communicating partners. In addition to that, they identify those messages that lead to significant state changes at the communicating partners (i.e., events). The proxies forward event data to a *Hub*. The Hub stores these data, and acts as a common access point for *Monitors*, which analyze and visualize them.

Figure 4 shows the components of 3DSoftVis, omitting the VRML part. Figure 5 shows the same system, now attached to the monitoring environment.

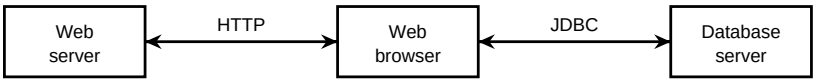


Figure 4: Architecture of a dynamic distributed system.

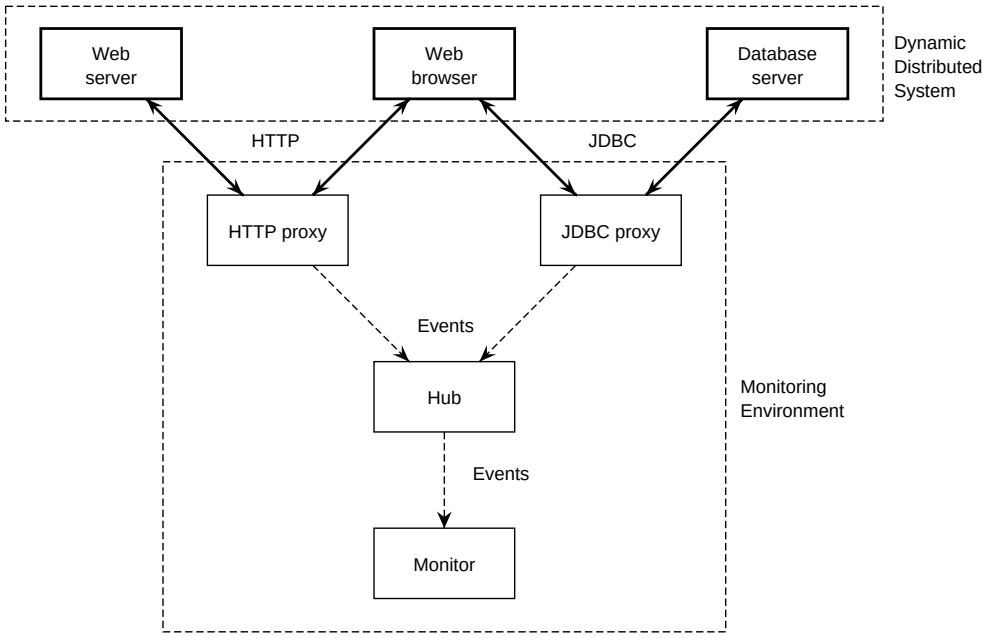


Figure 5: Architecture of a dynamic distributed system being monitored.

# 5 Current status

We have tried the proxy approach for HTTP. While implementating an HTTP proxy took longer than expected, largely due to complexities in the HTTP/1.1 protocol specification, the results are promising. Our next step is to add JDBC. With these two protocols, we hope to gain more experience with PBRM.

We have identified the following “hard” problems:

- An interaction between an HTTP client and server usually spans several Web pages. Such sessions seem to be difficult to identify. The obvious choice to use the client machine’s IP address as a unique identifier fails for machines behind a firewall. Even the process identifier of the browser, were it available to the monitoring system, would not be enough, since one browser may display multiple windows, each containing a seperate session. Our initial hope that Cookies would resolve this problem turned out to be false as well.
- One of the goals for proxies is transparency: The communicating components should not be aware of the presence of the proxy, at least as far as the communication protocol is concerned. Unfortunately, this goal cannot be completely met. Especially, errors can result in modified behavior. Consider, for example, a HTTP request to a non-existent host. In the original system, no connection will ever be established. In the monitored system, the client connects to the proxy. Only then does the proxy find out that the destination host doesn’t exist, and sends an error reply back to the client.

# 6 Summary

We have described protocol-based runtime monitoring as a new approach for debugging dynamic distributed systems. We are currently implementing a tool to assess the usefulness of PBRM. We have described some of our design decisions and current open problems in our project.

# Acknowledgments

This work was supported by the ARES ESPRIT Project 20477. We would like to thank Claudio Riva for implementing 3DSoftVis and explaining its inner workings to us.

# References


[Basten 93] T. Basten. *Hierarchical Event-Based Behavioral Abstraction in Interactive Distributed Debugging: A Theoretical Approach.* Eindhoven University of Technology, August 1993.

[Basten 94] T. Basten, T. Kunz, J. P. Black, M. H. Coffin, D. J. Taylor. *Time and the Order of Abstract Events in Distributed Computations.* Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands. Computer Science Note number 94/06. February 1994.

[Black et al. 93] J. P. Black, M. H. Coffin, D. J. Taylor, T. Kunz, A. A. Basten: *Linking Specification, Abstraction, and Debugging.* CCNG Technical ReportE-232, Computer Communications and Networks Group, University of Waterloo, November 1993.

[Diaz 94] Michael Diaz, Guy Juanole, Jean-Pierre Courtait: *Observer—A Concept for Formal On-Line Validation of Distributed Systems.* IEEE Transactions on Software Engineering, December 1994, Vol 20, No. 12.

[Fidge 91] C. J. Fidge: *Logical time in distributed computing systems.* IEEE Computer, August 1991, Vol. 24, No. 8, pp. 28–33.

[Fidge 96] Collin Fidge: *Fundamentals of Distributed System Observation.* IEEE Software, November 1996, Vol. 13, No. 6, pp. 77–83.

[Jazayeri 98] Mehdi Jazayeri, Claudio Riva: *Developing Native World Wide Web Applications.* Technical Report, Distributed Systems Group, Vienna University of Technology.

[Lamport 78] Leslie Lamport: *Time, clocks, and the ordering of events in a distributed system.* Communications of the ACM, July 1978 / Vol. 21, No. 7, pp. 558–565.

[Poet 97] T. Kunz, J. P. Black, D. J. Taylor, T. Basten: *Poet: Target-System-Independent Visualisations of Complex Distributed-Application Executions.* In Proceedings of the 30th Hawaii International Conference on System and Science. Volume 1, Maui, Hawaii, USA, pages 452–461, January 1997.

[Summers 92] J. A. Summers: *Precendence-Preserving Abstraction for Distributed Debugging.* Master's Thesis, Dept. of Computer Science, University of Waterloo, Ontario, 1992.

[Widmer 98] Bernfried Widmer: *Management of CORBA-Based Distributed Object Systems.* Master's Thesis, Information Systems Institute, Technical University of Vienna, Vienna, 1998.

# THE IMPLEMENTATION AND EVALUATION OF THE USE OF CORBA IN AN ENGINEERING DESIGN APPLICATION

Susan D. Urban, Ling Fu
Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406
s.urban@asu.edu

Jami J. Shah
Department of Mechanical and Aerospace Engineering
Arizona State University
Tempe, AZ 85287
Jami.shah@asu.edu

Ed Harter, Tom Bluhm, Brett Hartman
Boeing Defense and Space Group
Engineering Computing
P. O. Box 3999, MS 4A-19
Seattle, WA 98124-2499

## ABSTRACT

Many computer applications today require some form of distributed computing to allow different software components to communicate. Several different commercial products now exist based on the Common Object Request Broker Architecture (CORBA) of the Object Management Group. The use of such tools, however, often requires the modification of existing systems, rather than the development of new applications. The objective of this research has been to integrate the use of a CORBA tool into an existing engineering design application for the purpose of 1) evaluating the amount of re-engineering that is involved to effectively integrate distributed object computing into an existing appliations, and 2) evaluating the use and performance of distributed object computing in an engineering domain, which often requires the transfer of large amounts of information. The results of this work demonstrate that CORBA technology can be easily integrated into existing applications. The ease of the integration as well as the efficiency of the resulting system, however, depends on the degree of modification that developers are willing to consider in the re-engineering process. The most transparent approach to the use of CORBA requires less modification and generally produces less efficient performance. The less transparent approach to the use of CORBA can potentially require significant system modification but produce greater performance gains. This work outlines issues that must be considered for the partitioning of functionality between the client and the server, development of an IDL interface, development of client and servers sider wrappers, and support for concurrent, multi-user access. In addition, this work also provides performance and implementation comparisons of different techniques for the use of wrappers and for the transfer of large data files between the client and the server. Performance comparisons for the incorporation of concurrent access are also presented.

**KEYWORDS:** distributed object computing, engineering design, software re-engineering, efficient file transfer, client/server performance analysis.

## 1. INTRODUCTION

Many computer applications today require some form of distributed computing to allow different software components to communicate. Engineering design applications provide such an example, where product design efforts of large manufacturers require that designers must cooperate very closely. These designers, however, may be physically distributed at different locations, using varied types of computers and network operating systems, as well as different software packages. Manufacturers need to provide environments that allow designers to work together in efficient and flexible ways at distant locations.

In order to build such sophisticated distributed applications, the concept of distributed object-oriented computing has evolved. Distributed objects are software components that can be located at physically different sites within a network of computers. These objects contain properties and methods, which generally provide a set of predefined services to client applications. The use of distributed object computing concepts involves the definition of how and when distributed objects communicate with one another.

The Object Management Group (OMG) was formed as a means of standardizing the way that distributed objects communicate [1]. In particular, OMG has defined what is known as an Object Request Broker (ORB) [2]. An ORB serves as the middleware between a server and a client. An object sends a request to an ORB, which then locates the appropriate server object to receive the request. The server executes the request and then gives the response back to the ORB. The ORB finally forwards the response back to the original client object. The concept of an ORB has been extended into the Common Object Request Broker Architecture (CORBA) [1]. CORBA is a specification for an application-level communication infrastructure. CORBA permits objects written in different programming languages to communicate. Furthermore, the communication between a client and a server in CORBA occurs in a transparent manner.

Several different commercial products now exist based on CORBA, such as Orbix from IONA [3]. The use of such tools, however, often requires the modification of existing systems, rather than the development of new applications. The objective of this research has been to integrate the use of a tool such as Orbix into an engineering design application for the purpose of:

1) evaluating the amount of re-engineering that is involved to effectively integrate distributed object computing into an existing application, and

2) evaluating the use and performance of distributed object computing in an engineering domain, which often requires the transfer of large amounts of information.

This research has been performed in the context of the Integrated Product Data Environment (IPDE) [4], which is part of the Rapid Design Exploration and Optimization (RaDEO) research project at Arizona State University. The project is supported by the Defense Advanced Research Projects Agency

(DARPA), with ASU serving as a subcontractor to the Boeing Defense and Space Group. The purpose of IPDE is to provide a data storage facility that supports the exchange of data between different computer-aided design and analysis (CAD/CAA) tools [5]. The IPDE provides an interface to the system, known as the Domain Access Interface (DAI). The DAI communicates the transfer of product data to the Shared Data Manager (SDM), which controls access to the integrated product database (IPDB) and establishes relationships between the different types of data stored in the IPDB. The IPDB was developed using object-relational database technology.

In the first version of the IPDE, DAI's were compiled together with the IPDE as a part of one large monolithic application. In practice, however, DAI's must be distributed to support multi-user access from different locations, with a need to transfer large files (sometimes as large as 100M bytes) between the DAI and the IPDB. This research has transformed the IPDE into a distributed application, where DAI's communicate with the IPDE using CORBA technology to achieve application integration. The specific tasks of this research were to [6]:

1) redesign the current DAI/IPDE interaction into a client/server architecture, and evaluate the redesign effort,

2) modify the IPDE to support multiple DAI access to the IPDE with the client/server architecture,

3) implement and evaluate different techniques for the transfer of large files between DAI's and the IPDE, and

4) evaluate the overall effectiveness and efficiency of the use of CORBA technology in a multi-user engineering application, such as the IPDE.

The results of this work demonstrate that CORBA technology can be easily integrated into existing applications. The speed of the integration as well as the efficiency of the resulting system, however, depend on the degree of modification that developers are willing to consider in the reengineering process. The most transparent approach to the use of CORBA requires less modification and generally less efficient performance. The less transparent approach to the use of CORBA can potentially require significant system modification but greater performance gains. This work outlines issues that must be considered for the partitioning of functionality between the client and the server, development of an IDL interface, development of client and server side wrappers, and support of concurrent, multi-user access. In addition, this work also provides performance and implementation comparison of different techniques for the use of wrappers, and for the transfer of large data files between the client and the server. Implementation consideration and performance comparisons for the incorporation of concurrent, multi-user access are also presented.

The remainder of this paper is organized as follows. Related work on distributed computing with CORBA is presented in Section 2. Section 3 then presents a detailed description of the IPDE architecture,

while Section 4 addresses the redesign of the IPDE system for client/server distribution with CORBA. This section specifically addresses the different design issues that were investigated as part of the redesign process. Section 5 provides an evaluation of the effectiveness and efficiency of using CORBA with the IPDE environment. Lastly, Section 6 provides a summary of the research results and suggestions for future research.

## 2. RELATED WORK

CORBA specifications set by the Object Management Group have the express purpose of allowing distributed object computing to take place transparently in a multi-vendor, multi-platform network environment [1]. The main focus of CORBA is the concept of objects. For software, an object refers to a software component. For an enterprise, an object refers to real-world entities. One of the characteristics of an object is that an object can send messages to other objects with requests for service. Objects send messages back to the requesting object with their response.

The CORBA paradigm follows the distributed client/server methodology, which is based on message passing found in most UNIX systems [7, 8, 9]. Objects, by themselves, are servers responding to message requests. We distinguish between objects that live on the server-side or on the client-side. Server-side objects offer services and resources. Client-side objects request services and resources. Client-side and server-side objects inter-operate as in all object-oriented systems, with the caveat that the requester and provider may live on separate machines within the network.

Objects in a distributed environment generally adhere to concepts found in object-oriented programming. As a result, distributed objects typically support [10, 11]:

1) Encapsulation— the attributes (state) of an object are only accessible through methods of the object.
2) Inheritance— a new (specialized) class can be defined by reusing a previous (generalized) class while adding additional behavior or attributes.
3) Polymorphism— one message can cause different behaviors according to the actual type of the object.

In order for objects to plug and play together, clients have to know exactly what they get from every object for a service. In CORBA, the services that an object can provide are expressed as an interface using the Interface Definition Language (IDL) [1, 2, 12]. The interface defines the format of messages that can be called to request services. The communicating infrastructure can translate data formats when it is necessary to provide transparent connections between the sender and receiver. Each object needs a unique handle that a client can pass to the infrastructure to route a message to it. This handle, however, is not an address. When the object moves from one location to another, the handle is the same. CORBA therefore provides the basic model of a networked computing environment. The nodes in the network are objects having their own interfaces, where each object is identified by a unique handle

[2]. A message can be sent from one object to another object by formatted messages into an interface known to the system.

The CORBA standard has been implemented as numerous products from different companies, including Orbix by IONA [3], HP Distributed Smalltalk [13], SUN Network Enabled Objects (NEO) [14]. Researchers from industry and universities as well as users are developing distributed system applications with CORBA. Two engineering projects with the use of CORBA are presented below.

The Version and Integration Control System (VICS) [15] is a project developed by the Applied Research Laboratory at the University of Texas (ARL:UT). VICS is a multi-user distributed application that has been built using the World Wide Web (WWW) Common Gateway Interface (CGI). In the CGI approach, the graphical user interface (GUI) layout control of hyper-text marking language (HTML) forms is very simple. The server is invoked only when the client asks for an HTML form or a CGI program. It treats the requirements as individual ones, but does not treat the requirements from the same client process as one sequential group. In this way, the server then can't keep track of the status of each client.

The VICS project indicated a need for a more typical client-server approach. They chose to use CORBA to avoid the direct use of sockets. VICS also uses JAVA to keep the WWW's biggest advantage: automatic download of the client user interface. After re-implementing the VICS project using CORBA, the group at ARL:UT also compared the new approach against CGI in terms of how well each approach supports the development, maintenance, and execution of multi-user distributed applications. First of all, while the GUI and remote operation definitions are coupled in HTML forms in CGI, the Java/CORBA approach defines the GUI and remote operations separately. The client in the Java/CORBA approach also has more flexibility in managing the UI and invoking remote operations while CGI's client could only display the GUI and invoke remote operations. With respect to responsiveness, a single GUI update in the CGI approach requires downloading a new HTML file. The client is also single-threaded in the CGI approach, while in the Java/CORBA approach, the GUI updates are handled by the client applet. Remote invocations may be handled by an already executing server process, and the client can exploit multi-threading. This project concluded that CORBA provided several advantages. First of all, client and server programs can exploit both multi-threading and the continuous execution state. CORBA also reduced the complexity of the server-side software. The interaction between the client and the server also occurred through the ORB which carried structure types and classes in the remote method argument, while the CGI could only carried string types.

The project TAMGRAM [16] is in the distributed telecommunications area. This application performs a multimedia communication service, which has been designed to essentially enable the exchanges of audio and video data between a group of participants. CORBA is adapted to provide the

interoperability among the distributed telecommunication applications on a variety of hardware platforms including SUN's and Windows personal computers. The developers were also able to take advantage of object-oriented design, since the object is able to represent continuous flows of data of which the actual data structure e.g., information in audio or video types, can be of no interest. An object-oriented construction of objects also provides the reusability of designed computational objects.

In addition to these two examples, many manufacturers and companies are also using CORBA to provide interoperability as well as adaptability [17] among distributed applications on different hardware platforms. For example, Boeing has adopted CORBA to integrate applications across different operating systems and different hardware. The project involves the use of 70 sites with 45,000 users [18]. Oracle's Network Computing Architecture (NCA) is using CORBA as its backbone, with other Oracle primary tools also using Java/CORBA [19]. Other examples of projects using CORBA include the Legacy System Integration with Object-Oriented Methods (LIOM) Project, which is a metadata modeling project for healthcare applications in a federated database system [20], and the Dresdner Bank Frankfurt project which is a CORBA-based data transfer project for financial risk management [21].

This research has used Orbix [3], which is a relatively complete implementation of CORBA. With Orbix, programmers can develop distributed applications using object-oriented client-server technology. Programming can also use object technology to compose new applications from existing components and subsystems. Orbix provides a C++ language binding and also a JAVA language binding for CORBA and is supported on most platforms [3].

Orbix provides most of the functionality presented in the CORBA specification:

1) Orbix provides an Interface Definition Language (IDL) to collect client and server code.

2) Orbix supports a Dynamic Invocation Interface (DII) to compose requests and do dynamic type checking without prior knowledge of operation signatures.

3) Orbix provides a Basic Object Adaptor that provides the interface to Orbix for servers, and controls the mapping of the objects to the system processes.

4) Programmers have control over 'proxies'/'surrogates'. Proxies are local representatives for remote objects.

5) Orbix provides co-location of client and server code.

6) Orbix provides process level filters. Programmers can develop their own filter code for incoming and outgoing messages for both clients and servers, which facilitates integration of thread packages.

7) Orbix also supports most of the existing platforms and OS's: Solaris, HP-UX, AIX, DEC, SGI, NT, W95, OS/2, Mac, VMS, MVS.

This research has adopted the C++ language binding to remain compatible with the current implementation environment. The DAI's in the IPDE environment were initially implemented using

UIM/X [22, 23], a visual X-windows user interface tool. UIM/X is specifically designed to interface with Orbix. Since the DAI's and the current SDM interface are implemented in C++, the Orbix implementation in this project has also used C++.

## 3. OVERVIEW OF THE RADEO PROJECT

Since this project is based on the use of the IPDE of the RaDeo project, this section provides an overview of the IPDE. Section 4 will describe how we used the IPDE as part of this research.

### 3.1 Architecture of the IPDE

As described in the introduction, the purpose of the IPDE is to provide a database environment to support the exchange of large data files between CAD tools that are used in the engineering design and analysis process. Figure 1 provides an architectural overview of the IPDE. As indicated in Figure 1, the IPDE consists of three main components: the integrated product database (IPDB), the shared data manager (SDM), and a set of domain access interfaces (DAI's).

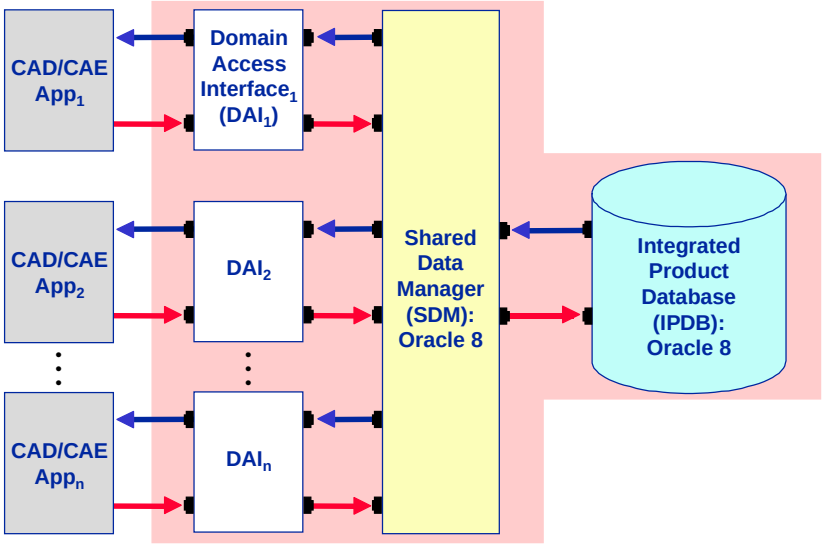


Figure 1: Architecture of the IPDE Project

The IPDB is an object-relational database [24] for storing engineering design data. The SDM, which also adopts object-relational technology, is a data manager that controls access to the IPDB and keeps track of relationships between product/process data as well as configuration information [25]. The

SDM is acting as the middle layer between the IPDB and DAI's. DAI's serve as a windows-based interface for communicating with the SDM about data transfer into and out of the IPDB.

### 3.2 Units of Functionality

The IPDB is the primary data component of the environment, storing the data files that must be exchanged between different design and analysis tools as well as the relationships that exist between such files. Versioning and configuration information is also an important data component of the IPDB. In particular, the data of the IPDB is based on the concept of application protocols (AP's) as defined within the Standard for the Exchange of Product Model Data (STEP), which is a comprehensive ISO standard that describes how to represent and exchange digital product information [26]. For example, some of the APs used in this project include AP 203 (Configuration Controlled 3D Designs of Mechanical Parts and Assemblies), AP 209 (Design Through Analysis of Composite and Metallic Structures), and AP 214 (Core Data for Automotive Mechanical Design Process). An AP provides a standard schema definition in EXPRESS for the data that is used within a particular design and/or analysis domain. EXPRESS, which is part of STEP, is an International Standard that has been used to describe information in designing, building, and maintaining product data [27, 28]. EXPRESS contains a well-defined textual language and a graphical representation language call EXPRESS-G.

A unit of functionality (UoF) is a logical subcomponent (i.e., subschema) of an AP, representing a sharable unit of data that can also be used to establish relationships to other sharable units of data. When a user checks a file into the IPDB, the user identifies the specific UoFs that must be stored. The SDM is responsible for extracting these subcomponents and storing them in the IPDB together with their relationships. Other users can then check out these UoFs and use them to generate additional UoF's and relationships related to the product design. The IPDB therefore maintains a history of the design files generated during the design process, together with information about the dependencies that exist between files.

The actual data within the IPDB and the SDM is stored at two different levels of detail. The metadata and control data of the SDM and some components of the IPDB, such as those that maintain data about specific products, relationships between UoFs, versions, and configurations, are stored as fine-grained objects. Other data, such as the large files that contain UoFs, are stored as binary large objects (BLOB) since there is no need to access the internal details of such files. In either case, however, the data that must be stored in the SDM and the IPDB is conceptually described using EXPRESS.

### 3.3 Functionality of the SDM

The SDM contains the security information for controlling access to the SDM and the IPDB. A fundamental task of the SDM is the management of bookmarks. A bookmark establishes a series of references to objects in the IPDB that are needed to support the process of checking data into and out of

the IPDB. The SDM can access the IPDB directly in this way. The SDM also controls other aspects of access to the IPDB, such as user management and session management.

A graphical view of a bookmark is shown in Figure 2. The left-hand side of Figure 2 is a simplified view of a bookmark described using EXPRESS. The rectangles on the left side of Figure 2 are EXPRESS entities. Each line with a small circle defines the "has-a" relationship. For example, the Product_Definition_Formation entity is an attribute of the Product entity. The right-hand part of Figure 2 displays specific instances of a bookmark. As described in Figure 2, a bookmark contains several references to IPDB objects. The top of a bookmark points to a specific product. The middle portion of a bookmark points to EXPRESS entities that define specific product versions (i.e., *Product Definition Formation*, *Product Definition*, *Property Definition*). A bookmark finally points to specific UoF's (i.e., large files that contain the actual design version). A bookmark itself doesn't store real values. Instead, a bookmark points to the paths through a bookmark schema that define specific configurations. These different paths help the users to check-in UoF's related to a specific product design, or to check-out the UoF's identified in the bookmark. All reading and writing operations to the IPDB must be done through bookmarks.

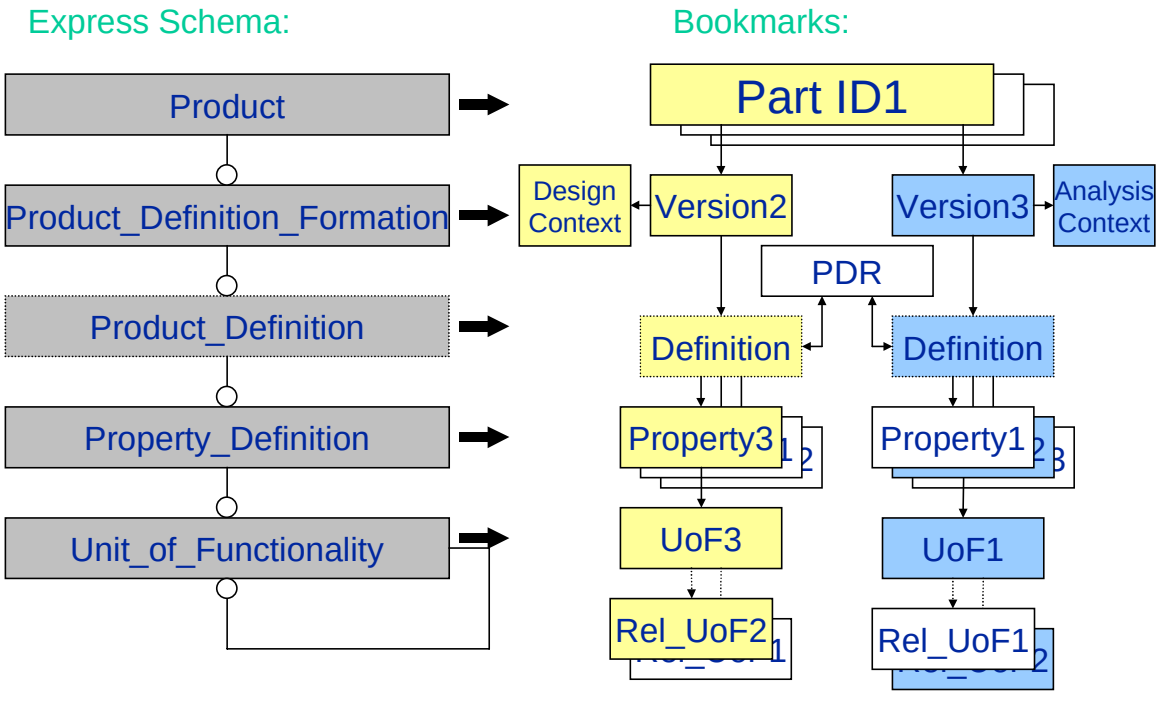


Figure 2: EXPRESS Schema and Bookmark Instances

| SDM_Operations |
| --- |
| |
| user_login (string:user_id, string:password): Result Integer<br>*// For user login to the SDM*<br><br>user_logout ( ): Result Integer<br>*// For user logout from the SDM*<br><br>get_domains (stringArr:domains): Result Integer<br>*// Fixed number of domain name exist in the SDM, which contains different groups of UoF's*<br><br>set_configuration (string:domain_name, string:parse_log, string:checkin_log,<br>string:checkout_log, string:in_dir, string:out_dir) : Result Integer<br>*// Set the configuration information including the domain names, the log files, and input file*<br>*// location and output file location*<br><br>view_configuration (string:domain_name, string:parse_log, string:checkin_log,<br>string:checkout_log, string:in_dir, string:out_dir) : Result Integer<br>*// Give a complete view of the configuration information*<br><br>clear_log_file (short:parse_log_path, short:check_in_log_path, short:check_out_log_path) : Result Integer<br>*// Clears up the log files that are created by parse/check_in/check_out process*<br><br>ask_change_passwd (string:passwd) : Result Integer<br>*// Invokes the password changing process. Asks the user to input the old one, and validate it*<br><br>change_passwd (string:passwd, string:repeat_passwd) : Result Integer<br>*// Repeat inputting the password, if they are the same, the password is changed to the new one*<br><br>display_cur_bookmark (string:bookmark_name, string:product_id, string:product_name,<br>string:product_version, string:product_definition_id, string:product_definition_description,<br>string:property_name, UofList:uoflist) : Result Integer<br>*// Displays the context in the current book, e.g., bookmark name*<br><br>add_bookmark (string:bm_name) : Result Integer<br>*// A bookmark is added in the SDM. The process of adding bookmark is from this step*<br>*// and can be terminated at any level* |

Table 1: The SDM API

### 3.4 Functionality of DAI's

A DAI is a graphical interface for interacting with the SDM. DAI's provides the means for users to load design files from different CAD tools into the IPDE. A DAI communicates with the SDM through an API that is composed of the SDM database operations as well as operations from ST-Developer [29]. ST-Developer is an additional engineering software package that provides tools for working with EXPRESS schemas. Table 1 presents the API of the SDM [6]. Table 1 also explains the functionality of each operation.

| SDM_Operations |
|---|
| |
| display_products (stringArr:part_number, stringArr:part_desc) : Result Integer<br>*// Displays the products' contexts (top level of the bookmark in Figure 2) in the IPDB.*<br><br>display_part_versions (string:part_number, stringArr:pversion, stringArr:pversion_desc) : Result Integer<br>*// Displays the versions' contexts (2nd level of the bookmark in Figure 2) related to the // selected product*<br><br>display_property_defs (string:design_id, string:analysis_id, stringArr:property_names,<br>stringArr:subtypes) : Result Integer<br>*// Displays the property definitions' contexts ( 4th level of the bookmark in Figure 2) related*<br>*// to the selected product definition*<br><br>display_uofs (string:prop_name, UofList:uoflist) : Result Integer<br>*// Displays the contexts of the UoF's (last level of the bookmark in Figure 2) related to the*<br>*// selected property definition*<br><br>select_uofs (UofNode:uof) : Result Integer<br>*// Selects one UoF from the UoF's list*<br><br>display_bookmarks (stringArr:bm_names, intArr:core_ids) : Result Integer<br>*// Display all the bookmarks related to the user in the SDM*<br><br>select_bookmark (short:bm_number) : Result Integer<br>*// Select the bookmark in the bookmark list, it then becomes the "current bookmark"*<br><br>delete_bookmark (short:bm_number) : Result Integer<br>*// Delete a bookmark from the bookmark list*<br><br>delete_current_bm (short:pre_core) : Result Integer<br>*// Delete the current bookmark in the bookmark list, then there is no "current bookmark"*<br><br>parse_exchange_file (string:ap_file_name, string:ap_file_location, string:err_log_name,<br>stringArr:uofs) : Result Integer<br>*// Inputs the AP files by name path, e.g. AP 209, extracts and returns the UoF's from the files*<br><br>check_in_exchange_file (stringArr:uof_instance, intArr:selected_ids, intArr:uof_ids, string:ap_file_name, string:ap_file_loc, string:err_log_name, string:err_log_loc): Result Integer<br>*// Stores the UoF's in the IPDB, establishes the relationships among the objects in the IPDB*<br><br>check_out_exchange_file (intArr:uofIds, string:p21_file_name,<br>string:constr_file_name, string:files_loc, string:err_string) : Result Integer<br>*// According to the relationships among the UoF's, checks out the UoF's and recomposes them to AP files*<br><br>ftp_invoke (string:hostname, string:username, string:password, string:directory, string file_name) : Result Integer<br>*// Invokes the ftp process, and transfers the data files between the client and the server* |

Table 1: The SDM API (Continued)

DAI's call the SDM functions directly and also perform other operations that do not involve access to the IPDB. The main screen of a DAI provides three different menus. The "File" menu invokes operations for accessing and viewing files. The "IPDB" menu invokes the primary SDM operations,

including the bookmark operations, an IPDB browsing operation, and the UoF check-in and check-out operations. The "Options" menu calls the operations for configuration of the environment such as changing passwords, and establishing default file locations.

The following chapter describes the investigation of the reengineering work involved in transforming the IPDE into a distributed client/server architecture using CORBA, with the Dai as the client as the SDM as the server. In particular, we outline the design tradeoffs and evaluate the efficiency of different techniques for incorporating the use of Orbix into the IPDE.

## 4. INTEGRATING CORBA INTO THE IPDB

This section presents the research issues associated with the redesign and implementation of the IPDE as a client/server architecture. Section 4.1 addresses issues associated with the architectural design of the system. Section 4.2 present the investigation of different alternatives for large file transfer. Issues related to concurrent user access are presented in Section 4.3. A more detailed evaluation of this research is presented in Section 5.

### 4.1 A Client/Server Architecture for the IPDE

To support design decisions in the reengineering of the IPDE, we established an objective of minimizing the number of changes to be made to the existing code of the IPDE. This objective was established since:

1) it was not clear at the start of this research how extensive the changes might be, and

2) there were limited resources within the scope of the RaDEO project to support extensive modifications.

Our approach, therefore, has been to take the path of least resistance, but to analyze the tradeoffs along the way. As a result, the client/server redesign of the IPDE is functional but not necessarily the most efficient or the best in terms of object-oriented design.

This subsection outlines the design decisions encountered in the research and the choices that were made in an effort to minimize the reimplementation effort. This first subsection addresses the partitioning of functionality between the client and the server. The second subsection addresses the tradeoffs involved with object-oriented design vs. code reuse. The third subsection discusses issues associated with the server and the client side wrapper.

#### 4.1.1 Partitioning of Functionality between the Client and the Server

The first logical step was to partition the functionality between the client (the DAI) and the server (the SDM). Most of this functionality was already defined by the current system. In particular, the DAI handles user interface issues as well as the logic associated with the control of user access to the SDM.

The SDM on the other hand, handles all operations associated with access to the databases of the SDM and the IPDB.

In the initial design process, we were fortunate to have access to an Orbix consultant working at Boeing. His initial suggestion was to redesign the system so that the server was responsible for the control logic associated with access to the SDM. This also frees the client from having to be concerned with knowledge associated with how to apply a sequence of operation calls to the server. In this way, any program could serve as a client to the SDM, while the SDM would maintain complete control over the access to the database. This shifting of functionality from the DAI to the SDM was rejected, however, since it required a significant amount of modification to both the SDM and the DAI. For example, this would have required adding state variables to the SDM for each user so that the SDM could recall the most recent sequence of operations for each user, as well as the code to analyze the sequence of operations to determine if an illegal sequence of operations has occurred. This change would have also required modification to the way in which the user interface was presented in the DAI.

The fundamental problem with this type of modificaiton is that the DAI was originally designed to capture the control logic associated with access to the SDM. This original design decision may therefore limit the generality of the SDM as a server in any distributed environment. The DAI, however, was originally designed as an important component of the IPDE, offloading application control logic from the SDM to the DAI. The decision was therefore made to leave the partition of functionality as it was in the original version of the system [30, 31]. In addition, the DAI would handle the tasks associated with file editing (non-database access) as well as the transfer of files between the client and the server. It was decided that the file transfer procedure (ftp) would be invoked and executed by the client. This would eliminate placing additional file transfer responsibility on the side of the already highly-burdened server.

After evaluating all options for partitioning functionality, we therefore decided to provide:

1) the SDM as a server, responsible for data management functions of the SDM and IPDB databases, including data integrity, retrieval, insertion, deletion and updating.

2) the DAI as a client, responsible for the user interface control logic associated with access to the SDM, and general non-database operations.

### 4.1.2 OO Design Vs. Code Reuse

The next major step in the redesign process involved the object-oriented design of the IDL interface. The first design below presents an ideal object-oriented view of the SDM as a server. We address the problems associated with achieving such an object-oriented design. We conclude by describing our final decision about how to design the IDL interface for the SDM.

*An Ideal Object-Oriented View of the SDM*

In an IDL file, an interface is similar to a class. The interface specifies the set of attributes and operations that all objects of that type provide to clients. IDL supports multiple inheritance, so one interface may inherit attributes and operations from others. After the IDL file is compiled, the interfaces are implemented as classes.

According to the principle of object-oriented design, an interface should provide improvement in basic component definition and interfacing. As a result, operations should be grouped into classes according to their logical relationships. For example, using the operations shown in Table 1, the bookmark's attributes and functions can be organized into an interface. The check-in/check-out functions define an additional interface. Other related functions, such as those that operate on configuration and login information define a third interface.

Figure 3 illustrates the implementation of these three separate interfaces: the "Bookmark" class, the "Check-in/Check-out" class and the "SDMOps" class. The one-to-one relationship between these three Orbix objects and their corresponding internal objects is displayed in Figure 4. This approach promises the security of each object and the potential for parallel processing between the three different server objects.

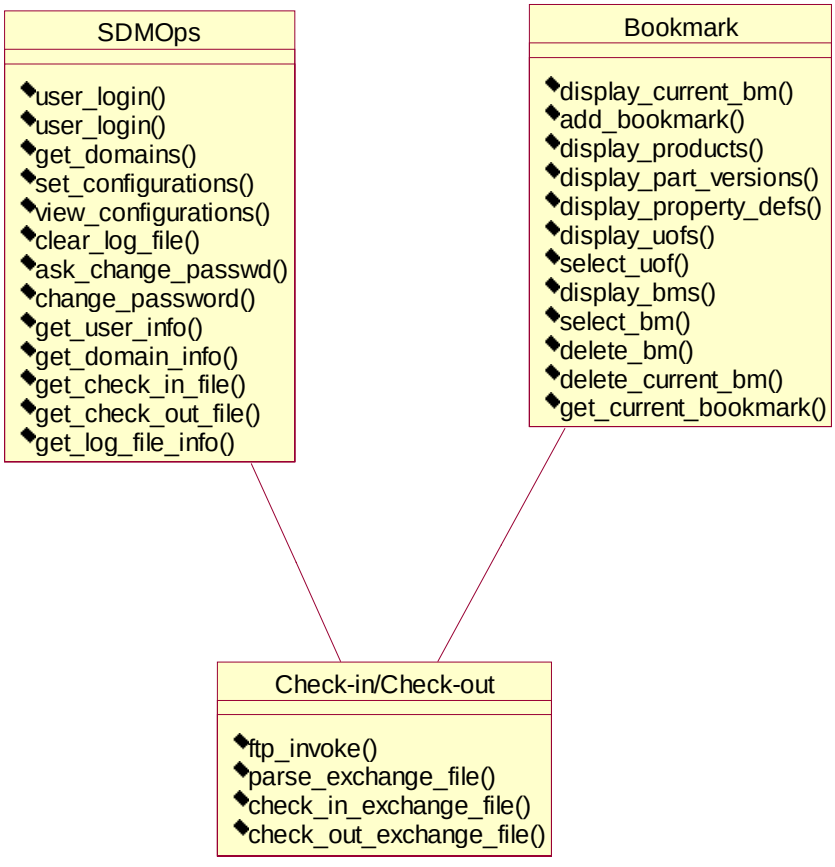


Figure 3: Ideal Design of the IDL Interface for the SDM

The original implementation, however, causes major restrictions for the implementation of this approach, requiring a significant reengineering process of the original API code. The problem is that the operations in the original API were all tightly coupled to each other and were not implemented following strict object-oriented concepts. For example, the check_in/check_out operations need to access bookmark information, including the current bookmark context and all of the configuration information. Access to this information is performed in the original check-in/check-out operation by directly accessing the corresponding objects in the Oracle databases of the IPDB. If we decouple the current API into three classes, access to the bookmark and the configuration information requires interactions between different objects. This decoupling will introduce new function calls in each of the classes and also require the recoding of the existing operations. In addition, reengineering work to the DAI is also unavoidable. The functions called by the DAI must be rearranged into three classes according to the change of the class definitions. New functions for associations among the classes will also be required by the DAI code. The implementation of this approach causes significant modifications of approximately half of the IPDE code.

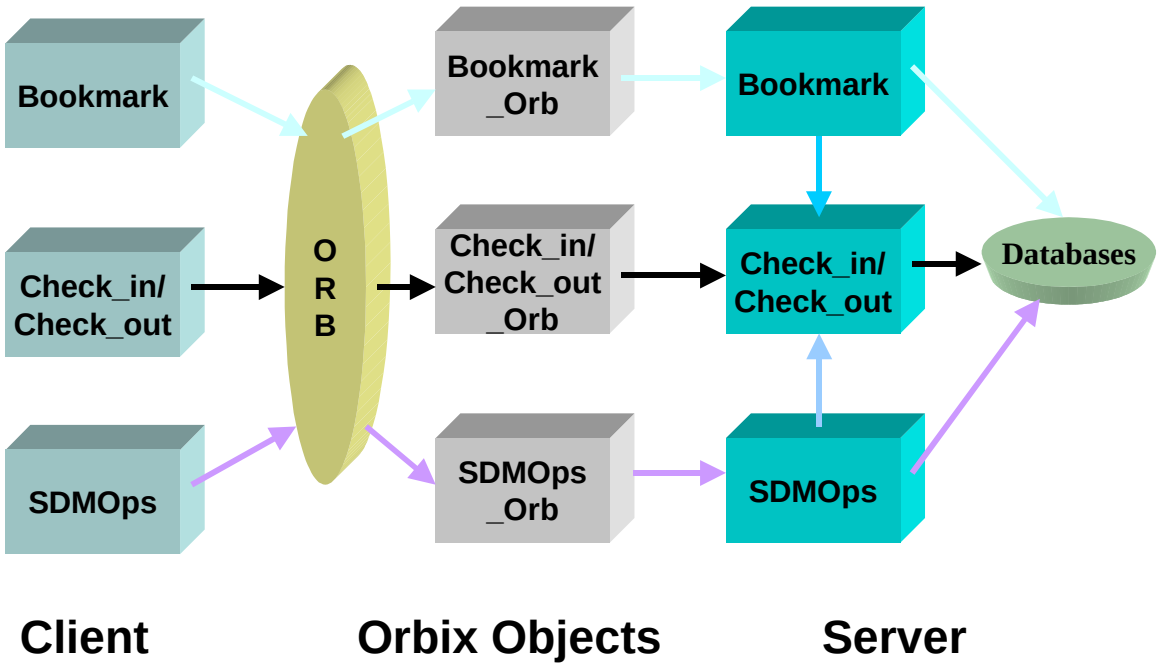


Figure 4: Ideal Design of the IPDE System

*The Actual IDL Interface to the SDM*

Since the original implementation provides only one internal class, "SDMOps", to accept all API invocations, the three stubs and three skeletons that will result from Figure 4 must still call the same internal object. The separation of the operations into three interfaces, therefore, provides only a logical grouping of the operations according to functionality. To avoid major recoding, the interface is implemented as shown in Figure 5. The interface therefore corresponds to the existing API class with all of the available functions defined as part of this interface. In this way, we could maximize the reuse of existing code, with the original API left mostly untouched.

This design hides the function calls in or among the other internal classes which are invisible to the DAI. From a functional point of view, however, the interface supports the interactions between the client and the server, even though the elegance of an objected-oriented design is sacrificed. Additionally, since the client still invokes one single interface rather than three, the DAI code can also be reused with minimal changes. We will provide a detailed description of how we implemented this approach in the next subsection, which discusses wrapping issues.

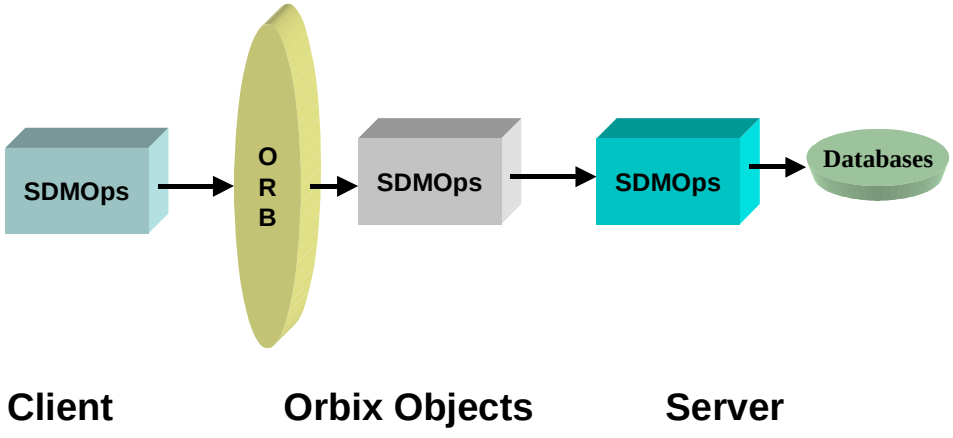


Figure 5: Actual System Model

### 4.1.3 Re-engineer SDM with Orbix

Following the initial IDL design, it was necessary to investigate the implementation differences between the existing API, as well as new functions that would serve the client/server model. For example, additional parameters were needed in some functions to support multiple user access as well as the changes to the database which support concurrent access. Additional functions were also needed to support large file transfer. Such changes must be reflected in the IDL interface. Existing functions of the API also required modification.

In the SDM, one example of incorporating this type of change involved the transfer of files between the client and the server. The new client/server environment requires that the server must access files that originate from remote sites. Input files originate from the client and are sent to the server. Output files from the server must also be sent back to the client. A file transfer operation is invoked when the transfer of a file is needed. The client needs to know information from the server, such as login information as well as where an input file should be sent or where an output file can be accessed. As a result, a new function was added to the server to retrieve the information for the ftp.

Implementation differences between the existing DAI and the SDM are primarily associated with the use of data types. In particular, the types supported by the IDL interface do not match exactly with the types of the parameters in the original implementation of the SDM. The function calls in the DAI and the function implementation in the SDM could be recoded to conform to the types of the IDL interface, since the DAI was developed using UIM/Orbix [22] which supports Orbix calls. Using this approach, the DAI

invokes requests to the server object directly and wrappers on the client side are not necessary. The advantage of this choice is that the system will work more efficient with one less wrapper on the client side. The disadvantage is that it requires modifications and retesting of the DAI. The integration of Orbix into the existing system therefore loses its transparency. To avoid significant recoding to the DAI, we chose to develop client side wrappers. The development of the client side wrappers also provided a more transparent approach to the integration of CORBA into the existing IPDE. After the creation of the stub and the skeleton, two layers were created corresponding to the stub and the skeleton, respectively. Figure 6 shows the basic procedure of creating the application with Orbix.

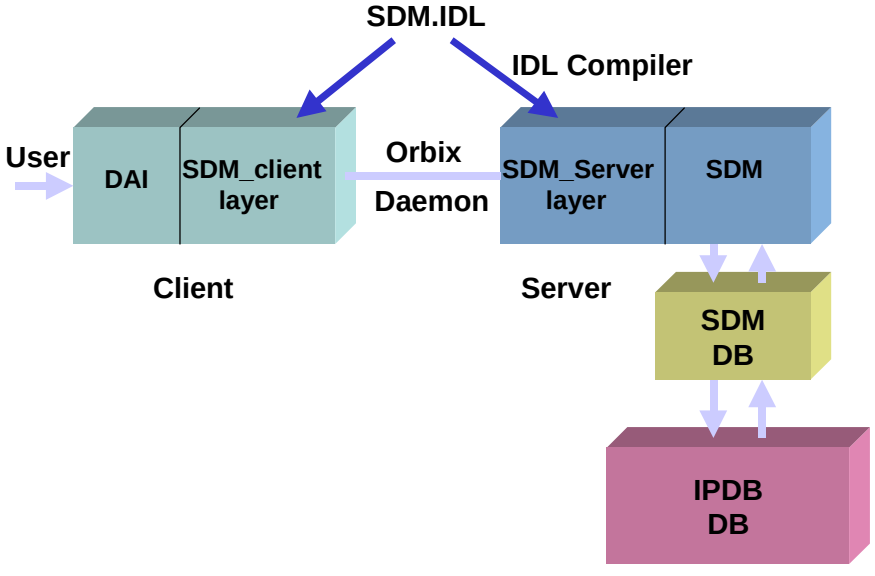


Figure 6: Creation of Client/Server with Orbix

The specific changes that occurred were:

1) On the server site, the skeleton implementation contains the call to the corresponding methods of the SDM object. The wrapper transforms the Orbix parameters to those of the original SDM interface. The output values of the SDM function are then wrapped back to Orbix format.

2) On the client site, a similar layer is added to accomplish transparency to the DAI code. Every function in this layer has exactly the same name as the corresponding function in the server. The functions in this layer, however, transform the data types of the DAI into the data types of the IDL interface.

Using this approach, the boundaries among the DAI, the client-side wrapper, the server-side wrapper, and the server are very clear. The reengineering process is therefore transparent to the existing system. The DAI and the SDM required little modification except for the recoding required to support concurrent access. The testing process to the new system is also limited to the Orbix middlelayer. The distributed computing process is straightforward, expeditious, and maximizes the reuse of the existing code as well.

To support the implementation of the wrappers, several general-purpose functions were developed. The key to a correct transformation is to be aware of the special characteristics of the data types or objects from both the original API and Orbix. For example, stringType, which is an abstract class, was implemented to have a strong capability to serve the programmer in the original code. The operations to a stringType object, e.g. "+", "<", "=", are overloaded to work as string concatenation, string comparison, and string assignment, respectively. stringType is a string object which can be assigned by '=', but string in Orbix can only be assigned by the function strcpy. The operations to a stringType object and the ones to a string variable in Orbix are therefore completely different. As another example, the length of UofList, which is a list class in the existing system, can be incremented dynamically, while its corresponding UList in Orbix be allocated an adequate amount of space before its use. The complete investigation of every difference between the original classes and the classes defined in Orbix is necessary to support a correct transformation.

After the client and server side wrappers were implemented, an initial client/server version of the IPDE existed. The problem of large file transfer between the clients and the server was addressed as the next step.

## 4.2 File Transfer Between the DAI and SDM

The check-in and check-out services provided by the SDM require the transfer of large files. The original implementation of the IPDE assumed the access of such files within the same file system. A major modification of the client/server implementation of the IPDE involved the capability of transferring check-in files from remote sources and transferring check-out files to remote sources. This transfer process must also be done in an efficient manner since the files to be transferred can be quite large (up to 100 megabytes).

There were three choices for implementation of the file transfer. Two of the three choices involved the transfer of the files as parameters in CORBA. In one case, the file is sent as one large data

string. In the other case, the file is broken into smaller parts and iteration is needed to send the entire file. This approach is a more generic method. However, the transaction time associated with this may not be acceptable, especially for the transfer of large files. A third choice is to automatically invoke an ftp operation external to CORBA, simply using CORBA as a notification mechanism. This approach provides a better solution for large files. The ftp approach also supports transferring files in any format while the other two approaches only support ASCII files. However, the disadvantage of this approach is the portability of the code. This section describes the design and implementation of the file transfer mechanism for this research. The problems associated with this method are also addressed. A performance analysis of the three approaches is described in Section 5.

### 4.2.1 Design and Implementation of the File Transfer

The transfer of files in the IPDE takes place before and after the execution of a check-in operation and after the execution of a check-out operation. Since the server may have multiple processes running at the same time while a client normally runs only one process, the server has a much greater computing burden than the client. Therefore, it was decided that the ftp procedure would be invoked and executed by the client. This approach reduces the server's computing work. The server can then focus on the check-in/check-out tasks.

The ftp is transparent to the DAI. Before requesting a check-in operation, the user must establish the check-in file location. When a check-in operation is requested, the ftp process is performed before the server begins the parsing of the check-in file. Similiarly, after a check-out operation is performed, the output file is automatically transferred back to the client. The user therefore requests the check-in or check-out process without explicitly requesting an ftp operation. Since the ftp function is a generic function which is embedded in the wrapper on the client side, no change to the DAI is required. The ftp operation is also independent from the check-in/check-out operations. As a result, no change in the check-in/check-out functions was required.

To implement the ftp procedure, the server needs a common directory to store the files 'ftped' from the client, as well as the error log files created by the check-in/check-out process and the output files to be transferred to the client. Once a new user logins to the system for the first time, the server will establish a new directory inside the common directory according to the user's name. Each user, therefore, has a separate working directory. The client must also be able to ask the server for the login information. The client then ftp's the file to the common directory on the server. If the file has been transferred back to the client, then the file must also be deleted from the server 's directory. The server object knows nothing about the ftp procedure. The parse operation simply accesses files from the server's working directory.

The check-out procedure also writes to the common directory. The client is responsible for accessing the file.

To support the transfer process, there is a profile on the UNIX platform called ".netrc". This file is specifically used for the invocation of the ftp mechanism. This file does not necessarily exist in the root directory, since the ftp function can write this file with the ftp commands into the root directory manually. Once the ".netrc" file exists in the system, it becomes available to an ftp process. A system call such as "ftp hostname" invokes the ftp command lines in the ".netrc" file. This file must contain the correct commands, including the information to login to the server (since the client invokes the execution), the ftp command, the file names to be transferred, and also the "delete" command which cleans up the files in the common directory. The information is stored in the SDM. Once an ftp operation is invoked, the client gets the information from the server through the Orb, and then rewrites the ".netrc" file in the root directory of the client.

The completion of the ftp procedure also depends on the correctness of the server information and the correctness of the file allocation. If one of these operations fails, the ftp will not be successful. Error messages from the ftp procedure are trapped and output to the user.

### 4.2.2 The Disadvantage of the FTP Approach

Section 5 demonstrates that the ftp approach is the most efficient approach for large files. There are, however, three disadvantages to the ftp technique implemented in this research. First of all, the portability of this approach is limited to UNIX platforms. The ".netrc" file is a part of the ftp package which is installed in all UNIX systems. Since the current implementation is UNIX-based, this approach is not a problem. This decision could cause problems, however, for future portability to Windows platforms.

Second, we need the login information of the server to fill in the ".netrc" file in the client's root directory. Since the client is responsible for the file transfer, the client must send a request to the server to fetch the login information, which is maintained in the SDM database. If the login information of the server changes, the SDM administrator must make the corresponding changes in the database. So the changes to the system and the database must be coordinated as a maintenance issue.

Finally, this approach requires that an administrator must keep the common directory secure.

## 4.3 Support For Multiple Concurrent Users

The IPDE must be capable of supporting multiple DAI's that access the database at the same time [32]. Some of the functionality for concurrent access is provided by Oracle 8 [24], while other functionality must be provided by Orbix as well as by the modification to the SDM.

### 4.3.1 Multi-processing Vs. Multi-threading

There are two approaches for implementing concurrent access in Orbix: multi-threading and multi-processing [33]. Multi-processing means that multiple processes exist synchronously [33]. When a client logins to the system, the system invokes a server object. A process is then created for this object. If another client logins to the system, another server object is invoked and a new process is also created for that object. So for a given server, there can be multiple processes existing simultaneously to serve different clients.

A multithreaded process, however, means the process has two or more threads (program counters), sharing the same address space, object handles, and other resources. Each thread has its own set of central processing unit (CPU) registers and its own stack. Each thread is also independently scheduled by the operating system [33]. The operating system divides the CPU time into time slots and allocates them to different threads. So the threads seem to be running synchronously to the users. Multi-threading can be adapted to a system that supports multi-processing. Many modern operating systems allow a process to create many lightweight threads. Orbix-MT allows clients and servers to create threads safely.

A server can be programmed to create a thread automatically per client request with Orbix [22]. For example, in the IPDE with CORBA, the server can be designed to have a ServerManageObject. When a client logins to the system, the client invokes a ServerManageObject. The object is then inserted into a list of ServerManageObjects. When a new client logins to the system, the new client invokes another ServerManageObject, which is again inserted into the list. The clients will have the capability to identify their corresponding ServerManageObject in the object list. In this approach, there is only one process at the server site. So every client's request will make the server invoke a thread to retrieve the corresponding ServerManageObject in the list. This scenario is illustrated in Figure 7.

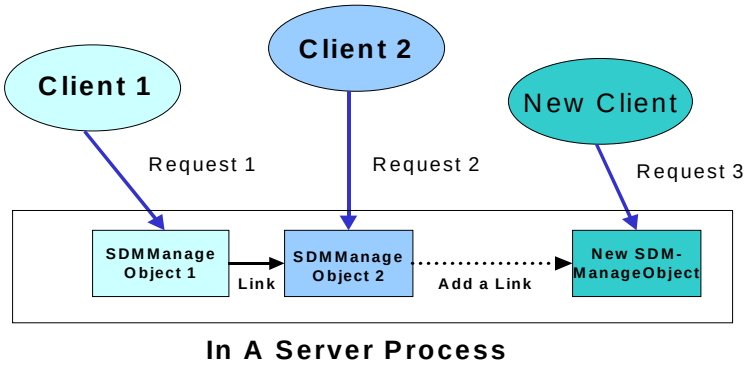


Figure 7: Relationships of the Clients, SDMManageObject, and a Server Process

To a system supporting multi-user access, the multi-threading approach is more scalable than the multi-processing approach. The threads in a process will share a large amount of resource (e.g., the text segmentation, the data segmentation). Using multi-processing, each process must have its own text segmentation and data segmentation independent from each other. Both of these resources obtain large amount of memories. In the multi-processing approach, it takes time to swap in or swap out the text segmentation and the data segmentations for multiple processes, while the threads are sharing the same resources which do not need to be swapped in the multi-threading approach. The multi-threading approach therefore is more extendable and lightweight than the multi-processing one. Since the IPDE project has the potential for being a heavyweight application, with multiple users accessing the system concurrently, the multi-threading approach therefore can reduce system resource requirements when compared with multi-processing.

In this research, the multi-processing approach was selected primarily since our current version of Orbix does not support multi-threading. If it was supported, however, the implementation of multi-threading would violate our objective of minimal changes to the code. Since each thread is independently scheduled by the operating system, it can run in parallel with the other threads in its process. Once multi-threading is introduced to the system, parallel programming difficulties, such as thread synchronization, critical section control, and deadlock prevention, will also be introduced to get the benefit of better throughput of the system. Multi-processing also introduces similar programming difficulties, but its situation is less complicated as multi-threading. For example, multiple threads implicitly share the data segmentation in which the global valuables are stored, while multiple processes will not share these

global valuables. The developer is responsible for synchronizing such data in multi-threading programming.

With Orbix, it is easy to invoke multiple processes. A single command line added into the server's registration information through the Orbix Daemon accomplishes this task. A switch is set to be "-per-client-pid", which specifies that a separate server process is to be used for each client process [34]. In this way, each client's request will invoke a separate server process, which will not share data structures or any other resources among the processes.

### 4.3.2 Changes to the SDM for Concurrent Accesses

The SDM and the IPDB database support concurrent access because Oracle 8 provides automatic row-level locking. Whenever a client A tries to get access to a record in the SDM or the IPDB which is being accessed by another client B, the client A has to wait until the completion of the transaction by the client B. The real transaction time is generally so fast that the locking is not apparent to users.

The primary difficulty associated with support for multiple DAI's is caused by the fact that the same use can potentially login more than once to the SDM, thus causing inconsistency for the data managed by an individual user. For example, the inconsistency exists for parallel use of viewing the IPDB and bookmark operations. The "View IPDB" operation adopts the same mechanism as adding a bookmark: a new bookmark is added to the SDM and becomes the current bookmark. The "View IPDB" bookmark is also invisible to a user for a single access. Once the user exits the screen, the bookmark is deleted from the SDM. Suppose, however, a user logins to the system twice, uses the "View IPDB" screen to traverse the IPDB and, at the same time, also tries to browse the current bookmark in another DAI. The user then finds that the current bookmark is not as expected. This problem has been solved by adding a flag into the "sdm_user" object in the SDM. Once the user logins, the flag is updated. Every time the user tries to login to the IPDE, the flag is checked to confirm that the user doesn't try to login more than once. After the user logouts from the IPDE, the flag is returned to its default value.

The last consideration for concurrent access issue is concerned with the shared data in the databases. In the current version of the IPDE, a UoF can only be modified by its owner, so the concurrent access by multiple users to more than one UoF's is not a problem. The same investigation to other shared data is done to make sure there is no conflict in any scenario of the transactions according the rules of the IPDE. In future versions of the IPDE, however, as several problems are resolved with regard to universal object identification, it may be possible for multiple users to try to modify the same UoF at the same time. In this case, write-locking to UoF's will be required.

## 5. EVALUATION OF USING CORBA WITHIN THE IPDE

The previous section presented the design and reengineering issues associated with incorporating CORBA technology into an existing application such as the IPDE. This section presents a more detailed evaluation of the design choices illustrated in Section 4. In particular, Section 5.1 provides a subjective evaluation of system modification issues when reengineering an existing system with CORBA. Section 5.2 presents a performance evaluation of file transfer options. Performance issues associated with the use of wrappers are addressed in Section 5.3. Section 5.4 then presents a performance evaluation of the concurrent, multi-user access provided by the new CORBA-version of the IPDE.

### 5.1 Incorporating CORBA into an Existing System

One of the objectives of the project was to evaluate the amount of effort required to incorporate CORBA technology into existing systems. This particular aspect of the evaluation is subjective in nature based on our experience with integrating CORBA into the IPDE. The issues outlined in this section, however, provide general guidelines that should be considered when modifying any existing system for use with a tool such as Orbix.

The issues to be considered can be roughly identified as:

1) partitioning of functionality between the client and the server,
2) developing an object-oriented IDL interface for the server,
3) developing client side wrappers and server side wrappers, and
4) insuring adequate support for concurrent, multi-user access.

When developing a new CORBA application, there is a lot of flexibility in the design process. For an existing application, however, the division of functionality may already be fixed and may require extensive modification if it is changed. In the IPDE, we had to choose between an ultra-thin client and a client with the capability to control the logic associated with calls to the SDM. To support an ultra-thin client, the server will be responsible for the control logic associated with access to the SDM. This can free the client from having to be concerned with knowledge of how to apply a sequence of operation calls to the server. However, this approach requires the server to have the capability to control the transaction sequences and also the capability to find illegal sequences. This shifting of functionality would have required significant changes to the SDM due to the division of functionality that was already determined in the original design of the system. The experience with this project demonstrates that to maximize reusability and reduce reengineering, it may be necessary to accept the original division of functionality for the components involved in the system.

After determining the functionality of the server, we needed to extract the operations from the server that the client would invoke from the user interface. IDL interfaces are designed to contain these operations as well as data types needed for parameters. An existing application may not able to provide a

true object-oriented interface. At best, the IDL may reflect the API of the existing system, or may require the development of an API layered on top of existing code. But if the existing code is not originally developed in an object-oriented manner, logical reorganization of the interface as separate objects may not be helpful to optimize the system. In the IPDE project, we also encountered the choice between an ideal IDL interface design according to object-oriented design and an IDL interface design considering the existing API operation class as discussed in Section 4.1.2. A redesign to separate the existing API into separate objects would have caused significant recoding to the API and the DAI. We chose to sacrifice a true object-oriented interface in order to maximize the reuse of the existing code and reduce the possibility of the new errors.

The wrappers of the client side and the server side are important components for adding distributed computing to an existing system. The wrapper of the server side, which communicates through the ORB, is a natural part of developing a skeleton. However, the wrapper for the client side keeps the Orbix programming transparent to the client. Though the system with this wrapper is less efficient compared to the system with the direct invocation to the server object by the client, the advantage is found in the implementation of this approach. The reuse of the client code is maximized. The clear boundaries between the client, the client-side wrapper, the server-side wrapper and the server present a modularized architecture that can accelerate the development cycle to extend the system in the future.

The development of wrappers is straightforward. Some general functions responsible for the conversion of the major data types must be developed. The implementation of converting functions depends on the complexity of the objects of both sites. To accomplish the wrapping correctly, a complete investigation of the data types in both formats must be done. Conversion functions are used when an operation of a Orbix object calls its corresponding operation of the API object. No changes to the existing API functions are necessary.

The multi-processing approach adopted for this research provided a quick solution to support for multiple users. Systems with many users, however, may need further investigation into the multi-threading approach with the understanding that multi-threading requires extensive modifications. In the IPDE project, for example, a complete recoding to support the parallel execution problem would be necessary if multi-threading is adopted. This approach also requires an evaluation of the existing code to determine alternatives for the support that it already provides for concurrent access as well as the types of changes that may be necessary to provide for concurrent access through CORBA.

In summary, CORBA technology can be easily integrated into existing systems. The easier and more transparent techniques, however, may be less efficient. If the objective of the CORBA transformation is to provide a more object-oriented design and to support efficient communication, significant redesign and recoding may be required.

## 5.2 Performance Evaluation of File Transfer Operation

As described in Section 4, there were three techniques available for transferring files between the DAI and the SDM:

1) a file is sent by an automatic invocation of an ftp operation external to CORBA

2) a file is sent as one large data string parameter, or

3) a file is broken into smaller parts and iteratively sent as a parameter.

The first subsection below describes the implementation issues associated with each of the approaches. The second subsection provides a performance comparison of each approach.

### 5.2.1 Implementation Efforts for Different Approaches

The file transfer approach implemented within the IPDE involved direct invocation of the Unix ftp facility. The use of this approach is rather straightforward. The coding changes were minimal and performed on the client side. The server simply needs to provide the necessary information for the remote login.

When the file is transferred as one large parameter, the server provides the major service. The implementation is separated into two parts involving the “get” and the “put” operations. In the “get” operation, for example, the server breaks the file into small parts, links the parts into an unbounded string list, and sends the list back as a parameter. The “put” operation is implemented in a similar manner. This approach was tested as a separate client/server program. The implementation of this approach requires approximately twice the effort of the ftp approach.

The third approach is to break the files into small, equal-sized parts and transfer each part using a parameter. The client is responsible for distributing the file into parts and composing the parts into a file again. This method is used to avoid the large buffer space required in the server. In this case, when space is a critical resource, the server needs to allocate one fixed buffer size for all parts of the file. This approach is therefore better than the second approach when the active file is sent as one parameter. However, the coding of this approach is more complicated. The transfer time is faster than the second approach but slower than the first approach as discussed in the next subsection.

The advantage of the second and third approaches described above is that the file transfer process can be handled by a server separate from the SDM server. The SDM server is already highly burdened with the service it provides. In particular, the parsing, check-in, and check-out procedures can be quite time-consuming, especially when large files are involved. When multiple users perform these operations at the same time, the server can be slowed down even more for it must be responsible for managing the buffer space required for the file transfer. However, synchronization control must be provided to the

multiple working servers, e.g., the SDM server and the ftp server. This approach was rejected since the synchronization control among multiple servers is very difficult to implement.

### 5.2.2 Performance Comparison of Each Approach

As part of this research, we performed time studies to compare the time involved to transfer files under the three techniques described above. Since the transfer process in each approach is invoked by an ftp function call or an iteration of the ftp function call, the starting points of the time testing are set to be prior to the ftp function call or the iteration. The ending points are set at the end of the ftp function call or the iteration. The system call "gettimeofday" is used to identify the starting time and the ending time. The time unit is also set to be 1/1000 second. In all three approaches, the client invoked the system call for starting and ending time. During the transfer process, the machine used for testing may also be dealing with resource demands of the users. To obtain consistent results, all tests were run while the system was running under similar loads.

Table 2 shows the results of the tests graphically that were used to compare the transfer time of the three different techniques. For each case, the timing was performed for files of size 0.75 megabytes (MB), 7.5 MB, 25 MB, and 50 MB, 75 MB, and also 100 MB.

| | 0.76 MB | 7.5 MB | 25 MB | 50 MB | 75 MB | 100 MB |
|---|---|---|---|---|---|---|
| Approach 1 | 1.888 | 85.233 | 235.346 | 467.745 | 678.456 | 926.343 |
| Approach 2 | 2.547 | 126.319 | 342.532 | 628.934 | 875.343 | 1157.587 |
| Approach 3 | 3.567 | 98.893 | 287.341 | 556.349 | 737.396 | 979.987 |

Table 2: Test Results of File Transfer with Different Approaches

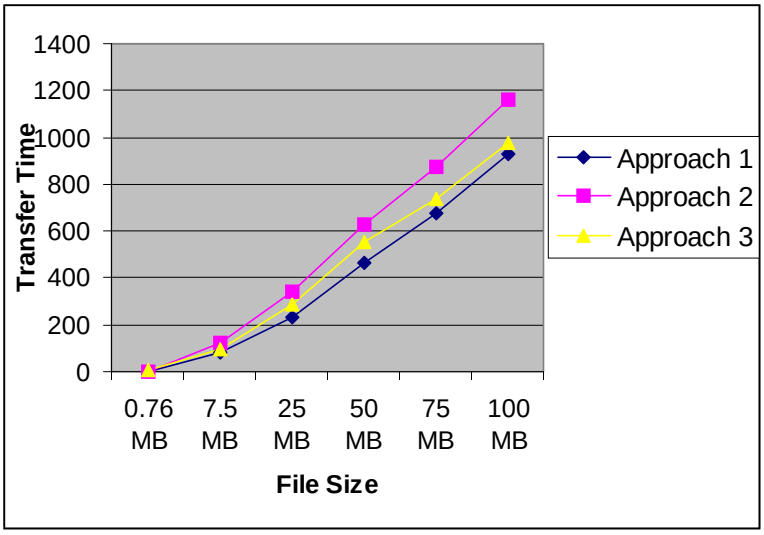


Figure 8: Time Comparison of Different FTP Approaches

The lines in Figure 8 represent the general growth of the transfer time according to the growth of the file size. From Figure 8, we also see that the transfer time for each approach is similar with smaller file sizes. When the files become larger (about 10 megabytes), the time required for each approach increases. Differences also begin to appear in the different techniques. The use of the Unix ftp facility in the first approach is the fastest. The second approach of sending the entire file as one parameter is the slowest of the three approaches. This is primarily due to the large amount of buffer space that is required to implement this technique. For the transfer of several very large files, this approach could become so overwhelming that other users may be prohibited from simultaneously running processes on the same machine. The third technique of breaking the file into several parts is comparable in time to the Unix ftp approach, although generally slower. The size of the buffer can be limited to a reasonable size, so the large buffer problem of the second approach will not occur. However, the size of the buffer is, in fact, an important factor for the transfer speed. For a file more than 100 megabyte, a 10-kilobyte buffer will cause the server and the client to be involved in more than 10,000 loops, therefore, slowing down the transfer time. The buffer size must be determined by the length of the files involved in the transfer. In our project, a 64-kilobyte buffer is reasonable for both small and large files. In general, the size of the buffer may be difficult to predict.

Since the three approaches are comparable, either approach could be adopted for the IPDE. For different resource requirements, we can choose different approaches. When the time and space are both

critical resources, approach 1 should be adopted. When portability is the most important issue and the files are relatively small, approach 2 could be chosen. Approach 3 is a more generic approach that is portable and works well for files of any size.

### 5.3 Performance Issues for Wrappers

Where the server with a wrapper is a necessary part of the client/server architecture, the client with a wrapper is optional to developers. As described in Section 4, an alternative implementation approach is to modify the client code to directly invoke Orbix calls. As part of this research, we performed studies to compare the communication time between the server and the client with a wrapper, and the time between the server and the client without a wrapper. In the IPDE, we could embed the functions of the Orbix objects into the DAI code. However, the DAI is an interactive GUI which would make the studies complicated if we did this. As a result, we developed a textual driver code for the client with the wrapper and another one for the client without the wrapper. The studies were therefore performed outside of the GUI environment. Iteration was introduced to invokes sequence of operations and therefore make the test results more apparent. The starting point was set prior to the iteration of the function call and the ending point was set right behind the iteration call. The system call "gettimeofday" was used as the starting point and the ending point. The time unit was set to be 1/1000 second. The number of loops was set as 100. The system load and the system environment were equal in each approach.

Table 3 shows the results of the tests. The timing was performed for several typical function calls in the SDM (display_products, display_uofs, display_current_bm, display_bms). These functions deal with operations on bookmarks. Figure 9 displays a graphical comparison of this test data.

| | display_products | display_uofs() | display_current_bm() | display_bookmarks() |
|---|---|---|---|---|
| Without wrapper | 5,447 | 7,335 | 6,786 | 4,495 |
| With wrapper | 14,424 | 22,165 | 15,767 | 20,643 |

Table 3: Testing Results of Wrapper Technology

The test data illustrates that the transaction time of the client with the wrapper is about two times slower than the approach without the wrapper. In the client with the wrapper, the transaction times of the functions are slower since different type conversion functions must be called. The complexity of conversion affects the execution time. However, as discussed in Section 4, the wrapper of the client side also maximizes the reuse of the existing client code. The wrappers of the client side and the server side provide clearer system boundaries and greater transparency to the Orbix internal process. The client side

wrappers are also faster to implement. This is one area, however, that could be recoded to improve the performance of the client/server implementation of the IPDE.

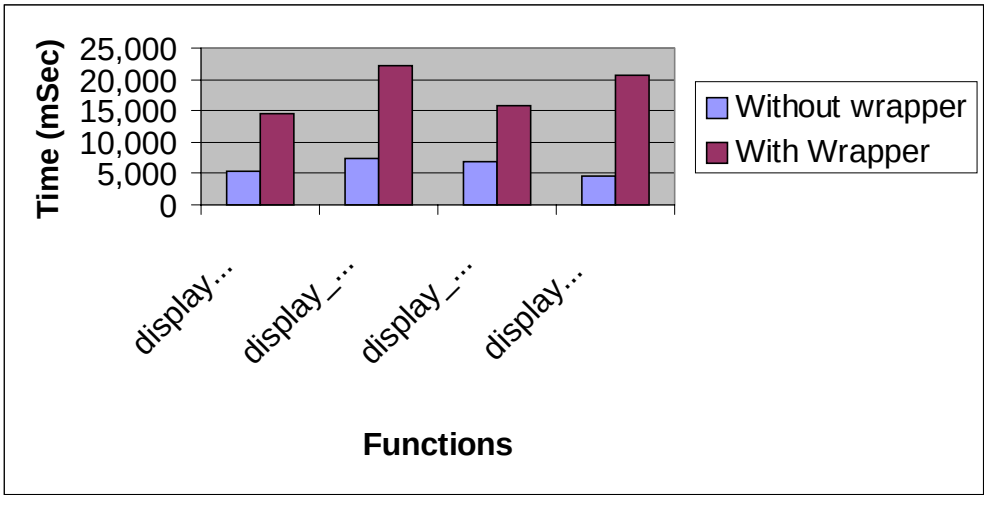


Figure 9: Graphical View of Test Results of Wrapper Technology

### 5.4 Effectiveness of Multi-user Access

The final evaluation issue addresses the effectiveness of using CORBA within the IPDE to support multiple users. As part of this research, we compared transaction times for the following cases:

1) the transaction time for single users in the original non-CORBA mode of operations,

2) the transaction time for single users in the non-concurrent CORBA mode of operation, and

3) the transaction time for multiple users in the concurrent CORBA mode of operation.

We developed a textual code driver for each mode to eliminate the complexity caused by the interactive GUI interface and the DAI. As in the previous study, iteration was introduced to make test results more apparent. The starting point was set up prior to the iteration of the function call and the ending point was set right behind the iteration call. The system call “gettimeofday” was used as the starting point and the ending point with the time unit set to 1/1000 second. The number of loops was set at 100 times and the system load was equal for each study.

Table 4 shows the data results of the time study. Figure 10 displays the time comparison of the results. For concurrent-mode cases, there were four concurrent processes active at the same time.

| | display_products | display_uofs() | display_current_bm() | check_in_exchange_file() |
|---|---|---|---|---|
| Mode 1 | 3,223 | 4,313 | 4,765 | 183,343 |
| Mode 2 | 14,424 | 22,165 | 15,767 | 230,643 |
| Mode 3 | 17,341 | 24,987 | 18,221 | 247,327 |

Table 4: Testing Results of Three Modes for Different Functions

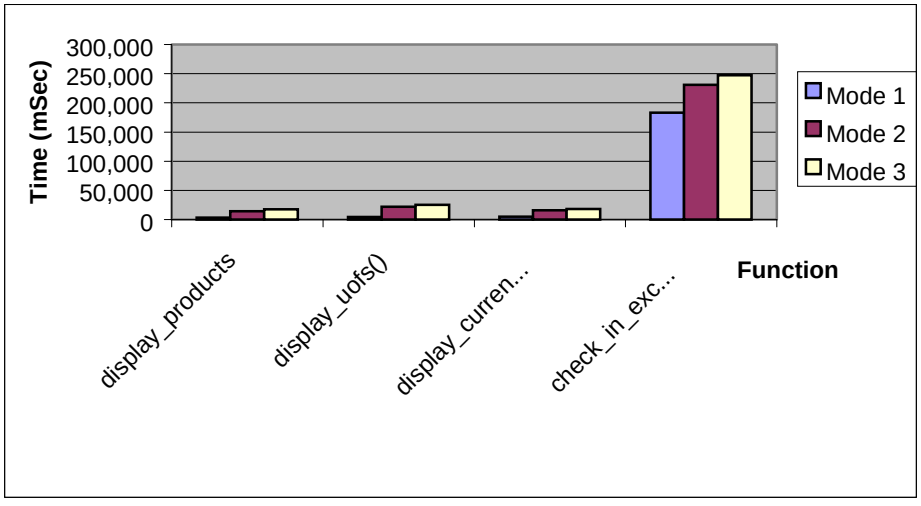


Figure 10: Comparison of Single and Concurrent Accesses

We analyze the reason for the results from Table 4 and Figure 10 below:

1) The transaction time for single users in the current CORBA mode of operation.

In this mode, the DAI, the SDM, and the databases are tightly coupled. The system allows only a single user to login. No extra layers are added between the DAI and the SDM. The system has fast transaction speed compared with the CORBA version of the IPDE with the obvious disadvantage of no concurrent access.

2) The transaction time for single users in the non-concurrent CORBA mode of operation.

The transaction time is slowed down significantly after the CORBA layer with wrappers are added to the DAI and the SDM. Since the IPDE is basically a database system, there is a large amount of data throughput. The middlelayers have to take time to wrap the data up into the Orbix format, then transfer them through the network, and finally wrap them back to the format that the DAI could use. The obvious advantage, however, is that users can access the IPDE from remote locations.

3) The transaction time for multiple users in the concurrent CORBA mode of operations.

This mode is the most complicated among the three since it involves concurrent access to the IPDE. The concurrent CORBA mode of operation is obviously the most time-consuming mode of operation, although the time for four users is not significantly greater than the transaction time for one user. As described in the previous section, the performance of the system could be improved by elimination of the client side wrapper and by the implementation of multi-threading.

## 6. SUMMARY AND FUTURE WORK

This research has presented the results of an experiment involving the reengineering of an existing engineering design application into a client/server architecture using CORBA. The specific application was the IPDE, supporting a database approach to the exchange of design data. The original IPDE was reengineered with the user interface (DAI) as the client and the database component (SDM) as the server. Evaluation of the reengineering work to the original system was performed. The efficiency of the different implementation options was also investigated.

The results of this work demonstrate that CORBA technology can be easily integrated into existing applications. The speed of the integration as well as the efficiency of the resulting system, however, depend on the degree of modification that developers are willing to consider in the reengineering process. The most transparent approach to the use of CORBA requires less modification and generally less efficient performance. The less transparent approach to the use of CORBA can potentially require significant system modification but greater performance gains. This work outlines issues that must be considered for the partitioning of functionality between the client and the server, development of an IDL interface, development of client and server side wrappers, and support of concurrent, multi-user access. In addition, this work also provides performance and implementation comparison of different techniques for the use of wrappers, and for the transfer of large data files between the client and the server. Performance comparisons for the incorporation of concurrent, multi-user access are also presented.

The results of this work also provide several recommendations for future versions of the IPDE. Performance can be improved by removing the client side wrappers and providing direct invocation of the Orbix calls into the DAI. In general, the system should be redesigned to provide a more object-oriented approach to the support of SDM services. Multi-threading should also be investigated for more efficient support of a large number of users.

## ACKNOWLEDGEMENTS

This research was supported by DARPA's Rapid Design Exploration and Optimization Program (RaDEO). The partners on this project are Boeing Defense and Space Group, MacNealschwindler Corporation, and Arizona State University.

## REFERENCES

[1] Object Management Group, The Common Object Request Broker: Architecture and Specification, Updated Revision 2.0, July 1996.

[2] Jon Siegel, CORBA Fundamentals and Programming, Wiley Computer Publishing Group, 1996.

[3] IONA Technologies Ltd, Orbix 2 distributed object technology, Programming Guide, November, 1995.

[4] Boeing Defense & Space Group, Manufacturing Automation and Design Engineering (MADE), Integrated Product Definition Model (IPDM), DARPA proposal, September 1995.

[5] Philip Gill, "A Better Way to Fly", Oracle Magazine, May/June, 1997.

[6] Ling Fu, The Implementation and Evaluation of the Use of CORBA in an Engineering Design Application, M.S. Thesis, Department of Computer Science and Engineering, Arizona State University, Tempe, AZ, Summer 1998.

[7] Douglas C. Schmidt, Silvano Maffeis, "Constructing Reliable Distributed Communication Systems with CORBA", IEEE Communications Magazine, vol. 4, no. 2, February, 1997.

[8] Steve Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments", IEEE Communications Magazine, February, 1997.

[9] Thomas J. Mowbray, Ron Zahavi, The Essential CORBA: Systems Integration Using Distributed Objects, John Wiley & Sons, Inc, 1995.

[10] G. Booch, Object-Oriented Analysis and Design with Applications, 2nd edition, Benjamin Cummings, 1994.

[11] Arnold Hutt,. Object Oriented Analysis and Design. John Wiley & Sons, Inc., 1994.

[12] Naji Ghazal, Basic Concepts of CORBA's IDL, GTE Labs' Distributed Object Computing Group, Dallas, June, 1996.

[13] Hewlett Packard Press Release, HP Introduces CORBA 2.0-Compliant HP Distributed Smalltalk Flexible Tool to Develop Three-Tier, Multilanguage, Multiplatform Application Architectures, http://www.hp.com/csopress/95aug07.html, August 7, 1995.

[14] Don Kretsch, Solaris NEO Interoperability with CORBA ORBs and COM, http://www.sun.com/software/events/presentations/OP4.Kretsch/OP4.Kretsch.html Jan 18th, 1998.

[15] Eric Evans and Daniel Rogers, "Using JAVA Applets and CORBA for Multi-user Distributed Applications", IEEE Internet Computing, vol.1, no.3, May-June, 1997.

[16] M. Khayrat Durmosch, Christian Egelhaaf, "Design and Implementation of a Multimedia Communication Service in a Distributed Environment Based on the TINA-C Architecture," Trends in distributed Systems: CORBA and Beyond, International Workshop TreDS '96, Springer, October, 1996.

[17] Steve Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments", IEEE Communications Magazine, Feburary 1st, 1997, vol. 35 no2.

[18] Boeing Corporation, “BOEING Pushes For Objects,” Informationweek, April 29th, 1996, n577, Page 24.

[19] OMG Organization, “CORBA Forms the Backbone of NCA,” Software Magazine, vol.17, no1, January 1st, 1997.

[20] M. Roantree and P.Hickey, “Metadata Modeling for Healthcare Applications in a Federated Database System, Trends in Distributed Systems: CORBA and Beyond,” International Workshop TreDS’96, Aachen, Germany, October, 1996.

[21] M. Leclerc, C. Linnhoff-Popien, CORBA-Based Data Transfer for Financial Risk Management, Trends in Distributed Systems: CORBA and Beyond, International Workshop TreDS’96, Aachen, Germany, October, 1996.

[22] Black & White, Getting Started with UIM/Orbix, Black & White and IONA Technologies Ltd., 1996.

[23] Black & White Technologies, Orbix Programming Guide, Release 2.0, Black & White and IONA Technologies Ltd., 1996.

[24] Oracle Corporation, Oracle 8.0.3 User Manual, 1997.

[25] Michael Tjahjadi, The Implementation and Evaluation of an Express to Oracle8 Mapping, M.S. Thesis, Department of Computer Science and Engineering, Arizona State University, Summer, 1997.

[26] International Standard ISO, STEP Part 1: Overview and Fundamentals,” ISO TC184/SC4, 9-15-1992.

[27] D. Schenck and P. Wilson, Information Modeling the EXPRESS Way, Oxford University Press, 1994.

[28] International Standard ISO, “STEP Part 11: EXPRESS Languange Specification,” ISO10303-11, 1993.

[29] STEP Tools, Inc., ST-Developer Tools User Manual, STEP Tools, Inc., 1997.

[30] Jim Clarke, Todd Bowman and Jim Stikeleather, A White Paper for Business Professionals --- Client/Server Architectures, The Technical Resource Connection, Inc, 1996.

[31] T.J. Mowbray and R. Zahavi, The Essential CORBA: Systems Integration Using Distributed Objects, John Wiley, New York, NY, 1995.

[32] Aart Van Halteren and Peter Foliant, “Experiences with Supporting Multiple Interfaces in a CORBA Environment”, the ECOOP’97 Workshop on CORBA, Jyväskylä, Finland, June, 1997.

[33] Helen Custer, Inside Windows NT, Microsoft Press, 1993.

[34] Jim Clarke, Jim Stikeleather and Peter Fingar, Distributed Object Computing For Business, The Technical Resource Connection, Inc, 1996.

# ClearNet: A Multi-Tiered Infrastructure for Browser-Based Applications


## Naser S. Barghouti and Bill Moss

## Bear, Stearns & Co., NY, NY


ClearNet is an infrastructure developed over the last 2 years at Bear, Stearns & Co. to deploy financial services to orrespondent clients over the Internet and private access lines. The services include order and trade entry systems, client portfolio management tools, broker management tools, on-line financial statements, and private securities trading systems. The architecture of ClearNet, which is based on distributed objects, aims to achieve 6 objectives:

1. Ease of Use: to provide a familiar user interface and a friendly navigation and use experience.
2. Security: to ensure that a client accesses only the services to which he or she is entitled.
3. Reliability: to guarantee a high quality of service, in terms of availability of services, that meets the customer's needs.
4. Performance: to provide our services in a speedy and efficient manner.
5. Scalability: to allow for future growth of our customer base without impacting performance or reliability.
6. Maintainability: to sustain our leading edge by absorbing new best-in-breed technologies as seamlessly as possible.

Technologies like Java (client- and server-side), CORBA, Enterprise JavaBearns, and client-side and server-side digital certificates were used in building ClearNet applications. This presentation will describe the architecture, especailly our use of istributed objects, and share our experiences in developing and deploying ClearNet to thousands of customers worldwide.

# A Software Architecture for A Real Time Data Distributed Objects System

***By Neil Roodyn***

## 1. Introduction

This paper presents a software architecture for using distributed objects within a real time system. It first presents the problem and then addresses the issues typically associated with both real time systems and distributed object systems. The commercial aspects are then examined along with the specific commercial requirements made for such a system. Next the thoughts and ideas behind the design are presented, followed by examining the objects, the object relationships and the interfaces exposed by the objects. Specific features of the implementation carried out by Cognitech Ltd. are considered, and possible future work is discussed.

## 2. The Problem

It was the problem of creating a system which both provided data and created derived data in a timely manner that led to the architecture described within this paper. In the past Cognitech has written systems which utilise the classic client/server model. The real time data feeds would be collected on a server which would then both archive the data and provide the data for collection by a number of clients. These systems had problems; i) The clients had no way of being notified of changes in the data without implementing a proprietary solution, ii) any derived data had to be calculated either on the server, wasting valuable CPU time, or duplicated on each client, resulting in the same data being created in many places & iii) the client/server model doesn't fit in well with object models used when designing and implementing the system.

After working on several such client/server solutions it became obvious that a better solution could be found by utilising a distributed objects model. It is this model that is described within the following text.

## 3. Issues with Real Time and Distributed Objects Systems

In order to fully understand the nature of both real time and distributed objects systems, as well as methods used to achieve the goals of this project, I here examine how existing systems manage to achieve similar tasks. The financial sector is Cognitech's target market for real time systems, these systems provide market data as it changes along with news and financial reports. This data is usually provided in the form of a 'feed', which is simply a stream of the data.

The data involved will generally come from an external source, the user will wish to view the changes in that data within a specified time frame. Each application will tend to be different, but generally real time data becomes historical data once that specified time frame has elapsed. The aim of the software is therefore to get the data to the user within that time frame.

Until recently it has not been feasible to create a real time distributed object system to cater with real time financial data. The available distributed object systems did not provide a stable enough platform and the timely delivery of data could not be guaranteed. Another problem was that the commercial acceptance of such platforms was low. This is now changing with statements such as:

*'It is self-evident that there is a significant potential market for a DCOM which is "real-time" -- at the very least in the minimal sense of DCOM having service latencies which are tightly upper bounded and the smaller the better.' [1]*

The architecture described here does not provide a guaranteed latency period and so the system must be considered as a *soft*[2] real time system. Even so the system implementation would not be of use if this latency becomes too great.

## 4. Commercial Aspects and Requirements

In previous projects Cognitech had handled data in multiple formats from multiple sources. We now examine the commercial requirements and aspects of the distributed objects system that was created. Windows NT is now becoming the standard environment at all levels of industry. It was a definite commercial requirement that this system ran on the Windows NT platform. Another good reason for selecting Windows NT as an environment is the powerful API it provides, the breadth of technical support available and the many skilled programmers who know the API well. This has lead to the fact that many other applications are already available which may complement a system such as the one being discussed herein.

Microsoft have already provided some guidelines for the creation of real time systems for financial data, in a paper entitled 'WOSA Extensions for Real Time Market Data'[4], otherwise known as WOSA/XRT. On top of this architecture I had already worked on the RTD (Real Time Data) System[5] for Black Ace Software Engineering Ltd. (a sister company of Cognitech). It was considered important to learn from these experiences and build on them to create the new system. With the general acceptance of, and migration towards, Microsoft solutions, COM has become a feasible standard for implementing commercial real time data systems. In essence a virtuous cycle has caused more developers to use COM which in effect has forced COM to become more stable and faster, which enables more developers to use COM and so on.

The requirements for where to create a system that could:

1. Take differing types of financial data; whether the data was a price, a currency exchange rate or description of a fund's activities.
2. Accept data from more than one feed; an input interface would need to cope with input from multiple sources.
3. Provide a mechanism for derived data to be calculated; these calculations would have to occur within set time frames upon receipt of the data. Calculations could be dependant upon another calculations derived data, but there were no cyclic dependencies.
4. Client programs should be able to filter the data for information that is of interest to the user.
5. Provide the data, derived and source, to one or more client programs; an interface to allow client programs to be notified in a timely manner would be required.
6. Store the data for historical analysis; some form of database system would be needed to archive the data. This archived data would need to be available to other third party application through a standard interface.

The existing business model is shown in figure 1. The data is collected by research and then entered into a database. The research department also performs calculations on the raw data to create derived data which is also placed in the database. The data in the database is then used by the research team , the sales team and the trading team to generate reports on screen and on paper.

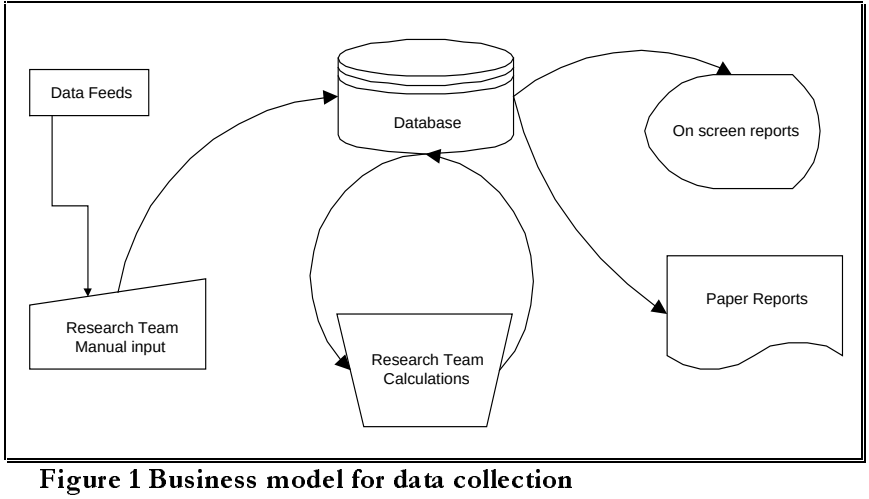


**Figure 1 Business model for data collection**

## 5. Designs

From the requirements and previous project experience figure 2 was sketched out as an outline for data flow within the system. As can be seen a three tier approach has been taken. This provides several advantages over the more typical client server model:

i) A simple system that inserts the data into a database will collect the feed data. There is little that can go wrong, Even if the rest of system fails the raw data will still be collected.

ii) Complex calculations are all carried out by one layer, this doesn't have to sit on one physical machine and can be scaled across multiple servers.

iii) Each piece of derived data only has to be calculated once. Unlike a client server model where the calculations occur on 'fat' clients.

iv) The clients can be 'thin'; they don't need to have much intelligence and they can become display mechanisms.

Each of the data connections is now examined.

### *Data Feed → Live Data Base*

This would be a straight movement of data directly into the Live DB from the feed. Some sort of feed decoding would be required. We could also have some module receiving the feed, which performs the Live DB update and the Data Server update.

### *Live DB → Data Server*

Straight after (or before) the data is input to the Live DB, the same data is passed to the Data Server. The method could be by single field value (or record value - depending on structure of feed), or by batch of field (record) values. Return values are not required. As mentioned before, we could have some module performing the Live DB update as well as the Data Server update.

### *Data Server → Live DB*

This data flow represents fields that have changed as a result of other fields being updated or data that has been input by the user. The Data Server will have to be intelligent enough to know when to update the Live DB (and when it does so, it must not cause the Live DB to update the already updated clients.)

### *Data Server → Historic Data Base*

This kind of data flow will occur when the client changes some data and this gets passed back to the Data Server and forwarded from there to the Historical Data Base. Again, some control may be required to identify this data as having the Historical Data Base as the data sink.

### *Historic Data Base → Data Server*

This data flow occurs when data has been requested by the Data Source from the Historical Data Base. A simple SQL interface will suffice.

### *Data Server → Calculation Objects*

This is an interesting data flow. The origin of the flow is a single field that has changed. This field may influence other fields and consequently must result in the updating of the clients and the Live DB. Issues that arise here are; i) Where do we decide which, if any, fields are affected by the changed field, ii) if the field does affect other fields which method do we use to decide which other fields are affected, iii) how are the recalculations carried out, do intelligent objects within the calculation objects perform the calculations, or are they performed by some 'turn the handle' operation.

### *Calculation Objects →Data Server*

This data flow consists of the fields that have changed as a result of calculation requests. These could be provided as a return value of the function performing the calculations mentioned in the previous section. Alternatives include passing the variables by reference, if that is possible in the environment.

### *Data Server →Clients*

Send the clients the data that has been updated and let them decide if they need to refresh their displays.

### *Clients →Data Server*

Any client will fire off an event when some of its data has changed. This event will be received by the Data Server and handled by requesting all clients to update. In addition to this, the corresponding Live DB, Historical DB and dependant fields must be updated.

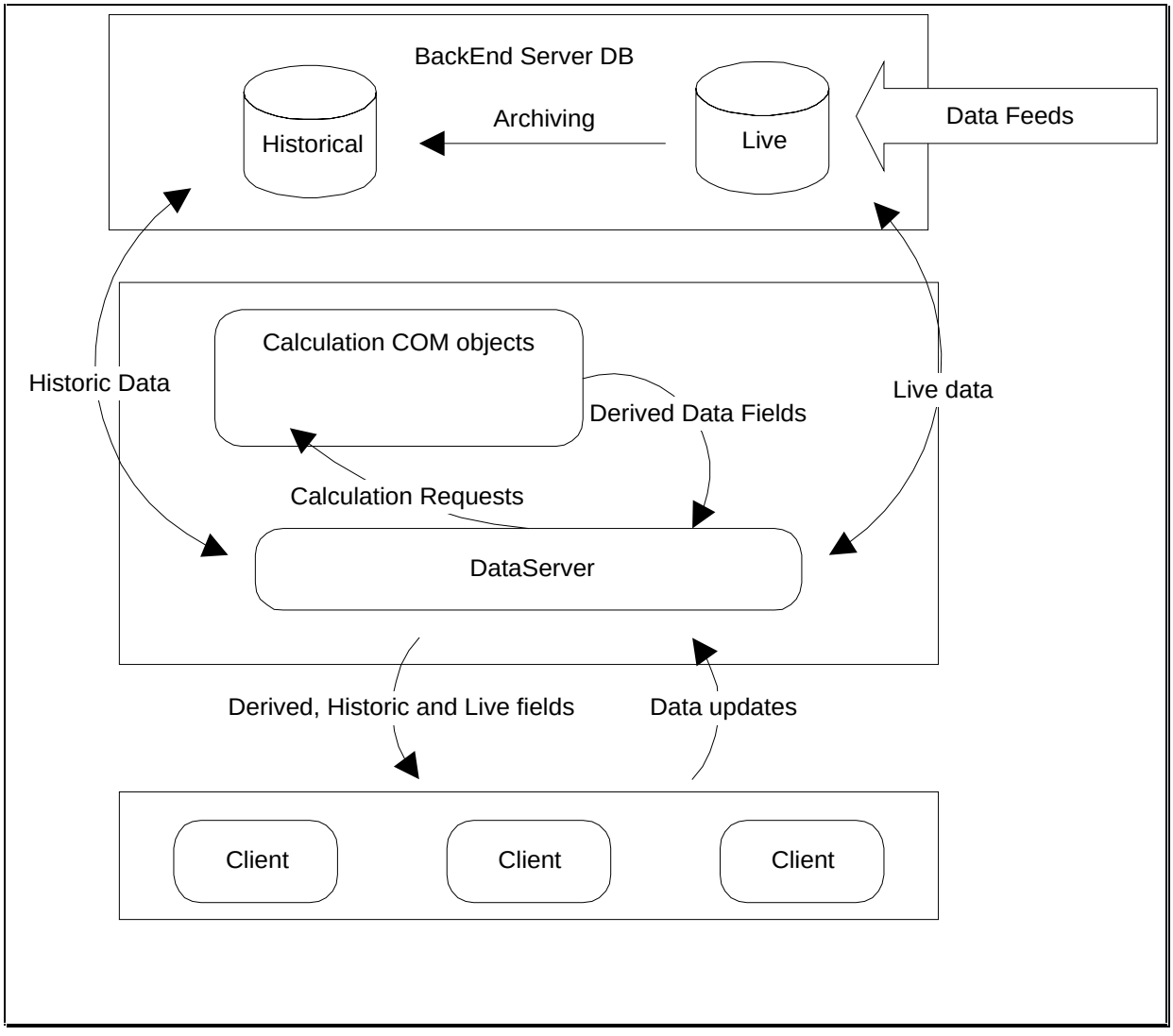


**Figure 2 Inter-module Data Flow Diagram**

## 6. The Objects

The system designed is split into separate modules; BackEnd Database Server, Data Server, Calculation Engines and Clients. Each of these modules contains one or more objects that exposes the functionality of the module to the other modules via a COM interface. These are listed in Table 1.

| Module | Object | Description |
|---|---|---|
| BackEnd Database Server | | |
| | ConnectionManager | A connection point for data servers to connect to for advises of data changes. |
| | DataAdvisor | Starts and stops advises being sent through to the Data Server. |
| | DBNotify | Database uses to notify of changes to the data. |
| Data Server | | |
| | ConnectionManager | A connection point for clients to connect to for advises of data changes. |
| | ClientManager | Exposes an automation interface for clients. |
| | DataStore | Supports an automation interface for feeding data and requesting the latest values. |
| | ItemSink | Exposes a sink interface for the ConnectionManager object of the BackEnd Database Server to advise of changes. |
| Calculation Engines | | |
| | ItemSink | Exposes a sink interface for the ConnectionManager object of the Data Server to advise of changes. |
| Client | | |
| | ItemSink | Exposes a sink interface for the ConnectionManager object of the Data Server to advise of changes. |

Table 1. Modules and Objects

## 7. Object Relationships and Interfaces

### *BackEnd Server*

The BackEnd Server Module comprises the Live and Historic Databases, plus the handling of data feeds. The Database Manager handles the interface between the databases and the Data Server machine.

Database Manager

The data manager handles routing of data to the database and to the Data Server. Its function is to ensure that the Data Server is only informed of changed values, by keeping a cache of most recent updates. In addition, it decides, according to the source of the data item whether or not to inform the Data Server of the new addition to the database.

### *Data Server*

The Data Server interfaces to the rest of the system are shown in figure 3.

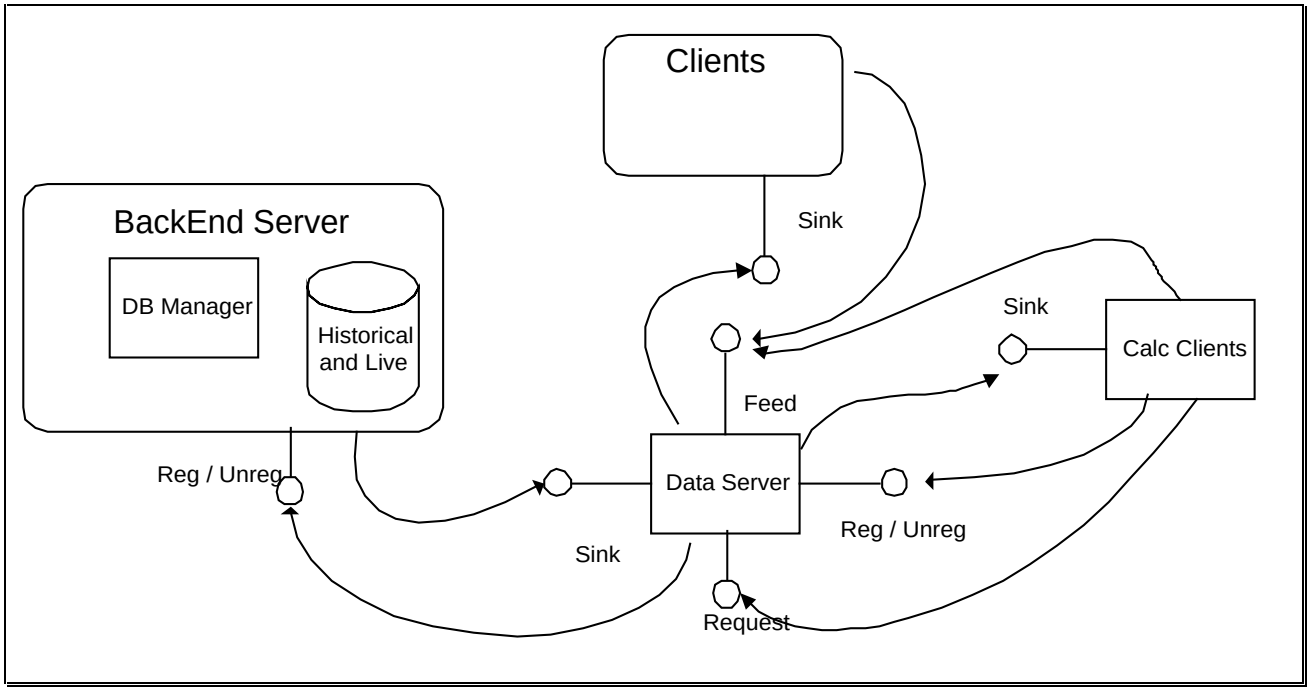


**Figure 3 Inter component Interface relationships**

The Back End Server and Data Server use the same mechanism for pumping data out to the Client programs, and the Calculation Engines, both of which must export the following DCOM interface:

**ItemSink:** Data Advise notifications are sent to this interface every time the Server is aware of a new piece of data. In addition, any broadcast messages are sent through this interface.

The Data Server exports three DCOM interfaces:
**Request:** This is a request for data from the Data Server (handled either by the cache, or by a pass-through request to the database). Each request spawns a new thread, which handles the request, pumps the data back to the caller, and then dies.
**Register / Unregister Client:** Each Client or calculation engine must register with the Data Server in order to receive notifications of changed data.
**Feed:** The feed interface is used to add data to the Data Server, either from a feed, from the result of a calculation, or from a client program.

In addition to the DCOM interfaces, the DataServer links directly into the databases in order to perform ODBC queries to fulfil data requests.

### *Calculation Engines*

The Data Server treats the calculation engines as clients. Each one receives data updates from the server, decides whether to calculate, and pumps results back as new data items. Each calculation performed by the system is a separate Calculation Engine.

## 8. Features of Implementation

The system has been designed with flexibility in mind, each component could run on a different machine or they could all run on one machine. The practical up shot of this is to allow the load to be spread. In practice the system created has the BackEnd server and the feeds running on the same machine as the database, and the Data Server running on a machine with the calculation engines, the clients are then all be running on

separate machines. In this way one machine becomes dedicated to data collection, one to deriving data and each user has their own client machine.
The entire system has been implemented using C++ and COM. The database we used was Microsoft's SQL server, this provides a mechanism for calling COM Automation functions within a stored procedure, which allowed us to communicate from the database into the BackEnd Server. It was easily tested by writing Visual Basic test harness code to test each component individually and groups of components together.

## 9. Future Work

This system is one of the first implementations of a real time distributed objects system, we at Cognitech strongly believe that we shall be creating more such systems in the future and will be improving on the system presented here.
Work is already under way to create the next generation of more generic real time systems using distributed objects.
With feedback provided from our clients we will be continually enhancing the architecture and working on new ideas to provide faster response times, further stability and create neater solutions.

# BIBLIOGRAPHY


[1] E. Douglas Jensen, 'A New Prospect for Real-Time DCOM?', May 1998 http://www.real-time.org/no_frames/noteworthy/rt-dcom.htm

[2] Cooling, J E. *Software Design for Real-time Systems.* International Thomson Computer Press, 1995.

[3] Brockschmidt, Kraig. *Inside OLE* Microsoft Press, 1995.

[4] WOSA Extensions for Real Time Market Data (WOSA/XRT) Design Specification. Open Market Data Council For Windows

[5] Neil Roodyn, Wolfgang Emmerich, An Architectural Style for Multiple Real-Time Data Feeds, 1999

[6] Pressman, *Software Engineering, A Practitioners Approach*, McGraw Hill

[7] *OLE2 Programmer's Reference Vols One & Two.* Microsoft Press, 1993.

[8] Crittenden, John. *Information Overload* Feature Articles PacificByte 1996

[9] Real-Time Systems and Microsoft Windows NT, Microsoft Corporation, 1995

[10] Denning, A. 'ActiveX Controls Inside Out', Microsoft Press

[11] Kruglinski, David. 'Inside Visual C++', Microsoft Press

[12] McCarthy, Jim. 'Dynamics of Software Development', Microsoft Press

[13] Maguire, Steve. 'Writing Solid Code', Microsoft Press

[14] Guy Eddon & Henry Eddon. *Inside Distributed COM ,* Microsoft Press, 1998

[15] Don Box. *Essential COM*, Addison Wesley, 1998

[16] Jonathan Pinnock. *Professional DCOM Application Development*, WROX, 1998.