Object Caching in a Transactional, Object-Relational CORBA Environment

Thomas Sandholm

[1]Department of Computer and Systems Sciences
University of Stockholm

October 1998

Abstract

The OMG's CORBA and CORBA services like the OTS are a technology standard that enable the building of transactional systems running in distributed and heterogeneous environments. In large-scale CORBA systems that integrate relational databases, however, careful attention must be paid to network traffic and the number of I/O-operations (like database access) performed, as these can degrade system performance significantly. Caching is a well-known concept to improve performance in e.g. database systems. Caching in a transactional object-relational CORBA environment has, however, not been studied in the literature so far.

This thesis investigates concepts to improve performance and reliability in large-scale CORBA systems. An object caching strategy for transactional, object-relational CORBA systems is developed. It employs distributed transaction management to replicate data, and asynchronous multicast notifications to update caches that are distributed to load balanced servers. The caching strategy is implemented and tested using a case study with real-world assumptions, and described as a generic, software architectural abstraction that can be reused in different CORBA system developments. Using the caching strategy proposed, the performance of the system can be drastically increased, and system scalability and reliability be well improved.

[1] This thesis corresponds to the effort of twenty full-time working weeks.

Contents

1 Introduction

This thesis was developed within a cooperation between IONA Technologies Dublin and Berlin University of Technology (TU Berlin), as part of the project CORBA Object Transaction Monitor Experimentation. In this project, advanced software design and implementation issues of building CORBA OTM applications are studied. The thesis describes the development of an object caching strategy using IONA's OrbixOTM environment, and using software architectural modeling and abstraction techniques developed at TU Berlin.

This chapter discusses as introduction, how the concept of CORBA OTM, and the research field of software architecture, help to develop reliable and performant large-scale software systems, and how they were combined in this work.

1.1 Background

Today's software is developed on various platforms using different operating systems, programming languages, and development tools to best meet the system requirements. Further, systems are expensive to develop, and play an important role in the day-to-day business in many companies. Typically, new applications therefore have to be integrated with existing *legacy systems.* In such heterogeneous environments, concepts are needed that support exchange of information, or concurrent access to shared data, while assuring integrity of data and performant computation.

Standards like the *Object Management Group's* (OMG*) Common Object Request Broker Architecture* (CORBA) address distributed computing in heterogeneous environments [OMG 98b, Vinoski 97]. In CORBA, *middleware* functionality and features are specified, and mappings and common interfaces for interoperation between diverse software are defined.

Protecting data integrity and constructing reliable (distributed) applications is, on the other hand, typically done by use of the concept of transactions. *Transaction Processing Monitors* (TP Monitors) as common in traditional client/server systems e.g. guarantee *ACID*-properties (Atomicity, Consistency, Isolation and Durability) to all programs that run under its protection, thus provide mechanisms to begin, commit, and rollback transactional requests [Gray & Reuter 93, Bernstein & Newcomer 97].

Common TP Monitors are, however, not designed to manage transactions in large-scale CORBA systems. In order to support transactions in heterogeneous CORBA environments, CORBA and TP Monitor concepts can be combined, as exemplified by IONA's Object Transaction Monitor (OTM) developments [IONA 98e].

Because of the combination of many different technologies, OTM systems are intrinsically complex. The field of *software architecture* aims at supporting the design of such complex systems by describing reusable software structures using architectural abstractions of *components* and *connectors* (component interactions) [Garlan & Shaw 96, Bass et al 98].

1.2 Problem Statement

One common reason for using transactions is to manage persistence of data. Today, this is typically done by a relational database. When integrating relational databases into an OTM

system, issues regarding mapping of persistent entities, scalability, and management of distributed data must be considered, while assuring performance and reliability [Orfali & Harkey 98, Shussel 96].

In large-scale distributed systems, network communication and database access degrade performance significantly. Performance has traditionally been achieved by using caches, e.g. in the development of operating systems. Further, *object caching* is a commonly used technique in object-oriented databases [Versant 98]. Object caching in a transactional, object-relational CORBA environment has, however, not been studied in the literature so far. The problem addressed in this thesis is - how to design object caching in a CORBA system integrated with relational databases, while considering reliability and scalability issues.

1.3 Objective

The objective of this work is to investigate, develop, and test an object caching strategy improving reliability, scalability, and performance in a CORBA system, and to capture the results in a design pattern.

1.4 Approach and Thesis Outline

The work described consists of two main parts: transaction demonstrator development, and software architectural design. Both parts comprise theoretical studies (part I, chapter 2-4), as well as practical experiments (part II, chapter 5-7).

1.4.1 Transaction Demonstrator Development

This part, done at IONA Technologies in Dublin, focuses on designing and implementing a distributed transaction demonstrator for CORBA environments. The system, originally developed for the IONA World trade-show held in Boston in March 1998, is further extended with an object caching feature.

In order to choose a caching strategy, theoretical studies are carried out before. Distributed objects and transactions (chapter 2) are investigated to ensure reliability of the cache. Studies of *Object-relational mapping* (section 3.1) help understand performance and flexibility issues related to *impedance mismatch* between objects and relational structures. Scalability, availability and performance issues are considered by investigating *load balancing* (section 3.2), *replication* (section 3.3), and *caching* (section 3.4).

The practical experiments begin with evaluating and testing event manager, and database access tools to be used with a distributed transaction tool (OrbixOTS) in the implementation (chapter 5). Thereafter an implementation with the selected tool-chain is carried out. The final implementation is tested by simulating a “real-world” scenario. The tests focus on measuring the influence of caching on performance and throughput (chapter 6).

1.4.2 Software Architectural Design

This part, carried out at TU Berlin, concerns object-oriented and software architectural modeling. Software architectural concepts are investigated (chapter 4). These concepts are then applied using traditional object-oriented modeling, as well as the software architectural modeling approach developed at TU Berlin in the research group Computation and Information Structures (CIS) [Tai 98b]. As a final step, architectural abstractions are made to form a generic design of the caching implementation (chapter 7).

1.5 Contributions

This work contributes to the design and development of CORBA OTM systems in two ways: (1) it shows how to extend an existing large-scale OTM system with a caching-strategy that improves performance and throughput, while assuring data consistency; (2) it provides a reusable connector abstraction [Tai 98a] for designing object caching of transactionally replicated data in an OTM environment. In practice, the work is currently used at IONA for OTM demonstrations, and it will contribute to a “design-handbook” of ORB-based systems developed as part of the research at TU/CIS Berlin.

PART I - CONCEPTS

The first part of this thesis introduces concepts for designing and implementing large-scale transactional systems. Chapter 2 presents the *CORBA* standard, and two *CORBAservices* for managing transactions and events. Chapter 3 surveys techniques for developing scalable and performant distributed systems using persistent data. In Chapter 4, the basic concepts of software architecture are introduced.

2 Distributed Object Systems and Transactions

In this chapter the basics of the *CORBA* (Common Object Request Broker) standard and two CORBA services; the *Object Transaction Service* (OTS) and the *Event Service* are explained. These two services can be deployed in any CORBA system and are commonly used to ensure reliability in distributed systems. Further, *TP Monitors* and *Object Transaction Monitors*, which provide similar functionality, are discussed.

2.1 CORBA

The CORBA standard has been developed by the (OMG) Object Management Group, which was established in 1989. OMG is today a consortium of almost 700 software companies and about 100 universities [OMG 98a]. Companies propose standards to the group and then the proposal goes through a process where the OMG members vote for or against standard adoption. A significant part of this process is that all standards finally adopted by the OMG must have gone through a "proof of concept". This means that the company or companies proposing the standard must have an implementation with which they can prove that the technique is versatile.

2.1.1 What is CORBA?

The CORBA standard is the core standard of all OMG standards. It consists of specifications for an Object Request Broker (ORB). An ORB is a software bus through which distributed objects communicate. The clients talk to the server objects by first "plugging into" the bus. Thereafter, they can theoretically talk to any objects residing in servers also plugged into the bus. The main objective of the bus is to encapsulate how the communication between clients and servers is realized. Clients and servers can thereby communicate without having to deal with mismatches caused by different programming languages or operating systems.

This functionality is often referred to as *middleware*, since the ORB operates above the level of implementation techniques, but below the level of applications in a tier between clients and servers. Apart from this basic functionality the current CORBA specification also addresses issues like *inter-ORB* operability [OMG 98b].

2.1.2 Why CORBA?

CORBA was introduced to standardize the development and deployment of applications operating in distributed heterogeneous environments [Vinoski 97]. The main idea is to standardize how clients and servers interoperate in a generic and object-oriented way, i.e. how distributed objects communicate. Using an ORB, new software can easily be added to the system (plugged into the bus). Further, old legacy software can be integrated into newer systems in a flexible way because of the language independence. The CORBA standard was developed with the component based software development paradigm in mind. Software is developed in components or packages with predefined external interfaces. Hiding the implementation details, these components should be easy to combine and "plug in" anywhere to get the desired functionality. A distributed CORBA object is a component in that sense [Orfali et al 96]. By developing components with clearly defined interfaces the historically expensive integration phase of software development can be eliminated [Baker 97].

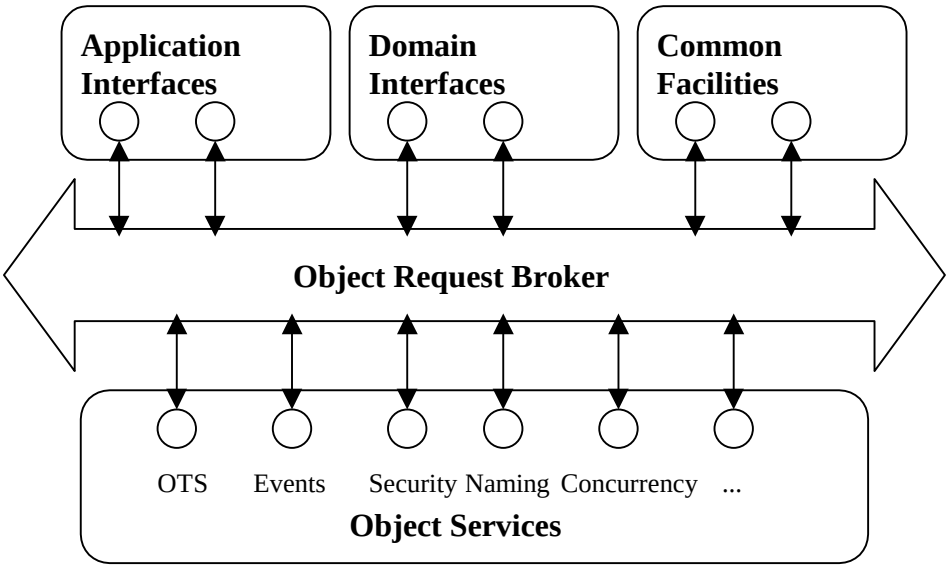


Figure 2.1 The OMG Object Management Architecture

2.1.3 Object Management Architecture

The ORB fits in to a higher level architecture defined by OMG called the *Object Management Architecture* (OMA) shown in figure 2.1.

Four different kinds of CORBA objects can be plugged into the ORB: application specific objects, standardized domain objects (e.g. for the medical domain), standardized common facility objects (e.g. system management), and finally the generic CORBA services that can be deployed in any CORBA system (e.g. OTS, Events). All CORBA objects must be defined with external interfaces in order to be connected to the ORB [OMG 97b].

2.1.4 CORBA IDL and ROI

The definition of interfaces is a key behind interoperability and language independence. This is done with the *interface definition language* (IDL) specified by the OMG. IDL is a declarative language with a syntax similar to the one of C++, however, IDL only specifies behavior not implementation. Languages that can be used for implementing CORBA objects must have corresponding IDL mappings. CORBA (revision 2.2 February 1998) currently specifies mappings for C++, Smalltalk, COBOL, Ada, and Java [OMG 98b].

The IDL specifications are used to generate code that plugs client and server code into the bus. On the client side, this code is called stub code, and on the server side, it is called skeleton code [Vinoski 97]. The client stub marshals the request and sends it through the ORB to the server. The server skeleton unmarshals the request, and sends it to the so called target object (the implementation of the object that the client wants to access) within the server process. When the target object has processed the request, the return value is sent to the skeleton. The skeleton marshals the value, and sends it through the ORB back to the client. On the client side, the stub now unmarshals the reply, and passes it on to the client. The entire process is called a [1]*remote object invocation* (ROI) and is performed transparently to the client and server code.

[1] The process is sometimes called remote method invocation (RMI). This is, however, also the term for the distributed object feature built-in in Java , and it was therefore avoided.

2.1.5 CORBA Implementations

A wide variety of organizations and companies have implemented the CORBA standard. There are today two major commercial ORBs: Orbix from IONA, and VisiBroker from Visigenic (owned by Inprise). Other vendors are ICL with DAIS (Distributed Application Integration System), and Expersoft with CORBAplus. IONA currently has products supporting the IDL to C++, Java, and COBOL mappings. Visigenic has products that support the C++, and Java mappings. [OMG 98c, Inprise 98, IONA 98a]

2.2 CORBA Services

As described in the previous section, there are some generic services that can be used in any CORBA environment. In this section, the *CORBA Transaction* and *Event services* are presented. These services play an important role by supporting reliability in the implementations presented in chapter 6.

2.2.1 Transactions

The *Object Transaction Service* (OTS) was specified to enable transaction processing in CORBA environments. This, for instance, involves management of distributed resources like databases.

What is a Transaction?
A transaction is defined to be a series of operations that can be performed as one unit. Transaction processing is the basis for reliable processing. By using transactions the client is guaranteed:

- that all or none of the operations within the transaction will be performed, ***a****tomicity;*
- that the transaction will bring the system into a ***c****onsistent* state;
- that all work inside of a transaction is ***i****solated* from other transactions as long as the work is not committed;
- and finally, that when the transaction completes successfully (commits), its modifications to the state are ***d****urable*, that is survive failures.

These guarantees are called the *ACID* properties of a transaction [Gray & Reuter 93].

OTS Constituents
The OTS defines three major components, and the interactions between these. The *transaction originator* who is responsible for beginning, committing, and rolling back transactions; the *recoverable server* that is responsible for connecting resources to the transaction; and the *transaction service* (TS) that is responsible for keeping the transactions atomic and durable.[OMG 97a]

Resource Integration
Resources, typically databases, can be integrated into OTS transactions in two ways: as CORBA [1]`Resource` objects, or by using the X/Open DTP standard XA interface [OpenGroup 92]. When using the CORBA `Resource` approach, the atomicity and recovery of a transaction typically has to be implemented explicitly on a CORBA level by the programmer. When using the XA approach, this functionality is normally supplied by the resource vendor. The recoverable servers register new resources with the transactions. When

[1] The courier font is used for denoting correspondences to interface or implementation entities.

it is time for a transaction to commit, the TS component calls these resources in order to carry out the two phase commit protocol described below.

Two Phase Commit

How does OTS assure atomicity and durability of a distributed transaction; i.e. a transaction with resources residing on several nodes in a network? This is done by driving the *two phase commit protocol* (2PC). The 2PC involves the following steps.

(1) The transaction originator calls commit after having completed its work within a transaction.
(2) The *transaction coordinator* (a part of the TS responsible for coordinating several resources participating in a transaction) asks all resources registered with the transaction whether they can commit. The resources then respond either with a "yes" or "no" vote. This phase is called the *prepare phase*.
(3) When the TS has got all the votes, and all votes were votes to commit, it calls all resources again that they can commit. If one or more resources voted to roll back, the TS component sends a request to all resources to roll back their work. When the resources finally have committed or rolled back their work, they send back a message that they are done to the TS.
(4) When the TS has got all done messages from the resources, it can forget the transaction.

Distributed Transactions

In order for distributed objects to participate in the same transaction, transactional information has to be exchanged. This is achieved by passing a so called *transaction context* explicitly or implicitly with every request that is part of a transaction. The TS component in the process that receives the request transparently associates the work performed in that process with the transaction identified by the transaction context. This is called *interpositioning*. A recoverable server that has been interpositioned will be called by the TS when it is time to carry out the 2PC.

2.2.2 Events

The second CORBA service to be presented is the Event service. Normally when invoking on distributed objects in CORBA, the clients send their requests to a specific server, and wait for the reply before carrying on. This message model is called *synchronous*. The CORBA Event service messaging model differs in two ways from this scenario. Firstly, the clients don't have to wait for a response, i.e. the message is sent *asynchronously*. Secondly, the clients and servers are only indirectly coupled. Multiple clients can be connected to multiple servers, without any of them knowing the identity of the others.

Why do we need events?

So, why is this messaging model so useful? Most applications, especially GUI programs are event driven, i.e. when a certain event occurs, a predefined action should be performed. For flexibility, it should be possible to add any number of event suppliers, and any number of event consumers to the application. Most of today's applications are developed in a modular way. There might for instance be one module for database logic, one for business logic, and one for GUI. The idea behind this structure is to keep the modules independent of one another. This would not be possible without a messaging model using loose coupling between event suppliers and consumers.

CORBA Event Service Constituents

The CORBA Event service specifies three basic components: The `Suppliers` that generate events, the `Consumers` that can receive events and the `Channel` that is the only

component that has knowledge about the different participants in a communication. The `Suppliers` register themselves with a specific `Channel` (many suppliers can register themselves with the same `Channel`). The `Consumers` can get a reference to the same `Channel` object and register themselves as `Consumers` of the `Channel`. When one `Supplier` sends an event to the `Channel` all `Consumers` will receive a message. [OMG 97a]

Models of Interaction
There are two different kinds of messaging models: the push model, and the pull model. In the push model the events are pushed to their receivers, and in the pull model the receivers wait for events to occur (or pull events periodically). Both the push and the pull models can be applied independently of each other, both on the supplier side and on the consumer side of the `Channel`. There could, for instance, be one `Supplier` sending events to a `Channel` (pushing `Supplier`), another `Supplier` waiting for clients to pull events (pulled `Supplier`), one `Consumer` that gets notified of events (a pushed `Consumer`), and one `Consumer` that waits for events to occur, or makes explicit queries for events in a periodic manner (a pulling `Consumer`), all connected to the same `Channel` and participating in the same interactions.

2.3 Transaction Processing Monitors

In a distributed transactional system, some mechanism for processing multiple client requests in an efficient way, while maintaining the transactional properties (ACID) is required in order to achieve reliability. *Transaction Processing Monitors* (TP Monitors) are software developed for this purpose. A TP Monitor works like a workflow manager or router between the clients and the server where the transactional program resides. All access to data resources goes through this server, but is coordinated by the TP Monitor. The software and hardware built in this architecture (see figure 2.2) comprise a *Transaction Processing System* (TP System) [Bernstein & Newcomer 97].

In a TP System, the *resource manager* (RM), e.g. a database manager, is kept separated from the *transaction manager* (TM), the part of the system that, for instance, is responsible for driving the two phase commit protocol. These two components have different responsibilities for helping the application programmers develop reliable systems. The TM assures atomicity and durability (logging of transaction information to enable recovery). Keeping the data consistent lies on the responsibility of the application programmer, and to some extent on the RM (e.g. database triggers). The RM further assures durability (before and after images of data) and isolation (no uncommitted data is visible).

One of the most important tasks of a TP Monitor working in a distributed environment is to carry out the two phase commit protocol. This, for instance, means that recovery has to be supported by maintaining logs. A TP Monitor also has to keep track of all resource managers that the application talks to in order to complete the 2PC protocol. Other common features offered by TP Monitors are: load balancing, fault-tolerance, and security [Gray & Reuter 93].

2.4 Object Transaction Monitors

An *Object Transaction Monitor* (OTM) combines the CORBA and TP Monitor concepts. The main responsibility of an OTM is to manage the server side objects transparently to clients.

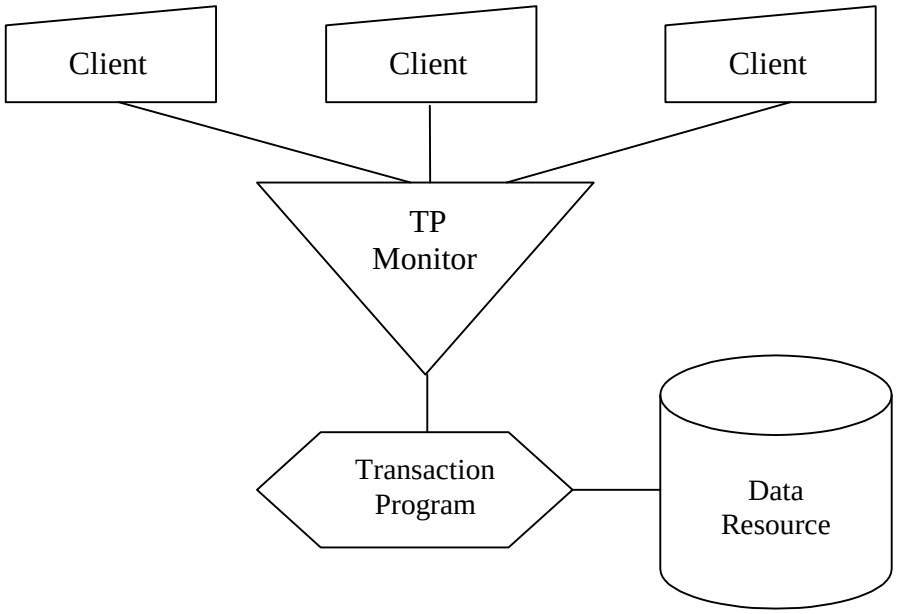


Figure 2.2: TP System Architecture

Automatic object management is crucial in large-scale systems with a large number of server objects.

Typically an OTM would: activate and deactivate components (distributed objects), coordinate distributed transactions, notify components of events, and automatically manage the state of components [Orfali & Harkey 98]. The OTM should thereby introduce scalability, load balancing, fault-tolerance, security, and persistence into a CORBA environment. Both TP Monitor and ORB vendors currently move their products towards the OTM framework. One example of this is the cooperation between Transarc (TP Monitor vendor) and IONA resulting in the product OrbixOTM. OrbixOTM for currently offers: object naming, event notification, distributed transactions, security, and system management on top of the Orbix ORB [IONA 98e].

2.5 Summary

In this chapter, the basics of the CORBA standard, and how the standard solves the problems faced when integrating heterogeneous systems were presented. The basic idea is to define interfaces between components or distributed objects in a standardized and language independent way. An object bus or ORB takes care of the communication between these objects. This architecture makes it possible to extend the system, and incorporate legacy systems in a straightforward and flexible way. The ORB functionality is often not sufficient in a large-scale distributed systems. Two commonly used services are the Event service and the Object Transaction Service (OTS). These services offer asynchronous messaging and distributed transaction processing respectively.

TP Monitors, used for run time execution of programs (i.e. routing) to improve scalability and reliability, were discussed. The OTM could be seen as a way to introduce this functionality in the CORBA world. This chapter was mainly concerned with reliability of OTM systems. In the next chapter, issues regarding persistence, scalability, and performance in such environments are discussed in some more detail.

3 Object Persistence and Caching

In this chapter, the theories behind *object-relational mapping*, *load balancing*, *replication*, and *caching* are presented and discussed. How to map the object model of a system into a relational database is often of vital importance to the performance. Performance and scalability are main goals for all these techniques. Caching, replication, and load balancing are closely related and often applied in large-scale systems in conjunction with each other. These sections give a basic idea of general design issues and common approaches in order to follow the discussions in chapter 5 and 6 (Technical Solutions and Case Study).

3.1 Object-Relational Mapping

Object-relational mapping has become increasingly important as object-oriented languages and tools are becoming more frequently used in companies, whilst relational databases for a long time have been dominating the database field. In this section, reasons for using a consistent mapping policy, and aspects that have to be considered when designing and implementing object-relational mapping are discussed. Further, common mapping approaches, different ways of accessing the database, CORBA to persistent objects mapping, and an example of a mapping tool called Persistence PowerTier are presented.

3.1.1 Why is Object-Relational mapping needed?

Object-oriented languages and relational databases have many advantages and are today de facto standard in their respective fields. OO-languages offer encapsulation, polymorphism and inheritance, which yields a natural mapping to the real-world domain. Relational databases build on a simple concept (the relational model), and also offer techniques like concurrency and replication off the shelf. The main reason why object-relational mapping is needed is hence that these two concepts are commonly used, and developers frequently face the problem of integrating an existing relational database into an object-oriented application.

Performance is an important reasons for using a suitable object-relational mapping. The main goal is to minimize database queries and maximize in-memory object queries. For this reason, an object-relational mapping approach often goes hand in hand with a caching approach.

A loose coupling between the application logic and the data schema supports system evolution. Additionally, general mapping solutions minimize error prone repetitive work. Flexibility and ease of integration are thus further reasons for considering an object-relational mapping policy. This is further discussed in section 3.1.4, accessing the database.

3.1.2 Design Issues

How the database schema is represented in the object model, or vice versa, relies heavily on how the persistent data is managed, e.g., whether there are only a fixed number of queries, or whether queries are constructed ad hoc in a flexible manner. If many ad hoc queries are used, a one-to-one mapping between classes and database tables would probably be unsuitable [Leser et al 98]. Similarly, whether the queries are closely connected to objects of one class only, or whether they involve traversing an object-graph must also be taken into consideration when choosing a mapping approach.

In some cases denormalized tables and redundant data can be accepted to optimize access time [ONTOS 98, Agarwal & Keller 98]. There is, however, always a trade-off between flexibility and performance. Flexibility would be increased if the tables were kept

normalized. For ease of maintenance, one single mapping approach could be applied for the entire system, which may be a reason for choosing a more generic and flexible approach.

3.1.3 Mapping Approaches

There are different ways to map object-oriented concepts into relational concepts. Below the most common approaches for mapping classes to relational concepts, and how the problem with object identities can be solved are discussed.

One-to-One
The mapping of classes into relational tables can be very simple. If all attributes of a class are of basic types like integer and string, a one-to-one mapping could be used[1]. One class corresponds to one table in the database and the attributes of the class correspond to columns in the table. Further, tuples or rows of the table represent the objects.

One-to-Many and Many-to-Many
If a class has attributes that are collections of basic types, or represent many-to-many object associations, a one-to-one mapping is not sufficient. Normally a complex class, e.g. a composition of several classes, is represented by multiple database tables. For performance reasons the opposite may also be the case, i.e. multiple classes could be represented in one denormalized database table. The relations on an object level are normally represented by foreign keys in the relational database.

Inheritance
Mapping inheritance is not as straightforward as the other mappings. A table in the database may represent an entire class hierarchy. Further, all the classes in the hierarchy may be represented by a table each, possibly with foreign keys to other tables to represent the hierarchy. A third approach is to only let the lowest level classes or the leaves of the tree be represented by database tables. This approach would lead to redundant data, but could still be an alternative to gain performance advantages.

OID vs. Primary Keys
The notion of uniqueness is quite different in the two worlds. The relational databases use a value-based approach by specifying some columns as primary keys to assure uniqueness within a table. In the object-oriented world uniqueness is kept orthogonal to the data itself by using object ids. There are basically two solutions to this mismatch. First, an object id generated at object creation time could be stored in one column for each database table [Ambler 98]. Second, a mapping algorithm between primary keys and the object ids could be used [Fahl & Risch 97]. The first approach is the easiest one to implement but requires database changes, which may not be possible to do when integrating legacy databases. The second approach is harder to implement. In this approach the mapping, for example, becomes invalid if primary keys are reused.

3.1.4 Accessing the Database

There are basically three different approaches for accessing databases from an object-oriented programming environment: direct access, wrapped access, and tool-based access. They are discussed in turn below.

[1] If the attributes are objects that represent one-to-one associations, then this could also be seen as a one-to-one mapping.

Direct Access

The easiest approach to implement, and possibly the most commonly used, is to issue direct calls to the database via embedded SQL directly from the client code. This is not a very suitable approach for a larger system, though, because no code-reuse can be done, i.e. a new mapping basically has to be implemented for each call. If the mapping is simple, and there is a limited amount of persistent objects in the system, this approach may be acceptable, however.

Wrapped Access

Embedded SQL code is not object-oriented, and the mapping of return values from SQL calls into objects can be repetitive and error prone. Therefore some tools offer the programmer the possibility to issue SQL calls by using objects that wrap in the database calls. The object-relational mapping is still limited in this case, as the programmer is always aware of the underlying database structure.

Tool-based Mapping Access

The most sophisticated way of accessing databases through an object-oriented language is to use a mapping tool that generates classes and access methods from relational database schemata, or generate schemata from object models. The mapping, in this case, is static but easy to maintain because of the code generation. Simple queries and updates of attributes can be made from the client code by using the methods of the generated classes. For greater flexibility, SQL commands can normally be issued as well. In theory, though, the database structure could be unknown to the clients.

3.1.5 CORBA to Persistent Objects Mapping

So far, only the mapping between OO-language classes (representing persistent objects) and relational database constructs has been discussed. There is, however, a second mapping that has to be considered in a distributed object system integrated with a database: mapping distributed objects specified by IDL to classes dealing with persistence.

Granularity Problem

Accessing a distributed object is very costly with regards to network traffic and server resources. For an object to be available for remote access, it has to be registered with the ORB, and has to be linked with skeleton code to dispatch incoming requests. Because of the high resource demand, and the extra work needed to access CORBA objects, these are normally coarse grained. Fine-grained implementation details are thereby hidden from the clients by providing a high-level external interface. Persistent objects having their counterparts in database tables and rows are normally much finer grained though. The basic problem is how to access large collections of small database objects in an efficient way through the ORB transparently to the clients. This issue is currently addressed in the new CORBA *Portable Object Adapter* (POA) specification [Schmidt & Vinoski 97, OMG 98b], the proposal for the new CORBA *Persistent State Service* (PSS) [IONA et al 98], and in the standard specifications from the object database community's counterpart to OMG; ODMG (Object Data Management Group) [Cattel & Barry 97]. One solution to this problem is to let the object implementers decide which subset of database objects that should be accessed directly as CORBA objects, and which objects that should be accessed through higher level delegating CORBA objects. Another solution is to provide an object caching mechanism to improve the performance. These two solutions ideally should be used together, but in an orthogonal way to ensure flexibility. They should further be transparent to the clients.

Object References Problem

CORBA objects are identified by an *interoperable object reference* (IOR), not necessarily an object id as expected in a non-distributed environment. Being able to tell which objects are the same is crucial for a caching implementation, and for mapping the objects to primary keys in the database. This is not always the case with CORBA IORs. This problem is addressed in the new POA specification [OMG 98b] by stating that a part of the IOR must embed an object id unique to the POA. IONA's Object Database Adapter Framework [IONA 97a] makes use of IONA proprietary markers, which are embedded in a similar way, for mapping unique objects to unique database ids.

State Representation

A CORBA object contains a lot of information that is used at runtime and for distributed purposes only. Further, in IDL, we specify the behavior of objects. These circumstances lead to the fact that only certain parts of the distributed objects should be made persistent, i.e. represent the state. So, which components of a CORBA object represent state? This is currently dealt with in a standard proposal to the OMG for passing objects by value [1][IONA et al 98]. It contains an extension of the IDL so that the state of an object can be defined explicitly. This is for instance useful for caching purposes; the local object cache can first request the remote object by value and then store its state locally. (In chapter 6, this process is called localize. The case study implementation of localize would have been simplified if the object-by-value semantics could have been used.)

Client or Server Controlled Persistence?

When integrating persistent objects into a CORBA system, two general approaches can be taken: client transparent persistency or client exposed persistency. In the client transparent approach, the ORB is responsible for fetching the persistent objects into memory when clients start accessing them, and flush them to the database when the clients are finished. In some cases, a more sophisticated caching mechanism may have to be implemented on the client side. The clients then need to be aware of the persistence details of a CORBA object. The IDL extension of state representation, for instance, provides the clients with such information. For flexibility and maintainability clients should not directly control server side persistence, though.

3.1.6 Persistence PowerTier Implementation

With Persistence PowerTier, developers can describe their object model in a graphical tool. Then a database schema, as well as classes that can be used to access the database are generated. Classes and their attributes are specified similar to how database tables are modeled. In addition to the class modeling, a modeling tool for relations is also available. Cardinalities, access operations for the relation, and foreign keys can be specified with this tool. The modeling is data oriented, and only the persistent objects should be modeled. There is, however, an extension to the product that can be used to transform UML diagrams modeled in Rational Rose to the Persistence format.

One-to-one and one-to-many relations between objects are supported by generating or specifying foreign keys with the relations tool. A concrete class modeled in Persistence can only be represented in one single database table. This table can, however, be changed at runtime and many concrete classes can be mapped to it. Further, Persistence supports inheritance by only representing the leaves as database tables. Only the leaves are represented as concrete classes.

[1] The reason for not referencing the specification itself is that it is not yet publicly available for non OMG members.

The mapping is performed in a tier between the database server and the client to gain extra performance advantages. This tier also provides an object cache discussed in greater detail in section 3.4.8.

An extension called DOCK (Distributed Object Connectivity Kit) provides a CORBA to persistent objects mapping. This mapping is a one-to-one mapping, though, with the exception that IDL does not have to be generated for some classes that are defined. The granularity problem discussed earlier, however, is not addressed further than providing an object cache. [Persistence 98]

3.2 Load Balancing

In this section server selection as a special form of load balancing is discussed. Server selection refers to selecting the target server from a group of identical servers in order to balance the load.

An analysis of how performance problems of large scale distributed systems can be solved by using a proper server load balancing strategy and scheme is done. Scheme is defined as the implementation of a strategy. The main goals of the use of load balancing, and what implications that must be dealt with when trying to achieve these goals are presented. Different approaches that can be chosen to resolve these difficulties are compared. Further, some strategies are presented and related concerning implementation and use in different contexts. Thereafter, an example of a load balancing implementation is presented; the load balancing feature of the OrbixNames product from IONA Technologies.

3.2.1 What Is Load Balancing?

Load balancing deals with the distribution of requests among servers. The servers can all reside on a single host or be distributed within a group of hosts. A simple server selection load balancing scenario is depicted in figure 3.1. We focus on this form of load balancing because selecting servers is a core part of the CORBA architecture and is a factor that easily can be tailored by the CORBA developer.

Many load balancing algorithms for distributing the load between different hosts on an operating system level have been investigated in various research projects. These algorithms are of minor interest in a CORBA context, as they are applicable on another level of granularity, and are not discussed here.

The selection of servers can be either static or dynamic. Static server selection means that clients are always given the same server to invoke on, and the load balancing is achieved by giving servers the responsibility to serve a certain group of clients at compile time. This form of server selection is sometimes referred to as partitioning [IONA 98c]. A more flexible type of server selection is dynamic selection. By dynamic selection, the decision of which server a client invokes on is taken at runtime, and is totally decoupled from the client's properties.

Load balancing is often done by the software responsible for workflow control like a TP Monitor or an ORB. Load balancing schemes are used primarily to augment the throughput, i.e. the number of requests handled in a correct way before deadline. Studies [Friedman & Mosse 96] have, however, shown that there is a correlation between load balancing policies and fault-tolerance in a system. A load balancing scheme only describes which of several available servers to use, not any heuristics for what to do if the server called happens to be down and unable to complete its task. Therefore a load balancing strategy

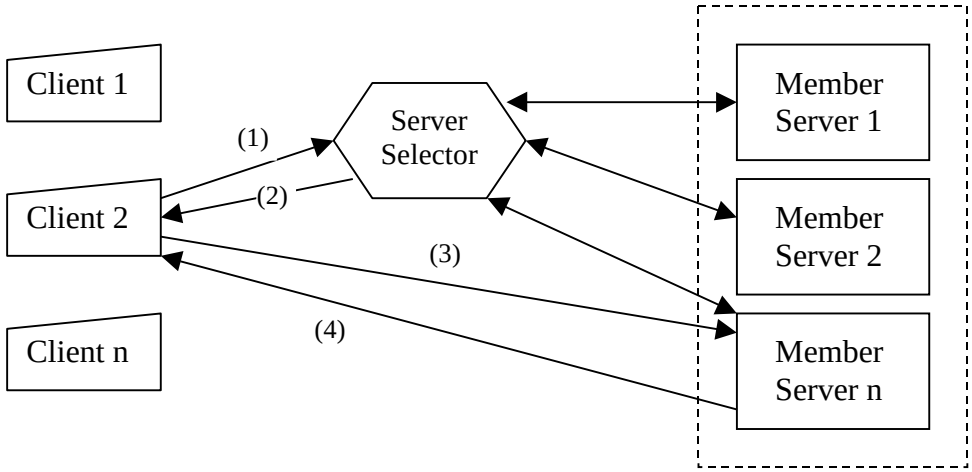


Figure 3.1: Load Balancing Scenario
(1) The client issues a lookup command to a Server Selector.
(2) The Server Selector selects a server from a group of servers that are able to perform the request, and returns a reference to the selected server to the client.
(3) The client invokes on the selected server.
(4) The selected server returns its reply to the invoking client.
The arrows between the Server Selector and the Member Servers indicate that information, on e.g. server load, that is used for the selection could be exchanged.

could be combined with some kind of *fault-tolerance* scheme making these decisions. Handling fault-tolerance in this explicit way is beyond the scope of this work, though.

3.2.2 Problems Addressed

Large-scale distributed systems are often “mission critical”. That means that it is crucial that they have a high availability during the time that their services are offered. If, for instance, the service is offered permanently the system must never go down for support and update jobs. The ability to distribute the load to other servers when some are down, e.g. to do updates, is therefore an important issue that load balancing strategies have to address.

The next problem addressed is overload. If some servers are overloaded, the risk of client timeouts and possible client crashes increases, i.e. the clients will never be served because the service is regarded as being down by the system after having waited too long for the reply. Hence the load balancing strategy should prevent overloaded servers from getting requests. If a server crash occurs, the scheme ideally should regard this server as overloaded. The load balancing scheme in this case works as an implicit error detection mechanism normally handled by the previously mentioned fault-tolerance schemes.

3.2.3 Goals

There is one major goal of load balancing; to increase throughput and thereby decrease the number of errors due to overloaded servers in the system. From the client’s point of view, load balancing should be transparent. The invocations sent should look the same regardless of whether load balancing is implemented on the server side or not. Another important goal and reason for deploying a load balancing scheme is to decrease the user request response time.

3.2.4 Design Issues

When implementing a load balancing strategy, server selection overhead, network overhead, and replicated data have to be considered, while taking the system context into account.

Server Selection Overhead
The mechanism that keeps the server distribution transparent to the client should be kept as simple as possible. Complicated algorithms can lead to the selector becoming a bottleneck in the system, as most strategies demand that all client requests be located in a serialized way. A system with a stable load that doesn't change much needs a less advanced algorithm.

Network Overhead
A possible scenario in a load balancing implementation is that the client sends a request to the server selector which returns the chosen server to the client. The client then makes its actual call to the server that was retrieved. Except from this network traffic the load information exchange between the server selector and the replicated servers also contributes to performance degradation (see figure 3.1). The network could be a serious time consumer when many clients and many replicated servers are involved [Garland et al 95].

Replicated data
Replicating stateful servers implies that a decision has to be made whether the data that is in common also should be replicated. Replicated servers operating on the same data could result in losing the concurrency intended, e.g. because of database locking. Therefore the data also must be replicated, in some cases. Using replicated data means managing replicated copies and considering hard disc space availability. Replicated data is also used to make the system more fault-tolerant. This topic is discussed in more detail in section 3.3.

3.2.5 Approaches

To be able to fulfill the requirements stated for a load balancing implementation, policies in three different areas are chosen. These include:

(a) how to get information about the load of candidate servers (*information policy*),
(b) decide when it is appropriate to re-locate a job (*threshold policy*) and
(c) choosing algorithms for deciding which server is getting the request (*location policy*).

If stateful servers are involved, a migration policy has to be considered as well to solve consistency issues [IONA 98d].

If no information policy is chosen, i.e. the schedule doesn't collect any information about the load on the servers, then the load balancing is called load independent. If information about the current system state is used to determine which server is getting the request, the load balancing strategy is said to be load dependent, because it is able to adjust according to changes in load.

3.2.6 Strategies

Only strategies concerning policies (a) and (c), mentioned in the previous section, are discussed because they are the most commonly used for server selection load balancing.

Four common load balancing strategies are presented: random, round-robin, load, and request queues. They all choose different location policies (c). Some of them also choose an information policy (a).

Random (also referred to as random splitting), takes (c) into consideration
In this strategy one server in the server pool is chosen at *random*. The advantage of this technique is that it is easy to implement and has little overhead. Because of the fact that a server that is overloaded or down could be chosen for a second time in a row, this strategy is the least fault-tolerant, though.

Round-Robin (cyclic splitting, cyclic service), (c)
In the *round-robin* strategy, the servers are invoked starting with one server and then invoking all other servers in turn before invoking the first one again. If all servers perform their tasks in a similar amount of time, this strategy is suitable. It has the same problem as the random strategy, though, it cannot cope with the situation where one server goes down or becomes overloaded.

Load (lowest load), (a) and (c)
The *load* strategy is to collect information of the load from the individual servers in some way (information policy) and then invoke on the server that is the least loaded. When using this strategy, failed servers will be regarded overloaded, and thus will never be invoked.

Request Queues, (a) and (c)
Another way of balancing the load is to use *request queues*. This technique balances the load dynamically. The clients' requests are put in a queue and are then dequeued by the servers ready to process a request. If the queue is persistent this approach has another feature. The clients and/or servers can crash without the requests being lost. One big advantage of this strategy is that no load information has to be polled from the servers and hence a lot of network traffic is saved.

Of the four strategies, the first two are load independent and the third and forth are load dependent. Implementing the load and the request queues strategies are much harder, though, and the load balancing has more overhead.

3.2.7 OrbixNames Implementation

OrbixNames from IONA Technologies is an OMG CORBA compliant implementation of the *CORBA Naming service*. The Naming service offers a way to get object references from hierarchically structured strings. OrbixNames is therefore in a good position to do load balancing transparent to the clients when they are getting their object references. OrbixNames in its current version (1.1) supports the round-robin and the random schemes.

OrbixNames normally consists of a repository of names which map to objects. With the load balancing feature the names map to *object groups* instead of objects. An object group is a collection of servers offering exactly the same service. The method `pick()` gets a member from the object group using either the round-robin or the random scheme. To make the load balancing available to the clients, an object group must be created and bound in the Naming service. The object group members are then added to their group in the server mainline. When a client resolves a name in the naming service that happens to be an object group, the `pick()` method is automatically invoked on the object group transparently to the clients [IONA 98b].

3.3 Replication

In this section, the reasons for using replication are presented, and different approaches that can be made are discussed. Further, issues that have to be considered when implementing

replication are analyzed. Oracle8 Advanced Replication is presented in section 3.3.6 as an example for a replication implementation.

3.3.1 What Is Replication?

Replication refers to the maintenance of redundancy in a system. Replication techniques use multiple copies of data or invocations to make a distributed system more reliable and performant. The copies are often called *replicas* [Bernstein & Newcomer 97].

3.3.2 Problems Addressed

As mentioned in the previous section on load balancing (3.2), distributed systems are often mission critical, i.e. they have to be highly available. When using redundant data it is easy to shift to another replica when the one in use becomes unavailable due to a server crash or communication failure. Another problem with distributed systems is that the performance often degrades severely when many users at many nodes are connected to the system. The solution to that is to increase locality, i.e. to store data locally in order to reduce costly network traffic. How to distribute the replicas is an important problem that has to be investigated and tested when deploying a replication strategy.

3.3.3 Goals

There are two major goals of replication; increasing availability and increasing performance. Higher availability can be accomplished due to higher fault-tolerance when storing or processing data redundantly. Performance gains are achieved because of increased locality as described in the previous section. A system using replication should do this transparently to the clients in order to make it possible to easily test different replication strategies, and fine tune the system. A replication scheme should also be as application independent as possible. However, the replication approach chosen is often influenced by the application context. This makes it impossible to find a fully generic solution.

3.3.4 Design Issues

In order to give the users the impression that only one data resource is used instead of a set of replicas, there must be some *synchronization* of the replicas to maintain consistency. This is the most challenging task when implementing replication. Synchronizing replicas distributed over several nodes means a lot of communication overhead, and hence performance losses. If performance was the reason for deploying replication, this is unacceptable. If, on the other hand, availability was the most important goal, this overhead might not be of such great importance, though. In addition to the communication overhead, storage space availability also restricts the use of replication. If the processing is replicated too, processor load will increase as well when introducing replication.

Synchronization in distributed systems is often achieved by using the 2PC protocol. This technique has proved to scale poorly, though, because of its weak fault-tolerance semantics [Shussel 96]. When replicas at multiple nodes are updated following the 2PC rules, it is enough if one node is down to undo the whole propagation. In large-scale distributed systems with numerous replicas, this is often unacceptable. In systems demanding absolute consistency, this is the only possibility, though. Deploying replication is therefore always a trade-off between availability (or fault-tolerance) and consistency [Faegri 95].

3.3.5 Approaches

There are two main groups of replication approaches: *synchronous*, and *asynchronous* replication. The different approaches differ in how strong the consistency is maintained.

Synchronous Replication

In the synchronous approach the changes are propagated to the replicas within a 2PC transaction for absolute consistency. The propagation may be started by some database trigger before committing the data. This is applicable, e.g., in banking funds transfer and financial trading systems that have high demands on accuracy. However, the overhead of this approach is normally unacceptable, and it does not scale.

Asynchronous Replication

In asynchronous schemes the consistency is maintained in a weaker way. The updates can be triggered in some way or can be done periodically. There are two different groups of asynchronous replication: *master/slave* (or pessimistic replication) *and peer-to-peer* (or optimistic replication).

- **Master/Slave**

This approach is also sometimes referred to as *primary-copy* replication in the literature because one replica is selected to be the primary. Updates are only allowed on the primary replica (master) and the secondary replicas (slaves) are read-only. One example of master/slave replication supported by many database vendors is snapshots. The replication is pessimistic because it avoids conflicting updates. For some systems this approach could be to rigid and inflexible, though.

- **Peer-to-Peer**

A more flexible approach is the peer-to-peer approach where all replicas can be updated. In this case there might be concurrent conflicting updates. In order to resolve conflicting updates, which are detected when propagating, some reconciliation strategy has to be chosen. Example of such strategies are: *latest timestamp* (or Thomas's Write Rule), *earliest timestamp*, *priority group* (some groups of replicas have priority when a conflict arises) and *site priority* [Chen 96].

One example of an asynchronous system is a data warehouse system or *Decision Support Systems - Replication* (DSS-R). In a DSS-R system it is more important to have a consistent view at a certain point in time for decision making and analysis, and the currency is of less importance. DSS-R systems are often implemented as master/slave.

Another example of asynchronous replication is *Transaction Processing - Replication* (TP-R) [Shussel 96]. This approach is the one closest to the 2PC approach. In order to make the system more fault-tolerant there is not a single 2PC transaction, but instead one per replica that is to be updated. If the transaction rolls back at one replica node, the update request is stored in a queue and can be processed when the replica is available again. The original updates are made on one single replica in a local transaction. As soon as this transaction commits the propagation to the other replicas starts automatically. The TP-R model is often used in production systems, and can be implemented both as master/slave and peer-to-peer.

3.3.6 Oracle8 Advanced Replication Implementation

Oracle8 supports asynchronous row level replication. When an update is made each single row that has changed is stored locally to be propagated at some later time. Another interesting feature of Oracle8 is that the changes can be propagated in parallel to the different replicas, while assuring that updates that are dependent on other updates are made in the same order as they were made originally. All of the replicas can be updated concurrently at each node (peer-to-peer model). Hence conflicts can occur, and has to be detected and resolved.

In the Oracle8 implementation, conflicting updates are detected by comparing the before image from the original site with the current image at the remote location (this is done on a column level). If a conflict is detected, then some reconciliation strategies can be chosen by the database administrator, e.g. latest timestamp and site priority. [Oracle 97a]

3.4 Caching

In a CORBA environment *object caching* (as opposed to data caching, which often is provided by the database) is the most applicable caching concept due to the object-oriented nature of CORBA. This chapter focuses on caching objects.

The motivation for using a cache, the concepts of object caching, and what functionality a cache manager should offer are discussed. Different ways of managing the cache, that is keeping the cache accurate, and some cache location policies are presented. Thereafter the most complex problem of object caching is discussed in greater detail - how to keep caches consistent using event propagation. Finally, two implementations from the industry are exemplified; *live object caching* in Persistence and the object caching in Versant.

3.4.1 Why Use a Cache?

Caches have been used for many years in the memory management of operating systems to solve the problem of inexpensive memory vs. fast access. The well known idea is to transparently swap frequently accessed data to a faster medium, giving the impression that all data is available in fast memory.

In distributed environments like CORBA, accessing information in remote databases is often a bottleneck (as discussed in section 3.1, object-relational mapping). Hence a big problem of these systems today is performance. The goal is to simulate local access in such a way that the end user ideally doesn't notice that remote access is performed.

3.4.2 Object Caching

Object caching concerns the caching of programming level objects that either are stored in an object database or in a relational database. The purpose of an object cache is to make some objects faster to access, transparently to the clients. In this section, a brief description of how an object cache normally works is given.

What Objects should be Cached?
Ideally you want to cache a small amount of data that is being accessed frequently and updated infrequently. To model such behavior a rule called the "80/20 rule" could be applied. The rule says that 80 percent of the users access 20 percent of the data. Thus to optimize performance these 20 percent of the objects, which often are [1]"core business objects" (programming level objects that model the business logic), should be cached. Objects that change their state very often should not be kept in the cache, though. Whether a caching implementation is successful depends heavily on the application context. Applications where clients issue many read request to analyze some situation, and then call write once are well suited for caching, whereas applications where clients just read some client proprietary information once, and then write back to the database don't gain as much from a caching implementation [Keen 98].

[1] Not to be confused with CORBA Business Objects defined by the OMG.

Object Faulting

An object cache must provide some way to deal with *object faults*. That is to decide what should happen when a programmer references an object that is not available in the cache. Some techniques are based on UNIX proprietary page faults (e.g. [Kordale & Ahmad 95]) that can be caught as exceptions. In the exception handler, the object is fetched from the database into the cache. A more elegant technique is to view the cache as a logical database by using a table with entries for each cached object. Object faulting is detected by checking a memory pointer field in this table. If the pointer is invalid, then the object is fetched from the source database.

Consistency of Replicated Data

When using caches, data is replicated locally. The same object could be replicated for multiple clients at the same time on different machines. In such a situation, keeping the copies consistent becomes a challenging task. Different cache management approaches (described in section 3.4.4) solve this problem differently. How strong the consistency should be held is very context dependent. For example, when browsing for information on a web page it normally doesn't matter whether the page is fully accurate. Fast access is more important. In a transactional system, though, it is more important that the data used by the transactional participants is accurate in order to fulfill the ACID guarantees.

Object Eviction

The size of the cache is often limited, and in applications dealing with a lot of objects, it becomes impossible to keep all the objects in the cache at the same time. Therefore some strategy has to be deployed concerning which objects should be evicted, and when they should be evicted to prevent the cache from getting full. Objects can be evicted on a *First In First Out* (FIFO) basis, or every time a transaction commits or aborts. One additional solution is to time stamp all objects when they are used and then evict the *Least Recently Used* (LRU) object when some upper threshold of cache usage is reached or in a periodical manner. Another possibility is to register a *Time To Live* (TTL) value for each object and then periodically evict all the objects with elapsed TTL values. [IONA 97b]

3.4.3 Cache Manager Interface

The cache can bee seen as a local database, as described in the previous section, on which a set of operations can be performed. A table consists of one entry for each data item to be cached normally with additional meta data. Below some operations that are in common for most cache managers are listed [Terry 85, Bernstein & Newcomer 97]. These operations are normally used both internally by the cache manager using one of the caching policies (section 3.4.4), and by the cache users i.e. the clients. The semantics of these operations are presented in the succeeding sections.

purge (or deallocate) - deletes one cache entry.
fetch (refresh, reload) - retrieves new data from the real source and stores it in the cache.
flush - writes data in the cache to the source.
invalidate(complain) - gives cache users the possibility to inform the cache manager that data in the cache is invalid.
revalidate – is used by some caches in order to see whether the data is valid without accessing the source.
pin/unpin – are often used in a transactional context to ensure that the data is kept in the cache and not written to the source.
getStatus - gets information or meta data about an entry in the cache table.

3.4.4 Maintaining Consistency

A cache manager, i.e. software that manages the physical cache, has to decide when to check the values in the cache for consistency, and what actions to take if inconsistency is detected. This can be done in four different ways: passively, on demand, periodically, or using pushes.

Passive
A *passive* cache manager waits until the user complains (invokes operation invalidate) about invalid data, and then decides to purge, refresh, or invalidate depending on what the user complained about. If the user, for instance, just noticed that there was an object in the cache that was erroneous, but isn't interested in invoking any methods on that object, it would be enough for the cache manager to call purge on the cache.

On Demand
In the *on demand* approach, the user says explicitly what operation should be performed on the cache. For example, a web browser that provides a refresh or reload button. This technique is applicable when the clients' demand on consistency is low [Terry 85].

Periodic
The cache manager could *periodically* check the accuracy of the data, and then refresh the data that changed. The danger with this technique is that it can introduce a lot of unnecessary network traffic.

Push
The *push* approach ensures consistency and accuracy of the data in the strongest way of the presented techniques. The cache manager gets a callback when the data has changed in the database, and it can then call refresh or invalidate on the cache. This is also the most expensive technique considering network traffic consumption and demands fast tailor-made messaging techniques, like multicasting. The push technique is further elaborated in section 3.4.6.

3.4.5 Cache Location Policy

The cache manager can work on different levels of granularity. By placing the cache on different locations, it can be made accessible to different groups of clients. A *cache manager* is here for simplicity defined to manage only one cache.

Per-process
In the *per-process* approach every client process has its own cache manager. The cache can then not be shared between clients. This could be an appropriate approach for long lived clients accessing the same data multiple times, e.g. browsing for information in a web-browser.

Per-processor/machine
All the clients on the same machine can share a cache manager. This is applicable when clients on the same machine have similar tasks, and use similar data.

Per-site
If a fast LAN is available, the network traffic within the site could have minor impact on the user response time. In that case it could be a good idea to have one cache in common for all the clients on that site. Sites connected with a fast network could also share a cache [Terry 85].

The approaches mentioned so far could easily be combined into a hierarchy. If the object is not available in the cache of the finer grained cache manager, the manager on the next higher level is contacted to get the object. This is done in [Chankhunthod et al 94] for caching web pages located on web servers that are ordered in a hierarchy.

Per-transaction
In transactional systems it might be appropriate to let all participants in a transaction share the same cache. When a participant references an object the cache manager first looks whether the object is available in the cache. If the object is not there, it calls fetch and increments the pin counter by calling pin. When the participant does not need the object anymore unpin is called. Later when committing the transaction, the cache is flushed (this can only be done if the pin counter is 0) and purged. Per-transaction cache management is closely related to managing recovery and logging. Before unpin is called the *recovery manager* checks whether the data has changed. If the data has changed (commonly referred to as being dirty), a log record is written (containing before and after images of the data) in order to enable recovery after a crash. This is done because unpinned data could potentially be flushed [Bernstein & Newcomer 97].

Per-application
Application servers accessing data in a database on behalf of clients are inherently candidates for offering caching. An application server could, for instance, combine a per-process location policy (where the server is the cache client) and a per-transaction cache, as exemplified by the Persistence PowerTier tool (section 3.4.7).

3.4.6 Cache Consistency - the Push Approach

In this section, different approaches that can be taken to keep multiple caches consistent are discussed. This involves synchronizing multiple concurrent cache users and dealing with conflicts that can occur when different users have different views of the same object. Further, updates have to be propagated, and the lifetime of objects controlled.

Optimistic vs. Pessimistic Approach
To prevent inconsistency of data different users must be prevented from updating the same data at the same time. This is easily done in non-distributed environments because all databases support some kind of synchronization mechanism to block users trying to read data that is updated by another user at the same time. One common synchronization technique is locking. Locking all the distributed caches of an object when one client wants to make an update, or query some data is both very costly, and also difficult to implement.

As the main objective of deploying caching is to increase performance, it would be unfortunate to limit the concurrency by blocking all users who want to read data currently being updated by some user, and thereby causing a severe bottleneck. If concurrent updates and read requests are unlikely to happen, however, this so called pessimistic approach, where all caches are locked during updates, could be used. This could be the case in a scenario where single clients access the same data frequently.

A more suitable approach, when having multiple distributed caches and many concurrent users, is known as the *optimistic approach* or *optimistic locking*. When a client reads some data into the cache it is not locked but instead some kind of log is written containing a version number or a time stamp. At the time the client is ready to update, the logged version of the cache is checked against the version currently available in the database. If these two versions are consistent, the update can take place, otherwise another user has done an intervening update, and the operation has to be canceled.

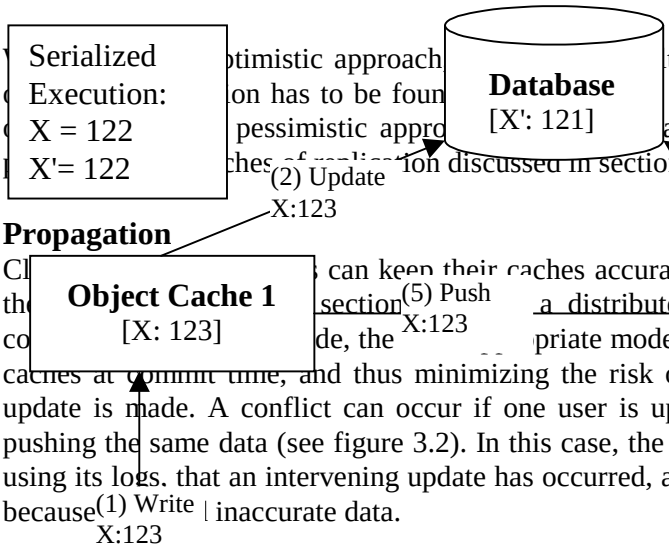


Figure 3.2: Window of failure, concurrent updates by many clients

Propagation

Cl can keep their caches accurate by ation of th section a distributed tr t where co de, the priate model wo es to all caches at commit time, and thus minimizing the risk of conflict when the actual database update is made. A conflict can occur if one user is updating concurrently to another user pushing the same data (see figure 3.2). In this case, the second update should be able to tell, using its logs. that an intervening update has occurred, and cancel the second write operation because inaccurate data.

When using the push approach, some additional decisions have to be made like when to update the caches, and how to carry out the update. The caches could be updated just before, or just after commit e transaction. If pushing is done before commit hen there is a risk m being started concurrently by of the updates is th en become ambiguous. If a co s made before prop be detected before starting th eration can be canceled before creating incorrect push events. If the database update is very costly compared he second ovide them with the new value. If all caches are just notified of a change, they have to access the database for the new value, which could be costly. Propagating the value with the notification, prevents this. Therefore this would be preferable, provided that the pushing technology used is able to transmit all the data changed in an efficient way, and the changed data is not too complex to send within a push. Pushing changed data could be very dangerous with respect to the consistency. Normally the order of which the events arrive has to be controlled as exemplified by Oracle8 advanced replication described in section 3.3.6.

Granularity of Events

Another important decision, to be made when deploying a caching strategy with an event notification mechanism is - on which level of granularity should the events or pushes be generated? Two contrasting factors should be considered. First, the overhead implied by the sender generating many different kinds of events (potentially on multiple channels). Second, the number of pushes invoked on clients not interested in the event. Another way to put it is; who should do the filtering -the receiver or the sender of the push? The following granularities could be appropriate depending on application context.

- **Per-Object**

As soon as an object changes its state an event is generated. If the application has few important objects used by many clients, this would be appropriate. An example could be stock-brokers interested in the stock of a company.

- **Per-Class**

If objects of a certain class are frequently read and less frequently changed, per-class propagation may be a good idea. Per-class notifications could be useful when, for instance, clients maintain a list of objects of a certain class. One example application could be a booking system where objects of the class that can be booked are cached.

- **Per-Table/Row**

If the database already supports events like triggers, it could in same cases be a good idea to use these built-in database events for better performance. A per-table trigger corresponds to a per-class event, and a per-row trigger corresponds to a per-object event if a one-to-one mapping is used. This might not always be the case, as discussed in section 3.1.3.

- **Logical Grouping**

There could be other groupings of objects suitable for pushing through the same event channel for optimal benefit concerning number of events, and overhead vs. notifications interesting to the receivers. This is useful in applications where the per-class propagation sends too many events not interesting to receivers, and all filtering is done at the receiver side. By ordering events into logical groups, some filtering could be done on the sender side. An example could be a booking system that sends out different events for each booking made in a different geographical area.

Lifetime of Objects

The lifetime of an object is closely related to the eviction policies chosen (described in section 3.4.2) and the cache location (described in 3.4.5). If the per-transaction caching model is used, the object lifetime could be the same as the time of the transaction. If the same object is changed and read by many transactions concurrently (for instance in a per-site cache), this would be an inappropriate approach, though, and a more general LRU strategy would be suitable. If an object is not used much, evicting it could lead to better performance, as update events don't have to be handled. Hence lifetime policies could have a big impact on consistency policies.

3.4.7 Object Caching Implementations

In this section, two different object cache implementations are presented. The object cache from Versant implemented in their object database, and the Live Object Caching concept implemented in Persistence PowerTier (described in section 3.1.6).

Versant Object Cache

Whenever a database session is started, Versant allocates a pointer to a new object cache in the virtual memory of the machine where the application is running. The same session can be used by many clients who then share the same object cache. This object cache is managed using a table containing one entry for each cached object. These entries contain one pointer to the object in memory, and one logical object identifier to locate the object in the database. Each entry also contains meta data like lock information, and pinning status. When a client makes a request on an object, the cache table is queried, and if the memory pointer is null, the logical object identifier is used to get the object from the database (see object faulting in section 3.4.2). After the object is retrieved pin is called. When the transaction commits, the

meta data in the cache table is queried to check which objects changed. The objects that did change are flushed, and then purge is called. The cache can optionally be kept after committing. [Versant 98]

Persistence PowerTier Live Object Caching

The *live object cache* of Persistence was developed to solve the performance problems due to accessing a relational database from an object-oriented environment, and having to map the relational structure to objects. With a local cache containing the most frequently used objects, a major performance improvement is possible. In addition to the faster access when the network traffic is reduced, being able to query an object in the cache instead of doing SQL joins also improves the performance. Another performance improving technique used is optimistic locking on the object level, which is implemented using version stamps. The live object cache implementation furthermore contains a notification mechanism which can be used to maintain the consistency of the cache.

There are two different physical caches in order to support transactions. A shared cache accessible by all clients, and a transactional cache only accessible by the client starting the transaction. If the transaction is committed, then the transaction cache is cleared, and the cache entry is copied to the shared cache [Persistence 98]. See figure 3.3.

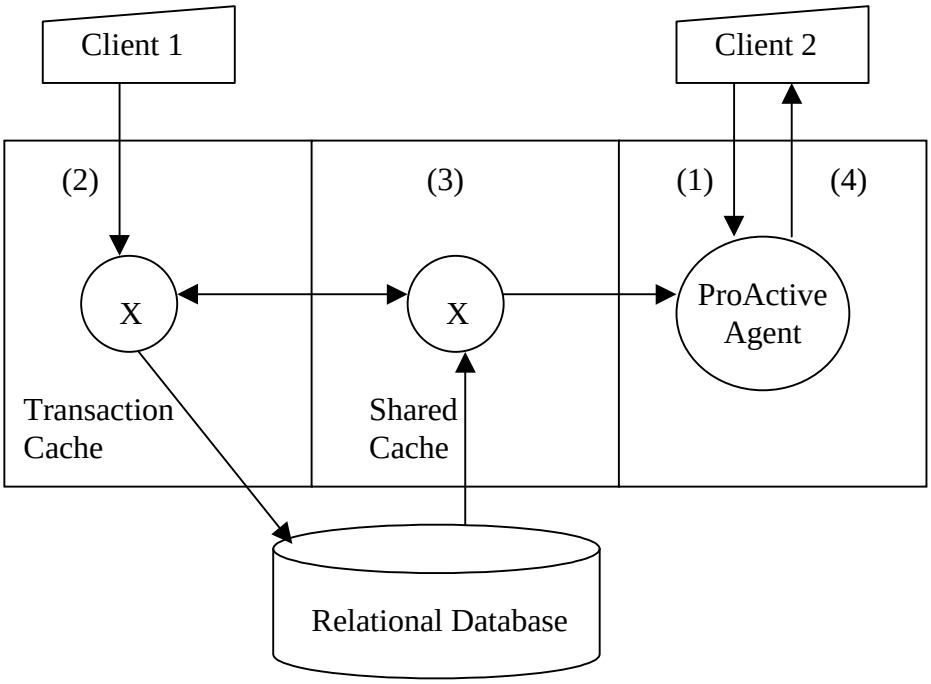


Figure 3.3: Persistence Live Object Cache Synchronization with ProActive Agents
(1) Client 2 registers a criteria with the agent.
(2) Client 1 starts a transaction on Object x. The object is fetched from the database into the shared cache, and copied into the transaction cache.
(3) The transaction is committed. The transaction cache is cleared, and the object is copied back into the shared cache.
(4) An event is generated and pushed to the clients registered for the criteria. These clients could then update their caches.

3.5 Summary

The focus of this chapter was on how to improve scalability and performance in large-scale distributed systems using persistent data

When mapping objects to relational databases basically two things are important to keep in mind. First, do not make the application logic dependent on the database schema, i.e. separate behavior (e.g. IDL) from how the data is actually stored. Second, try to retrieve as much data in as few database calls as possible to avoid the object-relational mapping becoming a bottle-neck (e.g. prefetch data that is likely to be read by others).

Three closely related techniques for improving performance and scalability in distributed systems were discussed. Load balancing is mainly used to duplicate the processing for better scalability. Replication involves duplicating data in order to improve performance and availability. Finally, caching is used to duplicate data in in-memory structures for better performance and to reduce network traffic.

Caching is the main focus of this thesis and was therefore discussed in greater detail. When implementing caching, one major decision has to be taken: how the caches are going to be updated when the source changes. This decision always leads to a trade-off between

consistency and performance. If the caches always have to be kept consistent with the source at any point in time, then the system is likely to suffer from serious performance degradation, and will not scale. Therefore usually less strong consistency is accepted which is often referred to as the optimistic approach.

4 Software Architecture

In this chapter, general concepts of software architecture are introduced. The concepts are elaborated and applied in Chapter 7: Architectural Abstractions from the Case Study. There are many different definitions of what software architecture is. Properties that all definitions have in common are described in the first section of this chapter. The second section presents the concepts of views and patterns. The last section describes three different techniques for modeling architectures: *UML*, *Catalysis*, and the *Connector framework*.

4.1 What is Software Architecture?

In the literature, there are many different opinions on what a software architecture is. All have in common that a software architecture should describe the software system in an abstract way. In current practice, this is normally done with informal "box-and-line" diagrams. The field of software architecture aims at formalizing these diagrams, and to make them more expressive. The boxes are typically referred to as *components*, whereas the lines are called *connectors*. Components are interrelated elements of software with externally visible properties. Components should be able to describe the software system on different levels of granularity. The highest level would typically hide all details that concern implementation. Connectors are used to describe the interactions between two or more components.

OO-programming languages and traditional OO-modeling techniques emphasize the modeling of entities in the system. The descriptions of how these entities interact is often distributed and embedded in the entity definitions. This approach is useful when trying to design common behavior of single entities (by using inheritance). It is not so suitable, however, for detecting common behavior involving several components. The software architectural approach, on the other hand, makes a clear distinction between components and connectors, and treats them independently.

A software architectural description should be able to address the composition of components, the general control structure, communication between components, synchronization, physical distribution, as well as scaling and performance issues [Garlan & Shaw 96]. Changing an architecture becomes increasingly costly as the development of a system progresses. Therefore, being able to consider design issues like those mentioned above is crucial to detect design flaws early in the development lifecycle. Architectural abstractions help in defining terminology for discussing the design among people involved in the development project. Another important objective of software architecture is to enable reuse of design. This is done by abstracting design solutions into generic patterns (discussed in the next section). By reusing design, the time for developing new similar systems can be reduced immensely. Furthermore, maintenance becomes easier when the system architecture is well documented.

In a broader perspective, the research in software architecture aims at making software development to an engineering discipline. That is the software development process should follow strict scientifically proven steps in order to always yield satisfying solutions at a reasonable cost. Many concepts therefore origin from other engineering disciplines.

4.2 Views and Patterns

In this section, two of the most essential techniques for creating software architectures are presented. *Views* are fundamental in capturing different aspects of the system, and *patterns* are used as "building-blocks" for composing architectures.

4.2.1 Views

A software system does not comprise one single architectural structure but several. Depending on what the entities and relations depicted represent, different structures or *views* of the system can be described. Examples of views are *conceptual*, *class*, *physical*, *module*, *process*, *data flow*, and *control flow* [Bass et al 98]. They are all explained in some more detail below.

Conceptual

Conceptual views are commonly used to model the problem domain. They are useful when the functionality of a system is to be outlined without having to consider, e.g., which implementation language is to be used. The entities refer to functional units and the relations typically mean that information is shared between the entities.

Class

The class view is probably the most common view in traditional object-oriented modeling. The entities are classes and the relations refer to methods or attributes. The diagrams in this view are implementation dependent but still offer some abstraction from the algorithms in the code.

Physical

In distributed systems where the system runs on multiple processors, machines, or hosts, a physical view of the system would normally be created. In a physical view the entities are hardware, and the relations represent communication. These views are important to be able to consider performance and scalability issues.

Module

A module view could be used for *information hiding*. The software system is grouped into modules encapsulating parts that are likely to change. When a change is made, only the module in which the change resided, and possibly its sub-modules, should be affected. The relations in the view are used to build a hierarchy of modules and sub-modules. The modules on the highest level in the hierarchy are commonly parts of the system that have been assigned to a group in an organization.

Data and Control Flow

A control flow view depicts states and how they transition into new states. Data flow diagrams typically describe entities sending or receiving data, in order to see which entities are involved in fulfilling a functional task.

Other commonly used views are: uses, calls, and process views [Bass et al 98]. By letting different diagrams focus on different quality concerns like performance and modifiability, a clearer overall view of the system can be gained. Furthermore, the views support the design process by incrementally defining new aspects of a complex system.

4.2.2 Patterns

As discussed earlier in this section, an important goal of software architecture is to reuse designs. This is done by documenting solutions found in a generic way. The result is called a pattern. A *pattern* is not created but detected. Finding patterns could be seen as forming reoccurring chunks of software into units that are associated with a representative name. Patterns hence both make it easier to describe and discuss complex systems, as well as record experiences from previous designs. By making patterns easy to use and understand, it should be easy to design suitable systems by composing patterns from a repository.

Patterns occur on different levels of granularity, i.e. have different scopes. Three major levels can be discerned: *reference models*, *architectural styles*, and *design patterns*.

Reference Models
This level is sometimes also referred to as the enterprise or global level [Malveau & Mowbray 97]. It is the highest and most abstract level. The reference models are standards defined for different domains. The standards could be defined by accredited bodies, consortia, the market (de facto), or within an organization. All architectures designed in an organization must conform to the chosen reference model. One example of a reference model is the Object Management Architecture (OMA) described in section 2.1. All the standards adopted by the OMG must conform to OMA.

Architectural Styles
Architectural styles describe patterns on a system level. This is a relatively new field of research. It arose from the need of having a terminology to describe common computational concepts on a higher level than the programming code. A catalogue of architectural styles and their definitions was created in [Garlan & Shaw 96]. Styles described there are: *dataflow systems*, *call-and-return systems*, *independent components*, *virtual machines*, and *repositories*. These styles are further refined into subcategories where, e.g., OO-systems are defined.

Design Patterns
Design patterns describe design solutions to common problems on a class level. Each pattern comprises some classes and their relations, as well as a textual description including code examples. The design patterns can be directly implemented in any programming language (but an object oriented language would be preferable). An extensive catalogue of design patterns can be found in [Gamma et al 95]. Patterns defined there include: *observer*, *proxy*, *adapter*, *abstract factory*, *iterator*, and *bridge*.

The idea behind patterns originally came from the architect Christopher Alexander who defined patterns for building houses in 1977. As a software research field, it is fairly new, though. Patterns have become increasingly popular in the industry as well, especially on the design level, because they can easily be applied and delivered with tools and standard class libraries.

4.3 Describing Architectures

In this section, common requirements and properties of languages and modeling techniques to describe software architectures are discussed. Thereafter three modeling approaches are presented. None of these was developed as an *architectural description language* (ADL), but can be used for describing interesting aspects of architectures. *UML* is the standard object-oriented modeling language. *Catalysis* is an extension of UML offering some more notations suitable for describing architectures. Finally, the *Connector framework* is a framework to be

used in conjunction with object-oriented modeling languages to enhance their architectural expressiveness.

4.3.1 Ideal Properties of Architectural Descriptions

The definitions of what exactly a language should support in order to be classified as an ADL vary immensely. Some common ADL properties [Garlan & Shaw 96] are discussed below.

Support for composition of components
Components should be easy to combine by specifying interfaces that are used, and those that are exposed by the component. By defining interfaces, contracts are defined between providers and users. The contract should be independent of the realization of a component. By combining components, large-scale and complex systems can be easier described.

Describing a system at different levels of granularity
The idea behind describing systems at different abstraction levels is to suppress unnecessary detail during analysis and in earlier phases of the design. Later the descriptions can be refined incrementally towards an implementation. At each level of abstraction different interactions and behavioral aspects will be exposed. Further, designers may be used to work on various levels of detail.

View support
In section 4.2.1, structures or views of a system were discussed. An ADL should have support for documenting these views. With views, concerns can be separated to get a clear picture of a complex system.

Identifying roles
Components in the systems can take part in several *collaborations* or interactions with other components. In each collaboration the component contributes in different ways to the joint behavior. A component can be said to play different *roles*. A component is an abstraction above the notion of a class and often has a complex behavior. By specifying which roles a component can play, the behavior can be decomposed. Each composition focuses on a different aspect of the behavior.

Reuse of components and component patterns
The concept of patterns was described in section 4.2.2. An architectural description is abstract and therefore often generic. In order to reuse generic patterns, an ADL should support notions of instantiating components or groups of component. If the instantiation of a component can be parameterized, the pattern could be seen as a *stereotype*. Further, if a collaboration of components can be parameterized and instantiated, the pattern is typically referred to as a *framework*.

Combining heterogeneous parts
A large-scale and complex system often comprises heterogeneous parts. To support the integration of these parts, an ADL should be able to describe how components realized in different environments interact. Typically, all components are viewed uniformly at a higher level of abstraction. The heterogeneity becomes visible in refinement processes where details are added. To make a system modifiable and extensible, it is important to have a common infrastructure that all the heterogeneous parts comply to.

Traceability from architecture to design
Traceability means that there should be a documented bi-directional path between the most abstract levels of analysis to the implementation. Traceability is useful when re-engineering the system, and simplifies maintenance.

4.3.2 UML

The *Unified Modeling Language* (UML) is an effort to combine concepts from several *object-oriented analysis and design* (OOAD) methods into one unified language. The work with UML started in 1994 and primarily consisted in unifying the *Booch*, *OMT* (Object Modeling Technique), and *OOSE* (Object-Oriented Software Engineering) methods. UML is developed by a consortium of industry and academia, and is currently in a standard adoption process of the OMG.

In UML, as in traditional OO-modeling languages in general, most of the modeling focuses on *class diagrams* (and hence implementation). More interesting from an architectural view-point are the *behavior diagrams* defined in UML. One of these diagrams is the *collaboration diagram* to model interactions between objects playing roles. The collaborations normally are on an implementation level, though. Classes participating in a collaboration as well as the relationships can be parameterized to form patterns.

UML has support for many different views, e.g. class, module, data flow, control flow, and physical views. The physical view can be described in two diagrams: *component diagrams* and *deployment diagrams*. The component diagrams depict components, the objects they contain, and the interfaces they expose. In deployment diagrams, the run-time component distribution of a system can be shown. Components and the hosts they reside on are described. Further, dependencies between components and interfaces from other components can be depicted. In the *state* and *sequence diagrams*, concurrency solutions can be shown. [OMG 97c]

4.3.3 Catalysis

Catalysis is an extension of UML developed by ICON Computing and Trireme Object Technology. Catalysis uses UML, but introduces some additions.

Catalysis focuses on modeling abstract behavior in *type models*. A type model specifies a contract between suppliers and consumers of a component. The external behavior of an object conforming to the type is defined. The contract is described by specifying the operations supported by the type, and pre- and postconditions for each operation. The type model suppresses all details about implementation. Types described in a type model can be implemented by many classes, and one class can implement multiple types. *Collaborations* can further describe how these objects interact depending on their external behavior. A third important construct in Catalysis is *refinement.* Refinement is the process of providing more detailed models. A refinement can be done on any model by assuring that the result – the *realization* – is conformant to the original model – the *abstraction*. A refinement hence provides *traceability* between models of different granularity. Type models, collaborations, and refinements can all be parameterized and reused as patterns. The parameterized models are called *frameworks* [D'Souza & Wills 98].

4.3.4 Connector Framework

The Connector framework has been developed at TU Berlin in the research group Computation and Information Structures (CIS). The framework can be used in conjunction with other OO-modeling languages like UML and Catalysis to elaborate certain parts of the

descriptions. There are two important constructs in the framework: *components* and *connectors*.

Components model the computational parts in a system (e.g. clients and servers). A component is described in terms of its *exported* and *imported interfaces*, a *representation-map*, and a *representation*. The interfaces provide a means to describe the behavior of the component in an implementation independent way. The representation-map describes how the interfaces are mapped to constructs in a programming language. Finally, the representation is a model of how the behavior is implemented. All these three parts can internally be described using constructs from modeling languages like UML and Catalysis.

Connectors describe the interaction between components playing different roles. A Connector specification comprises: a description of the roles participating in an interaction, role interfaces, and interaction protocols specifying the collaboration. Roles are component independent and used to provide an abstract view of participants in an interaction. This promotes the use of connectors as patterns. Interfaces of the roles are specified and used for describing the role dependencies in the interaction.

Connectors are instantiated by specifying the components playing the roles defined in the connector. The instantiation is called an *abstract architecture*. The abstract architecture is refined to form a *concrete architecture*. In a concrete architecture the realization of the components, as a result of the roles they are playing, is shown. [Tai 96, Tai 98b, Tai 98a]

4.4 Summary

In this chapter, software architectural concepts were discussed. There are many different interpretations on what a software architecture is. They, however, all have in common that a software architecture is an abstraction of the structures of the system. These structures are described using *components* and interactions between components, often referred to as *connectors*.

The main purpose of software architecture is to provide implementation independent views of the system to enable reuse of design, and support early design decisions. ADLs are languages developed to support this process.

Three modeling approaches were discussed: UML, Catalysis, and the Connector framework. They were not developed as ADLs, but they all support useful properties for describing architectures. Furthermore, they can be used in conjunction with each other. UML is suitable for describing different views of a system. Catalysis has useful notations for describing contracts between suppliers and users of a component. Finally, the Connector framework provides a means to combine properties from other modeling languages. Further, it has sophisticated abstract notions for describing interactions, which promotes the use of patterns.

In chapter 7, UML and the Connector framework were used to model the case study in this thesis, as they are suitable for modeling object-oriented CORBA systems.

PART II - PRACTICAL EXPERIMENTS

The second part of this thesis applies the concepts from part I in two steps. In the first step, an object caching strategy is implemented in a case study. Thereafter, software architectural modeling is used to describe a generic reusable design based on the case study. Chapter 5 presents the tools used for the implementation. Chapter 6 introduces the case study implementation. Finally, chapter 7 introduces an approach to modeling the case study.

5 Technical Solutions

In this chapter, various technical solutions and tools available for an implementation of the case study described in the next chapter are discussed. Investigated tools are presented by considering how they solve the problems discussed in the previous chapters. The products are grouped into two main categories based on the problems they address: distributed object systems and transactions, and object persistence and caching. In the first category, the products: OrbixOTS [IONA 98f], OrbixEvents [IONA 98g], and OrbixTalk [IONA 98h] from IONA are reviewed. In the second category, the products: Oracle OCI [Oracle 97b], Oracle Embedded SQL [Oracle 97c], Persistence PowerTier [Persistence 98], and RogueWave's DBTools.h++ [RogueWave 98] are discussed.

Each tool is evaluated regarding its suitability for the object caching strategy implementation to be done. The most decisive decision to be made was whether to use an object cache provided by a tool, or to write an object cache from “scratch”.

5.1 Distributed Object Systems and Transactions

In this category, it was quite clear that OrbixOTS was to be used for distributed transaction support as this product served as basis for all the implementations made in this thesis. OrbixOTS is also the core part of OrbixOTM (see section 2.4). The main question was whether to use OrbixEvents or OrbixTalk for the event propagation. Tests showed that both worked well with OrbixOTS. The question could also be put: whether a CORBA compliant product (OrbixEvents), or a more performant and scalable product (OrbixTalk) was to be used. OrbixTalk was finally chosen for its scalability. Below a short description of each of the products is given.

5.1.1 OrbixOTS

OrbixOTS is an implementation of the CORBA Object Transaction Service (see section 2.2.1). In compliance with the standards it supports distributed transactions by driving the 2PC protocol, and by propagating transaction information to enable interpositioning. OrbixOTS can integrate database resources in two ways; either by wrapping them into a Resource object or by using the X/Open DTP standard XA interface, which specifies the interaction between the transaction manager and the resource manager transparently to the database clients.

OrbixOTS thus can assure atomicity in a distributed environment. The isolation property of a distributed transaction, on the other hand, is normally the responsibility of the database implementing an XA interface. Durability is assured both by logging made in the XA implementation of the database (before and after-images of changed data) and logging done by the OTS. The OTS logs the current status of transactions using log files connected to the recoverable servers. Keeping the data consistent will very much rely on the application code. However, assuring the other three transaction properties eases up the burden of the application programmer to maintain consistency.

5.1.2 OrbixEvents and OrbixTalk

One way to maintain consistency in a distributed system (e.g. among distributed caches of the same data) is to let an event service notify distributed sites using the data of changes. In a large-scale system, this has to be done efficiently not to cause an overload on network traffic.

OrbixEvents is an implementation of the CORBA Event service (see section 2.2.2). It uses IIOP (the internet inter orb protocol standardized by OMG) to send events over the network, and could be used as a tool to maintain consistency. With OrbixEvents, multiple event generators (suppliers) can be connected to multiple event subscribers (consumers) in an asynchronous and decoupled manner by registering them with an event channel. If a supplier sends an event to the channel, one message to each consumer registered is sent by the channel.

OrbixTalk is an extension of OrbixEvents that additionally provides an implementation of asynchronous messaging using the multicast protocol. Instead of sending the event notifications to all subscribers using IIOP as stated by the CORBA standard, the event message is sent only once to a multicast port. The underlying network protocol then transmits the message further on to the listeners to this port. This reduces network traffic in a drastic way, and OrbixTalk therefore scales better than an IIOP solution. OrbixTalk also has additional features to ensure reliable delivery of the messages.

5.2 Object Persistence and Caching

In this category, the most difficult decision had to be made: which of numerous database tools should be used to access the database? Some tools like Persistence and Oracle OCI already supported an object cache off the shelf, this on the other hand doesn't imply that the built-in caches would be easier to use in the case study than a tailor-made solution written from scratch. Another crucial part was which of the tools supported an XA interface and could thus be smoothly integrated with OrbixOTS. At the time of implementation only OCI and Oracle PRO*C had support, though. Oracle PRO*C embedded SQL was finally chosen because it was the easiest product to use. Therefore, no product supplied object cache could be used.

5.2.1 Oracle OCI

The *Oracle Call Interface* (OCI) offers a procedural interface to SQL. SQL statements are built up by issuing subsequent calls in order to bind host variables (programming language variables connected to the query), to prepare and execute a query etc. Oracle also enables access to database tables through an object interface. Oracle8 implements a built-in object cache that only can be used by issuing OCI calls. The major drawback of OCI is that it is intricate to use. A lot of programming is required even to perform very simple tasks. Further, it constrains the programmer to use a procedural style of programming far from the object oriented ideas. Instead of first creating an object and setting up its internal state, (e.g. representing all the database and transaction details, and then invoking subsequent queries on this object), OCI requires that handlers are created and initialized, and passed in to all the database functions as parameters. Some OCI functions requires up to 10 different handlers as parameters. Because of the tight coupling to Oracle, and high functionality, OCI is suited to be used when implementing object-oriented database access wrappers, though.

5.2.2 Oracle PRO*C

The most common way to access relational databases is through SQL. The SQL calls can be embedded in the programming language using a precompiler. Oracle offers a C/C++ precompiler called PRO*C. The advantage of this approach is that it is almost as simple to program as if SQL had been used directly. It requires some additional programming like declaring host variables, but this extra work is small compared to building up a query in OCI. The Oracle object cache mentioned in the previous section can, however, not be used with embedded SQL.

Both PRO*C and OCI can easily be used with Oracle's XA implementation for distributed transactions.

5.2.3 DBTools

DBTools.h++ is an object-oriented SQL wrapper. SQL queries can be performed using a simple object-oriented API. RogueWave now also supports distributed transactions with their new XA add-on product for DBTools. This was not supported when implementing the case study, and could therefore not be tested. The data objects a programmer works with are only wrappers though, and don't provide access to the data in a high level object-oriented and storage independent way. (See discussion in section 3.1.4.)

5.2.4 Persistence PowerTier

Persistence is an object-relational mapping tool that enables an object model to be specified for the persistent data. From this model, C++ code can be generated enabling object-oriented and storage independent database access. Persistence also has an object caching feature (live object caching).

The current drawback of Persistence is that it doesn't support distributed transactions through an XA interface. This support is announced to be supported in the next release. For further details on the Persistence tool see the sections 3.1.6 and 3.4.7.

5.3 Summary

The following tool-chain was chosen for the implementation: Orbix, OrbixOTS, OrbixTalk, and Oracle PRO*C (Orbix is the core ORB needed by all other Orbix add-on products like OrbixOTS and OrbixTalk). The use of OrbixOTS is central in the implementation to assure the ACID properties in a distributed CORBA environment. All other tools therefore had to be compatible with OTS. OrbixTalk was chosen to gain optimal performance by reducing the network traffic. The tool that supports caching and object-relational mapping in the best way of the ones investigated is Persistence. It did not support XA, however, and could thus not be integrated with OTS. When the implementation of the example scenario started, RogueWave XA was not available and OCI was considered too complex to use, therefore all the database access was done using PRO*C and embedded SQL. (See table 5.1.)

	Orbix Talk	**Orbix Events**	**OCI**	**PRO*C**	**DBTools**	**Persistence**
OTS Support	Yes	Yes	Yes	Yes	No	No
Scalable	Very	Yes	-	-	-	-
Object Cache	-	-	Yes	No	No	Yes
Easy to use	Very	Yes	No	Yes	Very	Very

Table 5.1: Comparison between tools that could be used for the object caching implementation

6 Case Study

In this chapter, the problems addressed in part I of this thesis are investigated in the more detail by designing and implementing an object caching strategy with the chosen tools described in the previous chapter.

As an example scenario, the “Personal Touch Travel Agency”, originally design for the IONA World 98 trade-show to demonstrate IONA’s OTM Product, was chosen as basis for the implementation. In order to make the implementation and the tests as realistic as possible, some assumptions were made on how the system could be used in a real-world scenario. These assumptions serve as input to the test configuration setup. The result is summarized regarding two different aspects: the caching strategy chosen, and the assurance of ACID properties of the implementation.

First, the problem domain is introduced. Object caching is added to this picture gradually. The caching concepts are introduced starting on an architectural level, and then going through the levels of design and implementation. Finally, caching is introduced on an application level by simulating a real-world scenario, and carrying out performance tests.

6.1 Problem Domain

In the Personal Touch Travel (PTT) Agency, `Customer`s can book `Cottage`s located at different `Resort`s. `Customer`s can also browse information on `Resort`s and `Cottage`s as well as the `Availability` of `Cottage`s on-line in order to make the `Booking` procedure easier (see figure 6.1).

The PTT Agency should be accessible from a wide geographical area, and should be able to serve a large number of `Customer`s concurrently. `SalesOffice`s were introduced to meet this demand. Their purpose is to serve local `Customer`s by using information from a `CentralOffice`. Information on `Customer`s and `Booking`s made should be kept at each `SalesOffice` in order to facilitate services like invoicing and book keeping in the future.

6.1.1 System Architecture

The `CentralOffice` maintains the central database (source database) with data for `Resort`s, `Cottage`s and `Availability` of `Cottage`s. The `SalesOffice`s, which were introduced for server load balancing, maintain their own local databases with their local `Customer`s and `Booking`s, which in fact is replicated from the `Availability` data in the `CentralOffice` (see figure 6.2).

The PTT System thus contains both replicated servers and replicated data. The load balancing between the servers is static and visible to the clients. In this scenario it is reasonable to assume that `Customer`s have knowledge about their local `SalesOffice`s, therefore the load balancing is visible. Further, `Customer`s always contact their `SalesOffice`s, thus the load balancing is static (known at compile time - as defined in section 3.2.1). The `CentralOffice` and hence the replication of data is totally transparent to the clients.

In order to maintain consistency between the `SalesOffice`s and the `CentralOffice` distributed transactions should be used. The `Booking` data in the `SalesOffice`s should at any point in time be consistent with the `Availability` data in the `CentralOffice`.

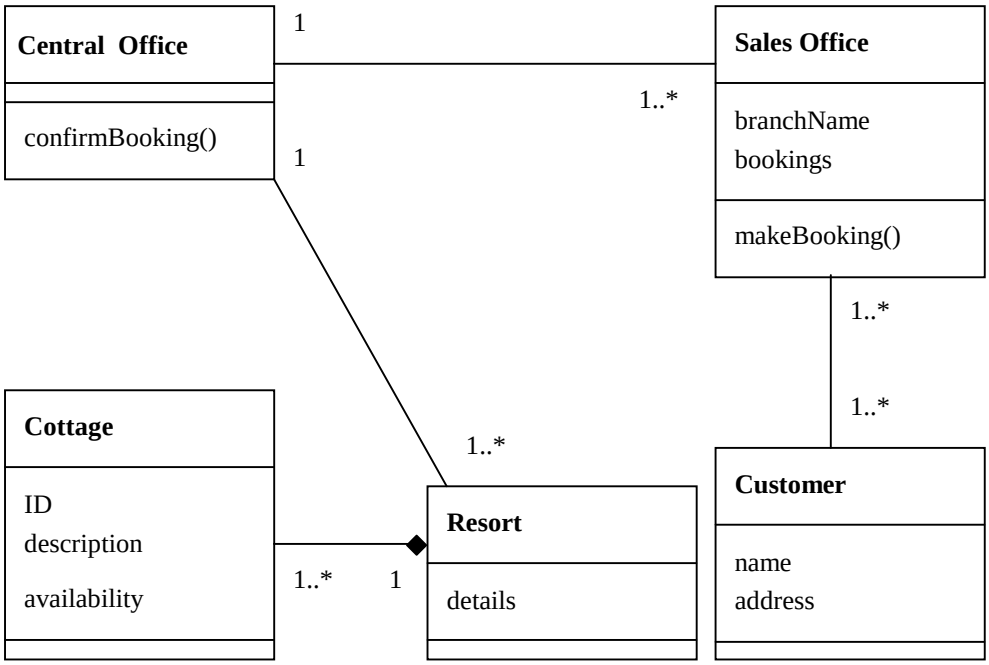


Figure 6.1: Logical view of PTT Agency

6.1.2 Constraints

The scenario described above simulates a large-scale transactional CORBA system. To understand the need of CORBA and distributed transactions, and performance issues implied by this environment, some real-world assumptions have to be made here. The main aim of these assumptions is to build a discussion basis for later implementation and test configurations, **not** to find exact figures. Hence, for simplicity, server failure and down-time has not been taken into consideration in the following calculations.

Number of Customers

To motivate the use of the architecture described above, we assume that the system has **1 million** users. The `Customer`s should furthermore be distributed evenly over a wide geographical area.

Use Pattern - Booking

Further assumptions rely on how these 1 million `Customer`s use the system. First a distinction between peak season and off-season has to be made. Three months in the summer time is peak season, whilst the rest of the year is assumed to be off-season. Every `Customer` makes in average 1 `Booking` for one week each year. The `Booking`s are in 80 per cent of the cases made in peak season. This means that during peak season there are about 61,540 `Booking`s per week (13 weeks June-August) and about 5,000 `Booking`s per week during off-season (the rest of the year). For every `Booking`, the `Customer` has to query information, i.e. use the `SalesOffice` server. A typical scenario would be that the user gets information on all `Resort`s, gets all `Cottage`s in one `Resort` and then checks the `Availability` for 10 `Cottage`s before the actual `Booking` is made. In this scenario the `Customer` issues **12 queries and 1 update for each Booking**.

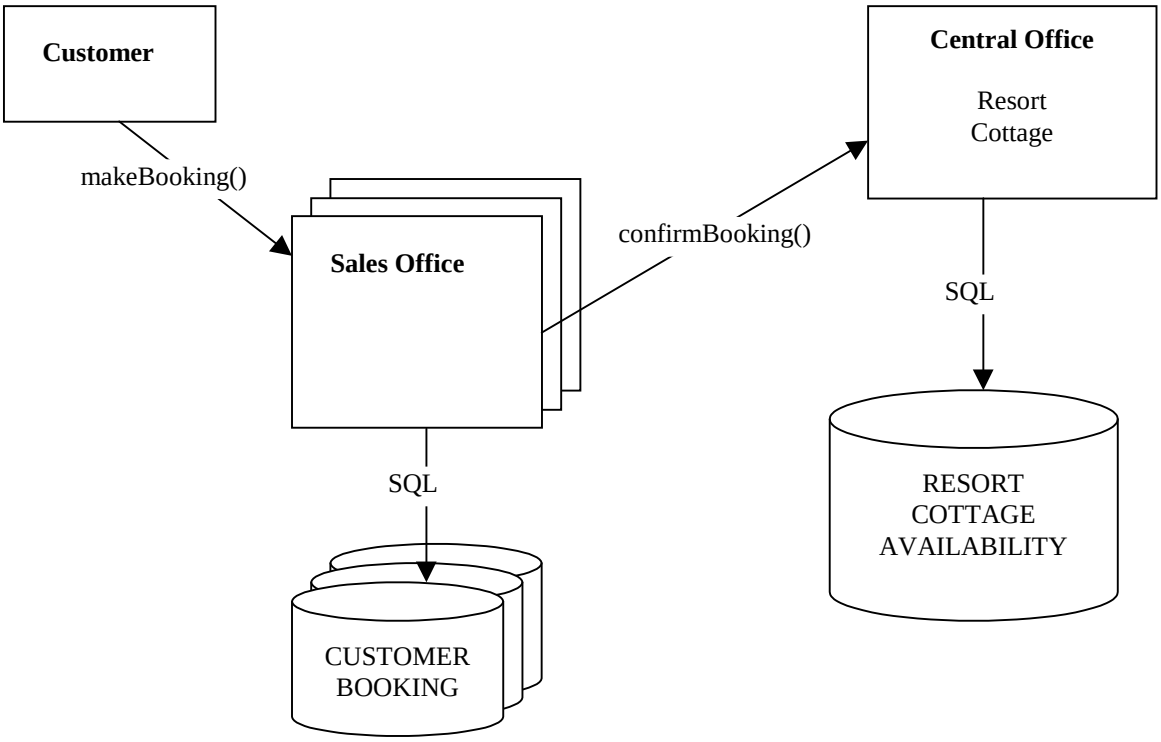


Figure 6.2: System Architecture

Number of Cottages

61, 540 `Booking`s could potentially be made each week. In order to meet this demand there should be at least **61, 600** `Cottage`s (some extra `Cottage`s are needed to meet local peaks and peaks within the peak season) in the system.

Number of Resorts

A `Resort` (small village) is assumed to consist of 100 `Cottage`s in average. That leaves us with **616** `Resort`s.

Load in the System

Totally in the system we have 61540 (bookings/week) / 5 (working days/week) / 8 / 60 / 60 (seconds/week) ≈ [1]**0.43 bookings/second**, i.e. **2.3 seconds between each booking**.

Number of Sales Offices

We assume that it would take the `SalesOffice` servers about 1 second to perform one query and 2 seconds to perform one update. A `Booking` would then take 12 * 1 + 1 * 2 = 14 seconds to execute (see Use Pattern). Therefore we need at least 14 (seconds/booking) * 0.43 (booking/s) ≈ **6** `SalesOffices.`

Load per Sales Office

With the above assumptions the load at each of the 6 `SalesOffice`s would be **14 seconds between each booking.**

[1] In these calculations, the exact value is always calculated and put into the next step of calculations. For simplicity the rounded value is shown when presented.

Peak Load in Peak Season
To simulate peak load within peak season we assume that twice as many `Booking`s could be made in the system, i.e. potentially **1.2 seconds between each booking**.

Booking procedure
Every time a `Customer` wants to book a `Cottage` the `CentralOffice` is contacted to confirm that the `Booking` can be made. Thereafter, a `Booking` is registered locally at the `SalesOffice` within the same transaction. All these transactions must be serialized and therefore could lead to the `CentralOffice` being both a single point of failure and becoming a severe bottleneck by heavy load. How many seconds it takes for a `SalesOffice` to perform a query or an update thus depends on how loaded the `CentralOffice` is. However, this drawback has not been taken into consideration at this stage because it is more of a design and implementation issue how to solve this issue.

6.2 Object Caching Development and Testing

In this section, an object cache is introduced into the PTT system. The changes to the system architecture, and the caching strategy that was chosen are described. Further, implementation issues are discussed, and the tests of the implementation are presented.

6.2.1 System Architecture

The system architecture changed in two ways:
(1) the `SalesOffice`s administer local copies of `Resort` and `Cottage` objects, (2) the `CentralOffice` sends out notifications when the source objects have changed (see figure 6.3). The source objects serve as masters in a master/slave approach.

This change was made to circumvent the `CentralOffice` becoming a bottleneck when many `Customer`s at many `SalesOffice`s use the system. The single-point-of-failure problem is only addressed indirectly by decreasing the `CentralOffice` load. Ideally some kind of primary-copy replication solution could be used (see section 3.3.5).

6.2.2 Design Issues

When implementing a caching mechanism for the example scenario, some important design decisions had to be made. These decisions together with the changed system architecture in figure 6.3 could be seen as the general caching strategy chosen.

Optimistic or Pessimistic Consistency Control?
This question is the same as: should updates be propagated within or outside the scope of the transaction changing the value? To keep the consistency of the system on an acceptable level the local caches have to be updated in some way when the source data changes. As discussed in the replication section (3.3), there are two possibilities: synchronous propagation of updates within a transaction for total consistency at any stage, and asynchronous propagation to trade-off consistency with performance. Because the `Booking` procedure ensures that no `Booking`s can be made without confirmation from the central server controlling the source data, the case when the cache is not consistent with the source only leads to the central server telling the local server that the `Booking` cannot be made. It is, however, important that the update notification reaches the local server as soon as possible so that as many `Booking`s as possible will be successful. As mentioned earlier, OrbixTalk was used to asynchronously propagate the updates. The propagation is started within the `Booking` transaction just after the central server has confirmed the `Booking` in order to notify the local servers as soon as

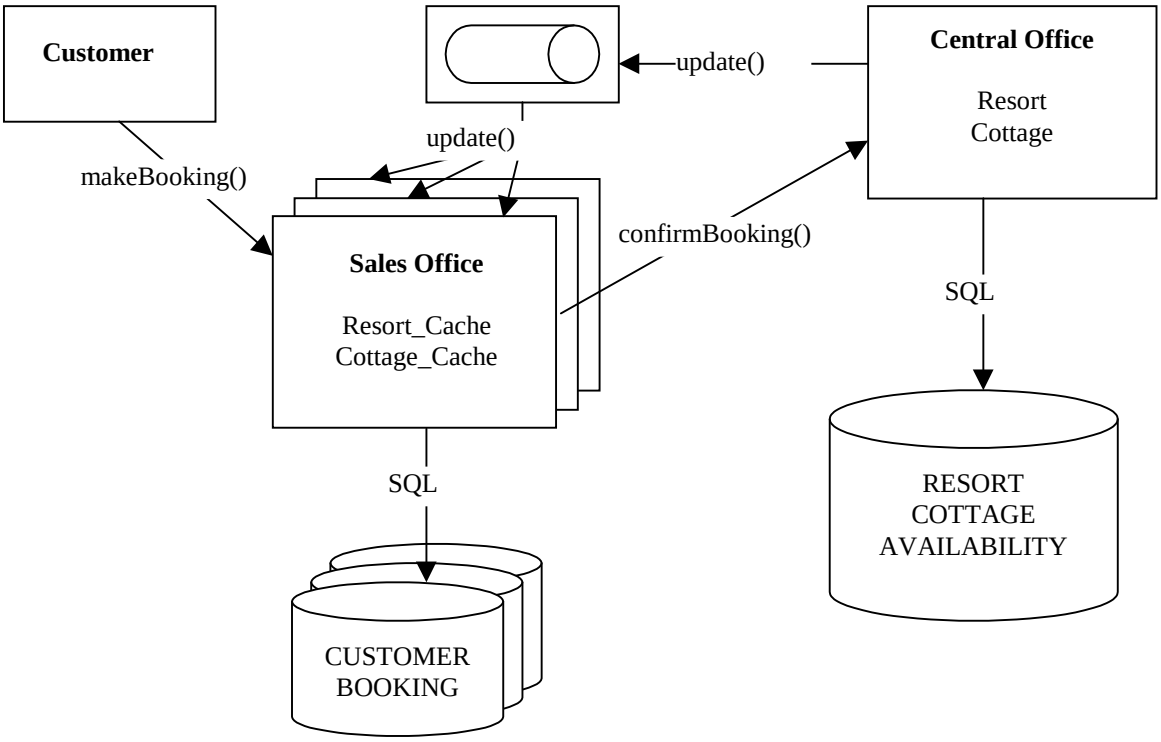


Figure 6.3: System Architecture with Caching

possible. For performance reasons, the transaction should not wait until all local servers have received the message, hence the propagation is non-transactional. The transactions will rarely abort after the central server has confirmed the `Booking`, and therefore this trade-off is acceptable.

What should be Cached?

Important properties of cached objects are that they are read frequently by many users and updated less frequently. Both the `Resort` objects and the `Cottage` objects fulfill this requirement and thus were cached at each local server. The `Resort` objects contain `Cottage`s, i.e. provide the `Customer`s with its `Cottage`s. Because of this property, the cache is hierarchically structured, and only cached `Resort`s can give access to cached `Cottage`s.

Caching in Relation to Load Balancing and Replication

The load on `SalesOffice`s is balanced to provide a better service to local `Customer`s. To make the load balancing as efficient as possible, as many of the requests as possible should be handled locally by the `SalesOffice`s without having to contact the `CentralOffice`. The object caches support this, and thus contribute to a better load balanced system. In the system, `Booking` data is replicated at each `SalesOffice` for availability reasons. The `SalesOffice`s could still do invoicing tasks using this replicated data even if the `CentralOffice` were down.

Object-Relational Mapping

The classes in the system have a 1-to-1 mapping to the database tables representing their state with one exception. The list of `Availability` contained in the `Cottage` class is represented in a separate table in the database. The clients are not aware of any database structure or persistent objects. All the database access is done transparently in the application servers.

The actual results from the SQL queries are performed in a separate module in the system, and returned in a struct format to the application servers that create objects to be put in the cache from the struct representation. This struct could be seen as the state part of the CORBA object as discussed in section 3.1.5.

6.2.3 Implementation

In this section, the problems faced when implementing the caching strategy that was chosen are discussed: *event granularity*, *localizing objects*, *serializing requests*, and how caches are *updated* or *invalidated.*

Granularity and Filtering of Events

As discussed in section 3.4.6 (cache consistency - the push approach), it is important to consider how the events generated by updates should be filtered, and how much information the event message should contain. On the one hand, not too many unnecessary messages should be sent, but on the other hand, the servers generating events should not have to do any filtering that is client dependent. In the example scenario, only one type of event (and one event channel) exists. The event is pushed from the `CentralOffice`, and caught by the `SalesOffice`s when a `Booking` has been made. Therefore the `CentralOffice` sends away just one event containing information on the `Cottage` that has been booked. The `SalesOffice`s then filter this message by checking if the `Cottage` is in the cache. When the `CentralOffice` pushes the event (just after confirmation of the `Booking`) all information on the `Booking` is available, and it is therefore straightforward to send this information with the message. The `SalesOffice` passes this information on to its cache implementation where the actual filtering is implemented. The implementation hence uses a per-class (`Cottage`) event channel but sends information on an object level to the receiver which, after filtering, enables the cache in the `SalesOffice` to update or invalidate `Cottage`s as if a per-object channel had been used. The idea behind this implementation was to send away a notification without any filtering on the server side. The notification is thereby tightly coupled to the application logic (no additional logic for sending away notifications). Further, as much information as possible was sent with the notification to the clients so that they could do the filtering and use the information passed with the event in a flexible way.

Caching/Localizing Objects

The `SalesOffice` must provide its clients with the same interface and functionality in the cached implementation as in the case when no caches are used. This means that the implementation of the `Resort` and `Cottage` classes has to be redefined to use the cache instead of contacting the database at the `CentralOffice`. The problem here is that the `SalesOffice` only returns `Resort` proxies to clients and then the clients make invocations on the resort proxy to get `Cottage`s. The `SalesOffice` must therefore return a proxy to an implementation of `Resort` that uses caching. This is done in a process called localize where the original proxies from the `CentralOffice` are converted into `SalesOffice` proxies for `Resort` and `Cottage` implementations using the cache. If the localize process is not done, the cache always returns proxies to `CentralOffice` objects, and the client still communicates with the `CentralOffice` when, for instance, accessing `Cottage`s. The cached `Resort` and cached `Cottage` implementations inherit from the original implementations but override the methods where the database is accessed, and use the cache instead. For the `Resort` class this method is `getCottages()` and for the `Cottage` class it is the `getAvailability()` method that is overridden.

```
template <class Value>
class CachedObject
{
public:
  //ctor/dtor:
  CachedObject () : m_isValid (0),
m_value (0) {}
  // member functions:
  Value* get () {
    if (!m_isValid)  {
      m_isValid = 1;
      fetch ();
    }
    return m_value;
  }
  Value* value() {
    return m_value;
  }
  int isEmpty() {
    return m_value ? 0 : 1;
  }
  void invalidate() {
    m_isValid = 0;
  }
protected:
  Value*       m_value;
  int          m_isValid;
  virtual void fetch()=0;
  virtual void localize()=0;
  virtual void purge()=0;
};
```

Figure 6.4: CachedObject Template

Serializing Non-Transactional Requests with Transactional Requests

A problem arose when the callbacks in the `SalesOffice` were executed at the same time as the `Booking` transaction was running. This caused problems because both operations access the cache. OTS only assures that transactional requests are serialized. As described above the update notifications are sent asynchronously independent of any transaction. Therefore non-transactional requests arriving at the `SalesOffice` that could access the cache had to be serialized. This was achieved by using a semaphore in an Orbix `Filter` (message interceptor) that serializes requests coming in to the `SalesOffice` server.

Update or Invalidate

There are two types of caches in the implementation. One that holds all cached `Resort`s and one that holds all `Cottage`s for a `Resort`. Both are implemented using a general cache template. A cache implementation provides a get method that is similar to a smart pointer, i.e. it hides the existence and the semantics of the cache from the clients, to access its value. This method gets the value from the source if the cache has been invalidated, otherwise the cached value is returned. A cache also has an operation to invalidate its value. Internally a cache maintains a list of remote proxies to the source objects, and a proxy to an object returning these source object proxies. Further a cache implementation has an internal method to localize remote proxies to proxies pointing to the cache. Figure 6.4 shows the C++ template for cached objects.

Every time a SalesOffice gets a notification of a Booking of a Cottage the list of cached Resorts it maintains is informed. The list of cached Resorts is then traversed to find the Cottage that has been booked. Recall that the event has Cottage information as a parameter, which makes this straightforward. If the Cottage is found in the cache, its cache entry is immediately updated (e.g. by making a remote call to the CentralOffice server to get the new value). Tests showed that it was too time consuming to invalidate the whole list of Cottages. Because the caches of all Cottages for a resort maintain a list of source object proxies, the changed entry in the cache easily can be updated without affecting the rest of the cache.

The implementation uses both invalidation and direct updating. The caches themselves - the cache of all Resorts and the caches of all Cottages for a Resort - can be invalidated. This is, for instance, used when the caches are filled for the first time; they are initialized as invalid and then after their first use, when they get their value from the source, they are valid until someone calls the invalidate operation. The single cache entries that are subject to frequent changes, i.e. the single Cottages are updated directly to keep the cache as accurate as possible. A strategy where the single cottage entries could be invalidated was also tested but it just led to extra overhead and less accurate cache entries in the tests.

Could Update Requests from Sales Offices Get Old Data?
Because the notifications, that a Booking has been made, are sent asynchronously outside the scope of a transaction after the Booking has been confirmed, it could happen that the update request from a cache reaches the CentralOffice before the original transaction has completed. In this case the cache should not be updated with the old value. This is prevented by the CentralOffice server serializing all transactions. All the caches that want to update their value will therefore wait until the original transaction has completed before the update is made.

Updating from Source?
When the event arrives at the cache and the cache entry is found, two scenarios are possible: the new value could be fetched from the source database, or the data sent with the event could be used to update the value directly. If the approach is chosen where a remote call to the source is made, the cache users are guaranteed that a cache entry (for a week to be booked) always exists if a Cottage is available. It could however happen that a week that is not possible to book is present in the cache. With the second approach, where event data is used, no guarantees regarding the cache entries can be made. However, because this approach saves one remote invocation, it is much faster than to update from the source, and also scales much better under heavier load. Both approaches are simulated in the implementation.

6.2.4 Selecting Test Configuration

To be able to do some realistic testing a test configuration was set up that took the constraints discussed in section 6.1.2 into consideration. In the real-world scenario there were 600 Resorts, 100 Cottages per Resort and about 1 Booking per second (from 6 SalesOffices) under peak load. To get an equivalent load in the test suite, the configurations described below were made.

To make the configuration easier, two SalesOffices that use two separate databases were used. Simulating the number of concurrent requests is no problem. The difficulty lies in simulating the size of the databases. For the caching strategy chosen, it is important to know how many Bookings that will be made at the same Resort, and how many Cottages that the Resort will have.

The scenario we have is that there are many queries potentially on the same Resort for Availability of Cottages before one Cottage is selected and booked. In order to maintain consistency in a safe way, the queries are using the cache if it is available, while the actual Booking always involves an update of the source data.

The caching implementation made relies on the assumption that Cottages within the same Resort will be queried many consecutive times, possibly by the same Customer. Hence all Cottages and their Availability within a Resort will be cached as soon as one request on a Cottage in the Resort comes in to the SalesOffice. The number of Resorts that can have their Cottages cached at the same time depends on memory availability and number of Cottages per Resort. If not enough memory is available, some kind of eviction policy has to be implemented. For instance the LRU (Least Recently Used) policy. This only makes sense, though, if we actually have a large system. To show performance gains with caching in our small test suite, we concluded that eviction would be too costly to use. The test environment restricts us to have cottage information in memory for about 6 Resorts with 10 Cottages each.

The load on the CentralOffice is vital to the performance of the SalesOffices. Therefore maximum load on the CentralOffice was simulated. Two tests were carried out. The first test let one SalesOffice make one Booking each second . The second test used two SalesOffices, each with the load of 1 Booking every second. The first test thus simulated peak load in peak season in the system (1 Booking each second), and the second test had twice that system load.

The fact that the Bookings are less spread over the Resorts and Cottages than in the real case, will be compensated by doing the measurements over a very short period of time, which eliminates the risk of running out of available Cottages. Table 6.1 summarizes the test configuration and the real-world assumptions.

	Real World	**Test Configuration**
#Sales Offices	6	1 and 2
#Resorts	600	6
#Cottages/Resort	100	10
# Bookings/sec	1	1 and 2

Table 6.1: Test Configuration

6.2.5 Results

The database configuration shown in Table 6.2. was set up to make measuring easy and to comply to the discussion in the previous section. For instance, one Customer was made responsible for each Booking at each SalesOffice, and no Bookings were stored in the SalesOffice before running the test in order to easily track the results in the database.

To do the caching implementation justice, an initial prefetch of Resorts, Cottages and Availability from the database was done. Further removing logging and debugging information written to disk could optimize these values. This information was crucial, though, in order to measure and monitor the tests. The figures should only be seen as a means to compare caching and non-caching servers.

Database	Table	Rows
Central Office	Resort	6
	Cottage	60
	Availability	2800
Sales Office 1	Customer	1
	Booking	0
Sales Office 2	Customer	1
	Booking	0

Table 6.2: Database Population before Test Run

With no caching and two `SalesOffice`s, the network traffic turned out to be too heavy and almost half of the `Customer`s' `Booking`s received communication failure exceptions. These exceptions were received after 1-4 seconds and they therefore make the average time of a `Booking` in this test much lower (see table 6.3).

	1 Caching Server (update from source)	1 Non-Caching Server	2 Caching Servers (update from source)	2 Non-Caching Servers
Performance (average response time per booking)	1.2 (1.3) sec	37 sec	3.5 (27) sec	46 sec
Throughput (percentage successful bookings)	100 (100) %	100 %	100 (90) %	52 %

Table 6.3: Test Results

The results show that caching improved performance drastically when many users accessed the same cache, as in the first test with one `SalesOffice`. The `Booking`s were made in 1.2 seconds with caching, and in 37 seconds without caching in average. All the requested `Booking`s were successful in this case, both with and without caching which proves that the cache was kept accurate in a satisfying way. The case with two `SalesOffice`s shows that the caching `SalesOffice`s handled the load much better, i.e. the caching implementation improved load balancing.

In the case when no caching was used, we had about twice as many remote calls as in the caching case. This proved to be very performance degrading. The database access and the 2PC part of a `Booking` was of minor importance to the overall performance.

In the non caching case every read operation has to go through the `CentralOffice` and is thus serialized. This is the same as having a distributed read lock as well as a distributed write lock in the system. As discussed in section 3.4.6 this was exactly what we wanted to avoid with our optimistic locking approach in the caching implementation in order to increase concurrency.

The fact that the `CentralOffice` easily can become a bottleneck is underlined by the two update approaches chosen. If the `CentralOffice` was contacted after each update notification, it took in average 27 seconds to complete a `Booking`, and the throughput was 90 per cent, in the case with 2 `SalesOffice`s. If the caches were updated locally using event parameters, then the corresponding figures were 3.5 seconds, and 100 per cent.

6.3 Summary

The case study is summarized in two sections: caching strategy, and ACID properties. In conjunction, the solutions listed in the two sections assure performance, scalability, and reliability of the system.

6.3.1 Caching Strategy

In the tests, objects accessed by many users like `Resort` and `Cottage` were cached with a performance improving result. The caches were located at an application server level and existed for the lifetime of the server without any eviction. Object faulting was implemented by a generic get method that first checked whether the value was valid, and if not, fetched the new value from the source. Other important operations in the cache interface are localize, fetch, update, and invalidate.

The caches were synchronized using a pushing CORBA Event service model. This proved to help the throughput in the tests. Events were generated on a per-class level but the events had a parameter that enabled clients to filter the message and do updates on an object level.

The cached CORBA objects (`Cottage` and `Resort`) were mapped to C++ structs. The `Cottage` object was mapped to two structs (`Cottage` and `Availability`). Further, the `Availability` struct was mapped both to the central `Availability` table and to the replicated `Booking` table. The IDL interfaces, however, suppress these details.

6.3.2 ACID Properties

Atomicity of the distributed transactions was assured by OTS. The confirmation of a `Booking` and the actual `Booking` were always carried out in an “all-or-nothing” fashion, which assured that consistency wasn't broken by the transaction.

Consistency must be assured on the application level or the database level (e.g. with triggers). It was assured on an application level by an “implicit” optimistic locking approach where clients read the cached value but couldn't change it before confirming that the value read conformed to the source. The caches were therefore kept consistent with the source in an asynchronous or optimistic way allowing inconsistency for a short period of time. The shorter the time of inconsistency, the likelier it is that the optimistic locking is successful, i.e. that the transaction in the end succeeds. Therefore an “as-soon-as-possible” policy was chosen were the updates of the source were pushed out to all the caches using an asynchronous event channel mechanism, and the caches were updated immediately when receiving the event.

Isolation was assured partly by OTS by serializing all transactional requests at both the `SalesOffice`s and the `CentralOffice`. The database also assured transactional isolation by holding read and write locks on data.

The application server layer assured durability (persistence) of objects by issuing SQL commands to the databases connected to the transaction by OTS. Durability was also assured partly by OTS by keeping logs to enable recovery.

The ACID properties were kept totally transparent to the clients who are making the `Booking`s. All the transactions originate either from one of the `SalesOffice`s, or from the `CentralOffice`.

7 Architectural Abstractions from the Case Study

This chapter presents an approach to software architectural modeling of the case study that was described in the previous chapter. The first section of this chapter describes the case study implementation using *UML* [OMG 97c]. In the second section, the *Connector framework* is used to describe the system on a higher and more abstract software architectural level. The implementation and architectural diagrams in combination form a pattern for designing an object cache of transactionally replicated data.

7.1 Implementation Modeling with UML

In this section, the PTT case study implementation described is modeled using three different views: *class view* (static structure), *uses view* (dynamic structure), and *physical view*. The first two views focus on the implementation of the `SalesOffice`, the locus of the cache functionality.

7.1.1 Class View

The static structure, i.e. the class dependencies at compile time (represented by attributes and inheritance), is depicted in figure 7.1. Three significant properties of the cache are exposed in this view.

First, there is a generic cache template that all implementations of cached values must inherit from. This template (`CachedObject<Value>`) encapsulates the semantics of the cache, and thereby makes the cache functionality transparent to the clients.

Second, there are compositional dependencies between `SalesOffice_i` (_i is used to denote an implementation of an interface), `ResortsCache` (manages a collection of cached `Resort`s), `Resort_i_Cache` (a cached `Resort`), `CottagesCache` (manages a collection of cached `Cottage`s), and `Cottage_i_Cache` (a cached `Cottage`). This means that no objects from the lower levels (higher up in the diagram) can exist if an object on a higher level doesn't exist.

Third, `SalesOffice_i` is an implementation of the IDL-interface `CacheUser`. A `CacheUser` will receive information on changes to the source.

7.1.2 Uses View

In figure 7.2 the run-time dependencies of the classes (dynamic structure) are depicted. The operations that are accessible remotely (system interfaces) are marked in bold. Further, +, * , and – denote public, protected, and private operations or attributes respectively. The `SalesOffice` communicates with the cache using three methods. With `get()` the cached `Resort`s are retrieved. The other two methods are used when the `SalesOffice` receives a notification that the source has changed. They correspond to the choice whether to update from the source, or to use event parameters to change the cache (discussed in section 6.2.3).

7.1.3 Physical View

Figure 7.3 shows the physical view of the whole PTT system. The `CentralOffice` resides on a central node. The `SalesOffice`s and `Customer`s are distributed over multiple nodes. The database at the central node contains all `Resort`s, `Cottage`s, and their `Availability`. Each `SalesOffice` manages a database containing their local

Customers and the Bookings made at that SalesOffice. The Booking table hence has replicated data from the central Availability table. Each time a booking is made a row in the Availability table is deleted, and a row in the local Booking table is added. This is done within a distributed transaction. These tables are therefore always consistent. The PTTSql component is responsible for separating database calls (embedded SQL) from application logic code in the CentralOffice and SalesOffice servers.

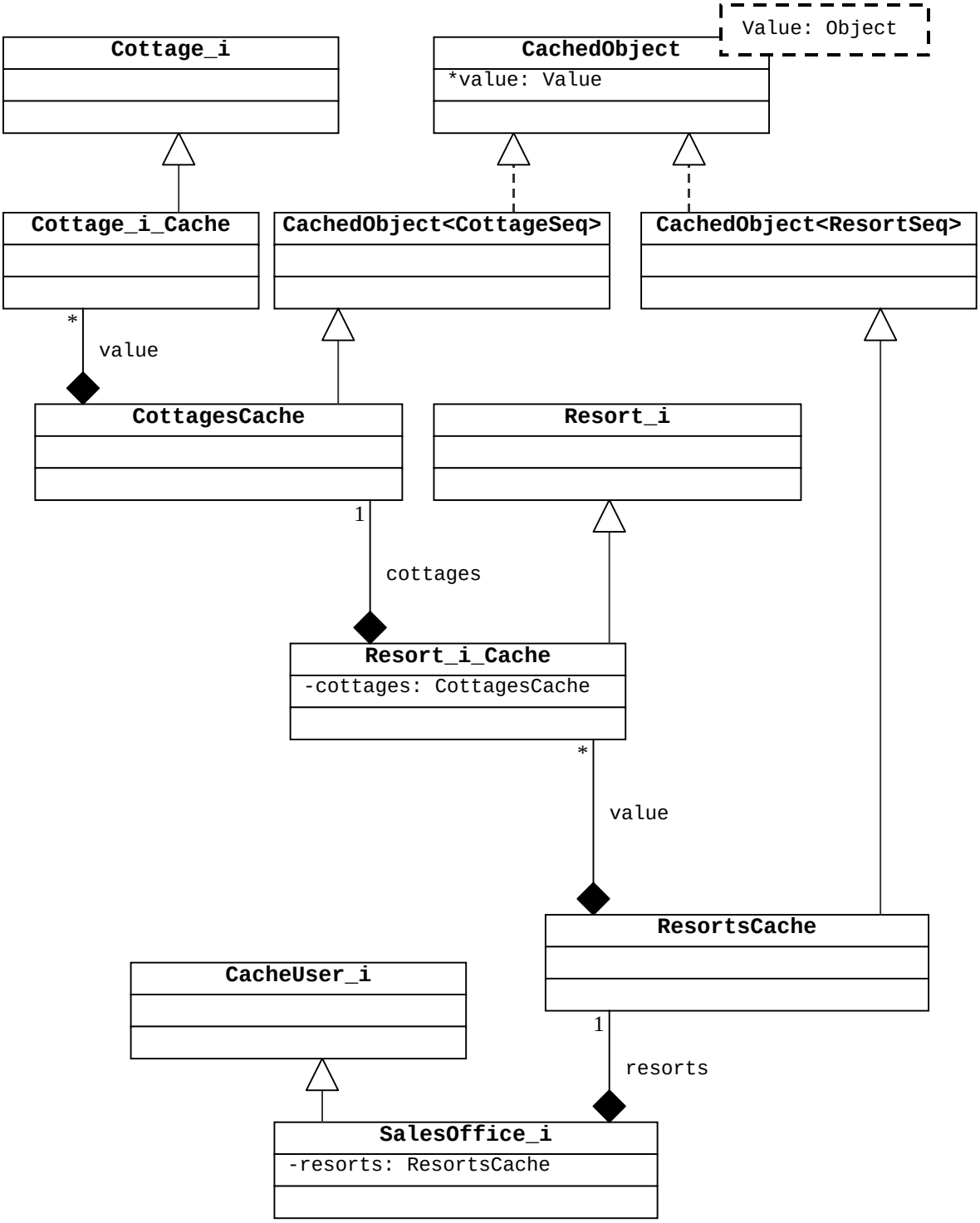


Figure 7.1: Cache Implementation in Sales Office - Class View

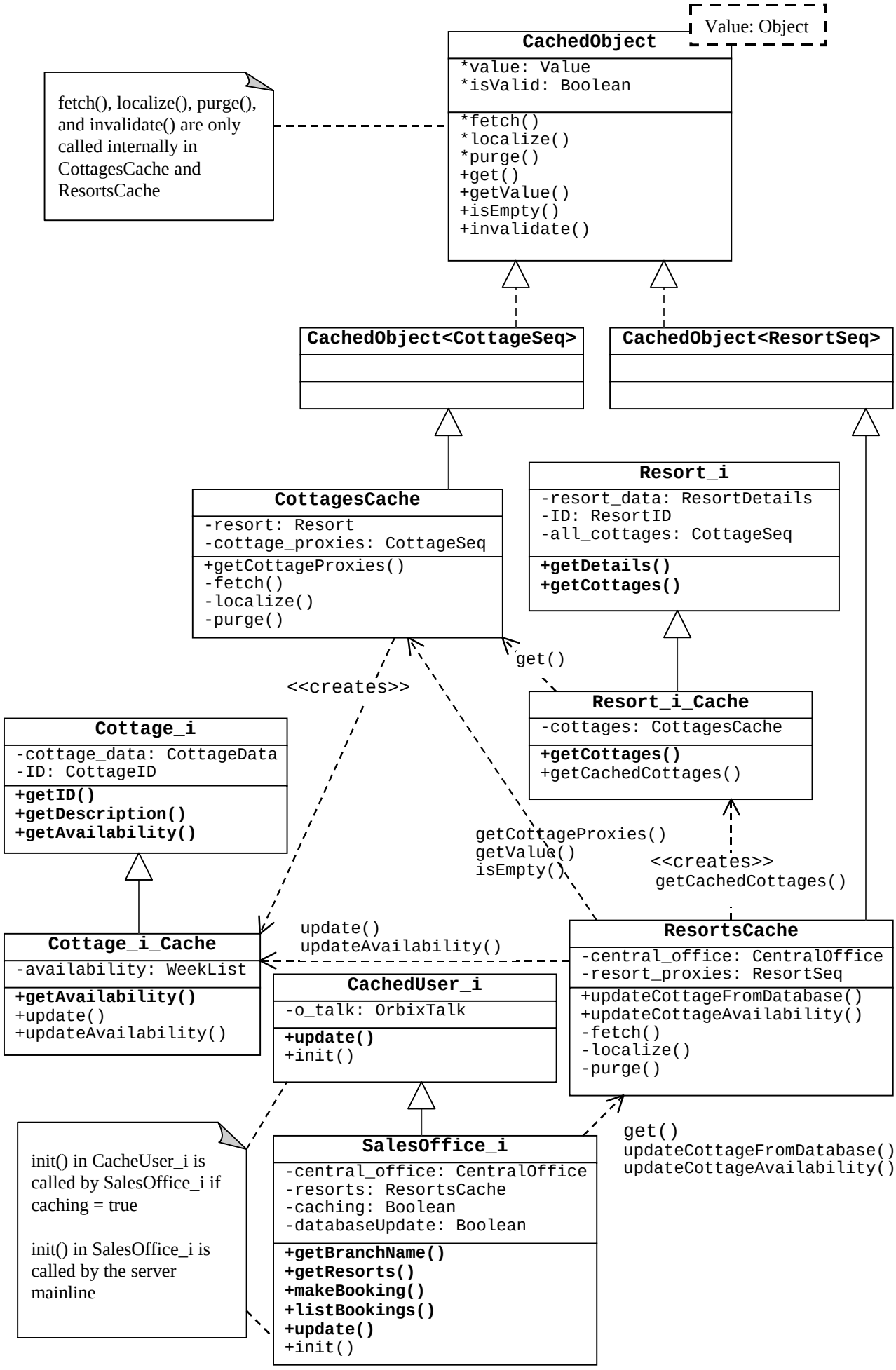


Figure 7.2: Cache Implementation in Sales Office - Uses View (with inheritance)

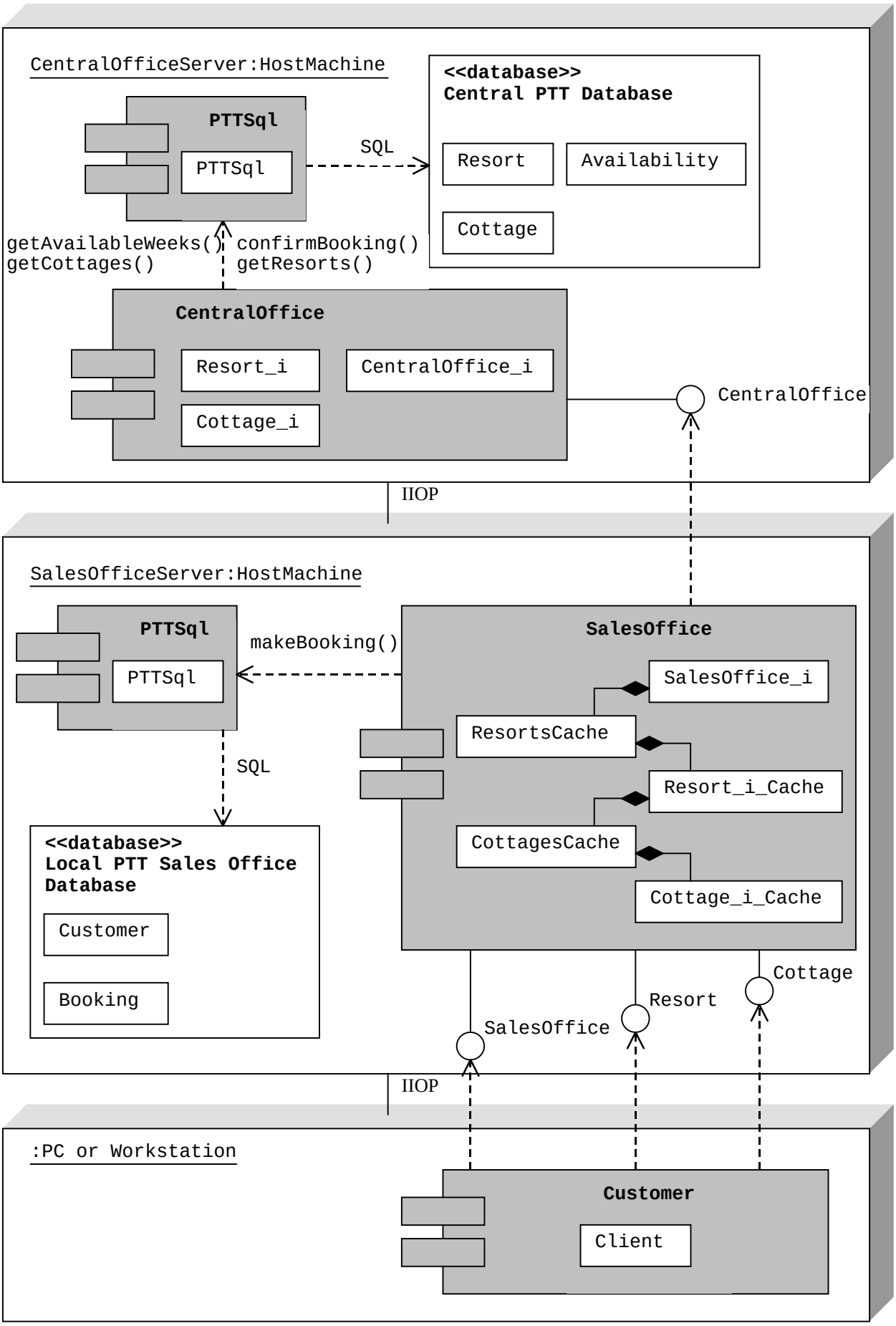


Figure 7.3: PTT System - Physical View

7.2 Architectural Modeling with the Connector Framework

This section introduces an approach to modeling distributed components on a software architectural level. The first step in the Connector framework (presented in section 4.3.4) approach is to describe the *components* involved as independent entities. A *connector* is then modeled to describe the interactions between the components in an abstract way using roles and interaction protocols. The third step is to instantiate the roles with the components to form an abstract architecture. Finally, the abstract architecture is refined by showing how the components are realized. The result is called a concrete architecture.

By describing the connector *ObjectCaching_with_TransactionalReplication,* a generic design is proposed. The design could potentially be reused in a CORBA environment for designing an object cache while assuring consistency of replicated data.

7.2.1 Components

As a first step to describe a system architecture, the components and their *core functionality* are analyzed and modeled. Core functionality refers to the domain-oriented component behavior. The exported and imported system level interfaces of each component are described. The interfaces could be seen as a contract that each component signs. If the required (imported) interfaces are available, then the component promises to offer some services (exported interfaces). Each interface should have a complete interface description (e.g. in some IDL) including exceptions, type definitions, and pre- and postconditions. Components in the case study that either import or export application defined system level interfaces are described below. In the depicted components, CORBA IDL is used to describe the interfaces. The complete IDL specifications for the PTT system are shown in Appendix B.

SalesOffice
Figure 7.4 shows the `SalesOffice` component. This view does not expose anything from the caching implementation. These more detailed interfaces will be introduced after having investigated in which interactions the `SalesOffice` takes part. The `SalesOffice` must call `confirmBooking()` on `CentralOffice` in order to make a `Booking`. Further, the `CentralOffice` is called to retrieve `Resort`s. These interactions must take place, regardless of the technical solution chosen to implement the component's interface. Therefore the `CentralOffice` interface is shown as an imported interface.

CentralOffice
The `CentralOffice` component is depicted in figure 7.5. This component does not depend on any other interfaces for its core functionality. The `CentralOffice` provides interfaces to `Resort` and `Cottage` objects in order for the `Customer` to make a `Booking`. If a component only exports interfaces, it is totally independent and easier to reuse.

Customer
The `Customer` is the client component in the system and is not subject to reuse.
This component (figure 7.6) does not export any interfaces, but imports the `Resort`, `Cottage`, and `SalesOffice` interfaces. The core functionality of a `Customer` is to browse for `Cottage`s in `Resort`s, and to make `Booking`s at the local `SalesOffice`.

Three other components are also used in the case study: the `OrbixTalk`, `OTS`, and `PTTSql` components. `OrbixTalk` and `OTS` correspond to *CORBAservices* and are therefore reusable components that are independent of this case study. Sample component descriptions of the

OTS and the Event service can be found in [Tai & Busse 97, Tai & Busse 98]. The PTTSql component (used for database access) is not described here, as it neither exports nor imports any system level interfaces.

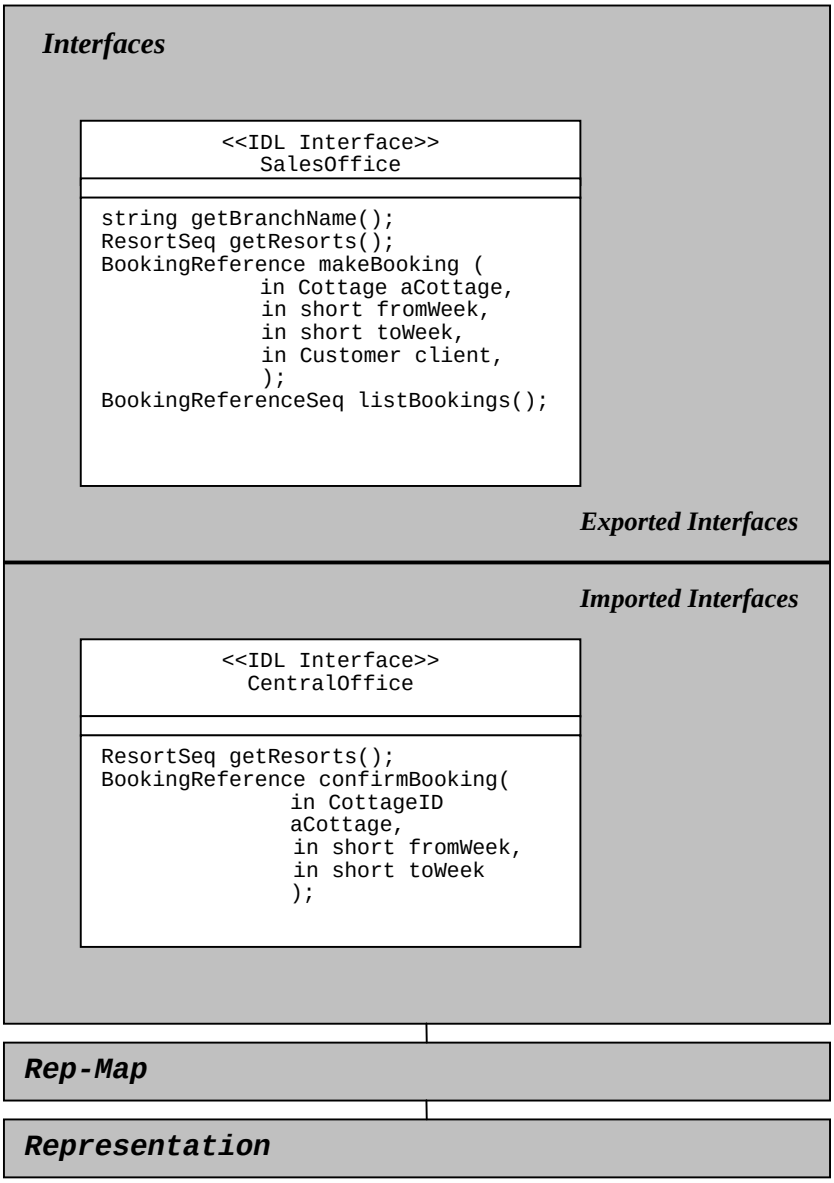


Figure 7.4: Component SALESOFFICE Core Functionality

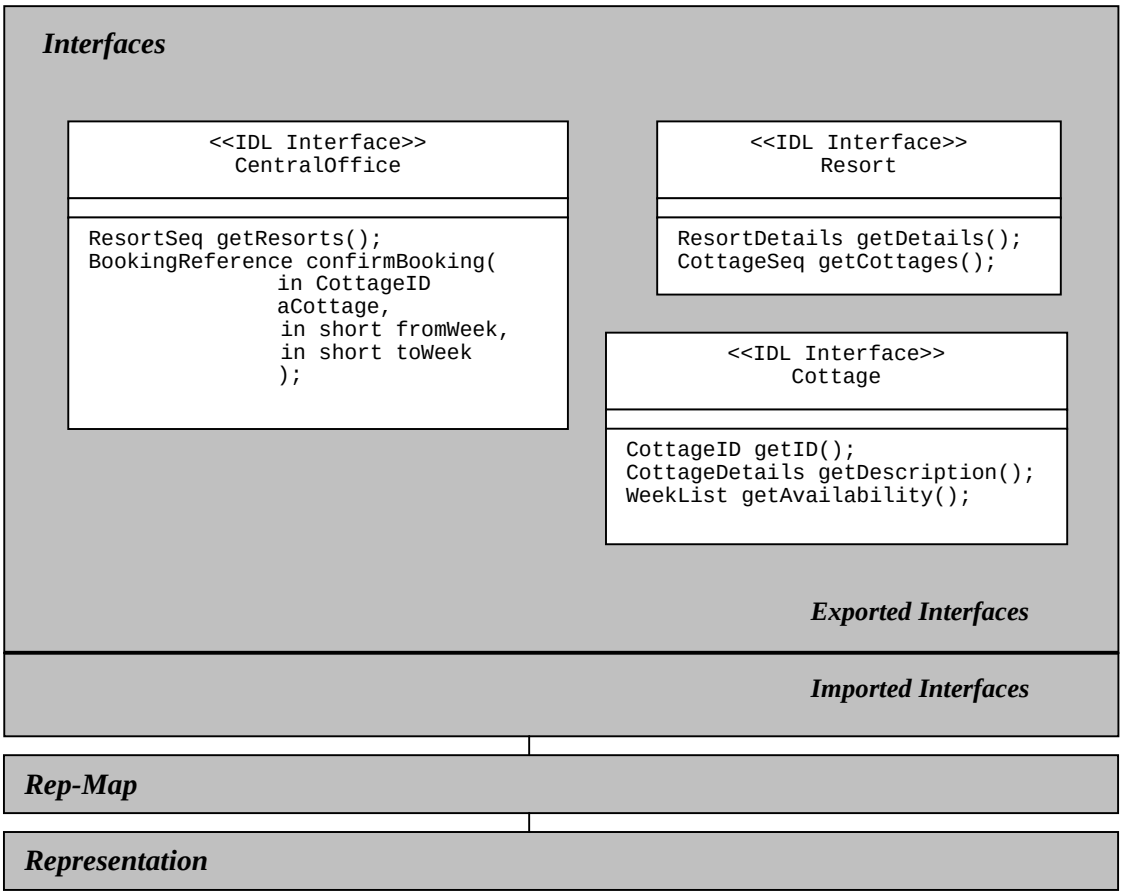


Figure 7.5: Component CENTRALOFFICE

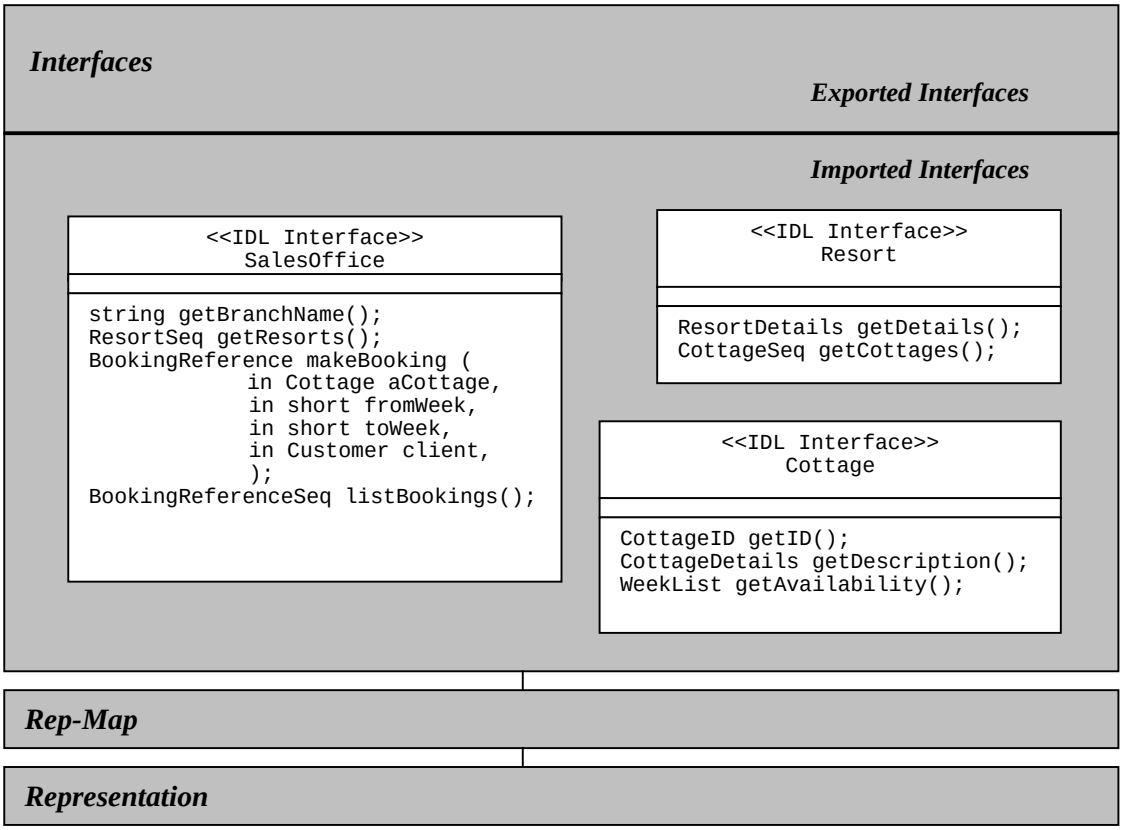


Figure 7.6: Component CUSTOMER

7.2.2 Connector ObjectCaching_with_TransactionalReplication

In this section, the *ObjectCaching_with_TransactionalReplication* connector is introduced. The connector is a description of interactions between generic components called roles. The connector can thus be applied to different caching and replication scenarios. *Roles*, *role interfaces*, *interface usage*, and *interactions* comprise a specification of a connector [Tai 98a].

Roles

Roles are generic participants of an abstract interaction. They are played by components. By describing roles instead of specific components, common patterns are exposed when specifying role interactions. The following roles can be identified:

TransManager, // drives 2PC protocol, e.g. OTS
TransObject, // executes transactional requests
EventManager, // propagates updates asynchronously
SourceObject, // in central server, e.g. Cottage_i
SourceManager, // e.g. CentralOffice
CachedObject, // in local server, e.g. Cottage_i_Cache
CacheManager, // e.g. SalesOffice
Client // e.g. Customer

Role Interfaces

The next step is to define interfaces for the roles. Components playing these roles must support the interfaces specified here. The interfaces are used to specify dependencies and interactions between roles. The arrows "<" and ">" refer to plug points that are replaced with application specific interfaces when the connector is instantiated. `<Object>State` denotes the state of `Object`. "| value |" denotes that value is optional. Further, OID refers to an object identifier. The interface operations are mainly used for inter-role communication, i.e. they are on a *system level*. However, some operations are only used internally, and are listed because they show important characteristics of the role.

- **Transaction interfaces**

```
TransManager.Current { //Current is an interface specified by OTS
      void begin();
      void commit(in boolean report_heuristics);
      void rollback();
      ...
}
TransObject.TransactionalObject {}  //TransactionalObject is an
                                    //interface specified by OTS.
                                    //It indicates that the object
                                    //is transactional
```

- **Event interfaces**

```
EventManager.<Registration> {
      //The Registration interface in the case study was OrbixTalk
      //This API wraps the CORBA Event service PushSupplier-
      //PushConsumer model.
      CORBA::Object registerTalker(in string subject,
                                   in string cacheUserInteface);
      void registerListener(in CORBA::Object cacheUser
                            |,in string subject|);
      //CORBA::Object is cast to a CacheUser proxy.
      //By registerListener, subject does not have to be
      //passed. E.g. in OrbixTalk the object marker is used.
}
EventManager.CacheUser {
```

```
      //In the CORBA Event service this interface corresponds to
      //PushConsumer and the operation is called push. Similar to
      //the previous interface, this interface could be seen as a
      //wrapper on top of the Event service API
      oneway void update(in any event);
}
```

- **Source Interfaces**

```
SourceObject.<SourceObject> {
      <SourceObject>State getState(in OID id);
      ...
}

SourceManager.<CentralManager>:TransactionalObject {
      //public use:
      <SourceObject> get(in OID id);
      boolean confirmModify(in OID id, in any value);
      //internal use:
      void makePersistent(in OID id, in any value);
}
```

- **Cache Interfaces**

```
CachedObject.<SourceObject> {
      <SourceObject>State getState(in OID id);
      ...
}
CacheManager.CacheUser {
      oneway void update(in Any event);
}
CacheManager.<LocalManager> {
      //public use:
      <SourceObject> get(in OID id);
```

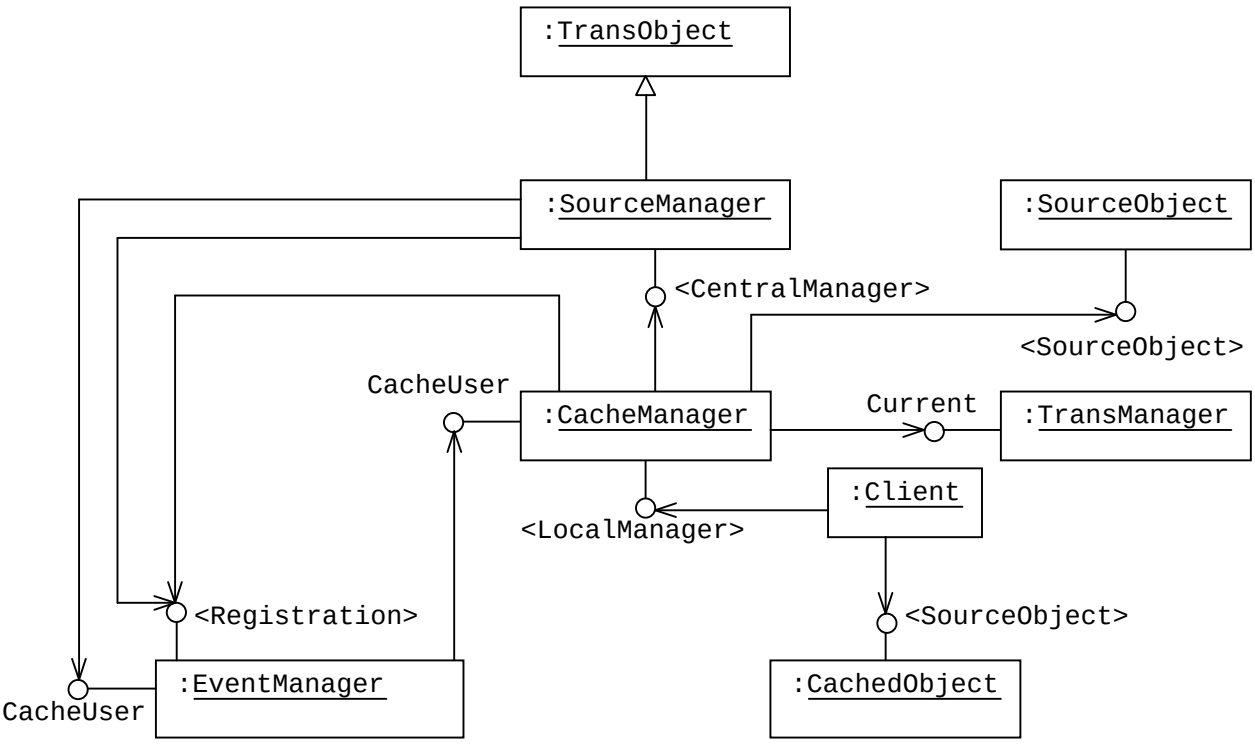


Figure 7.7: Role Interface Usage

```
        void modify(in OID id,in any value);
        //internal use:
        boolean cacheIsValid();
        <SourceObject> createCache(in OID id,
                                   in <SourceObject>State state);
        void updateCache(in OID id, in <SourceObject>State state);
        void convertEvent(in any event,
                          out OID id,
                          out <SourceObject>State state);
        void makePersistent(in OID id, in any value);
}
```

Interface Usage

Figure 7.7 shows the dependencies between the roles (UML notation), i.e. how the interfaces are used. Notable is that neither the `CacheManager` nor the `SourceManager` communicate with the objects they manage directly by using public interfaces. These managers are only responsible for creating and deleting the objects. Furthermore, they are responsible for adding transactional and event based behavior to the cache. Transactions and events are thereby kept transparent to the `Client`, `SourceObject`, and `CachedObject` roles.

Interactions

Three interactions of this connector are specified below using UML sequence diagrams: cache initialization and use (figure 7.8), replicated data modification (figure 7.9), and cache update (figure 7.10).

- **Cache Initialization and Use**

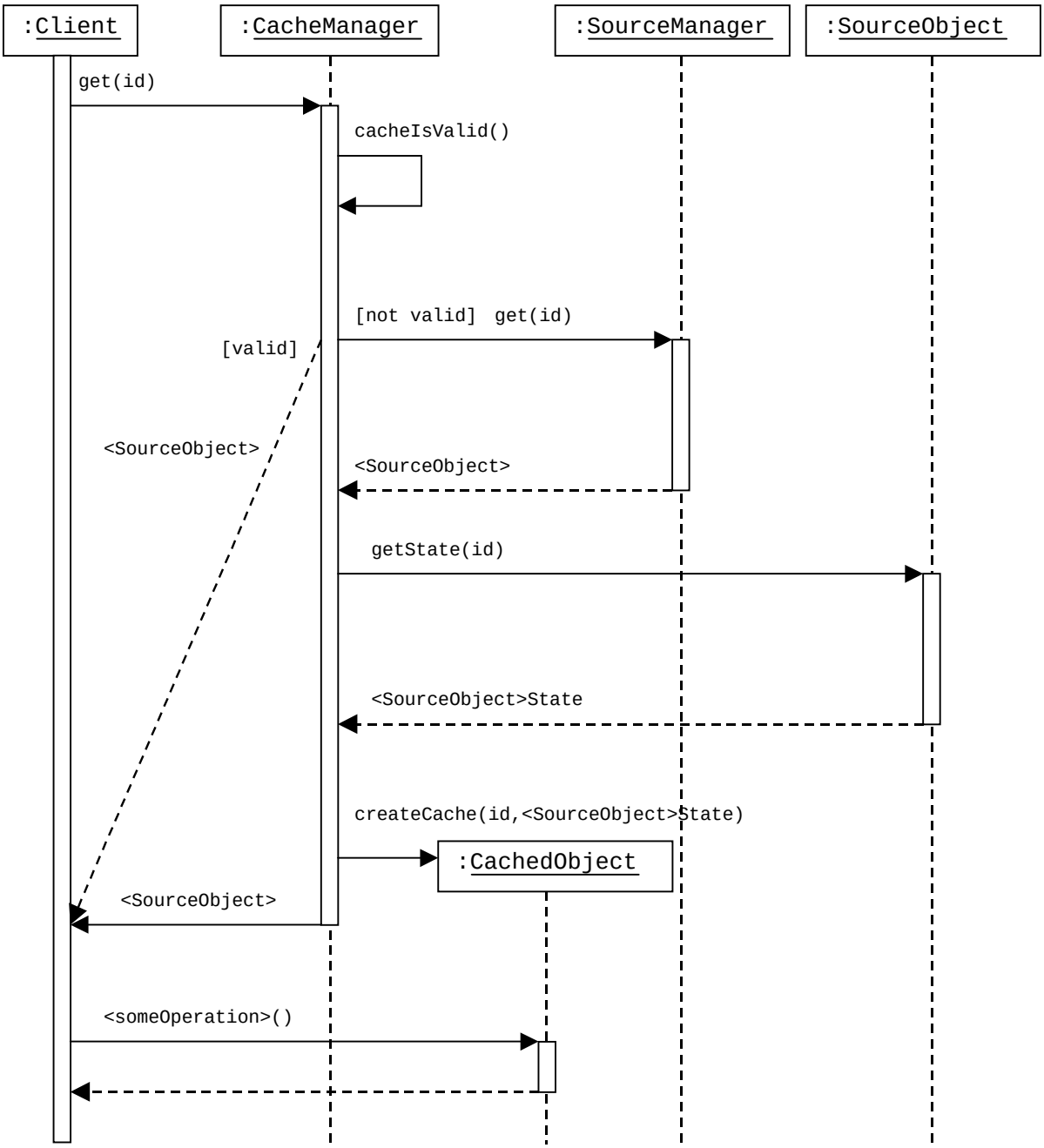
This interaction describes the basic cache functionality. The `Client` sends a requests for an object. If there is a valid object in the cache, it is returned directly by the `CacheManager`. Otherwise, the accurate state of the source object is fetched from the source. The state is used to create a cached object. A reference to the `CachedObject` is returned to the client. The next operation on the object will use the cache if it hasn't been invalidated.

- **Replicated Data Modification**

This interaction shows how the source data is kept consistent with the locally stored data. When a client wants to modify a value, a transaction is started. Within this transaction a confirmation with the `SourceManager` is done, and the local database is updated. Since these operations are performed in an "all-or-nothing" fashion, the replicated data is always consistent. The confirmation with the `SourceManager` serves to detect whether other clients have updated the source concurrently, and a conflict thereby has occurred. In the case study, such a conflict occurred when two clients selected the same `Cottage` and week from the cache, and then tried to book it concurrently. A conflict leads to a race condition where the first transaction to execute will succeed, and the second one will roll back.

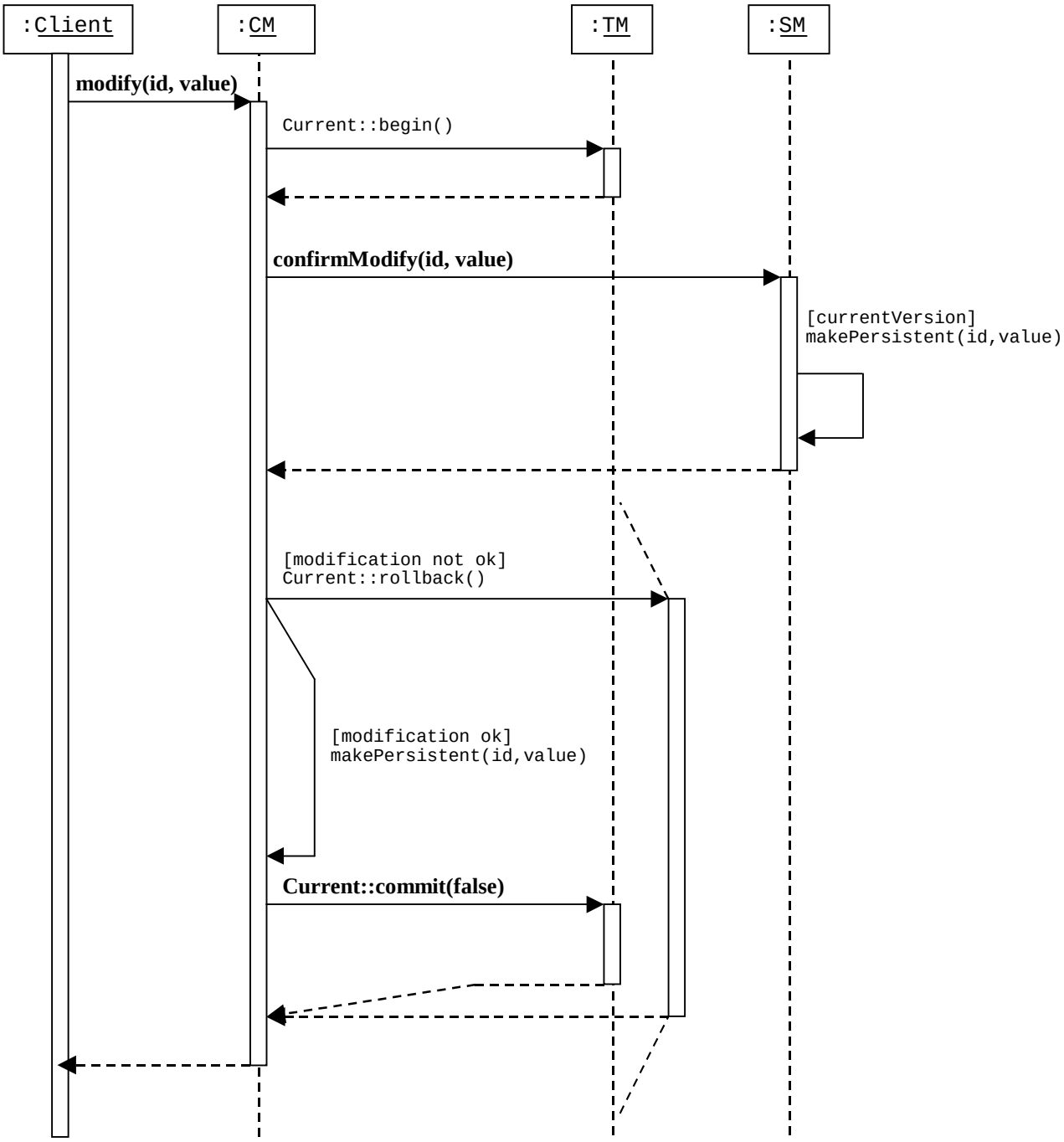
- **Cache Update**

The caches are updated by using event notifications. The events are pushed from the `SourceManager` to the `EventManager` when a SourceObject has changed. The `EventManager` then pushes the events to the registered `CacheManagers`. The `CacheManagers` must filter the event to find out whether the object that has changed is in the cache. If it is in the cache, the value can be updated in two ways. Firstly, it can be updated by getting the state from the source. Secondly, it can be updated locally by using the value passed by the event. The pros and cons of the two approaches were discussed in section 6.2.3.



Pre SourceManager and CacheManager are initialized
Client has proxy to CacheManager
Post Client communicates with CachedObject

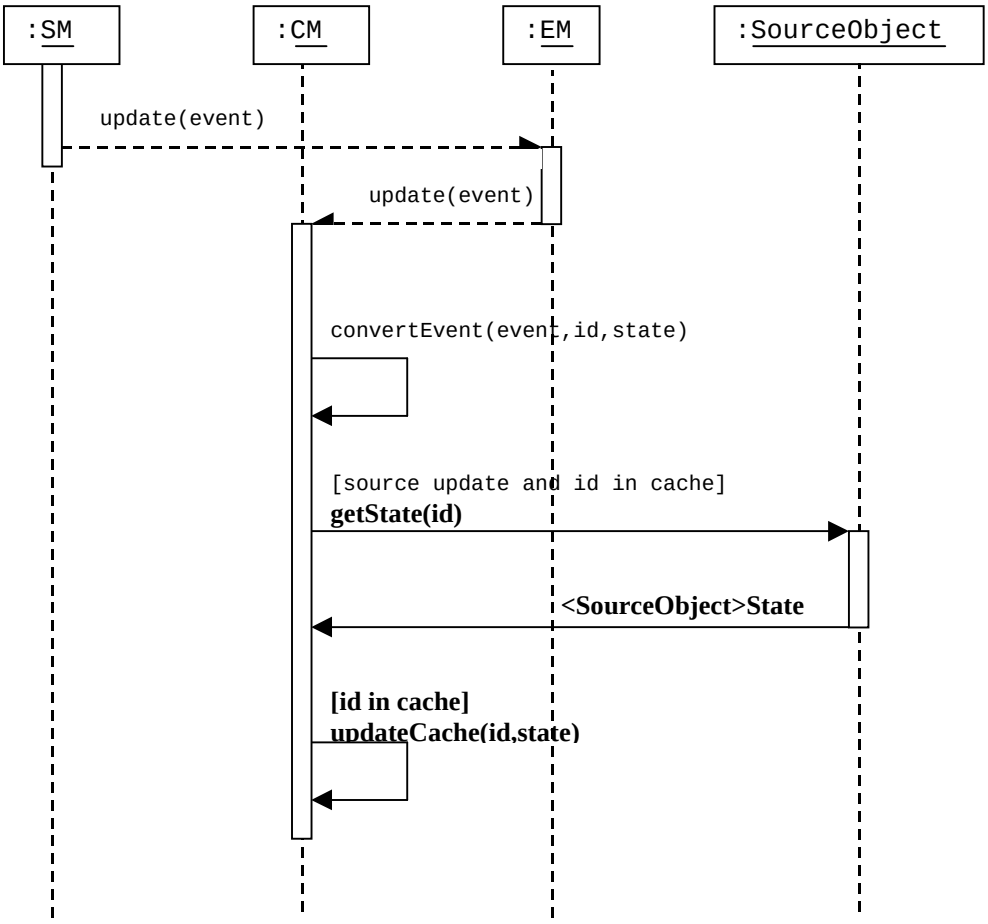
Figure 7.8: Interaction Cache Initialization and Use



TM – TransManager, CM – CacheManager, SM – SourceManager

Pre Interaction *Cache Initialization and Use*
Post Booking has been recorded by both CM and SM
or by neither of them

Figure 7.9: Interaction Replicated Data Modification



SM - Source Manager, CM - CacheManager, EM - EventManager

Pre SM has called registerTalker(subject,cacheUserInterface) and
CM has called registerListener(this |,subject|) and
SM has performed makePersistent(id,value) in
Interaction *Replicated Data Modification*

Post If object that changed was in the cache, it is
consistent with SourceObject

Figure 7.10: Interaction Cache Update

7.2.3 Abstract Architecture

With the connector now being specified, the next step is to instantiate the connector with specific components. The result is called an abstract architecture. The PTT abstract architecture using the connector *ObjectCaching_with_TransactionalReplication* is depicted in figure 7.11. By assigning a role to a component, all interfaces of the role are assured to be provided by the component. Further, the component must be able to participate in all interactions specified for the role. The plug points from the generic pattern is replaced by application specific interfaces. E.g., `<LocalManager>` is changed to `SalesOffice`, `<CentralManager>` becomes `CentralOffice`, and `<SourceObject>` is replaced by `Resort` and `Cottage` interfaces. The diagram in figure 7.11 specifies detailed and complex system behavior in a concise, but yet expressive model. This model does not describe how the behavior is implemented, though. In order to specify the system realization, the component models have to be refined.

7.2.4 Concrete Architecture

The abstract architecture is made to a concrete architecture by describing how the components' behavior is implemented. All components are described in three views: imported and exported interfaces (system level), imported and exported representation-map (programming level interfaces), and representation (implementation).

The concrete architecture for the `SalesOffice` component is depicted in figure 7.12 and 7.13. The caching functionality is now visible in the exported interfaces by exposing a `CacheUser` interface, and the interfaces of the cached objects. To keep the cache accurate the source objects have to be contacted. This is done by using the `Resort` and `Cottage` interfaces from the `CentralOffice` (imported interfaces). The `SalesOffice` uses the OTS-specified `Current` interface to start, commit and abort transactions. All the operations on the `CentralOffice` must be executed within a transaction. This is assured by inheriting from `TransactionalObject` (as specified by the OTS).

In the representation-map, the programming language classes that correspond to interfaces to other components are shown. Notable here is that the `OrbixOTS` and `OrbixTalk` interfaces are not exposed on a system interface level. They, however, still represent interfaces to other components, and are therefore depicted in the imported representation map. The tags, `<<Generated C++ Class>>`, denote classes generated from the IDL interfaces by an IDL compiler (<<>> is used for specifying UML stereotypes).

The representation part of the model describes the classes used to implement the specified behavior. The classes ending with `BOAImpl` are skeletons used to convert the remote requests to C++ method invocations on the implementation object. Dependencies on other components are shown by links to the representation map. E.g., the `CacheUser_i` class uses `OrbixTalk`, and `SalesOffice_i` uses `CosTransactions::Current`. The representation part corresponds to the models in figure 7.1 and 7.2.

7.3 Summary

In this chapter the implementation of the case study was modeled. Two different modeling approaches were used: implementation centric modeling with UML, and software architectural modeling with the Connector framework. The UML models showed class structures, and the physical distribution of components on different hosts. Although these models are useful to understand the implementation they offer little support for reuse.

Since the objective of this part of the work was to find abstract reusable designs to the case study, software architectural modeling was carried out. Using the Connector framework, the implementation was abstracted into a connector called ObjectCaching_with_Transactional-Replication. The connector is a generic interaction specification, and provides plug points for different components and implementations. Therefore the connector was proposed as a generic design of object caches for transactionally replicated data.

A connector is realized by instantiating the connector roles with components and refining the components to an implementation level. The realization of the specified connector was demonstrated by describing the `SalesOffice` component on three levels: the system interface, representation-map, and representation levels. This separation offers support for reuse, and is well suited for describing CORBA based systems. This is further underlined in [Tai & Busse 97, Tai & Busse 98, Tai 98b] where the CORBA Event service, CORBA OTS, and a CORBA OTM system are described using the Connector framework.

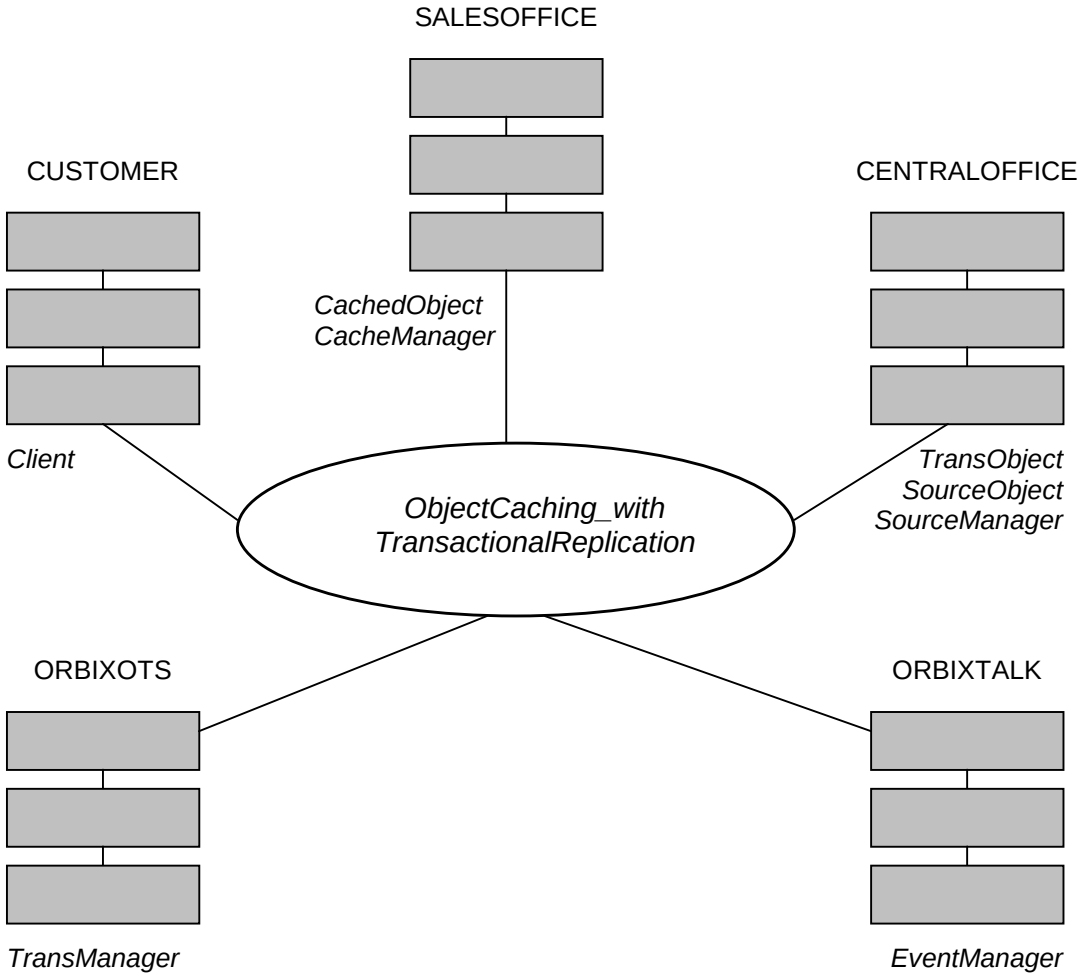


Figure 7.11: PTT System – Abstract Architecture

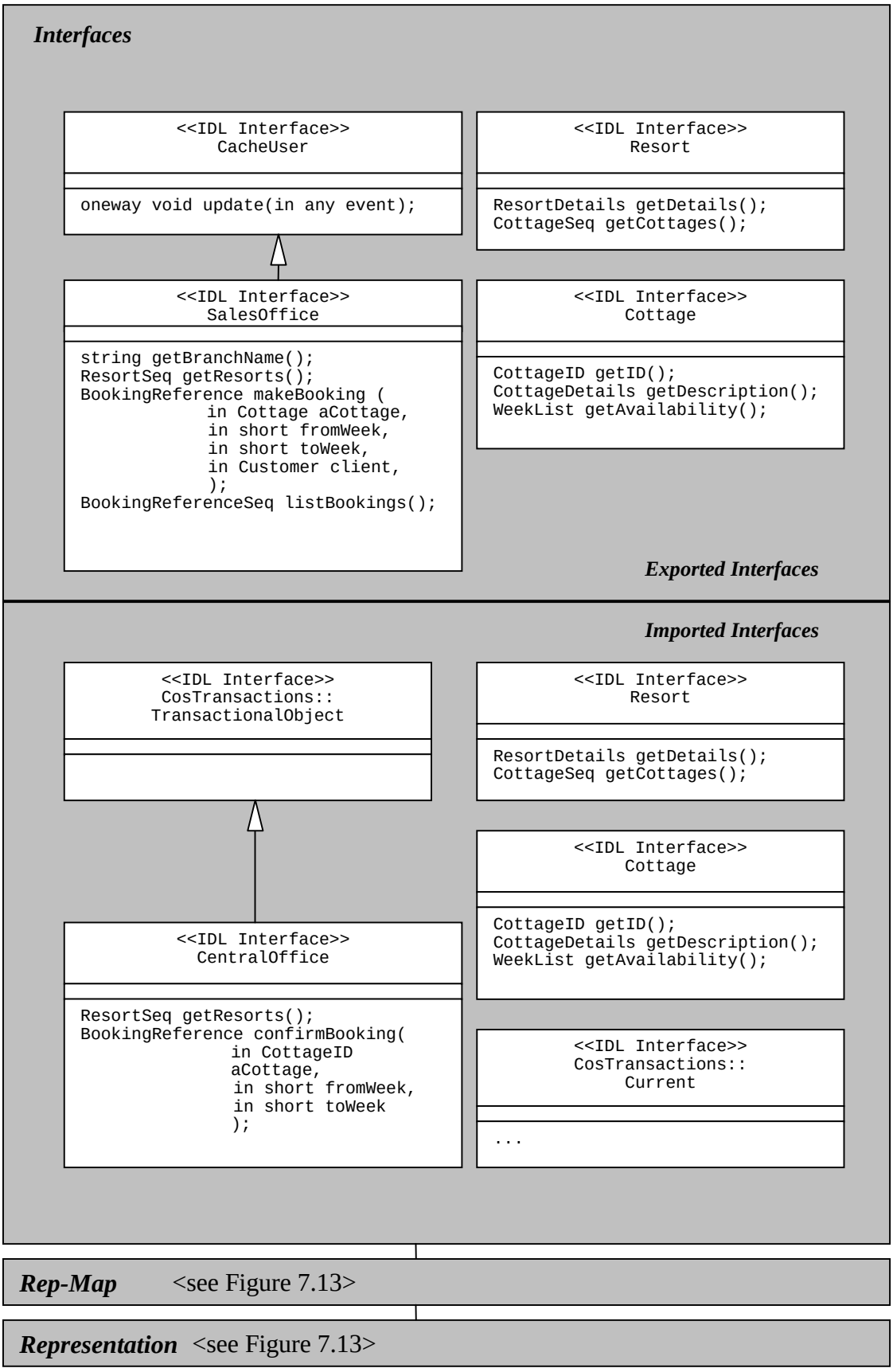


Figure 7.12: Component SALESOFFICE Concrete Architecture (Interfaces)

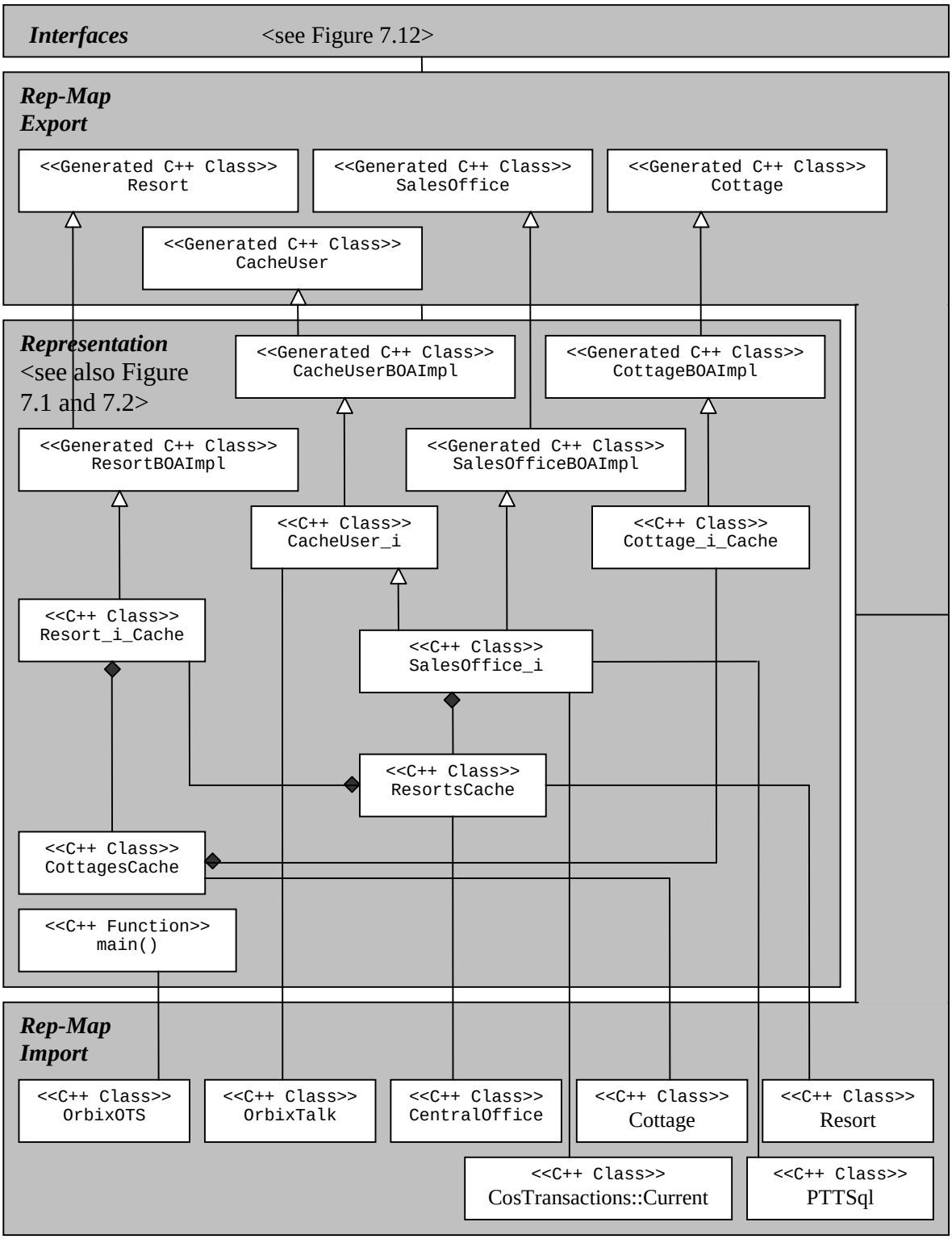


Figure 7.13: Component SALESOFFICE Concrete Architecture

8 Conclusion and Discussion

This chapter summarizes the investigations of influences of object caching on reliability, scalability, and performance in a transactional, object-relational CORBA environment, and discusses how this is captured in the generic design proposed in the previous chapter.

8.1 Reliability

Reliability of the object caching strategy was achieved by using distributed transactions and asynchronous update messages. The distributed persistent data corresponding to the cached objects was always kept synchronous by using the 2PC protocol. The caches themselves were, however, synchronized in a less stringent way by using asynchronous non-transactional pushed updates. The reason for this was to gain performance. To compensate for the inconsistencies of the distributed caches, an optimistic locking approach was chosen. Before modifications could be completed, a confirmation was done to detect inconsistencies. Furthermore, to ensure that as many modifications as possible were successful, giving the impression that the caches were absolutely synchronous with the source, the updates were sent as soon as possible after the source confirmation.

In terms of the proposed generic design, this behavior is captured as follows. The `CacheManager` confirms modification with the `SourceManager` and replicates data from the `SourceObject` using the 2PC protocol (Interaction replicated data modification). The `SourceManager` sends an update event to the `CacheManagers` through the `EventManager` after modification has been confirmed (Interaction cache update).

8.2 Scalability

Scalability was mainly achieved by using load balanced servers, i.e. multiple servers offering the same service. To improve scalability as many of the services offered as possible should be possible to perform locally without contacting the central server. Two service localization techniques were used to accomplish this. First, caches minimized central server load by performing all read requests locally, and only contacting the central server for updates. Second, modifications made at the local servers were replicated (made persistent) locally, and kept absolutely synchronous with the source. Querying modifications that the local servers have made is thus the same as querying the source.

In the generic design, the `CachedObject` offers the same interface as the `SourceObject`. The `CacheManager` only contacts the `SourceManager` during a modification, and if the cache is invalid. The replication is described in Interaction replicated data modification.

8.3 Performance

Performance can be improved by localizing services (as described in the scalability section), and trading off consistency (as described in the reliability section). Network communication is a major performance consumer in a distributed system. The goal is therefore to minimize remote invocations by using caches. Further, object-relational mapping policies can improve performance. By encapsulating the database access in one module, the rest of the system can access the persistent data in the form of objects. Furthermore, the cached objects offer traversal on an object level as opposed to traversing database tables, which improves

performance significantly. By letting the cached objects be coarser grained than the database entities, the overhead of distributed objects can be reduced (e.g. in the case study, `Availability` was not exposed as an object on a CORBA IDL level but accessed through the `Cottage` object.)

The service localization and consistency trade-offs are exposed in the generic design as described above. Object-relational mapping is not visible in the generic description of the system architecture. It is, however, visible on the realization level, and is documented by the mapping of system level interfaces (e.g. IDL) to their implementations. For example, the `Booking` table (figure 7.3) can be traced to its system level counterpart, the `SalesOffice` interface (figure 7.12).

This exemplifies an important property of performance. Performance issues are exposed both on a software architectural level, and on an implementation level. In order to enable reuse of a pattern more information must hence be documented than the abstract architecture. A sample realization of the abstract architecture is also part of the pattern. Both the case study and the architectural abstraction chapter of this thesis should therefore be seen as part of the same pattern. This pattern aims at helping to design object caches in a transactional object-relational CORBA environment.

8.4 Final Remarks

The proposed pattern is very coarse grained on its generic level (the connector Object-Caching_with_TransactionalReplication) in order to suit many different implementations. In the case study implementations, two technical implications were deliberately avoided. First, object eviction was not considered due to the scale of the test suite. Second, a generic version check was not implemented, as only `Booking`s were subject to change in the example scenario. Adding these two features to the caching implementation would, however, yield the same design on the highest level of abstraction.

The connector descriptions presented highlighted important design decisions made. These decisions would not have been as clearly documented if the case study had been modeled by using traditional class-based modeling like UML solely.

Acknowledgements

First of all, I would like to thank my supervisor at TU Berlin, Stefan Tai, for reviewing numerous drafts of this thesis, and for introducing me to the modeling approach of the Connector framework. Many thanks to Dirk Slama at IONA who supervised the technical parts of this thesis, and helped me understand the issues involved in developing large-scale transactional systems. Further, I am grateful to Eamon Walshe, for giving me useful comments on my implementation; Martin Bergljung, for helping me with the initial OTS demonstrator implementation; and other employees at IONA, for their feedback on presentations of my work. Finally, I would like to thank Prof. Janis Bubenko, my supervisor at Stockholm University, and Hercules Dalianis, for their comments.

References

[Agarwal & Keller 98] S. Agarwal, A. Keller. *The Power Tier Server, A Technical Overview*. White paper. Persistence Software, USA,1998.

[Ambler 98] S. W. Ambler. *Mapping Objects To Relational Databases*. White paper. AmbySoft Inc., Canada, 1998.

[Baker 97] S. Baker. *CORBA Distributed Objects - Using Orbix*. Addison-Wesley, UK, 1997.

[Bass et al 98] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. Addison Wesley, USA, 1998.

[Bernstein & Newcomer 97] P. Bernstein, E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publisher, Inc., San Francisco, California, 1997.

[Cattel & Barry 97] R.G.G. Cattel, D. K. Barry. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Francisco, 1997.

[Chankhunthod et al 94] A. Chankhunthod, P.B. Danzig, C. Neerdales, M. F. Schwartz, K. J. Worrel. *A hierarchical object cache*. Technical report, CU-CS-766-95. Department of Computer Science, University of Colorado, March 1994.

[Chen 96] J. Chen. *From Distributed Database to Replication*. CS445 reports. RMIT University, Melbourne, 1996.

[D'Souza & Wills 98] D. D'Souza, A. Wills. *Objects, Components, and Frameworks with UML – The Catalysis Approach*. Book draft. Addison Wesley, USA 1998.

[Faegri 95] T. E. Faegri. *Limitations for Inconsistency in Support Layers for Reliable Distributed Object Systems*. Position paper submitted to ECOOP'95. Department of Computer Science, University of Glasgow, 1995.

[Fahl & Risch 97] G. Fahl, T. Risch. *Query processing over object views of relational data*. The VLDB Journal 6:261-281. Springer-Verlag, Germany, 1997.

[Friedman & Mosse 96] R. Friedman, D. Mosse. *Load Balancing Schemes for High-Throughput Distributed Fault-Tolerant Servers*. Technical report, TR96-1616. Department of Computer Science, Cornell University, USA, 1996.

[Gamma et al 95] E. Gamma, R. Helm, R. Johnsson, J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, USA, 1995.

[Garlan & Shaw 96] D. Garlan, M. Shaw. *Software Architecture – Perspective on an Emerging Discipline.* Prentice Hall, USA, 1996.

[Garland et al 95] M. Garland, S. Grassia, R. Monroe, S. Puri. *Implementing Distributed Server Groups for the World Wide Web*. Technical report, CMU-CS-95-114. School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1995.

[Gray & Reuter 93] J. Gray, and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, USA, 1993.

[Inprise 98] Inprise. *VisiBroker*. Internet site (http://www.inprise.com). Inprise, USA, 1998.

[IONA 97a] IONA. *The Orbix Database Adapter Framework*. White paper. IONA Technologies, Ireland, 1997.

[IONA 97b] IONA. *The Evictor*. Internet site (http://www.iona.com/support/cookbook). IONA Technologies, Ireland, 1997.

[IONA 98a] IONA. *Products*. Internet site (http://www.iona.com/). IONA Technologies, Ireland, 1998.

[IONA 98b] IONA. *OrbixNames 1.1*. Manual. IONA Technologies, Dublin, Ireland, 1998.

[IONA 98c] IONA. *Orbix Advanced Workshop course 1.0*. Training course. IONA Technologies, Boston, USA, 1998.

[IONA 98d] IONA; *The Orbix Object Transaction Monitor (OTM)*. White paper. IONA Technologies, Ireland, 1998.

[IONA 98e] IONA. *OrbixOTM Guide*. Manual. IONA Technologies, Ireland, 1998.

[IONA 98f] IONA. *OrbixOTS Programmer's and Administrator's Guide*. Manual. IONA Technologies, Ireland, 1998.

[IONA 98g] IONA. *OrbixEvents Programmer's Guide*. Manual. IONA Technologies, Ireland, 1998.

[IONA 98h] IONA. OrbixTalk Programmer's Guide. Manual. IONA Technologies, Ireland, 1998.

[IONA et al 98] IONA, FUJITSU, INPRISE , Objectivity, Oracle, Persistence, Secant, Sun. *Joint Revised Submission Persistent State Service 2.0*. Object Management Group, USA, 1998.

[Keen 98] C. Keene. *Building Better Performance: Scalable Application Development with the Persistence PowerTier Server*. White paper. Persistence Software, San Mateo, California, 1998.

[Kordale & Ahmad 95] R. Kordale, M. Ahmad. *Object Caching in a CORBA Compliant System*. Technical report, GIT-CC-95-23. College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, 1995.

[Leser et al 98] U. Leser, S. Tai, S. Busse. *Design Issues of Database Access in a CORBA Environment*. Technical paper. Department of Computation and Information Structures, Technical University Berlin, 1998.

[Malveau & Mowbray 97] R. C. Malveau, T. J. Mowbray. *CORBA Design Patterns*. Wiley, USA, 1997.

[OMG 97a] OMG. *CORBAservices: Common Object Service Specification*. Object Management Group, USA, 1997.

[OMG 97b] OMG. *A Discussion of the Object Management Architecture*. Object Management Group, USA, 1997.

[OMG 97c] OMG. *UML Version 1.1*. Internet site (http://www.rational.com/uml), Object Management Group, September 1997.

[OMG 98a] OMG. *About OMG*. Internet site (http://www.ontos.com/mapcon.htm). Object Management Group, USA, 1998.

[OMG 98b] OMG. *The Common Object Request Broker: Architecture and Specification, Rev. 2.2*. Object Management Group, USA, 1998.

[OMG 98c] OMG. *Success Stories*. Internet site (http://www.ontos.com/mapcon.htm). Object Management Group, USA, 1998.

[ONTOS 98] ONTOS. *ONTOS*Integrator Object/Relational Mapping Concepts*. Internet site (http://www.ontos.com/mapcon.htm). ONTOS, 1998.

[OpenGroup 92] The Open Group. *Distributed Transaction Processing: The XA Specification*. X/Open Document C193, ISBN 1-85912-057-1. X/Open Company Ltd., Reading, U.K, 1992.

[Oracle 97a] Oracle. *Oracle8 Server Replication*. Manual. Oracle Corporation, Ireland, 1997.

[Oracle 97b] Oracle. *Oracle8 OCI*. Manual. Oracle Corporation, Ireland, 1997.

[Oracle 97c] Oracle. *Oracle8 Pro*C*. Manual. Oracle Corporation, Ireland, 1997.

[Orfali & Harkey 98] R. Orfali, D. Harkey. *Client/Server Programming with JAVA and CORBA*. 2nd edition. Wiley, USA, 1998.

[Orfali et al 96] R. Orfali, D. Harkey, J. Edwards. *The Essential Distributed Objects Survival Guide*. Wiley, USA, 1996.

[Persistence 98] Persistence. *Power Tier User Guide, Version 4.1*. Manual. Persistence Software, USA, 1998.

[RogueWave 98] Rogue Wave. *DBTools - Family of Products*. Brochure. Rogue Wave Software. USA. 1998.

[Schmidt & Vinoski 97] D. C. Schmidt, Steve Vinoski. *Object Adapters: Concepts and Terminology*. C ++ Report October. SIGS, USA, 1997.

[Shussel 96] G. Schussel. *Replication, The Next Generation of Distributed Database Technology*. Internet site (http://www.dciexpo.com/geos/replica.htm). USA, 1996

[Tai & Busse 97] S. Tai, S. Busse. *Connectors for Modeling Object Relations in CORBA-based Systems*. Proc. 24th Intl. Conference on the Technology of Object-Oriented Languages and Systems (TOOLS 24). IEEE Computer Society, Beijing, China, September 1997.

[Tai & Busse 98] S. Tai, S. Busse. *Software Architectural Modeling of the CORBA Object Transaction Service*. In Proc, 22nd Annual International Computer Software & Application Conference. IEEE Computer Society, Vienna, 1998.

[Tai 96] S. Tai. *Object abstractions in the design of corba systems for air traffic control simulation*. Technical report 27/96. EUROCONTROL Experimental Centre, Paris, 1996.

[Tai 98a] S. Tai. *A Connector Model for Object-Oriented Component Integration*. Proc. ICSE'98 Intl. Workshop on Component-Based Software Engineering (ICSE'98/CBSE-Workshop), Kyoto, Japan, April 1998.

[Tai 98b] S. Tai. *Architectural Representation of a CORBA OTM Application*. To appear. Technical University Berlin, 1998.

[Terry 85] D. Terry. *Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments*. Technical report, CSD-85-228. University of California, Berkley, 1985.

[Versant 98] Versant. *Versant ODBMS 5.0 Concepts and Usage*. Manual. Versant Object Technology Corporation, California, 1997.

[Vinoski 97] S. Vinoski. *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments*. IEEE Communications Magazine, 14:2. USA, 1997.

Appendix A: Glossary

2PC	Two Phase Commit
ACID	Atomicity, Consistency, Isolation, Durability
ADL	Architectural Description Language
BOA	Basic Object Adapter (CORBA Object Adapter)
CORBA	Common Object Request Broker (defined by OMG)
DTP	Distributed Transaction Processing (defined by X/Open - now the Open Group)
FIFO	First In First Out (eviction policy)
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference
LRU	Least Recently Used (eviction policy)
OCI	Oracle Call Interface
ODMG	Object Data Management Group
OMA	Object Management Architecture (defined by OMG)
OMG	Object Management Group
OOAD	Object-Oriented Analysis and Design
ORB	Object Request Broker (the object bus in CORBA)
OTM	Object Transaction Monitor (merge of CORBA and TP Monitor concepts)
OTS	Object Transaction Service (CORBAservice)
POA	Portable Object Adapter (CORBA Object Adapter)
PSS	Persistent State Service (CORBAservice)
RM	Resource Manager (defined in X/Open DTP model)
ROI	Remote Object Invocation
TM	Transaction Manager (defined in X/Open DTP model)
TP Monitor	Transaction Processing Monitor
TP System	Transaction Processing System
TS	Transaction Service (component of the OTS)
TTL	Time To Live (eviction policy)
UML	Unified Modeling Language
XA	Interface between RM and TM (defined in the X/Open DTP standard)

Appendix B: IDL Interfaces for the PTT System

PTT.idl

```
module PTT
{
    // ----------------------------------------
    // General
    // ----------------------------------------

    typedef sequence<short> WeekList;
    typedef string BookingReference;
    typedef sequence<BookingReference> BookingReferenceSeq;
    typedef long CottageID;

    interface Cottage;
    interface Resort;

    typedef sequence<Cottage> CottageSeq;
    typedef sequence<Resort> ResortSeq;

    exception NotAvailable {};

    // ----------------------------------------
    // Resort
    // ----------------------------------------
    struct ResortDetails {
      string name;
      string description;
    };

    interface Resort
    {
      ResortDetails getDetails();
      CottageSeq    getCottages ();
    };

    // ----------------------------------------
    // Cottage
    // ----------------------------------------
    struct CottageDetails {
      string     name;
      string     description;
      string     address;
    };

    interface Cottage
    {
      CottageID      getID();
      CottageDetails getDescription ();
      WeekList       getAvailability ();
    };
};
```

CacheUser.idl

```
interface CacheUser {
  oneway void invalidate(in any update);
};
```

SalesOffice.idl

```
#include "PTT.idl"
#include "CacheUser.idl"
module PTT_SalesOffice
```

```
{
    struct Customer {
      string lastName;
      string firstName;
      string address;
    };

    struct CreditCardDetails {
      string number;
      string expiryDate;
    };

    // -------------------------------
    // SalesOffice
    // -------------------------------
    interface SalesOffice : CacheUser
      {
      string getBranchName ();

      PTT::ResortSeq getResorts ();

      PTT::BookingReference makeBooking (
        in PTT::Cottage aCottage,
        in short fromWeek,
        in short toWeek,
        in Customer client,
        in CreditCardDetails card)
        raises (PTT::NotAvailable);

      PTT::BookingReferenceSeq listBookings ();
    };
};
```

CentralOffice.idl

```
#include "PTT.idl"
#include <ots/orbix/cos_ots.idl>

interface CentralOffice : CosTransactions::TransactionalObject
{
    PTT::ResortSeq getResorts ();

    PTT::BookingReference confirmBooking (
          in PTT::CottageID aCottage,
          in short fromWeek,
          in short toWeek)
      raises (PTT::NotAvailable);
};
```