Design of Object Caching in a CORBA OTM System

Thomas Sandholm[1], Stefan Tai[2], Dirk Slama[1], and Eamon Walshe[1]

[1] IONA Technologies plc
The IONA Building, Shelbourne Road, Dublin 4, Ireland
{tsndhlm, dslama, ewalshe}@iona.com
[2] Technische Universität Berlin
Sekr. E-N 7, Einsteinufer 17, D-10587 Berlin, Germany
stai@cs.tu-berlin.de

Abstract. CORBA Object Transaction Monitors (OTM) refer to a middleware technology that enable the building of transactional, object-oriented information systems running in distributed and heterogeneous environments. In this paper, we address large-scale OTM-based systems and focus attention on the important quality factors of system performance, system scalability, and system reliability. We develop an object caching strategy that employs OTM concepts such as distributed transactions and asynchronous event multicast, and show how this strategy improves an existing distributed CORBA system wrt. performance and scalability. We further describe our object caching solution as a transferable, reusable architectural abstraction, and demonstrate the application of software architectural concepts for design modeling of CORBA systems that introduce object caching.

1 Introduction

Software systems implemented in distributed and heterogeneous environments are becoming increasingly common as a result of the availability of communication technologies like the Internet and component technologies like distributed object middleware. This observation in particular holds for large-scale information systems, where data is distributed with software components to different nodes in a network. An important requirement here is to keep the distributed data consistent and to guarantee performance of the system.

In this paper, we focus attention on the development of a *CORBA Object Transaction Monitor (OTM)*-based system. CORBA OTM refers to an advanced middleware technology that has been adverted a major trend for next-generation distributed transaction processing [11]. CORBA OTM consists of the standard *object request broker (ORB)* providing mechanisms for remote object invocation [9], and a set of object services for distributed systems and data management, including the *CORBA Object Transaction Service (OTS)* [10].

We develop an object caching strategy that can be introduced to large-scale CORBA OTM-based systems [12]. The major objective is to improve system performance, while assuring system scalability and system reliability. The caching

M. Jarke, A. Oberweis (Eds.): CAiSE'99, LNCS 1626, pp. 241–254, 1999.

solution is described using the software architectural modeling concept of a *connector*, and its application is demonstrated with an example scenario that has been implemented using IONA's OrbixOTM product [7]. The work presented has been carried out as part of the project *"CORBA Object Transaction Monitor Experimentation"*, a cooperation between IONA Technologies Dublin and Technische Universität Berlin.

The paper is structured as follows. First, we introduce CORBA OTM and concepts relevant to improve system performance. Second, we present an object caching strategy and its implementation and test results for a sample OTM-based system. Third, we develop the software architectural connector "Object Caching with Transactional Replication" capturing interfaces and interoperation patterns of our caching solution, and show how this connector can be used to describe caching in CORBA OTM-based systems.

2 CORBA OTM

With the Object Management Group's (OMG) *Common Object Request Broker Architecture (CORBA)* [9], a standard middleware technology for the integration and interoperation of diverse software components in distributed and heterogeneous environments has been proposed. *CORBA Object Transaction Monitors* address *enterprise computing* based on CORBA, and provide additional support for security, transactions, availability, or manageability.

CORBA OTMs comprise a variety of (standard and non-standard) *object management services*, of which the CORBA Object Transaction Service (OTS) and the CORBA Events Service [10] are important examples. The OTS provides utilities for distributed transaction processing, i.e. for transaction management, transaction propagation, and for driving the two-phase commit protocol to coordinate different distributed resources, including databases. The Events service enables loosely coupled, asynchronous messaging between multiple event suppliers and multiple event consumers using event channels (being standard CORBA objects), based on a publish/subscribe paradigm.

CORBA OTMs can be compared to Transaction Processing (TP) Monitors of traditional client/server systems, but take the concept of a TP Monitor from procedural to open distributed object computing. A thorough treatment of enterprise computing with CORBA and CORBA OTM can be found in [14].

3 Improving Scalability and Performance

In large-scale distributed systems, special attention has to be paid to possible bottlenecks, due to the fact that a large number of concurrent requests have to be processed. Three main techniques can be used to circumvent this scalability and performance problem: *load balancing*, *replication*, and *caching*.

Load balancing involves duplicating processing in the system, e.g. by having many servers offering the same service. The main goal is to increase throughput, i.e. the number of successfully served requests, when multiple clients send

requests concurrently. In [5], different schemes that can be used for spreading the load between servers transparently to the clients are demonstrated.

If the servers have state (manage local data), then some techniques have to be considered regarding how to replicate the data among the distributed servers [2]. Data can be replicated both to increase availability and to improve performance due to service localization. The main problem with replication is how to keep the replicas mutually consistent. Two main approaches exist here: the replicas can be updated *synchronously*, e.g. within a transaction for absolute consistency, or, the updates can be sent out *asynchronously* to trade off consistency with performance and scalability. Further, either only one replica can be updated (*master/slave* replication), or all replicas can be updated (*peer-to-peer* replication).

Object caching in a distributed environment naturally relates to load balancing and replication, but has as its main goal the improvement of performance, or user response time. Two important issues here are (a) where to locate caches, and (b) which objects to put in the caches. A list of possible cache location levels, e.g. per-process, per-node, and per-node group is presented in [18]. Many different cache location levels can coexist for the same object, as shown in [4]. The decisions taken will influence which clients can share the same objects. Shared objects must be read frequently and be updated infrequently in order for the caching to be successful.

Keeping caches accurate and consistent with the source is of predominant importance when caching objects at distributed servers. The same *update approaches* as with replication can be used: asynchronous, or synchronous. The asynchronous approach can be compared to optimistic locking in the database field, and must handle the case when two conflicting updates are made concurrently. Careful attention has to be paid to what information should be sent with the updates in order to minimize network traffic, while keeping the caches accurate at all times. Network traffic can also be decreased by using the multicast protocol for propagating the updates [8].

Further important issues that have to be considered when implementing an object caching strategy are *object faulting* and *object lifetime* (eviction). Object faulting concerns how to fetch the accurate value from the source and place it in the cache transparently to the client. This mechanism can, for instance, be implemented by detecting operating system page faults, as demonstrated in [8]. The problem with this approach is, however, that pages are cached, but not objects, which results in non-object-oriented trade-offs in the code. Further, such an implementation is also very operating system dependent. The object lifetime policy to choose, i.e. when to evict objects from the cache, is determined (limited) by the cache memory available. Common lifetime policies are: FIFO, TTL (Time To Live), LRU (Least Recently Used), transaction based (object lifetime equals transaction lifetime), and application server based (object lifetime equals application server lifetime) [4], [2].

4 Object Caching Strategy

4.1 Example Scenario

We use a geographically distributed travel agency as a sample CORBA OTM-based system, which is to be improved with an object caching strategy.

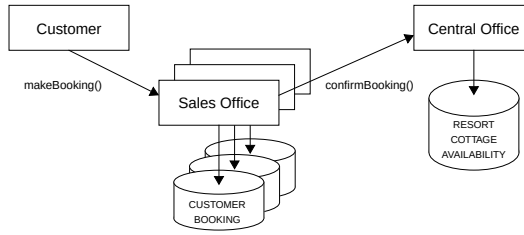

Fig. 1. A Distributed Travel Agency

The system architecture of the travel agency is shown in Fig. 1. `Customers` can book `Cottages` residing in `Resorts`, and browse `Cottage`, `Resort`, and cottage `Availability` information. Multiple `SalesOffices` have been introduced to decrease the load on the `CentralOffice`. Each *booking* for a cottage must first be issued on a `SalesOffice`; the *booking* is then confirmed at the `CentralOffice`. The `CentralOffice` manages persistent `Resort`, `Cottage`, and cottage `Availability` data in a relational database. The `SalesOffice` manages local `Customer` information, and maintains all `Bookings` that are made at the `SalesOffice`.

We assume that the system has one million `Customers`, and that one *booking* can be made each second in the system. For each *booking*, multiple queries on `Resort`, `Cottage`, and cottage `Availability` (`Booking`) information typically are issued. We assume that the average *booking* during peak system load consists of 12 queries, followed by one update (= a booking session).

4.2 Introducing Caching

The first decision concerns to cache `Resort` and `Cottage` objects (as these are frequently read objects, but updated infrequently) and to replicate cottage `Availability` data as `Booking` data at the distributed `SalesOffices`. Two different consistency policies are chosen to keep the cached objects and the replicated data consistent with the data in the `CentralOffice`.

Asynchronous propagation (optimistic approach) is used for keeping the cached objects accurate and consistent with the source. Asynchronous multicast propagation is selected because of performance and scalability reasons. The problem of concurrent updates here is handled when confirming *bookings* at the

`CentralOffice`. The `CentralOffice` detects when an inaccurate cache has been used, and returns an exception that the *booking* cannot be made.

Synchronous propagation (pessimistic approach) is used for updating the replicated data (`Bookings`). When a *booking* is to be made, a distributed transaction is started at the local `SalesOffice`. Within this transaction, the *booking* is confirmed at the `CentralOffice`, is made persistent in the `Availability` table, and is replicated locally in the `Booking` table. The data in the two databases are hence kept synchronous by the two phase commit protocol. This approach is more time consuming, and does not scale as well as the asynchronous approach. The approach, however, assures that the local data is always consistent with the source at any point in time. This enables book keeping or invoicing tasks, for example, at the local `SalesOffices` without contacting the `CentralOffice`. We use master/slave propagation, i.e. all changes must be made at the `CentralOffice` first. By doing so, we avoid conflicts that can occur due to concurrently propagated updates.

Two different strategies are implemented for updating the caches, once an update event has arrived at the `SalesOffice`. The source can be contacted to get the currently most accurate value, or local updates can be made using object information sent with the event. Source updates are safer when concurrent updates are made. Additionally, if the original transaction aborts after the notification has been sent away, the caches will still be valid when using the source update approach, as they will read the source data in a transaction scheduled after the original one. Local updates have a significant performance advantage, though.

If an object that is not in the cache is accessed, then it is fetched from the source transparently to the clients. Once the object is fetched, it remains in the cache for the lifetime of the `SalesOffice` servers. When a particular `Cottage` is queried, the cache is filled with information for all `Cottages` in the same `Resort`. `SalesOffices` thus only need to cache some `Resorts`, which compensates the fact that no direct object eviction is implemented. Furthermore, the `Availability` data are decreased (and thereby also the size of the caches) as more `Cottages` are booked.

In order to minimize network traffic, a proper *event granularity* must be chosen. In our example, we use only one event channel for propagating *bookings*, but send object information with the events to enable updates of single cache entries (for a single object). The `SalesOffices` are event consumers, and the `CentralOffice` is the only event supplier.

The caching strategy is summarized in Table 1.

This object caching strategy has been implemented and tested by simulating the assumptions about system usage. The relation between the number of queries, and the number of updates (12 to one) is crucial to the success of the caching implementation. The more queries that are made, the more does the caching pay off. The tests were carried out by simulating both peak system load, and twice that load, in order to measure scalability of the solution. The peak system load, i.e. one booking session is started every second, was derived from

Table 1. Caching strategy summarized

Problem	Solution
Cached Objects	`Resort` and `Cottage` objects
Cache Location	Application server
Replica Consistency	Synchronous propagation
Cache Consistency	Asynchronous multicast propagation
Update Policy Replication	Master/Slave
Update Policy Caches	Update from source/local updates
Object Faulting	Clients access objects via application servers that transparently fetch source state
Cache Eviction	Application server based
Event Granularity	One event channel per class of objects that can be modified, events carry object level information

assumptions made on how the one million customers use the travel agency system. Further, the system was tested before and after the introduction of caching, and local updates were compared to updates from source. Throughput (reliability), i.e. number of successful *bookings*, and the response time for the *bookings* (performance) were measured.

A client test suite was developed for the simulations. Each client in the suite implemented a *booking* session as follows: (1) a query to get all `Resorts` was made, and one of these `Resorts` was picked at random; (2) all `Cottages` for this `Resort` were collected in a query, and 10 `Cottages` in the chosen `Resort` were picked randomly; (3) availability data were retrieved for each of these `Cottages`; (4) finally a booking was attempted for one week chosen at random from the retrieved availability data. One of these clients was started asynchronously from a shell script each second. Information on the time it took for each client to complete its *booking* session (user response time), and whether the booking attempt in (4) was successful was traced. Further details on test environment are available in [12]. The results are depicted in Fig. 2.

The caching strategy improved reliability, scalability, and performance of the system compared to the system without caching. The caching strategy using local updates scaled better than the solution updating from source. Notable from the tests is that only half of the booking attempts were successful in twice peak system load when caching wasn't used.

The caching strategy could further be improved by redirecting clients accessing the same `Resorts` to the same `SalesOffice`. This functionality could be combined with a general load-balancing scheme to dynamically spread the load on the `SalesOffice` servers by using e.g. OrbixNames [7]. Also, a group of replicated `CentralOffice` servers could be introduced into the system, so that the `CentralOffice` no longer can become a bottleneck and single-point of failure.

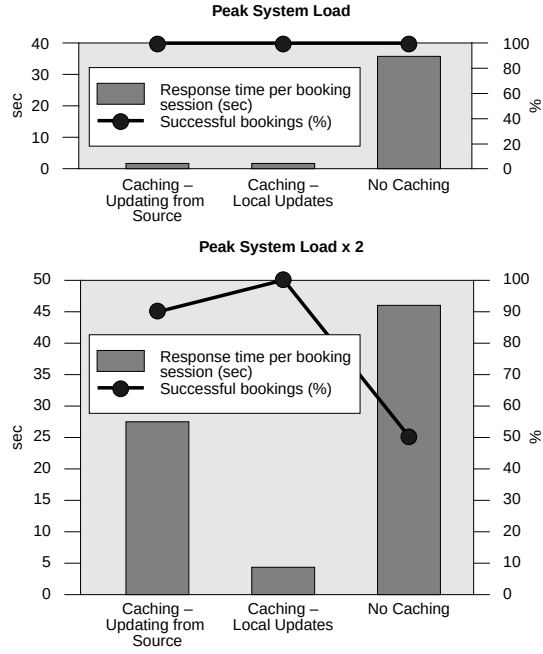

Fig. 2. Object Caching Test Results

5 Software Architectural Design

In the following, we present our object caching solution as a transferable, reusable *connector* abstraction for software architectural system design. The notion of connectors for modeling component collaborations has been mentioned in a variety of work in the area of software architecture [13], [1].

5.1 Connector "Object Caching with Transactional Replication"

Our particular connector concept has been proposed in [16], and has been exemplified for modeling CORBA object services in [15], [3]. Connectors are pattern-like descriptions of complex component collaborations. A connector comprises the definition of *roles*, *role interfaces*, and *interaction protocols*.

Fig. 3 depicts the roles of our caching connector. Each role describes a collaboration responsibility, which is taken on by components in the software architecture of a particular system.

For each role, a set of role interfaces is defined (Fig. 4). These interfaces must be provided by any particular component playing the role. The role interfaces are declared using OMG IDL [9]. `OID` in Fig. 4 refers to a secondary identifier of an object, which is not the object reference itself, but an identifier used to

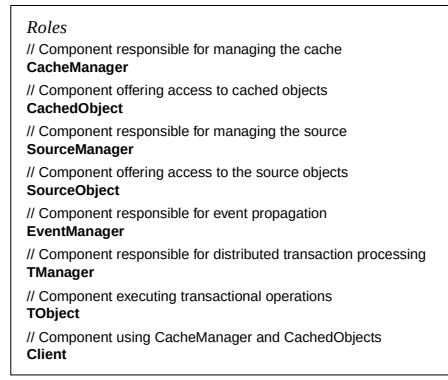

Fig. 3. Roles

map the object to a unique external entity (e.g. a primary key in a relational database).

Fig. 5 depicts the interaction protocols of our caching connector. Interaction protocols describe sequences of role interface requests along with pre- and postconditions. They are described using UML sequence diagrams [19].

The *Cache Initialization and Use* interaction describes the basic cache functionality. A component in role of *Client* sends a request for an object. If there is a valid object in the cache, it is returned directly by the component in role of *CacheManager*. Otherwise, the accurate state of the source object is fetched from the source. The state is used to create a cached object (cache initialization). 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.

The *Replicated Data Modification* 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 component in role of *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 kept consistent. The confirmation with the *SourceManager* serves to detect whether other clients have updated the source concurrently, and a conflict thereby has occurred. (In our example scenario, such a conflict occurred when two clients selected the same `Cottage` from the cache, and then tried to book it concurrently for the same calendar week.) A conflict leads to a race condition where the first transaction to execute will succeed, and the second one will roll back.

The interaction *Cache Update* depicts how caches are updated by using event notifications. The events are pushed from the component in role of *SourceManager* to the component in role of *EventManager* when a *SourceObject* has changed. The *EventManager* then pushes the events to the registered *Cache-*

Role Interfaces

```
// CacheManager interfaces
CacheManager.CacheUser {
      oneway void update(in any event);
};
CacheManager.<LocalManager> {
      <SourceObject> get(in OID id);
      void modify(in OID id, in any value);
      // for 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);
};
// CachedObject interfaces
CachedObject.<SourceObject> {
      <SourceObject>State getState(in OID id);
      . . .
};
// SourceManager interfaces
SourceManager.<CentralManager> : CosTransactions::TransactionalObject {
      <SourceObject> get(in OID id);
      boolean confirmModify(in OID id, in any value):
      // for internal use:
      void makePersistent(in OID id, in any value);
};
// SourceObject interfaces
SourceObject.<SourceObject> {
      <SourceObject>State getState(in OID id);
      . . .
};
```

```
// EventManager interfaces
EventManager.<Registration> {
      // corresponds to registration interface of IONA's OrbixTalk,
      // a product-specific API that wraps the CORBA Events Service
      CORBA::Object registerTalker
                  (in string subject, in string cacheUserInterface);
      void registerListener
            (in CORBA::Object CacheUser, in string subject);
};
EventManager.CacheUser {
      // corresponds to CORBA Events Service PushConsumer interface,
      // operation push
      oneway void update(in any event);
};
// TManager interfaces
TManager.CosTransactions::Current {
      // original OTS interface
      void begin();
      void commit(in boolean report_heuristics);
      void rollback();
      . . .
};
// TObject interfaces
TObject.CosTransactions::TransactionalObject {
      // original empty OTS interface
};
```

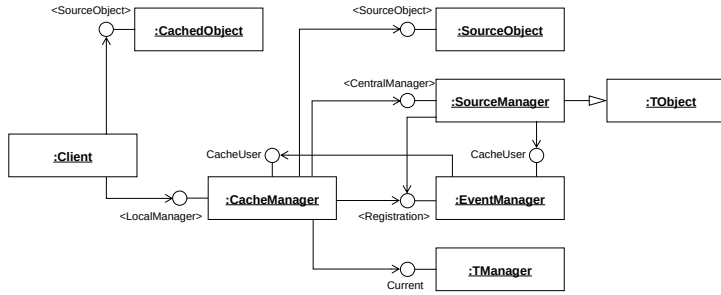

Fig. 4. Role Interfaces

Managers. 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. First, it can be updated by getting the state from the source. Second, 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 4.2.

5.2 Modeling the Example Scenario

We describe the software architecture of our example system by using the architectural framework of *components*, *connectors*, *abstract architectures*, and *concrete architectures* [17].

Components are design-time abstractions of computational system parts. Components are described using multiple views: the *core functionality view*, and

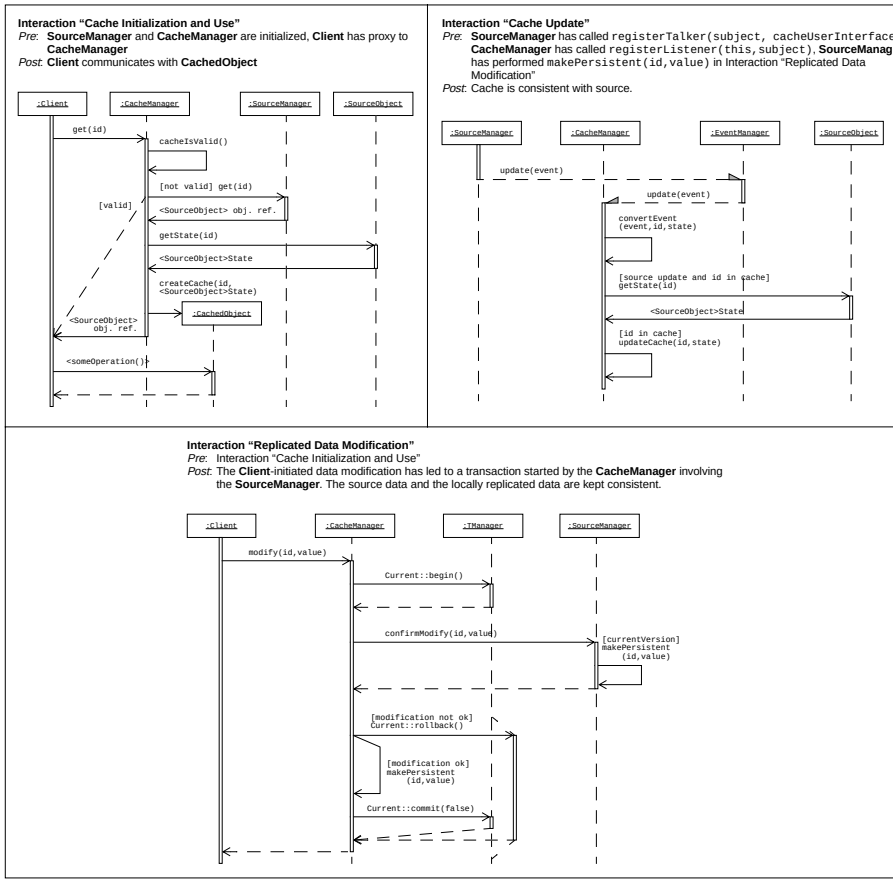

Fig. 5. Interaction Protocols

various component *collaboration views*. The core functionality view models the domain-oriented component features.

For our travel agency system, we can define the three application components `SalesOffice`, `CentalOffice`, and `Customer`, and the two service components `OrbixOTS` and `OrbixTalk` (off-the-shelf components implementing the CORBA OTS and Events Service, respectively) as design-time components. Fig. 6 depicts the exported and imported (required) system-level interfaces of the core functionality view of the `SalesOffice` component.

An *abstract architecture* of connector-based component composition is shown in Fig. 7. The components of our particular system are related by means of the generic connector "Object Caching with Transactional Replication", i.e. the connector roles are distributed to the components. This describes a requirement on the components to implement the respective role interfaces, and characterizes

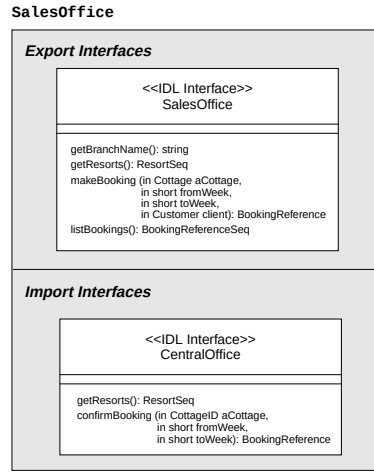

Fig. 6. Component SalesOffice – Core Functionality View

the component to interact with the other components as specified with the connector interaction protocols. Abstract architectures are software architectural descriptions on a very high level of abstraction.

Fig. 8 shows the "object caching with transactional replication" view on the `SalesOffice` component, i.e. the collaboration view resulting from the abstract architecture of Fig. 7. The caching view exhibits all component features that have been introduced because of the caching rationale (as opposed to the core functionality view). The caching functionality is now exposed with new exported and imported interfaces, such as the provided `CacheUser` interface, or the required OTS `Current` interface to start, commit and abort distributed transactions. The `SalesOffice` component now imports and exports the `Resort` and `Cottage` interfaces unchanged.

Fig. 8 also shows the `SalesOffice` component's *representation* and export and import *representation-map*[1], i.e. the internal realization design and program-level interfaces of the component. This diagram describes implementation details of the component as a distributed, transactional CORBA server, and is expressed using UML class modeling. In the representation part, we can e.g. see that the cache is structured in a hierarchical containment tree. The `SalesOffice` contains a collection of cached `Resorts`, and each cached `Resort` contains a collection of cached `Cottages`.

The set of all component descriptions of the same (object caching) view is called a *concrete architecture* to an abstract architecture. The software architecture of a particular system is thus described on two different levels of abstraction.

[1] We adapted the terminology of representation and representation-map from the ACME ADL [6].

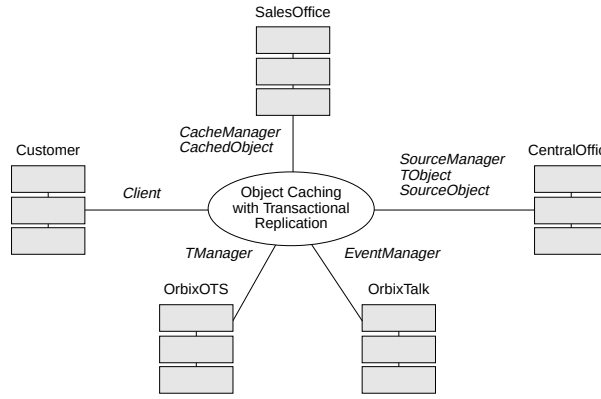

Fig. 7. Abstract Architecture of Travel Agency System

Overall, the architectural framework of design-time components, connectors, abstract and concrete architectures employs a clear separation of modeling concerns, and enables a pattern-oriented, structured approach to architectural software system representation.

6 Conclusion

In this paper, we developed an object caching strategy for CORBA OTM-based systems which addresses system reliability, system scalability, and, in particular, system performance. We demonstrated object caching for a sample distributed, transactional CORBA system, and showed how the caching solution proposed increased system performance notably. We abstracted the caching functionality and interoperation patterns into a software architectural connector, which was used to model the complex component collaborations of our example system, and also serves as a reusable design solution to object caching that can be applied to other CORBA OTM-based systems.

The caching solution can be summarized as follows:

Reliability of the solution was assured by asynchronous updates of the caches, and by synchronous modification of the replicated data. In terms of our connector, this behavior is captured as follows: the *SourceManager* sends an update event through the *EventManager* to the *CacheManagers* (interaction protocol "Cache Update"), and data modified in the database of the *SourceManager* is replicated in the *CacheManager*'s database within a distributed transaction started by the *CacheManager* (interaction protocol "Replicated Data Modification").

Scalability was improved by service localization. All queries can be performed locally because of the caches, which thereby improve load balancing. This is captured in the connector interaction protocol "Cache Initialization and Use". The

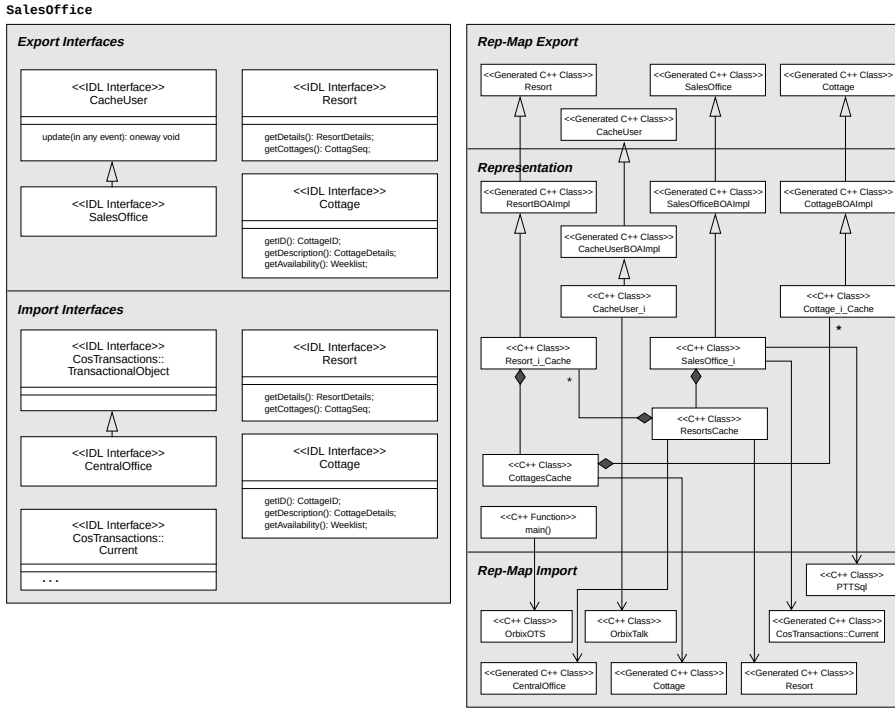

Fig. 8. Component SalesOffice – Object Caching with Transactional Replication View

CachedObject offers the same interface as the *SourceObject*, and the *CacheManager* is responsible for localizing the *SourceObject*, i.e. converts the *SourceObject* into a *CachedObject*. Service localization is also achieved by data replication, as mentioned previously.

Performance of the system was notably improved through the caches, and has been addressed in line with system scalability and system reliability. The design decisions regarding performance are hence documented in all three interaction protocols of our caching connector.

Acknowledgements. We would like to thank Prof. Herbert Weber, TU Berlin, and Fiona Hayes, IONA Technologies Dublin, for their continuous project support. We would also like to thank Prof. Janis Bubenko, University of Stockholm, for co-supervising the thesis underlying this paper.

References

1. L. Bass, P. Clements, R. Kazman. Software Architecture in Practice. Addison-Wesley, 1998.

2. P. Bernstein, E. Newcomer. Principles of Transaction Processing. Morgan Kaufman, 1997.
3. S. Busse, S. Tai. Software Architectural Modeling of the CORBA Object Transaction Service. In Proc. COMPSAC'98, IEEE Computer Society, 1998.
4. A. Chankhunthod, P.B. Danzig, C. Neerdales, M.F. Schwartz, K.J. Worrel. A Hierarchical Object Cache. Technical Report, CU-CS-766-95. University of Colorado, 1994.
5. R. Friedman, D. Mosse. Load Balancing Schemes for High-Throughput Distributed Fault-Tolerant Servers. Technical Report, TR96-1616, Cornell University, 1996.
6. D. Garlan, R. Monroe, D. Wile. Acme: An Architecture Description Interchange Language. In Proc. CASCON97, 1997.
7. IONA Technologies. OrbixOTM Guide. IONA Technologies plc., 1998.
8. R. Kordale, M. Ahmad. Object Caching in a CORBA compliant System. Technical Report, GIT-CC-95-23, Georgia Institute of Technology, 1995.
9. Object Management Group. The Common Object Request Broker: Architecture and Specification, rev.2.2. OMG, 1998. On-line at http://www.omg.org
10. Object Management Group. CORBAServices: Common Object Services Specification. OMG, 1997. On-line at http://www.omg.org
11. R. Orfali, D. Harkey. Client/Server Programming with Java and Corba, 2nd edition. Wiley, 1998.
12. T. Sandholm. Object Caching in a Transactional, Object-Relational CORBA Environment. Masters Thesis, University of Stockholm, 1998.
13. M. Shaw, D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.
14. D. Slama, J. Garbis, P. Russell. Enterprise CORBA. Prentice-Hall, 1999.
15. S. Tai, S. Busse. Connectors for Modeling Object Relations in CORBA-based Systems. In Proc. TOOLS 24, IEEE Computer Society, 1997.
16. S. Tai. A Connector Model for Object-Oriented Component Integration. In Proc. ICSE'98 Workshop on Component-Based Software Engineering, 1998.
17. S. Tai. Constructing Distributed Component Architectures in Continuous Software Engineering. PhD Thesis, TU Berlin, 1999. to appear.
18. D. Terry. Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments. Technical Report, CSD-85-228. University of California, Berkeley, 1985.
19. UML Partners. The Unified Modeling Language, v1.1. OMG, 1997. On-line at http://www.omg.org