Chapter 7: Evaluation

In Chapter 4, we motivated that enterprise application middleware should support custom and adaptable process topologies. In Chapter 5, we presented our FPT architecture as a basis for middleware that explicitly supports both custom and adaptable process topologies. Then, in Chapter 6, we described our *Free Data Objects* framework, which is a proof-of-concept implementation that realizes all concepts of our FPT architecture.

This chapter is devoted to an evaluation of our proof-of-concept implementation. Since the implementation is based on the FPT architecture, this chapter also evaluates (implicitly) our FPT architecture. We show that it is easy to construct enterprise applications with custom process topologies. Furthermore, we demonstrate that process topologies are adaptable with our approach – changing an existing process topology is straightforward, can be achieved through reconfiguration, and does not require any code changes to application code. In many transaction processing systems, efficiency is also a major concern. Therefore, we show that support for custom and adaptable process topologies does not come at the cost of degraded performance. For several scenarios, we analyze the performance of our proof-of-concept implementation. We show that our solution introduces relatively little overhead and provides good performance.

This chapter is structured as follows: We evaluate our solution with the help of seven scenarios – each one is described in a separate subsection: In Section 7.1, we evaluate support for custom and adaptable process topologies. We examine the implementation and evolution of an enterprise application based on the case study presented in Section 4.2. In the scenario described in Section 7.2, we analyze basic transaction throughput rate and transaction execution times. Section 7.3 examines how client response times are affected by introducing additional intermediate tiers into a process topology. Section 7.4 compares performance of coarse-grained remote access, which is typical of our approach, to fine-grained remote access, which is encouraged by object middleware. In Section 7.5, we evaluate load distribution among replicated processes. In Section 7.6, we show how transaction throughput is affected by data distribution and distributed transactions. In Section 7.7, we analyze transaction throughput and abort rates in the presence of "hotspot" data, i.e., data items that are frequently read and written by many concurrently clients. Finally, in Section 7.8, we give a summary of this chapter.

7.1 Scenario 1: Custom Topologies and Adaptability

In Section 4.2, we presented a case study of an enterprise application that implements a marketplace for accommodations. Figure 4-1 shows how the application starts as a tiny system and then, step by step, topology patterns are applied and the system evolves into a complex custom process topology. By means of the case study, we motivated why support for custom and adaptable process topologies is important.

To demonstrate that construction of custom process topologies and their adaptation is straightforward with our approach, we implemented a sample enterprise application (a simple marketplace for accommodations) on top of our framework. The application corresponds to that described in the case study. We started with a process topology as shown in Step 1 of the case study and then adapted it for each evolutionary step. In the remainder of this subsection, we first outline the implementation, then describe what actions were necessary to realize each evolutionary step, and finally draw conclusions.

7.1.1 Scenario Outline

We implemented a sample enterprise application that manages data objects of four different types: customers, accommodations, landlords, and reservations. Each accommodation is offered by a landlord and can be reserved by customers for given time periods. A time period is a sequence of days – it is defined by the first day and the last day of a stay. Definitions of these four data object types, as they appear in an object manager's configuration file, can be found in lines 4 to 20 of Listing 7-1.

We implemented six types of transactions that are executed on data objects:

(1) insertion of a new landlord

(2) insertion of a new accommodation

(3) insertion of a new customer

(4) reservation of a random accommodation for a random time period and a random customer

(5) cancellation of a random reservation

(6) update of the description of an accommodation

We implemented clients that constantly execute the transactions described above following a random scheme. For illustration purposes, the Java source code of all six transaction types is given in Appendix B. From the source code, we can see that developers have a comfortable, logical view on data objects.

A typical requirement for reservation systems is that conflicting reservations are always prevented – even if multiple transactions are concurrently executed. In our case, the enterprise application has to ensure that each accommodation is reserved at most once for each day. A simple implementation could rely on the guarantees of the strictest ANSI transaction isolation level, the *serializable* level. However, as is typically the case for systems based on optimistic concurrency control, our FDO framework does not support the *serializable* level (see Subsection 5.7.5). In practice, the *serializable* level is rarely used anyway – either because of performance concerns (concurrency may be reduced significantly) or because it is not supported by the underlying enterprise application middleware and/or data stores. Instead, in many cases, an equivalent level of consistency, without relying on *serializable,* can be achieved by making slightly changes to data structures and transactions. In this scenario, a solution is to store and write reservation data in a special way. In addition to storing one data object per reservation, we also introduce data objects that represent those time periods for which accommodations have *not* been reserved yet. The following example illustrates that approach. The example shows all six reservation data instances stored for an accommodation *A* as tuples of the form *(accommodation, firstDay, lastDay, customer)*:

(A, day 0, day 735, null), (A, day 736, day 739, C_1), (A, day 740, day 803, null),
(A, day 804, day 812, C_2), (A, day 813, day 820, C_3), (A, day 821, day 999999, null)

There are three reservations for *A*, which have been performed by customers C_1, C_2, and C_3, respectively. We require that all reservation data instances for a given accommodation describe disjoint time periods and together cover all days from 0 to 999999 (the end marker). Instances with *customer=null* represent time periods not reserved yet. Each continuous time period not reserved yet must be represented as a single instance, i.e., if there is an instance *(a, d_1, d_2, null)*, then there must not be an instance *(a, d_2+1, d_3, null).* Storing reservation data according to the rules described above prevents conflicting reservations, although the FDO framework provides only the *repeatable reads* isolation level. Tests performed by application code in combination with version checks for updated and deleted data objects (see subsections 5.7.4 and 6.5.1) guarantee that transactions of type 4 and type 5 (see Appendix B) cannot commit if they would lead to conflicting reservations.

7.1.2 Realization of Evolutionary Steps

In this subsection, we describe actions necessary to realize each evolutionary step described in Section 4.2. The first six steps are described and discussed in detail. Actions to be performed for steps 7 to 10 are similar to those described in the first six steps. Therefore, steps 7 to 10 are only outlined.

We recommend that, for each step described below, the reader first consults Figure 4-1 as well as the paragraph that corresponds to the step in Section 4.2.

Step 1 (Initial Two-Node Structure)

In the beginning, the enterprise application's process topology consists of a single application process and a single data store. We developed a Java client (class *Client*) that implements the six transaction types described in the previous subsection. In Section 5.1, we stated that an object manager component has to be placed in each application process. Our FDO framework implementation provides an implementation of such a component as part of a Java library, which has to be placed in the Java classpath of the client (and, in general, in the classpath of all FDO-based applications). On start-up, our enterprise application client instantiates an object manager component, passes configuration data to it, and starts executing transactions. Configuration data for the object manager (see Section 6.2 for details) is stored in a local text file, which is shown in Listing 7-1. The configuration file defines data objects (lines 4 to 20), assigns all data objects to a single domain associated with a single data store (lines 22 to 27), and defines Java interfaces for type-safe access to data objects (lines 29 to 32). The object manager instance is named *SingleClient* (line 36) and accesses a single Oracle database via JDBC (lines 38 to 44). The database schema and the source code for Java interfaces to data objects were automatically created using two tools (*DBSchemaCreator* and *InterfaceGenerator*) that we developed as part of our FDO framework. With the FDO library in its classpath and a database schema created, the client is ready to run.

```
01: # definition of data objects, domains, and interfaces
02: OBJECT MANAGER * {
03:
04:    DATA OBJECT Landlord { TYPECODE = 100;
05:       name:string; address:string; phone:string; email:string;
06:    }
07:    DATA OBJECT Accommodation { TYPECODE = 101;
08:       countryCode:short; address:string; description:string;
09:       accommodationType:short; coordX:double; coordY:double;
10:       numberOfBeds:short; petsAllowed:bool; landlord:Landlord;
11:       hasTv:bool; distanceToBeach:float; rentPerDay:float;
12:    }
13:    DATA OBJECT Reservation { TYPECODE = 102;
14:       accommodation:Accommodation; firstDay:int;
15:       lastDay:int; reservedBy:Customer;
16:    }
17:    DATA OBJECT Customer { TYPECODE = 103;
18:       name:string; address:string; contact:string;
19:       isPremiumCustomer:bool; customerNo:int;
20:    }
21:
22:    DOMAIN dom1 {
23:       CODE = 1001;
24:       DEF = SELECT FROM Landlord, SELECT FROM Accommodation,
25:             SELECT FROM Reservation, SELECT FROM Customer;
26:       RITREE = ds1;
27:    }
28:
29:    INTERFACE FOR Landlord:      testapps.sc1adaptability.Landlord;
30:    INTERFACE FOR Accommodation: testapps.sc1adaptability.Accommodation;
31:    INTERFACE FOR Reservation:   testapps.sc1adaptability.Reservation;
32:    INTERFACE FOR Customer:      testapps.sc1adaptability.Customer;
33: }
34:
35: # Single client, which directly connects to a database
36: OBJECT MANAGER SingleClient {
37:    ID = 10;
38:    SERVER CONNECTION {
39:       SERVERADDRESS = JDBCAddress
40:          "jdbc:oracle:thin:@//foo.mi.fu-berlin.de:1521/MYDB"
41:          "clemens" "gt7ye3pz" "ds1";
42:       IMPORTS = (dom1, ds1);
43:       ADAPTER = OracleJDBCConnection;
44:    }
45: }
```

Listing 7-1. Object manager configuration for a single client that accesses a single data store.

Step 2 (Multiple Clients)

In the second evolutionary step, additional clients running on separate personal computers are introduced. To realize these changes, we copy the client code, the FDO library, and the configuration file described in Step 1 to each of the new personal computers. Then, on each personal computer machine, we edit lines 36 and 37 of the configuration file to give each object manager instance a unique name and id. We choose *Client1*, *Client2*, *Client3* as names and *10*, *11*, *12* as ids. Now all clients can be started and terminated independently and can concurrently access all data in the central database – no code changes are necessary.

Step 3 (Three-Tier Topology)

In this step, the *wrapper insertion* topology pattern is used to place a new process – a "market server" process *M* – between the clients and the central database. We implemented a class *GenericServer* that represents the new process. The implementation of the *GenericServer* simply instantiates a local object manager component (which handles incoming requests in newly created threads) and then waits in the main thread until the process is interrupted. The server implementation and the FDO library are deployed on a dedicated server machine named *platinum*. The configuration file of one of the clients is copied to *platinum* and lines 36 and 37 of the copy are edited to give the new server's object manager the name *MarketServer* and the unique id *20*. Then all client configuration files are changed such that their server connections point to the market server on *platinum* (and thus not directly to the database any more). Listing 7-2 shows a new configuration file for a client and a new configuration file for the market server.

Object manager configuration file for one of the clients:

```
01: # definition of data objects, domains, and interfaces
02: OBJECT MANAGER * {
...    (contains lines 3 to 32 from Listing 7-1)
33: }
34:
35: # client, which connects to a central server process
36: OBJECT MANAGER Client3 {
37:    ID = 12;
38:    SERVER CONNECTION {
39:       SERVERADDRESS = SocketAddress “platinum.mi.fu-berlin.de” “13210”
40:       IMPORTS = (dom1, ds1);
41:       ADAPTER = SocketConnection;
42:    }
43: }
```

Object manager configuration file for the new market server:

```
01: # definition of data objects, domains, and interfaces
02: OBJECT MANAGER * {
...    (contains lines 3 to 32 from Listing 7-1)
33: }
34:
35: # central server, connects to a single database
36: OBJECT MANAGER MarketServer {
37:    ID = 20;
38:    EXPORTS = (dom1, ds1);
39:    SERVER CONNECTION {
40:       SERVERADDRESS = JDBCAddress
41:          "jdbc:oracle:thin:@//foo.mi.fu-berlin.de:1521/MYDB"
42:          "clemens" "gt7ye3pz" "ds1";
43:       IMPORTS = (dom1, ds1);
44:       ADAPTER = OracleJDBCConnection;
45:    }
46:    CLIENT LISTENER {
47:       LOCALADDRESS = SocketAddress “platinum.mi.fu-berlin.de” “13210”
48:       ADAPTER = SocketConnection
49:    }
50: }
```

Listing 7-2. Object manager configuration files for a client and the newly introduced market server.

Step 4 (Data Distribution)

The next evolutionary step requires applying the *data distribution* topology pattern, variant (b): Instead of storing all data objects in a single data store (the Oracle database *ds1* on machine *foo*), we distribute them across three Oracle databases *ds1*, *ds2*, and *ds3* located on machines *foo*, *bar*, and *wombat,* respectively. The following changes to configuration data are necessary:

- To define three data sets (see the partitioning step in Subsection 5.2.2), we modify the existing domain *dom1* (see lines 22 to 27 in Listing 7-1) and add two new domains *dom2* and *dom3*. Domain *dom1* contains all landlord data, *dom2* contains all accommodation and reservation data, and *dom3* contains all customer data. These changes are applied to all configuration files we have created so far.

- Then we move all accommodation and reservation data from database *ds1* to database *ds2* and all customer data from *ds1* to *ds3*.
- Also, we add two new *SERVER CONNECTION* sections (for connections to data stores *ds2* and *ds3*) to the market server's configuration data.
- Finally, we change all *IMPORTS* and *EXPORTS* directives in all configuration files to reflect that we have three different domains instead of one.

Listing 7-3 shows two object manager configuration files after the above changes have been applied (one file for one of the clients and the other file for the market server).

Object manager configuration file of a client:

```
01: # definition of data objects, domains, and interfaces
02: OBJECT MANAGER * {
...    (contains lines 3 to 20 from Listing 7-1)
21:
22:    DOMAIN dom1 { CODE = 3001;
23:       DEF = SELECT FROM Landlord;
24:       RITREE = ds1; }
25:    DOMAIN dom2 { CODE = 3002;
26:       DEF = SELECT FROM Accommodation, SELECT FROM Reservation;
27:       RITREE = ds2; }
28:    DOMAIN dom3 { CODE = 3003;
29:       DEF = SELECT FROM Customer;
30:       RITREE = ds3; }
31:
...    (contains lines 29 to 32 from Listing 7-1)
36: }
37: # Client, which connects to a central server process
38: OBJECT MANAGER Client3 {
39:    ID = 12;
40:    SERVER CONNECTION {
41:       SERVERADDRESS = SocketAddress “platinum.mi.fu-berlin.de” “13210”
42:       IMPORTS = (dom1, ds1) (dom2, ds2) (dom3, ds3);
43:       ADAPTER = SocketConnection;
44:    }
45: }
```

Object manager configuration file of the intermediate server:

```
...    (lines 1 to 36 are identical to lines 1 to 36 in the listing above)
37: # Central server, which connects to three databases
38: OBJECT MANAGER ReservationServer {
39:    ID = 20;
40:    EXPORTS = (dom1, ds1) (dom2, ds2) (dom3, ds3);
41:    SERVER CONNECTION {
42:       SERVERADDRESS = JDBCAddress
43:          "jdbc:oracle:thin:@//foo.mi.fu-berlin.de:1521/MYDB"
44:          "clemens" "gt7ye3pz" "ds1";
45:       IMPORTS = (dom1, ds1);
46:       ADAPTER = OracleJDBCConnection;
47:    }
48:    SERVER CONNECTION {
49:       SERVERADDRESS = JDBCAddress
50:          "jdbc:oracle:thin:@//bar.mi.fu-berlin.de:1560/ARDB"
51:          "clemens" "gt7ye3pz" "ds2";
52:       IMPORTS = (dom2, ds2);
53:       ADAPTER = OracleJDBCConnection;
54:    }
55:    SERVER CONNECTION {
56:       SERVERADDRESS = JDBCAddress
57:          "jdbc:oracle:thin:@//wombat.mi.fu-berlin.de:1561/DBTHREE"
58:          "fdouser" "hu7y9llw" "ds3";
59:       IMPORTS = (dom3, ds3);
60:       ADAPTER = OracleJDBCConnection;
61:    }
62:    CLIENT LISTENER {
63:       LOCALADDRESS = SocketAddress “platinum.mi.fu-berlin.de” “13210”
64:       ADAPTER = SocketConnection
65:    }
50: }
```

Listing 7-3. Object manager configuration files for a client and the intermediate server after the application of the *data distribution* pattern.

Step 5 (Cooperation with Another Marketplace)

In step number five, variant (a) of the *integration of subsystems* topology pattern is applied. More specifically, one of the clients *C* is granted read-only access to data managed by another FDO-based enterprise application *X* that implements a similar marketplace. More specifically, *C* is allowed to connect to a second server *S* that has access to all data of *X*. In this scenario, we assumed that *X* uses the same data object definitions and partitions data into the same domains as our enterprise application. (else it would be necessary to integrate schemas and/or to adjust domains first, see Subsection 5.2.4.) However, we also assumed that *X* defines a different data distribution scheme for the domains and replicates all its data objects to two data stores *ds4* and *ds5* for better availability.

For simplicity, we will focus on configuration data and changes relevant to our enterprise application and will ignore internal details of the other marketplace. In this step, the following changes to the object manager configuration of the client *C* are necessary:

- For all three domain definitions, the RI-trees are changed (lines 24, 27, and 30 in Listing 7-3). The new RI-trees are *I(ds1, R(ds4, ds5))*, *I(ds2, R(ds4, ds5))*, and *I(ds3, R(ds4, ds5))*, respectively. These changes are necessary to inform *C*'s object manager that, from now on, data items can be found not only in the data stores of our enterprise application (*ds1*, *ds2*, *ds3*). Instead, alternatively, each data object may be stored in data stores *ds4* and *ds5*. See Subsection 5.2.3 for more details on RI-trees and Subsection 5.2.4 for a discussion on how to integrate different data distribution schemes.
- A new server connection to server *S* is added. The *IMPORTS* directive of the server connection covers the following tuples: *(dom1, ds4) (dom2, ds4) (dom3, ds4)*.

Now our client *C* has access to data from both enterprise applications. Again, no code changes are necessary because data management is transparently handled by object managers. For instance, a query for all accommodation instances performed on *C*'s object manager would be sent to both servers of *C*. Then the two partial query results would be transparently integrated and finally returned to the application code of *C*.

Step 6 (Integration of the Other Marketplace)

In this step, the enterprise application *X* mentioned in the previous step is to be fully integrated so that all clients of our enterprise application are granted full access to data managed by *X*. To realize Step 6, object manager configuration data is adapted as follows:

- The changes to the three RI-trees we applied to *C*'s configuration data in the previous step are also applied to the configuration files of all other processes of the enterprise application[1].
- The *SERVER CONNECTION* section we added to *C* in the previous step is removed from *C* and added to the configuration of the market server *M* (which then has four server connections: to *ds1*, *ds2*, *ds3*, and to *S*). The *IMPORTS* directive of that section as well as *M*'s *EXPORTS* directive is changed such that they cover the following tuples: *(dom1, ds4) (dom2, ds4) (dom3, ds4) (dom1, ds5) (dom2, ds5) (dom3, ds5)*.
- The configuration data of all clients of *M* is changed such that the clients import *(dom1, ds4) (dom2, ds4) (dom3, ds4) (dom1, ds5) (dom2, ds5) (dom3, ds5)* from *M*.

After that reconfiguration step, all clients of our original enterprise application have full and transparent access to data from their marketplace and from the integrated marketplace.

[1] Using our implementation, also the configurations of all processes that belong to the integrated enterprise application *X* have to be changed. This is necessary to handle cases where data objects containing references to data objects stored in databases *ds1*, *ds2*, or *ds3* are inserted into *ds4* and *ds5*. Clients that belong to *X* cannot resolve such references since, with the chosen process topology, they have no access to *ds1*, *ds2*, or *ds3*. Using our implementation, either (1) all processes of *X* have to be informed of that fact by defining appropriate RI trees in their configuration files, (2) occurrences of such cross-application references in *ds4* and *ds5* have to be prevented, or (3) application developers have to ensure that processes of *X* never attempt to resolve cross-application references. However, alternatively, the process topology could just be redesigned such that all client processes of *X* also have access to *ds1*, *ds2*, and *ds3* – in this way, references to data objects stored in those data stores could be used without limitations.

Step 7 (Access from the Internet)

In Step 7, Internet clients are added to the enterprise application and the *wrapper insertion* pattern is applied to introduce a new intermediate server *I*. We change our system as follows: First, we set up a new server machine for *I* and copy our *GenericServer* implementation (including the FDO library) used for the market server to the new machine. Then we copy an object manager configuration file from one of *M*'s clients to the new machine and edit the file to define an appropriate name, id, and client listener.

For each Internet client, we take the client code, the FDO library, and the object manager configuration from one of *M*'s clients, replace *M*'s host name in the configuration with *I*'s host name, and add all files to an installation archive that can be downloaded and started by Internet users. Unique names and ids for Internet client object managers are assigned manually. (It would be straightforward to perform this task automatically through an installation script or on the first connect to the intermediate server *I*.)

Step 8 (Proxies for Internet Clients)

Proxy processes (see Subsection 3.4.1) for caching are inserted between Internet clients and the intermediate server *I*. Each proxy process serves clients from a specific geographical region. This evolutionary step corresponds to a combination of the *wrapper insertion* pattern and the *process replication* pattern.

The insertion of the proxy processes is carried out in the same way as the insertion of the market server in Step 3: New servers that run our *GenericServer* implementation are set up and then configured with server connections to *I* and with client listeners for receiving requests from Internet clients. All Internet clients are reconfigured such that they connect to an appropriate proxy process instead of *I* (more specifically, all clients in *r*-th region are configured to connect to the *r*-th proxy process).

After those changes, all requests issued by Internet clients are routed via the newly added proxy processes. Transparently, proxy processes cache data objects for the Internet clients. How caching influences transaction throughput and response time depends on many factors, including the approach and implementation of the caching mechanism. Our object manager implementation realizes an object cache that looks up individual data objects by their object identifier. This is advantageous especially for navigational access to data objects. A more advanced implementation could also speed up arbitrary queries by caching complete or partial query results.

Note that, with our implementation of Step 8, the mapping of a client to a geographical region (which defines which proxy process is used by a client) is performed manually through editing the server connection section in the client's configuration file. Other, more convenient options would be

- to provide different installation archives (see description of Step 7) with different configuration files for users of different geographical regions,
- to let client-side installation or start-up code determine the client's geographical region and accordingly change the local object manager's configuration file, or
- to allow servers to redirect a client's connect request to another server.

Step 9 (Replication of the Central Market Server Process)

In this step, a combination of the *process replication* pattern and the *meshing* pattern is applied. The central market server *M* is replicated and the replica *M'* is deployed on a separate machine. Except for the object manager's name, its id, and the host name defined in the client listener, *M*'s configuration is identical to that of *M*. Then we adapt the server connection section in *I*'s configuration such that *I* connects to *M*' instead of *M*. To the configuration of all other clients of *M* (which are operated by travel agents), a second new server connection section, pointing to *M',* is added. With the new server connections, we have created a mesh for each client because *M* and *M'* both provide access to the same data objects in the same data stores. How effective in terms of load distribution and fault tolerance a mesh is depends on the implementation of the object managers that form the mesh. Our implementation supports meshes by providing a simple load distribution mechanism based on a random function (see Section 6.9).

Step 10 (Replication of Proxy Processes)

In the last evolutionary step of our case study, the proxy processes placed between the Internet clients and the intermediate server *I* are replicated such that they form meshes. The setup and configuration changes to realize this step are analogous to those described in the previous step.

7.1.3 Scenario Conclusions

In this scenario, we demonstrated that our *Free Data Objects* framework provides good support for developing enterprise applications with custom and adaptable process topologies. We demonstrated that it is very simple to build custom process topologies with our framework because the structure of a process topology is defined through configuration. Topology and data-distribution details are hidden from application code, which has a comfortable, logical view on business data (see the source code given in Appendix B). We also demonstrated that adapting an existing process topology is mainly a matter of reconfiguration and setting up new servers. In particular, it was not necessary to change application code at any stage[1]. This property makes our approach particularly suited for those development projects where enterprise applications start as small (possibly prototypical) systems and then, successively, are scaled up to larger systems.

7.2 Scenario 2: Basic Transaction Performance

In this scenario, we approach performance, which is an important factor in many enterprise applications. We evaluate basic transaction throughput rates and transaction execution times for a common three-tier process topology without replication. The results provide us with a rough estimation of how many clients can be handled by a single FDO-based server. The results will also help us to judge performance results of scenarios 3 to 7, which address more specific aspects.

In this scenario, we examine two different kinds of transactions: *fine-grained* transactions that map to a single operation each and *coarse-grained* transactions, which consist of multiple operations.

7.2.1 Setup for Fine-Grained Transactions

We benchmark an enterprise application with a process topology where *n* clients access a central middle-tier server, which in turn is connected to a single Oracle database (see Figure 7-1). Each client sequentially executes transactions at a given rate and we measure how the central server's average transaction throughput and execution time develops for an increasing number of clients.

[1] In Step 3, we required the *GenericServer* implementation. However, that implementation is trivial, does not contain application-specific code, and can be reused for servers of other enterprise applications.

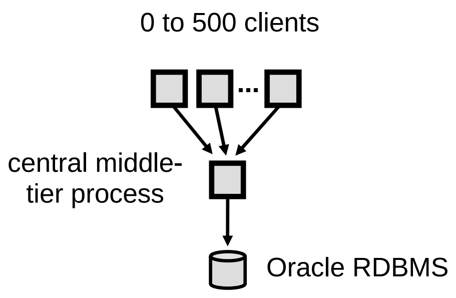

Figure 7-1. The three-tier topology used in Scenario 1 - clients access a single data store via a central server process.

As a prerequisite, we generated 5,000 data objects of type customer. Each customer instance always stores three string values (with 18, 27, and 15 characters, respectively), a Boolean value, and an integer value. We reused the data object definition from Scenario 1 (see Listing 7-1, lines 17 to 20) and changed the definition of the *customerNo* attribute from *customerNo:int* to *customerNo*:int*. The additional asterisk gives object managers a hint that customer data objects are frequently queried by their *customerNo* attribute. In our implementation, that hint is used to create a corresponding secondary index in the underlying Oracle database.

Each of the clients performs a mix of the following five simple operations that access customer data objects:

(1) The client queries one specific customer instance *C* by the value of its *customerNo* attribute. It is guaranteed that an instance with that *customerNo* exists – the query always returns one result object. Then all attribute values of the returned customer instance are read using *getXXX* calls.

(2) The customer instance *C* is updated by writing a new string value to its *contact* attribute.

(3) The client performs a query for customer instances. The query specifies a range of *customerNo* values. Always, exactly ten result objects are returned. The client then reads all attribute values of each result object using *getXXX* calls.

(4) The client deletes one random customer instance (except for *C*) of the ten instances queried in the previous step.

(5) The client inserts a new customer instance with appropriate values.

Each client loops through the five operations described above and executes each operation within a separate transaction. Operations (2) and (4) reuse object references obtained by previous transactions. After each transaction, a client waits approximately 1000 milliseconds before starting the next transaction – this simulates a short "think time" of an (imaginary) interactive user. To prevent clients from running too synchronously (i.e., process the same type of operation at the same time), we randomly use a "think time" of 800, 1000, or 1200 milliseconds for each client in each step.

A client terminates each transaction with an immediate commit request. Since our implementation does not support query caching, each transaction requires a roundtrip to the database: Transaction (1) and (3) for performing a query; (2), (4), and (5) for committing writes to the database. Using the low-level network analyzer tool *Ethereal* [SW01], we verified that the JDBC driver does not cache any data – all queries and write accesses are directly run against the database. Since there is only one database involved and the process topology does not contain meshes, coordinator transfer, and one-phase commit optimizations (see Subsection 6.7.3) apply in this scenario: On commit, clients always initiate a coordinator transfer to the central middle-tier server, which then terminates the transaction with a one-phase commit. For communication between clients and the central middle-tier process, the *NIOSocketConnection* plug-in (see Subsection 6.3) is used on both sides.

To avoid transaction conflicts in this scenario (which would lead to transaction aborts and, as a consequence, also to query result sizes other than one and ten), we let each client work on a separate subset of ten data object instances. However, we verified that, except for sporadic transaction aborts, the performance of a pure random access scenario does not differ from the scenario presented here. For a more detailed analysis of transaction conflicts, we would like to refer the reader to Scenario 7.

Detailed specifications of the hardware and software used in this scenario and the following scenarios can be found in Appendix C. Among other things, the appendix describes machines of seven different types (M1 to M7). In this scenario, all clients are deployed on 1GHz Pentium III desktop machines running Linux (machines of type M2). The central middle-tier server process runs on a 2×1GHz Pentium III machine also running Linux (machine of type M3). The database is an Oracle 9i database system that runs on a separate 2×1GHz Pentium III machine running Windows 2000 Server (machine of type M4). All machines are connected via a 100 Mbit Ethernet local area network.

Since only a limited number of machines were available for benchmarking purposes, we used five desktop machines of type M2 and ran up to 100 clients concurrently on each of those machines. All clients deployed on the same machine ran within a shared Java virtual machine process. Except for the virtual machine they shared, clients on the same machine were completely independent of each other: Each client ran within a separate Java thread, instantiated its own object manager component, and used a separate socket connection to communicate with the central middle-tier process. From the viewpoint of the middle-tier process, this setup is equivalent to a setup where each client runs on a dedicated desktop machine. Moreover, processor load on client machines is typically below ten percent, even with 100 clients running concurrently. All this means that collocating clients as described above has no significant negative impact on performance.

All throughput and time values are obtained using benchmark code (counters, calls to *System.currentTimeMillis()*, and writes of aggregated statistics to stdout) injected into appropriate places of our implementation. The benchmark code is lightweight and we verified that it has no significant impact on performance.

The central middle-tier server is started first and all clients are started afterwards. In all scenarios, we give the enterprise application a warm-up phase of about ten minutes before we start to measure values. During this time, connections are established, internal database caches are filled, the database allocates additional rollback segments (if necessary), and the Java hotspot server compiler gathers sufficient runtime statistics to compile crucial code paths. In all cases, we verified that the warm-up time was sufficient, i.e., that results do not change significantly with a longer warm-up time.

7.2.2 Results for Fine-Grained Transactions

In Figure 7-2, we present average transaction throughput rates of the central middle-tier server for different numbers of clients.

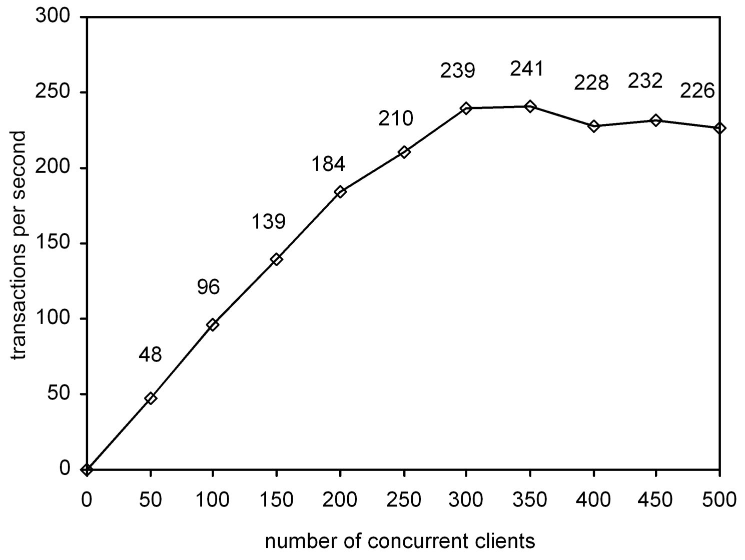

Figure 7-2. Fine-grained transactions: Transaction throughput of a single FDO-based server.

The numbers are averaged over 10,000 transactions[1]. From the diagram, we can see that, from 0 to 200 clients, transaction throughput increases nearly linearly, which means that the central middle-tier server can handle the additional load. Then the curve starts to flatten and reaches its peak at around 240 transactions per second for 300 to 350 concurrent clients. With more clients, the transaction throughput slightly drops again due to the overhead of maintaining many active connections. In an additional run with 750 clients (not shown in the diagram), we measured a throughput of about 215 transactions per second.

In the next diagram, in Figure 7-3, we show transaction execution time and processor idle time values – these numbers have been measured along with the throughput rates we presented above. In contrast to transaction throughput, which is measured at the middle-tier server, execution time values are measured directly by clients. We define "transaction execution time" as the time from a client's call to *beginTx* to the completion of the client's *commitTx* call. Execution time values are averaged over at least a hundred transactions sequentially executed by one selected client. Curve (a) in the diagram shows that, with up to 200 concurrent clients, the average transaction execution time is less than 100 milliseconds. In that range, the middle-tier process can process most incoming requests at once. With 200 to 300 clients, transaction execution time increases considerably but still remains acceptable for many applications. At that stage, client requests cannot be processed immediately any more and have to be queued (see Section 6.8). However, typically, there are only few requests in the queue and thus the delay is relatively small. When the middle-tier server has to serve more than 300 concurrent clients, the transaction execution time grows linearly with the number of clients. The reason for that is that, at some stage, most clients have a pending request in the middle-tier server's queue. Although requests are not strictly processed in FIFO order (see Subsection 6.8.2), a newly enqueued request will have to wait – on average – until *n* requests have been processed (where *n* is the number of pending requests at the time the request is enqueued).

[1] A throughput value of *v* (transactions per second) means that we measured 10,000/*v* seconds for executing 10,000 consecutive transactions.

Curve (b) illustrates processor idle time values, which are measured at the central middle-tier server using the Linux *vmstat* tool. The tool reports idle time as percentage values, e.g., an idle time value of 20% corresponds to 80% processor load. Idle time values are averaged over both processors of the server machine and over a time interval of several minutes. The curve shows that, from 0 to 200 concurrent clients, idle times decrease linearly, i.e., processor load increases linearly. This corresponds directly to the linear transaction throughput increase we observed in Figure 7-2. For 200 to 300 clients, idle times are further reduced, but not linearly any more. With 300 clients, load (like transaction throughput) reaches its maximum value. However, since the central middle-tier process heavily depends on IO from/to clients and the Oracle database, idle times in our scenario are never eliminated completely.

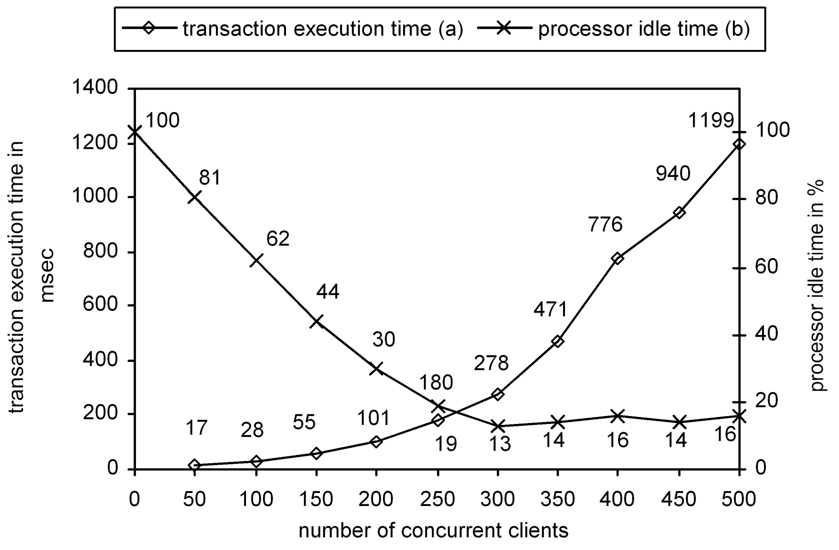

Figure 7-3. Fine-grained transactions: Average execution time per transaction (a) and processor idle time (b).

The transactions analyzed so far are very fine-grained – each transaction consists of a single operation only. In many applications, such transactions are quite common. This is especially the case when interactive users access data through simple graphical user interfaces where each transaction directly maps to a single action of the user. For example, entering a new or changing an existing customer data object and then pressing the “OK” button could trigger such a single, fine-grained transaction. However, there are also many applications where transactions are more coarse-grained and consist of multiple operations. Coarse-grained transactions are discussed in the following subsection.

7.2.3 Setup for Coarse-Grained Transactions

To compare fine-grained and coarse-grained transactions with respect to transaction throughput and transaction execution time, we slightly modified the setup used for fine-grained transactions described above. Instead of executing each operation within a separate transaction, we redefined transaction boundaries in the client application code and combined all five different operations (two queries, one update, one delete, one insert) into a single transaction. Such coarse-grained transactions are repeatedly executed by each client. To be able to directly compare the results to those for fine-grained transactions, we also increased the clients’ average “think time” after each transaction to 5,000 milliseconds (± 20% as described in Subsection 4.2.1).

7.2.4 Results for Coarse-Grained Transactions

Figure 7-4 shows average transaction throughput rates for different numbers of concurrent clients. The numbers are averaged over 2,000 transactions. Like with fine-grained transactions (see Figure 7-2), we observe a linear increase in throughput at the beginning. With 300 clients, the curve starts to flatten. With 350 clients, the throughput rate reaches its peak at 59 transactions per second, which is about one-forth of the peak rate of fine-grained transactions. We see that the ratio not 1:5 (as one might expect when five transactions are regrouped into a single transaction) but slightly better because only three instead of five roundtrips between client and middle-tier server are required per coarse-grained transaction: two for queries and one to commit the transaction. Also, commit is executed on the database only once instead of five times. (The SQL statements sent to the Oracle database are the same for fine- and coarse-grained transactions – although they are executed in a different order.) On the right y-axis, a scale for the operation throughput rate (five times the transaction throughput) is displayed for direct comparison with results for fine-grained transactions. We can see that, using coarse-grained transactions, nearly 300 operations can be processed per second while fine-grained transactions (see Figure 7-2) peak at only 241.

In Figure 7-5, transaction execution times and processor idle time values are displayed for coarse-grained transactions. Similar to our fine-grained transaction results, processor idle time decreases until a lower limit is reached. With up to 200 clients, the execution times of coarse-grained transactions are only about 20 to 50 milliseconds higher than those of single fine-grained transactions. With 250 clients, execution times start to increase significantly. For more than 350 clients (when the transaction throughput rate has reached its limit), we observe a linear increase and values of to three times the execution times of fine-grained transactions. We can also see that execution times of coarse-grained transactions are always higher than those of fine-grained transactions. However, when compared to executing all operations individually as separate fine-grained transactions, coarse-grained transactions always perform much better.

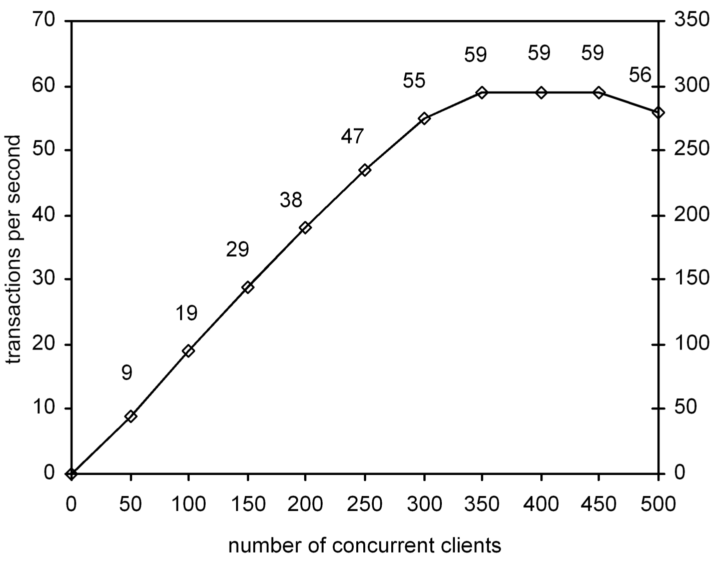

Figure 7-4. Coarse-grained transactions: Transaction and operation throughput of a single FDO-based server.

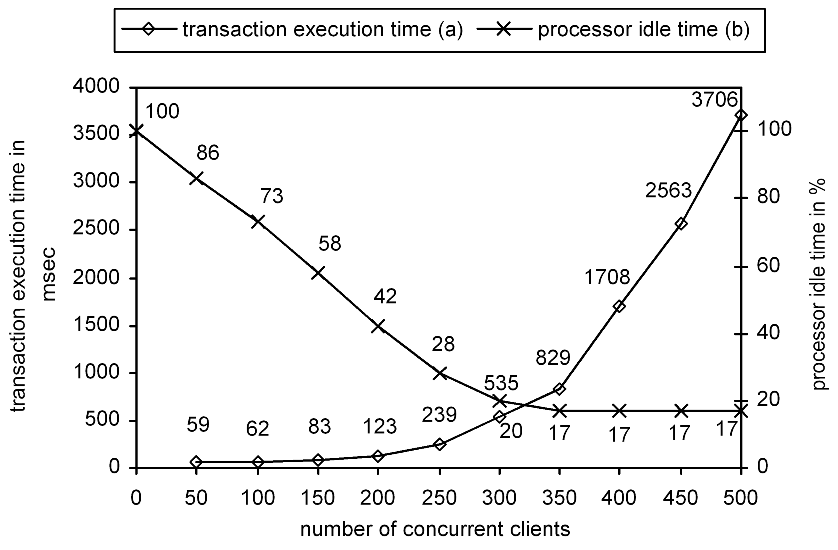

Figure 7-5. Coarse-grained transactions: Average transaction execution time per transaction (a) and processor idle time (b).

7.2.5 Scenario Conclusions

In this section, we showed that our implementation can handle hundreds of clients. We analyzed basic transaction throughput rates and execution times for an FDO-based enterprise application with a simple three-tier process topology. The numbers demonstrate that, even though both the database and the single middle-tier process run on low-end server machines, our implementation can easily handle up to 300 concurrent clients in this particular scenario. Technically, a process can handle many more clients. However, with more than 300 (active) clients, we can see that individual transaction execution times and thus response times degrade significantly with our setup. Note that with faster server machines, much higher throughput can be expected.

To how many clients a server scales in a concrete application scenario depends heavily on the activity of the clients. We simulated a “think time” of 1,000 milliseconds between two consecutive fine-grained transactions executed by each client (and 5,000 milliseconds for coarse-grained transactions). In some enterprise applications, clients will, on average, show more activity and thus put more load on a server. However, there are also many applications with clients that show far less activity in the average case.

The granularity of transactions is also essential. We learned that a single coarse-grained transaction is typically more expensive than a single fine-grained transaction because more operations have to be executed. On the other hand, we have also shown that, given a number of operations, executing them as part of coarse-grained transactions usually performs slightly better than executing them as separate fine-grained transactions.

7.3 Scenario 3: Impact of Additional Tiers on Response Time

In Chapter 3, we presented several patterns for constructing distributed process topologies. In Chapter 4, we argued that custom process topologies are needed to address application-specific requirements and that enterprise application middleware should therefore provide support for custom process topologies. Within complex custom process topologies, clients often do not access data stores directly. Instead, for various reasons (e.g., security, caching, management of replicated data), they access data stores via a path of one or more intermediate servers. In particular, applying the *wrapper insertion* pattern, the *facade* *insertion* pattern, or the *integration of subsystems* pattern (variant b) to a process topology increases the path length for clients. However, many developers are concerned that employing intermediate servers might introduce a considerable amount of latency and thus impair response time of an enterprise application. In this scenario, we analyze client response times for various numbers of intermediate servers and argue that such concerns are not justified.

7.3.1 Setup

Figure 7-6 depicts the different process topologies analyzed in this scenario: Multiple clients access a single data store via a chain of *n* intermediate processes ($n = 1, 2, 3, 4, 5$). When there is exactly one intermediate process, the topology is equivalent to that used in Scenario 2. The clients are deployed on five different machines. All clients, the database, and the last process in the chain of intermediate processes run on the machines described in Scenario 2. The other n-1 intermediate processes are deployed on dedicated 1GHz Pentium III desktop machines running Linux (machines of type M2, see Appendix C). Note that the process topologies selected for this scenario are rather artificial since intermediate processes mainly relay requests and responses in this case. In practice, it would not be reasonable to create chains of processes without assigning appropriate functionality to them. However, these topologies allow us to analyze the net overhead of introducing additional intermediate tiers without having to deal with specific functionality in tiers. Internally, messages are unmarshalled, dispatched, and marshalled again by intermediate processes. Also, messages are transformed into tasks (see Subsection 6.8.3), which are queued and processed asynchronously. Processing includes, among other things, performing a (trivial) object and query routing and also caching of data objects found in messages.[1]

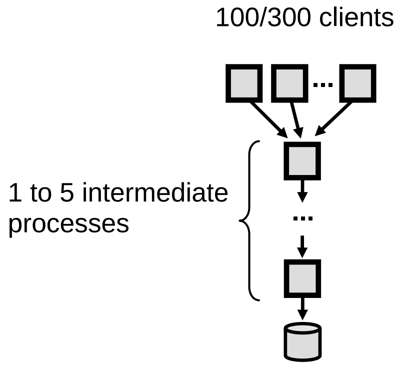

Figure 7-6. The process topologies analyzed in Scenario 3: Clients access a single data store via a chain of one or more intermediate processes.

The same transactions, client code, "think time", and data objects as for fine-grained transactions described in Scenario 2 are used here. However, instead of employing the *NIOSocketConnection* plug-in for communication between object managers, we used the *SocketConnection* plug-in.[2] For each number of intermediate processes, we measure transaction execution time (from the viewpoint of a client) and average transaction throughput. Since each transaction contains only a single operation, transaction execution time is identical to client response time in this case (assuming that each transaction maps to an action of a user). We provide results for both a high load situation (300 concurrent clients) and a medium load situation (100 concurrent clients).

[1] Object caches are maintained by intermediate processes, but clients do not benefit from caching in this scenario.
[2] The *SocketConnection* plug-in is less optimized than the *NIOSocketConnection* plug-in, but requires less memory per connection. This is important since 300 connections to clients are managed on the first intermediate server, which is a desktop machine.

7.3.2 Results

Response time results are shown in Figure 7-7. Values are averaged over at least a hundred transactions sequentially executed by one selected client. We can see that introducing intermediate processes clearly *has* an effect on client response time. With 100 clients, i.e., when load is moderate, each additional process in the chain costs about 10 to 15 milliseconds additional response time. Compared to 28 milliseconds response time for a topology with one intermediate process, this is relatively high. The absolute values, however, are still low. In a high-load situation with 300 clients, the costs of an additional tier tend to be slightly higher than with 100 clients. However, when compared to 369 milliseconds response time for a topology with one intermediate process, they are not too significant. When low response time is a priority, reducing or offloading some load will typically be much more effective than reducing the number of tiers, as we can see in Figure 7-3.

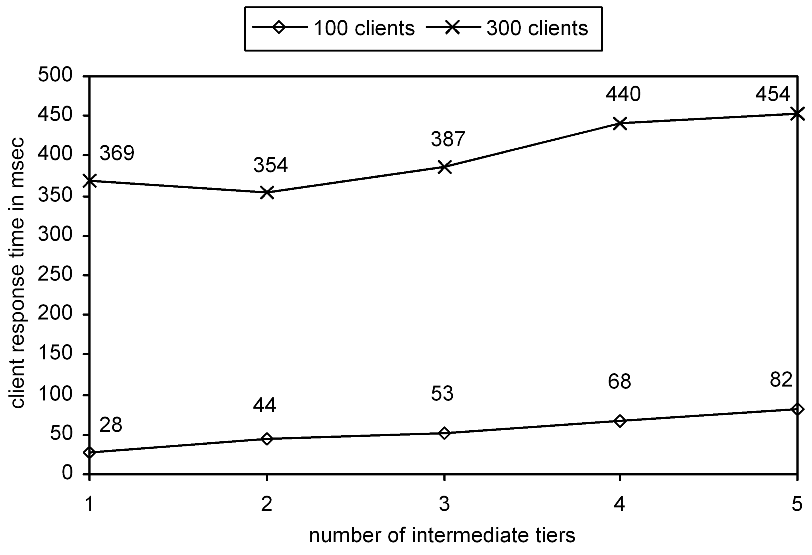

Figure 7-7. Client response times for different numbers of intermediate tiers.

In Figure 7-8, transaction throughput rates are displayed. The numbers are averaged over 2,000 transactions. We can see that introducing intermediate processes has only little influence on overall transaction throughput. With many intermediate processes, there is a slight decrease. This can be explained by the increased response time, which has an effect similar to a higher "think time" value.

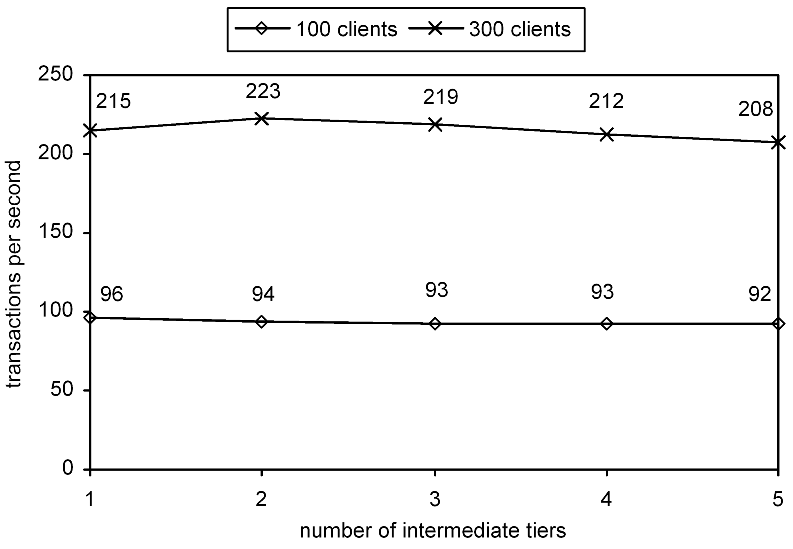

Figure 7-8. Transaction throughput for different numbers of intermediate tiers.

7.3.3 Scenario Conclusions

In this scenario, we have shown that, from a performance point of view, the introduction of one or more additional tiers between clients and data stores is acceptable. Additional tiers increase response times for clients, but not much. Furthermore, we have seen that transaction throughput is mostly independent of the number of intermediate tiers.

We think that the latency caused by intermediate tiers is often overestimated. Bad response times are much more likely to be a result of server overload (which can add seconds to response times) than of too many intermediate processes (which add milliseconds). We believe that software architects should focus on clear, scalable process topologies and should not attempt to keep path lengths short at all costs. In fact, in many cases, the introduction of an intermediate tier can help to *improve* response times. An example is a tier for transparent caching, which answers client queries immediately instead of routing them to a data store. Or, when data is distributed among multiple data stores to distribute load, a new tier could be introduced to shield clients from the complexities of distributed data management. As we can see in Figures 7-3 and 7-7, reducing a high server load can easily offset the small latency caused by an additional tier – provided the new tier is not a bottleneck itself.

7.4 Scenario 4: Coarse-Grained versus Fine-Grained Remote Data Access

In Subsection 4.4.1, we argued that a too fine-grained approach to accessing remote data often leads to significant performance problems. Fine-grained access is especially encouraged by object middleware where it is straightforward to represent data items as middleware remote objects, e.g., as CORBA objects or as Java RMI objects. In this section, we compare performance of coarse-grained remote access, which is typical of our approach, to fine-grained access based on RMI. In particular, we analyze the performance of queries that return multiple data object instances. We decided to focus on such queries in this scenario because they are frequently used in many typical enterprise applications and, if not implemented efficiently, can be a major performance bottleneck.

7.4.1 Setup

We use the same process topology (see Figure 7-1) and machines as the ones used in Scenario 2. All clients constantly query data from a central middle-tier server process, which in turn accesses a single Oracle RDBMS. There is no “think time” between transactions, which causes considerable load even with relatively few clients. All clients are deployed on five client machines. We measure values for both a *high load situation* (ten concurrent clients; two per machine) and an *overload situation* (125 concurrent clients; 25 per machine).

We reused data object definitions of Scenario 1 (see Listing 7-1) and filled the database with 5,000 data objects of type *Accommodation* and ten data objects of type *Landlord*. Each accommodation instance references a random landlord instance. Each landlord instance has a payload[1] of 90 bytes stored in four attributes. Accommodation instances have a payload of 87 bytes stored in twelve attributes. To speed up access to accommodation data, we added a hint to the definition of data objects that leads to the creation of a secondary index on the *countryCode* attribute in the Oracle database.

Clients follow an access pattern that is typical of users who browse through data that is presented to them in tabular form. Each row represents an accommodation and each column represents an attribute value. All actions necessary to fill the cells of a (GUI) table with accommodation data are executed within one transaction. The first operation of a transaction is a query for all accommodation instances that match a *countryCode* value randomly selected by the client. Then, for each accommodation instance in the result of the query, the client reads all attribute values, resolves the reference to the accommodation's landlord instance, and reads the landlord's name. After the client has executed one transaction, the client immediately starts the next transaction with a new random *countryCode* value.

We compared performance values for three different variants: Variants (a) and (b) are based purely on our FDO framework whereas variant (c) uses Java RMI for remote data access and employs the FDO framework only for direct database access. The differences between variants (a) and (b) on the one side and (c) on the other side are depicted in Figure 7-9. Only one client is shown in each case.

[1] We use the *payload* of a data object instance (in bytes) as a rough measure of the net amount of data stored in it – excluding any overhead introduced by data stores, frameworks, or network protocols. We define the payload of a data object instance as the sum of payloads of all its attribute values. Each *null* value counts as 1 byte, regardless of the attribute’s type. *Boolean*, *short*, *int*, *long*, *float*, and *double* attribute values count as 1, 2, 4, 8, 4, and 8 bytes, respectively. A *string* value that consists of *n* characters counts as *n* bytes. An object reference counts as 4 bytes. Note that the actual space required for storing an instance in a data store, caching it in an object manager, or transmitting it over the network can be much higher than the payload value.

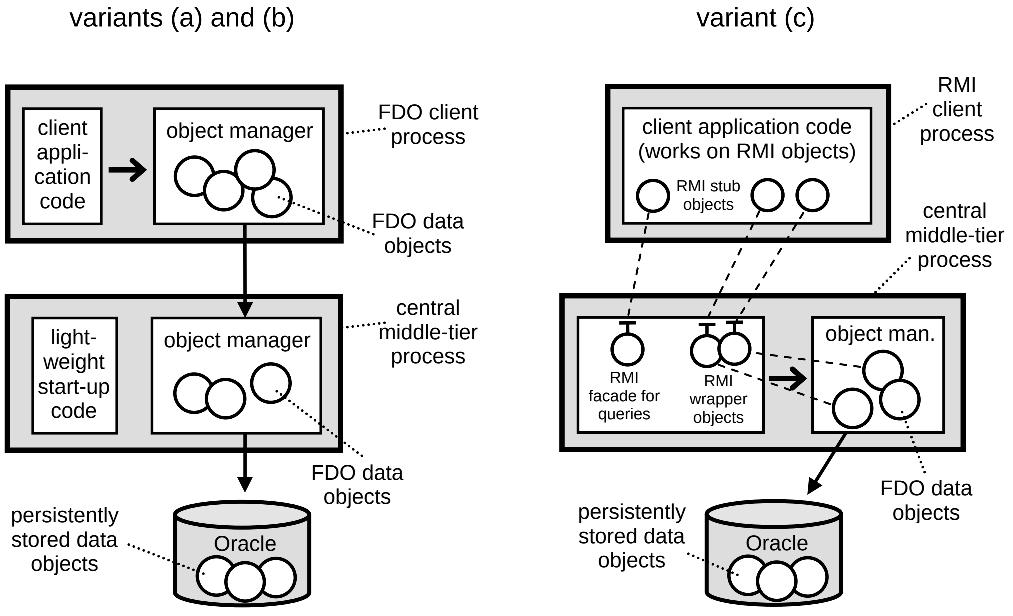

Figure 7-9. Setup of Scenario 4: While variants (a) and (b) are purely based on our FDO framework, variant (c) uses Java RMI for remote data access.

In the following, the three variants are described in more detail:

Variant (a): FDO SocketConnection

We implemented an FDO-based client and configured all clients and the central middle-tier server to use our *SocketConnection* protocol plug-in for communication (see Section 6.3). Internally, the plug-in uses Java serialization for transmitting data between client object managers and the server object manager. Implementation of transactions is straightforward:

1. a new transaction is opened,
2. a query of the form

   `SELECT FROM Accommodation WHERE countryCode=`*`randomValue`*` ORDER BY rentPerDay`

   is executed on the local object manager,
3. on each result object of the query, *getLandlord().getName()* is executed,
4. for each non-reference attribute *X* of each result object, *getX()* is invoked,
5. the current transaction is closed.

Variant (b): FDO NIOSocketConnection

The implementation is identical to variant (a). The only difference is that clients and the server are configured to use the more sophisticated *NIOSocketConnection protocol plug-in* (see Section 6.3) for communication. The plug-in uses a custom wire format instead of Java serialization and utilizes threads more efficiently than the *SocketConnection* implementation.

Variant (c): RMI

In this variant, data objects are represented as middleware remote objects (RMI objects) that reside in the central middle-tier process. Clients do not use any part of our FDO framework. Instead, they use Java RMI (stubs) to communicate with the application code in the middle-tier process. The server-side application code exposes an RMI facade to clients, which allows them to set transaction boundaries and to query accommodations. For realizing persistence and transaction management, the central middle-tier server relies on an FDO object manager component. Thus the object manager is used only as a local access layer on top of the Oracle database.

The server-side application code delegates all incoming remote invocations to its local object manager. For each FDO data object instance returned by the object manager (as a result of an explicit query or of a navigational access), a lightweight RMI object that wraps the FDO data object is instantiated by the server-side application code. To clients, remote references to those RMI wrapper objects are returned. The RMI interface of the facade and an excerpt of the RMI wrapper interfaces for accommodations and landlords are shown in Listing 7-4.

Remote interface of the RMI facade for querying accommodations and for transaction demarcation:

```
public interface QueryInterface extends Remote {
   Serializable beginTx() throws RemoteException;  // returns txId
   RemoteAccommodation[] getAccommodationsByCountryCodeOrderByRentPerDay(
      Serializable txId, short countryCode )
      throws RemoteException, FDOException;
   void commitTx( Serializable txId )
      throws RemoteException, CommitException;
  void rollbackTx( Serializable txId ) throws RemoteException;
}
```

Remote interface of RMI wrapper objects for FDO data objects of type *Accommodation*:

```
public interface RemoteAccommodation extends Remote {
   public String getAddress(Serializable txId )
      throws fdo.om.DeletedException, RemoteException;
   public void setAddress( Serializable txId, String address )
      throws fdo.om.DeletedException, RemoteException;
   public RemoteLandlord getLandlord(Serializable txId )
      throws fdo.om.DeletedException, RemoteException;
   public void setLandlord( Serializable txId, RemoteLandlord landlord )
      throws fdo.om.DeletedException, RemoteException;
...   (get/set methods for ten other attributes)
}
```

Remote interface of RMI wrapper objects for FDO data objects of type *Landlord*:

```
public interface RemoteLandlord extends Remote {
   public String getName(Serializable txId)
      throws fdo.om.DeletedException, RemoteException;
   public void setName( Serializable txId, String name )
      throws fdo.om.DeletedException, RemoteException;
...   (get/set methods for three other attributes)
}
```

Listing 7-4. Java RMI interfaces used for variant (c) of Scenario 4.

The RMI wrapper approach described above is one of many ways of how to integrate non-FDO processes into an FDO-based enterprise application (also see Subsection 6.10.3). Although clients do not use a local object manager or any class of our FDO framework, they can access FDO data objects in a transactional, object-oriented, and type-safe manner[1]. The implementation of transactions at the client side is equivalent to that of FDO-based clients, except that remote object references instead of references to local FDO data objects are used. Object identity (see requirement R2 in Subsection 4.2.2) is preserved by server-side application code, which guarantees that each FDO data object instance is wrapped by at most one RMI wrapper object at any time. An RMI wrapper object can be concurrently accessed by any number of clients.

[1] The RMI wrapper approach is a particularly lightweight approach. However, a disadvantage is that server-side interfaces as well as logic for wrapping data objects are left completely to application developers. Also, clients have to pass transaction identifiers explicitly as parameters to RMI calls.

For each of the three variants described above, we measure the total transaction throughput at the central middle-tier server. We varied the number of result objects returned by each query (0, 5, 10, 15, and 20 result objects). All throughput values are averaged over 10,000 transactions except for the RMI variant where values for 5 to 20 result objects are averaged over 1,000 transactions.

7.4.2 Results

In Figure 7-10, average transaction throughput per second for a high load situation (10 concurrent clients, no "think time" between transactions) is shown. Figure 7-11 displays values for an overload situation (125 concurrent clients, no "think time" between transactions). Both figures show values for each of the three variants described in the previous subsection. First, we discuss the results for the high load situation. Then we describe the results for the overload situation.

High Load Situation

The diagram in Figure 7-10 shows that, for queries that return no result objects, all three variants achieve nearly identical throughput of about 180 to 200 transactions per second. Although no data objects have to be transmitted in that case, it is still necessary to have a network roundtrip between a client and the central middle-tier server and to execute a database query per transaction.

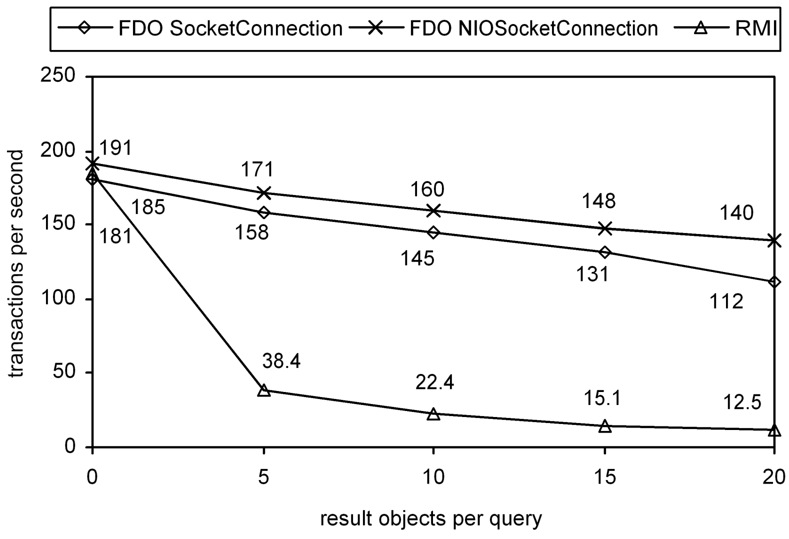

Figure 7-10. Query transaction throughput for different numbers of result objects – high load situation.

For queries that return five accommodation data objects, the situation changes completely: The throughput of the RMI variant drops by eighty percent to less than 40 transactions per second. In contrast to that, both FDO-based variants drop only slightly to 171 and 158 transactions, respectively. The significant performance drop of the RMI variant can be explained by the large number of network roundtrips between clients and the central middle-tier server; for each result object returned by a query, thirteen roundtrips are necessary to read attribute values (eleven for reading primitive attribute values of the accommodation instance, one for resolving the reference to a landlord instance, and one for reading the landlord's name attribute). All in all, 66 client/server roundtrips (1+5*13) are required to execute a single transaction! This fine-grained communication style imposes a heavy load on the central middle-tier server and thus limits its throughput. In contrast to that, both FDO-based variants still require only one roundtrip per transaction. Since query caching is not implemented in our framework, the FDO-based variants do not benefit from query caching – for each query, always all result objects are fetched from the database and transmitted to clients. However, to a limited extent, both FDO-based variants benefit from *object* caching because they are allowed to cache landlord instances. Thus, resolving references to landlord instances and reading a landlord's *name* attribute can typically be performed locally without contacting the server.

With 10, 15, and 20 result objects, the trend described above continues. The throughput of the RMI variant drops to only 12.5 transactions per second for 20 result objects per query while the FDO-based variants still achieve a throughput of 140 and 112, respectively. For the *FDO NIOSocketConnection* variant, we also measured values for 50 and 100 result objects per query (88 and 53 transactions per second, respectively; not shown in the diagram). Note that for all variants, the throughput decreases not only because of higher communication costs but also because more data objects per transaction have to be fetched from the Oracle database.

In all cases, the *FDO NIOSocketConnection* variant provides higher throughput than the *FDO SocketConnection* variant. This can be attributed to the more efficient marshalling code and smaller message sizes of that variant. With the *ethereal* network analyzer tool, we analyzed the size of messages exchanged between a client and the central middle-tier server (by adding the sizes of the data parts of all packets sent over the underlying TCP connection). With the *FDO NIOSocketConnection* variant, which employs a custom marshalling mechanism, a QueryRequestMessage contains 104 bytes and a QueryReplyMessage 49 bytes plus 162 bytes per result object of type *Accommodation*. With the *FDO SocketConnection* variant, which relies on Java Serialization, a QueryRequestMessage contains 397 bytes and a QueryReplyMessage 381 bytes plus about 205 bytes per result object of type *Accommodation*. While both FDO-based variants introduce a certain overhead (compared to the 87 bytes payload per result object), the RMI variant generates much more traffic: For each *getXXX* RMI request and also for each response, about 100 bytes are required in this scenario.

All in all, a single transaction for querying 20 accommodations maps to about 3.4 Kb network traffic with the *FDO NIOSocketConnection* variant, 4.9 Kb with the *FDO SocketConnection* variant, and more than 50Kb with the RMI variant. This means that the fine-grained communication style of the RMI variant does not only result in more messages but also requires significantly more data to be processed and to be sent over the network.

Overload Situation

In Scenario 2 (see Figure 7-2), we have seen that the average transaction throughput may decrease slightly when a fully loaded FDO-based server is given even more work. The ten clients used in the high load situation described above are enough to saturate the central middle-tier process (except for empty query results). Figure 7-11 shows results for a severe overload situation with 125 concurrent clients and no "think time" between transactions.

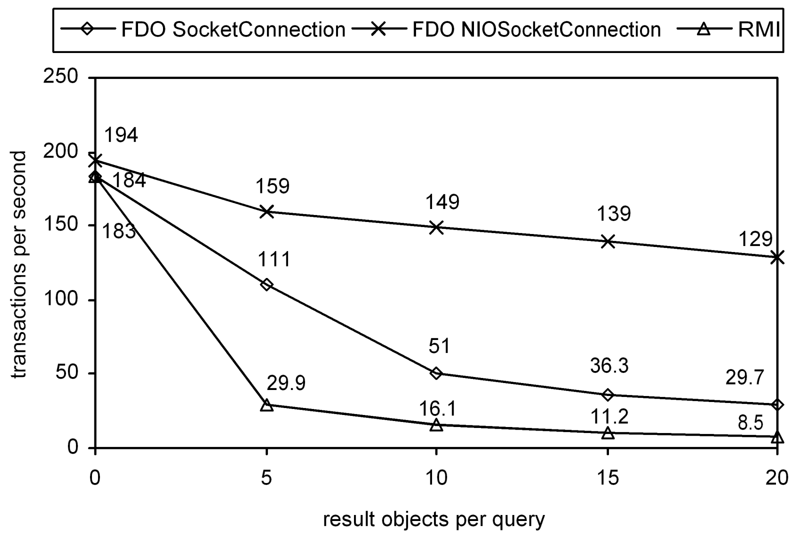

Figure 7-11. Query transaction throughput for different numbers of result objects – overload situation.

For queries that return no result objects, the performance of all three variants does not differ much from the high load situation because the central middle-tier process is not fully loaded then. However, for non-empty query results, all throughput values are lower than the values of the high load situation depicted in Figure 7-10. The *FDO NIOSocketConnection* variant is relatively stable and drops only by less than ten percent, compared to the high load situation. The RMI variant is more sensitive to overload and decreases by about 20 to 30 percent. The *FDO SocketConnection* variant does not cope well with the overload situation and achieves only a fraction of the corresponding throughput in the high load situation, which is mainly because the *SocketConnection* plug-in employs much more concurrent threads (two per connection) than the *NIOSocketConnection* plug-in. However, we can see that, even with overload, the *FDO SocketConnection* variant still performs more than three times better than the RMI variant.

7.4.3 Scenario Conclusions

In this section, we compared coarse-grained access to remote data objects, which is typical of our approach, with fine-grained access, which is encouraged by object middleware. In particular, we compared the throughput of query transactions for two FDO-based variants and an RMI-based variant. We showed that the RMI-based variant requires considerably more messages, sends more data over the network, and thus produces much more server load. As a result, the RMI-based variant achieves only a fraction of the throughput of the FDO-based variants.

Note that we used a low-latency, high-bandwidth network (see Appendix C) in this scenario. With more latency and/or lower bandwidth, the performance of the RMI-based variant would suffer most as it requires more network roundtrips and bandwidth than the FDO-based variants. For instance, in a wide area network with ping times of 100 milliseconds, displaying a GUI table with just five accommodation data objects would require at least six seconds (66 roundtrips; each with 100ms latency).

In principle, we could have compared an FDO-based implementation with an implementation purely based on object middleware (e.g., an application server that supports EJB entity beans). However, with such an approach, it is difficult to identify the impact of remote access granularity and analyze it independently of other aspects: With an EJB application server, many crucial aspects like database access, object/relational mapping, caching, consistency mechanisms, and concurrency control are handled and implemented differently. Therefore, we decided to employ an FDO object manager component in the middle-tier process of the RMI-based variant. In this way, all three variants use the same (server side) data management mechanisms – only client remote access to data objects is handled differently – and therefore are directly comparable.

7.5 Scenario 5: Process Replication

In Chapter 3, we presented several process topology patterns. Among other things, we presented the *process replication* pattern (Subsection 3.4.2) and the *meshing* pattern (Subsection 3.4.3), which can be used to distribute the load generated by clients horizontally among several replicated processes. In this section, we analyze transaction throughput of a scenario in which those patterns have been used to distribute load among multiple middle-tier processes of a three-tier process topology.

7.5.1 Setup

In this scenario, a three-tier process topology with 100 clients, up to 4 replicated middle-tier processes, and a single Oracle 9i RDBMS is used (see Figure 7-12). The client processes are deployed on four different 1GHz client machines of type M2 (25 clients on each machine). Each middle-tier process runs on a dedicated 450MHz machine of type M1[1]. The database runs on a dedicated 2x1GHz machine of type M4. Each client process is configured to connect to all middle-tier processes (*meshing* pattern) and each middle-tier process is configured to connect to the central database. The same transactions, client code, and data objects as for coarse-grained transactions described in Scenario 2 are used. There is no "think time" for clients between transactions. Object managers communicate by means of our *NIOSocketConnection* plug-in. The random routing mechanism employed in the implementation of our *Free Data Objects* framework ensures that each client request is routed to one[2] random middle-tier process, which means that client load is distributed equally among all middle-tier processes.

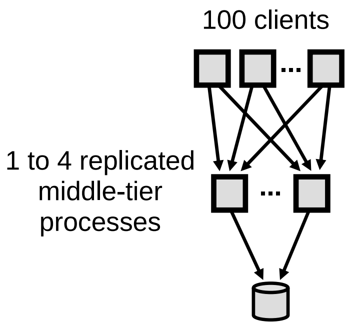

Figure 7-12. Client load is distributed horizontally among replicated middle-tier processes.

We measure the average transaction throughput for each of the replicated middle-tier processes. Then we add the individual throughput rates to obtain the total transaction throughput rate of the system. All values are averaged over 1,000 transactions.

[1] In contrast to most other scenarios, we decided to place the server processes on relatively slow machines of type M1. Thereby, we can better demonstrate the effects of load distribution. With server processes that run on faster machines (e.g., of type M3), process replication does not improve throughput because the central database immediately becomes a bottleneck. Since no faster database machine was available, we decided to use slower machines for server processes instead.

[2] In principle, our routing mechanism can route a request to multiple server processes, e.g., for querying multiple data stores or on commit of a distributed transaction. However, since only one database is used and all middle-tier processes import/export the same data, each request is always routed to exactly one middle-tier process in this scenario.

7.5.2 Results

The diagram in Figure 7-13 shows throughput of coarse-grained transactions for up to four middle-tier processes. With one middle-tier process, a throughput of 11.7 transactions can be achieved, which is only about 20 percent of the peak throughput we observed in Scenario 2 (see Figure 7-4). This is not surprising since the middle-tier process runs on a 450MHz machine in this scenario while Scenario 2 relies on a machine with 2×1GHz.

With more middle-tier processes, we observe an almost linear increase in throughput. In fact, the throughput is nearly proportional to the number of middle-tier processes employed. In this scenario, the database is clearly not a major bottleneck since we know from Scenario 2 that the database is able to process at least 59 transactions per second. However, each additional middle-tier processes increases the load on the database and slightly reduces the throughput per process. Therefore, the transaction throughput does not grow perfectly linearly in this scenario.

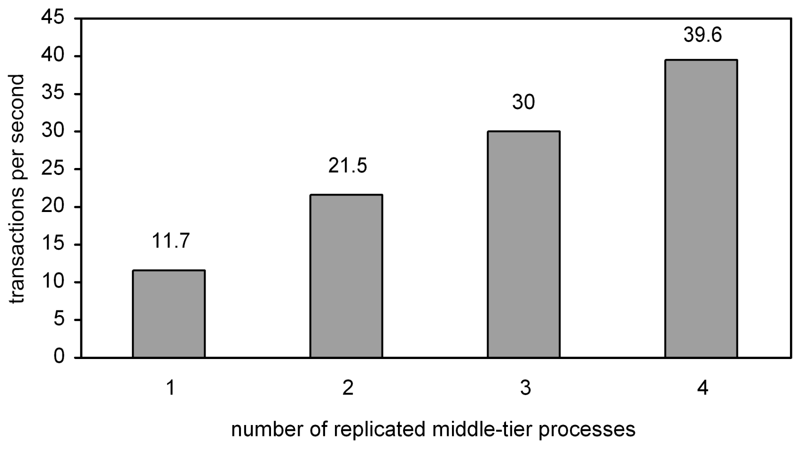

Figure 7-13. Throughput of coarse-grained transactions with up to four replicated middle-tier processes.

7.5.3 Scenario Conclusions

In this scenario, we demonstrated the capabilities of the *process replication* and the *meshing* patterns. Client load is horizontally distributed among replicated middle-tier processes of a three-tier topology. Ideally, transaction throughput should be proportional to the number of replicated middle-tier processes employed. We demonstrated that, in this scenario, our FDO framework comes close to the ideal throughput and achieves a nearly proportional throughput rate.

7.6 Scenario 6: Data Distribution

In Subsection 3.4.4, we presented the *data distribution* process topology pattern and its three variants. With that pattern, data is not stored in one central data store but is distributed among multiple data stores instead. Our FPT architecture and our FDO framework support all three variants of the pattern. In this scenario, we examine how transaction throughput of an FDO-based application is affected by data distribution in general and, in particular, by the three different variants.

7.6.1 Setup

Figure 7-14 shows the three different process topologies (*A*, *B*, and *C*) analyzed in this scenario. Topology *A* uses only a single data store and is included here so that we can compare data distribution with a non-distributed situation. In topology *B*, data is distributed among two data stores, which are accessed by a central middle-tier process. Topology *C* is similar to topology *B* but uses two middle-tier processes (process replication). Each middle-tier process is connected to all clients but only to one of the two data stores.

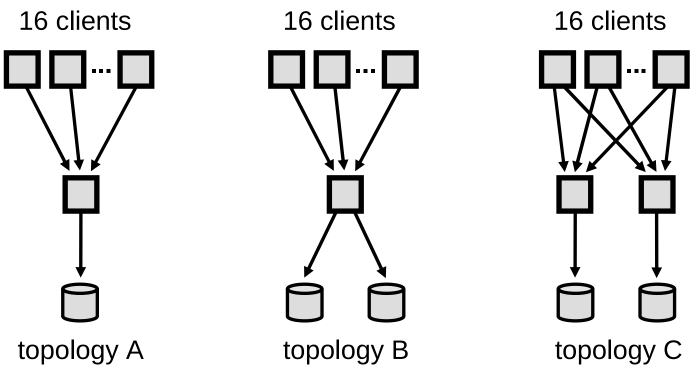

Figure 7-14. The three process topologies used in Scenario 6.

In this scenario, 16 clients are employed. They run on four different client machines of type M2 (see Appendix C); four clients are deployed on each machine. All clients use our *NIOSocketConnection* plug-in for communication. The middle-tier process of topologies *A* and *B* runs on a dedicated server machine of type M7. The two middle-tier processes used in topology *C* run on dedicated machines of type M3 and M7, respectively. In all topologies, Oracle 9i RDBMS are used as data stores. In topologies *B* and *C*, the databases run on two dedicated machines of type M5 and M6, respectively. For topology *A*, we measure values once for a database machine of type M5 and once for a database machine of type M6.

In the previous scenario, we decided to employ relatively slow machines for running middle-tier processes. Thereby, we ensured that the database was never a bottleneck in that scenario and scaled to several replicated middle-tier processes. In this scenario, we take a similar approach. However, instead of limiting the throughput of middle-tier processes, we limit the throughput of databases. While all other scenarios used a machine of type M4 for the database, we use two slower machines of type M5 and M6 in this scenario. Furthermore, we employ a machine of type M7 for a middle-tier process (see above). That machine is faster than the server machines used in other scenarios[1]. With such a setup, middle-tier machines do not slow down database machines. That allows us to demonstrate the effects of data distribution.

As a prerequisite, the database(s) is/are filled with 10,000 data objects of type *Customer*, which are assigned unique *customerNo* values from 1 to 10,000. We reuse the data object definition from Scenario 1 (see Listing 7-1, lines 17 to 20). Each customer instance always stores three string values (with 18, 27, and 15 characters, respectively), a Boolean value, and an integer value. In the Oracle database(s), a secondary index on the *name* attribute of customer objects is generated. For topologies *B* and *C*, we measure values for each of the three data distribution variants: replication, ad hoc distribution, and distribution based on a static partitioning criterion. In the latter case, we partition customers into two domains (for domains see Subsection 5.2.2) – one domain for each database. The first domain contains all customers with *customerNo* ≤ 5,000 and the second domain contains all other customers.

All clients constantly execute transactions. There is no "think time" between transactions. We measure values for two different transaction mixes:

1. The first mix contains only query transactions. Each transaction consists of a single range query that selects 20 customer data objects by their *customerNo* values. For the start of the range, a random value is used. Note that there is no database index on the *customerNo* attribute and thus

[1] Since that machine was available for benchmarking purposes only a very limited amount of time, we did not use in other scenarios.

the Oracle database performs a full table scan for each query. Although a table scan can typically be performed in memory (at some point, all pages that contain customer data reside in the database's cache), that puts additional load on the database(s).

2. The second transaction mix contains only a few query transactions and many update transactions. Each client first executes a query transaction as described above. Then, for each of the 20 customers returned by the query, the client executes a separate transaction to update the *contact* attribute of the corresponding customer. That means that 20 out of 21 transactions are (fine-grained) update transactions.

As described in Scenario 2, we made sure that concurrent transactions do not conflict.

7.6.2 Results

The diagram in Figure 7-15 shows transaction throughput per second for query transactions. All values are averaged over 10,000 transactions.

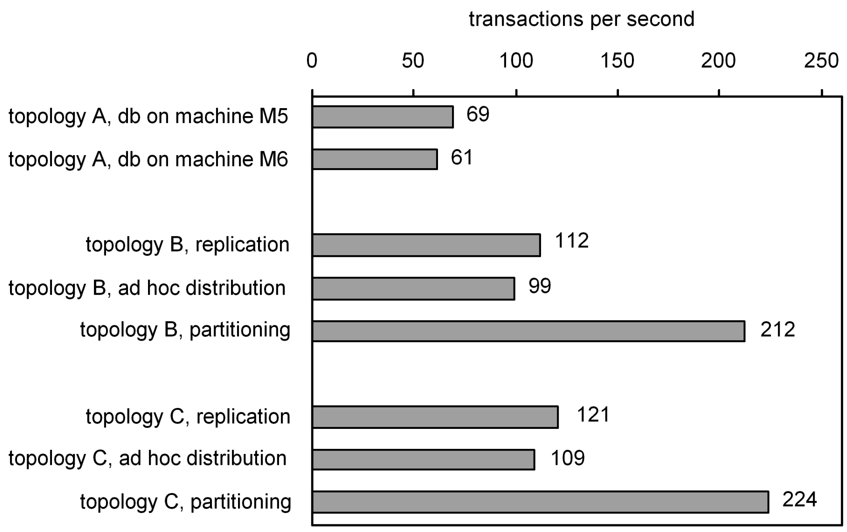

Figure 7-15. Transaction throughput for query transactions.

With a central database running on a machine of type M5, only 69 transactions per second can be achieved. With a database deployed on a machine of type M6, even less transactions (61 per second) are executed. These numbers are significantly lower than those of previous scenarios because the database performs a full table scan per query. Besides, the database machines used here are slower than the machine of type M4, which is employed in all other scenarios.

For process topology *B*, which distributes data among two data stores, transaction throughput rates are much higher than with a central database. When the replication variant is used for data distribution, 112 transactions per second are achieved. The value comes close to the "ideal" throughput of 2*61 transactions possible in this context with our current implementation[1].

Using the ad hoc data distribution variant performs nearly as well as the replication variant. At first glance, such a good throughput is surprising because, with ad hoc distribution, each query is sent to *both* data stores and the two partial query results have to be integrated. In contrast to that, the replication

[1] Each query is routed to a random data store with our current implementation. That implies that both data stores always process approximately the same number of queries per unit of time. Therefore, the slower data store limits the overall query throughput to at most twice the throughput of the slower machine. With a routing mechanism doing a more sophisticated load balancing, the overall throughput would be ideally the sum of individual throughput rates of both databases.

variant requires only one database to be contacted per query. The relatively good performance can be explained as follows: A database performs a full table scan for each query because no index can be utilized in this scenario. When data is evenly (ad hoc) distributed among two databases, each database contains only half of all data objects. While the replication variant requires one full table scan of 10,000 customers in one database, the ad hoc distribution variant requires each of the two databases to perform a full table scan of 5,000 customers. With an appropriate index, no full table scans would be required and thus the difference between the two variants would be much more significant.

The highest throughput for process topology *B* is achieved when data is distributed with the partitioning variant. With 212 transactions per second, that variant is about twice as fast as the other two data distribution variants. In fact, in this particular scenario, the partitioning variant can combine the advantages of the other two variants: First, only one database is accessed per query (advantage of the replication variant). Second, since each database contains only half of all data objects, the full table scan for a query is less expensive (advantage of the ad hoc distribution variant).

The throughput numbers for topology *C* are about 5 to 10 percent better than those for topology *B*. In contrast to topology *B*, routing decisions are made by clients and not by middle-tier processes in topology *C*. Also, each middle-tier process of topology *C* requires fewer database connections than the middle-tier process of topology *B*. Furthermore, middle-tier load is distributed among two processes/machines in topology *C*. Although no middle-tier machine is fully loaded with query transactions (i.e., the databases are always a bottleneck), load distribution helps to improve throughput slightly.

The diagram in Figure 7-16 shows transaction throughput per second for the transaction mix that consists mainly of update transactions. Again, all values are averaged over 10,000 transactions.

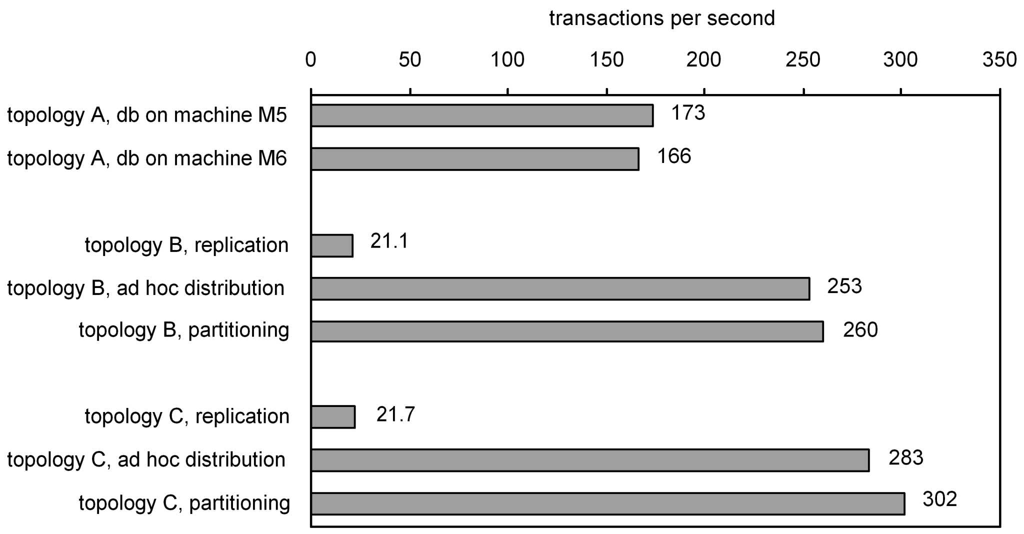

Figure 7-16. Transaction throughput for a mix dominated by update transactions.

The numbers for topology *A* (one database only) show that significantly more update transactions than query transactions (as discussed above) can be executed per second. There are two reasons for that: First, an update transaction affects only a single data object while a query transaction copies 20 data objects over the network. Second, the database can utilize a primary index for updating individual data objects whereas each query requires a full table scan.

With topology *B* and ad hoc distribution or partitioning, transaction throughput is higher than with topology *A* because updates are distributed among the databases. However, these two cases are the only ones in this scenario where the middle-tier machine is fully loaded and thus a bottleneck. With a faster middle-tier machine, the numbers could be higher.

The throughput for topology *B* in combination with the replication variant (21 transactions per second) is surprisingly low. Replication is likely to be slower than ad hoc distribution and partitioning because each update has to be performed on both databases instead of only one. Furthermore, update transactions are distributed transactions (two databases are updated atomically) and thus there is an additional overhead for driving the two-phase commit protocol, including force writes to the coordinating object manager's transaction log (see Section 6.11 for distributed transaction management). However, with our current implementation, these factors would justify a drop to about 100 transactions per second and do not explain the extraordinary drop to 21 transactions per second. The low value is caused by the fact that the Oracle 9i database employed in this scenario always places an exclusive lock on an *entire table* when a distributed transaction updates a row[1]. This surprising behavior severely limits concurrency at the database level and is responsible for the low throughput value. With a finer-grained locking style for distributed transactions, e.g., row-level locking, transaction throughput would improve significantly.

As is the case for query transactions, all values for topology *C* are slightly higher than those for topology *B* (for the same reasons as described for query transactions). Note that, for topology *C* in combination with the replication variant, the two-phase commit protocol is coordinated by client processes[2]. In contrast to topology *B*, clients cannot transfer coordination because no middle-tier process has direct or indirect access to all databases. We can see that client coordination and the resulting longer paths for propagating prepare/commit messages have no major (adverse) impact on throughput.

7.6.3 Scenario Conclusions

In this scenario, we showed that, in most cases, the distribution of data (and thus load) among two databases leads to significantly higher transaction throughput as compared to a central database. We presented results for combinations of three different process topologies, three different data distribution variants, and two different transaction mixes (queries and updates). With our specific setup, data distribution with the partitioning variant performed best. However, that particular result cannot be generalized. Each data distribution variant has specific advantages and drawbacks. With only slight modifications to our setup or with implementation of additional features in our framework, many numbers could change significantly, e.g.:

- An appropriate secondary index would prevent full table scans in the database for each query. All query transaction throughput values would be higher with such an index. As data distribution with the replication variant suffers most from table scans, that variant would benefit most from the addition of an index.
- With other types of queries, performance of the ad hoc distribution and partitioning variants would change. For instance, queries that cover more than one domain/partition (not implemented yet) would have to be sent to multiple databases with the partitioning variant instead of only to one database. The result is more load on databases. Or, the ad hoc distribution variant would benefit from queries where individual data objects are loaded by resolving an object reference (navigational access). In contrast to range queries, which require all databases to be accessed, such queries are sent to one database only (the database that stores the corresponding data object).

[1] The Oracle database creates the exclusive table lock on XA prepare and holds it until XA commit. This behavior is independent of the selected transaction isolation level and independent of the presence of primary/secondary indices. For non-distributed transactions, exclusive locks are placed on individual rows, as expected.

[2] In practice, we would recommend coordination by those processes only if they in turn are servers for other processes. Technically, it is possible to let pure clients coordinate distributed transactions. However, usually it is better to design transactions and a process topology in such a way that coordination can always be transferred from pure clients to server machines. This reduces the risk that a coordinator fails during the critical phase of the two-phase commit protocol, does not recover, and leaves a distributed transaction in the pending state.

- With an effective query caching, where most queries are answered by intermediate objects managers, all data distribution variants would lead to similar query performance.
- Our setup was favorable to the partitioning variant because we made sure that data objects and also queries are evenly distributed among the databases. With data skew or queries that concentrate on data in one of the databases, the transaction throughput could drop to that of a single database topology or even below that value.
- When more databases are employed and clients produce much more load, all transactions that require access to all data stores become more expensive and, if executed frequently, are a severe threat to the throughput and scalability of an enterprise application. E.g., for data that is frequently queried and rarely written, the replication variant scales particularly well because each query can always be answered by a single data store. Or, when many data objects have to be inserted as fast as possible, the ad hoc distribution variant might be suited best because, with an appropriate routing function (such as our random routing), even distribution of data and thus load can be achieved.

7.7 Scenario 7: Hotspot Data

In many enterprise applications, there are at least a few data items that are frequently read and written by many concurrent clients. Access to these data items is sometimes a significant performance bottleneck, in which case they are referred to as "hotspot" data. In systems with pessimistic locking, the locking mechanism serializes concurrent transactions when they access hotspot data. As a result, concurrency is reduced or even completely eliminated.

However, the problem is even worse in many systems purely based on optimistic concurrency control, because the assumption that transaction conflicts are relatively rare does not hold any more (see Subsection 5.9.1). In the worst case, most transactions are first executed and, when they attempt to commit, aborted because of transaction conflict (and then possibly tried again). When this happens, an enterprise application is not only burdened with unnecessary load, which reduces overall transaction throughput. In addition, individual applications and/or users may effectively not be able to perform successful write accesses any more, even with several retries.

Our FPT architecture and our implementation heavily rely on optimistic concurrency control. Therefore, in principle, the problems regarding hotspot data can be observed with our implementation, too. Moreover, data objects can be cached by processes and may become stale, which magnifies the problem. However, many hotspot related problems arise only when enterprise applications use the framework's automatic check mode (see Subsection 6.5.3) to handle transaction conflicts. In many typical cases, the framework's manual check mode is a simple way to avoid hotspot related problems.

In this scenario, we first show an example of how our implementation is affected by hotspot data. We implemented a sample enterprise application for managing reservation data that is frequently accessed by concurrent clients. First, we show transaction throughput and abort rates for a conventional implementation that purely relies on the FDO framework's default check mode. Then we present results of a slightly modified, optimized implementation that makes use of the framework's advanced concurrency control features (field calls) to access hotspot data. Note that, in both cases, we do not attempt to resolve hotspots by redesigning the way reservation data is stored – instead, we merely investigate the effects of the given hotspots.

7.7.1 Setup

In this scenario, we implemented a sample enterprise application that manages reservation data. Unlike in the preceding scenarios, we did not avoid contention in this one. Instead, we explicitly selected a situation where concurrent clients frequently access the same data items, which makes them a hotspot.

The (imaginary) organization that runs the enterprise application is a company that operates ferries on several different routes, e.g., from Dublin to Liverpool. For each route, several ferries depart per day. Travelers are strongly encouraged to make a reservation for a number of seats and a specific route, day, and time. This allows the company to plan ahead (at least by a day), to assign ferries with appropriate capacity to different routes, to schedule additional ferries when demand is high, or to cancel ferries when demand is low. To give customers maximum flexibility, travelers are allowed to change their reservations (free of charge) to other days and times as often as they wish and at short notice. This can be done via Internet, phone, using the short message service of cellular phones, or at the company's offices. In this scenario, we focus on changes to existing reservations. Initial reservations, cancellations, and changes made by the company as a reaction to reservation changes are not considered here.

To prevent that ferries are overbooked or that the company's capacity planning becomes obsolete due to too many spontaneous changes to reservations, some limitations are imposed. A reservation *r1* can only be changed into a reservation *r2* when the following conditions are met: First, the ferry reserved by *r2* must have the requested number of seats left for reservations. Second, if *r1* is for a ride today and *r2* is for a ride on a later day, then the total number of today's reservations for *r1*'s route must be higher than a given limit set by the company. The second condition ensures that today's capacity allocated to a route is utilized at least to a certain degree. For instance, when a sudden change in weather occurs, only a certain number of today's travelers would be allowed to change their reservation (free of charge) from today to tomorrow. However, in typical situations, reservation changes are possible. Many customers rely on that and change their plans frequently and spontaneously.

Figure 7-17 displays a UML class diagram of all data object types involved in a reservation: A *Customer* can make any number of *Reservations*. A *Reservation* is valid for a given number of seats on a specific *Ride*. Each *Ride* is carried out by a specific *Ferry*, on a given day and on a specific *Route*. Instead of associating *Rides* and *Routes* directly with each other, a *RouteDay* class is put in between. The *RouteDay* contains attributes for storing (a) the total number of reservations for that route and day and (b) the minimum number of reservations acceptable to the company. The number of reservations for specific rides is stored in the attribute *reservedSeats* of *Ride* and is an aggregation of all seat values of associated *Reservations*. The total number of reservations in *RouteDay* is in turn an aggregation of all *reservedSeats* values of all associated *Rides*.

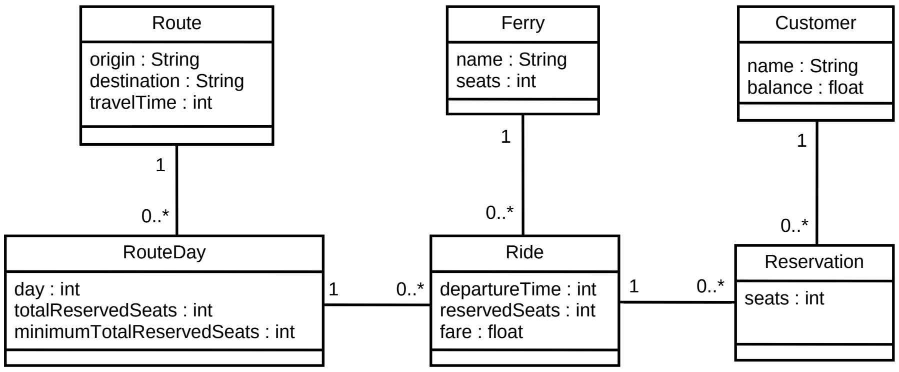

Figure 7-17. Data object types involved in a reservation. *Ride* and *RouteDay* data objects represent hotspots.

To change a reservation *r* from *oldRide* to *newRide*, first the following two tests are performed:

(1) It is tested whether *newRide.reservedSeats* + *r.seats* ≤ *newRide.ferry.seats*.

(2) If *oldRide.routeDay.day* is today and *newRide.routeDay.day* is a later day, then it is tested whether *oldRide.routeDay.totalReservedSeats* – *r.seats* ≥ *oldRide.routeDay.minimumTotalReservedSeats*.

If all tests are successful, then the following actions are performed:

- In *r*, the *ride* reference is changed from *oldRide* to *newRide*.
- *oldRide.reservedSeats* is decremented by *r.seats* and *newRide.reservedSeats* is incremented by *r.seats.*
- If *oldRide.routeDay.day* ≠ *newRide.routeDay.day*, then *oldRide.routeDay.totalReservedSeats* is decremented by *r.seats* and *newRide.routeDay.totalReservedSeats* is incremented by *r.seats.*

All tests and updates described above are performed by clients and within a single FPT transaction for each reservation change. We can see that a successful reservation change transaction requires write access to five data objects for an inter-day change and to three data objects for an intra-day change. The *Ride* and *RouteDay* data objects represent hotspots because they contain aggregated data that is frequently changed.

We implemented an enterprise application and populated the database with sample data for one route, three route days, 12 rides, 5 ferries, 2145 reservations, and 2145 customers. Reservations are for one to six seats, equally distributed. Data for the route, route days, rides, and ferries is given (in tabular form) in Figure 7-18. For better readability, object identifiers and object references are shown in a simplified way.

Route

oid	origin	destination	travelTime
0	Dublin	Liverpool	730

Ferry

oid	name	seats
300	MS Atlas	1500
301	MS Bingo	1000
302	MS Chronos	750
303	MS Desmona	750
304	MS Eire	250

RouteDay

oid	day	totalReser-vedSeats	minimumTotal-ReservedSeats	route (ref)
100	0	2684	2500	0
101	1	2354	1875	0
102	2	2462	1500	0

Ride

oid	departureTime	reservedSeats	fare	routeDay (ref)	ferry (ref)
200	430	268	100	100	300
201	930	678	120	100	302
202	1215	82	80	100	303
203	1645	510	99.95	100	301
204	2330	1146	130	100	300
205	415	1142	110	101	300
206	1100	430	105	101	303
207	2330	782	110	101	300
208	420	899	160	102	301
209	1345	155	105	102	304
210	1715	570	140	102	303
211	2335	838	90	102	301

Figure 7-18. Excerpt of data used in Scenario 7.

The first *RouteDay* entry (with *day*=0), represents the current day. The *minimumTotalReservedSeats* attribute value of each route day is set to 50 percent of the total seat capacity of the corresponding day and route.

The same process topology and machines as in Scenario 2 are used here. Up to ten clients (running on up to four machines) concurrently send reservation change transactions to a central middle-tier process[1], which is connected to an Oracle database. There is no "think time" between transactions. For communication between clients and the middle-tier process, our *SocketConnection* protocol plug-in is employed. Each of the clients is responsible for a subset of the 7,500 reservation instances. To provoke many conflicting transactions, the central middle-tier server periodically selects and announces a random

[1] Note that neither the standard nor the optimized implementation variant requires a central server process for accessing hotspot data. Both variants run in any process topology. Also, our *Free Data Objects* framework does not employ any optimizations that depend on the number of processes directly accessing hotspot data in data stores (such as bundling writes of different FPT transactions into a single data store transaction or serializing conflicting transactions above the data store level).

(existing) ride. Then each client attempts to change all its reservations (in random order) to that ride. Once a client detects that the ride is overbooked, i.e., test (1) fails, it stops and waits for the next announcement of the server. When test (2) fails for a reservation change, a client just proceeds with the next reservation change. As soon as the central middle-tier server detects that the announced ride is nearly overbooked (ten or less seats left), it announces another random ride and all clients start to change reservations again.

We implemented two variants of application clients:

- The *standard implementation variant* relies purely on the FDO framework's default check mode to handle transaction conflicts. The default check mode is most convenient for application developers because it transparently handles all checks in the background. However, the default check mode generates version checks for updated data objects, which is relatively conservative. To minimize the number of transaction aborts due to stale cache entries, we added code that explicitly reloads relevant hotspot data objects immediately before they are read and updated.
- The *optimized implementation variant* utilizes the FDO framework's advanced concurrency control mechanisms to access hotspot data. The clients update hotspot data objects by using delta writes (see Subsection 6.5.2) and explicitly associate predicate checks with them. As explained in Subsection 6.5.3, this combination corresponds to *field calls*, which are less conservative than version checks and allow for much more concurrency. With field calls, it is not essential for transactions to always work with the most current version of a data object. Therefore, hotspot data objects are not explicitly reloaded as is the case in the standard implementation variant. For hotspot data objects accessed to perform tests (1) and (2), a freshness requirement of one second is defined, which means that, typically, these data objects can be read from a client's cache and are (transparently) reloaded at most once per second.

In both variants, clients perform tests (1) and (2) on possibly stale data because of the FPT architecture's optimistic approach to concurrency control. In rare situations, this may cause a client to decide that a reservation cannot be changed although, a fraction of a second before, the database state has changed and the reservation change would be allowed. Generally, this is acceptable for a reservation system. On the basis of stale data, a client may also decide that a reservation change is allowed although it is not any more. However, on commit, the framework's concurrency control always detects a transaction conflict in that case (for both implementation variants; through version and/or predicate checks) and aborts the transaction.

7.7.2 Results for the Standard Implementation Variant

In Figure 7-19, transaction throughput rates of reservation change transactions are displayed. All numbers are averaged over 20,000 transactions. The diagram shows three curves: one curve for aborted update transactions, one for successful update transactions, and one for total update transactions (which represents the sum of the first two rates). Transactions that are rolled back at clients because test (1) or test (2) indicated that the reservation change is not allowed (see previous subsection) are not counted in the results.

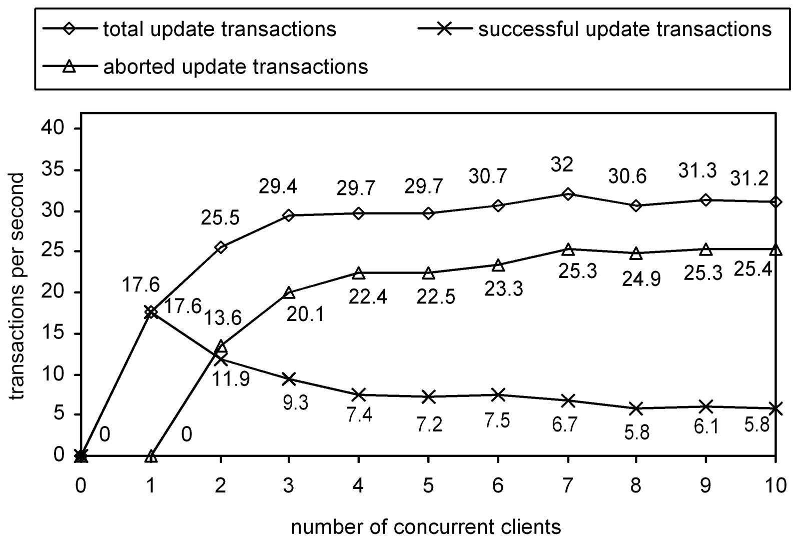

Figure 7-19. Throughput of update transactions in the presence of hotspot data – standard implementation variant.

Since reservation change transactions access hotspot data in the database and contain multiple operations, the total throughput is significantly lower than the throughput rates observed in Scenario 2. For one client, we observe a transaction throughput of 17 to 18 successful update transactions per second. Since there is no concurrency, no update transactions are aborted. With two concurrent clients, the total transaction throughput increases. But since many transactions are aborted, the number of successful transactions actually *decreases* to about 12 per second. That trend continues as more and more concurrent clients access hotspot data – with ten clients, more than eighty percent of all update transactions are aborted and the rate of successful update transaction drops to about six per second.

7.7.3 Results for the Optimized Implementation Variant

Figure 7-20 shows results for the optimized implementation variant. We can see that transaction abort rates are low, compared to the standard implementation variant. Transaction aborts increase along with the number of concurrent clients, but, even with ten clients, only a small fraction of all transactions are aborted. We can also see that, for any number of clients, the total transaction throughput is slightly higher than with the standard implementation variant. This is due to the fact that the standard implementation variant reloads more data objects than the optimized variant.

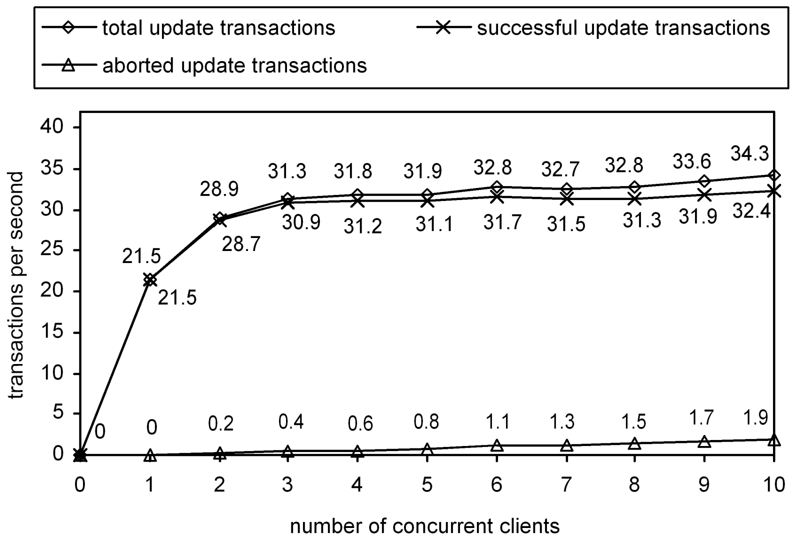

Figure 7-20. Throughput of update transactions in the presence of hotspot data – optimized implementation variant.

7.7.4 Scenario Conclusions

In this scenario, we showed that, by using the advanced concurrency control features of our *Free Data Objects* framework, even hotspot data can be accessed efficiently. We implemented two variants of an enterprise application that manages reservations and then analyzed transaction throughput and abort rates. The first implementation variant relies on the default check mode of our *Free Data Objects* framework. As expected for a framework based on optimistic concurrency control, that variant suffers from high transaction abort rates. In fact, the highest throughput of successful update transactions is achieved for one client, i.e., with no concurrency at all. The second implementation variant uses the framework's advanced concurrency control features to optimize access to hotspot data objects. A combination of delta writes and predicate checks is used that provides the functionality of field calls at the framework level. With that approach, transaction abort rates remain low when accessing hotspot data. Note that we did not attempt to *resolve* the hotspots in this scenario. Instead, we used optimizations to reduce (and nearly eliminate) the additional prohibitively negative impact of hotspots *at the framework level*. In the database, the hotspots remain.

In contrast to the default check mode, which is completely transparent and thus convenient for application developers, the use of delta writes and predicate checks is not transparent. But since only a few lines of application code have to be modified (those that access hotspot data), this will be acceptable in most cases.

Note that the particular optimizations we employed in the second implementation variant are not suitable for all kinds of hotspot data. However, they are especially suited for data objects that become hotspots because they represent some type of counter or quantity that is frequently increased or decreased (e.g., a counter for items in a warehouse or a balance value) or aggregated values. In practice, many typical hotspot data items belong to this class. There are other hotspots situations that can also be efficiently handled. For instance, for frequent unconditional writes to the same data objects, applying weaker existence checks instead of version checks can be a significant advantage. However, those hotspots, for which using existence or predicate checks is not an option, have to be addressed at the design level of an enterprise application.

7.8 Summary

In this chapter, we evaluated the FPT architecture and our proof-of-concept implementation. Using seven scenarios, we evaluated various aspects and demonstrated the effects of different process topology patterns. The first scenario focused on support for custom and adaptable process topologies. We implemented an enterprise application based on the case study presented in Chapter 4 – including all evolutionary steps described in the case study. Thereby, we demonstrated that complex custom topologies are well supported, straightforward to construct (through configuration), and easy to adapt to changes in requirements (through reconfiguration and without code changes) with our framework.

The next six scenarios we presented focused primarily on performance aspects. We showed that our solution introduces relatively little overhead and provides good performance. In particular, we analyzed transaction throughput for fine-grained and coarse-grained transactions, and showed that, in a simple standard topology, a single server can handle up to several hundreds of concurrent clients. Then we demonstrated that introducing additional intermediate tiers into a process topology is acceptable in terms of latency and has no major impact on transaction throughput. We also showed the performance advantages of coarse-grained remote access, which is typical of our approach, over fine-grained remote access, which is encouraged by object middleware. In another scenario, we showed how distributing load among multiple replicated processes significantly improved an enterprise application’s transaction throughput. Furthermore, we compared performance of different variants of distributing data (and, consequently, load) among multiple data stores. Finally, we showed that hotspot data can lead to many transaction aborts with our optimistic approach to concurrency control, but advanced concurrency control features of our framework (delta writes and predicate checks) help to overcome that problem.

Note that in most scenarios presented in this chapter we analyzed relatively simple process topologies. In particular, five of the seven scenarios are based on three-tier topologies. These topologies were sufficient to demonstrate important aspects of our framework and the effects of individual process topology patterns. However, a systematic evaluation of complex, large-scale process topologies (which would require either a dedicated testbed or a simulation-based approach) is future work – see Section 8.2.