Chapter 4: Custom and Adaptable Process Topologies

In the previous chapter, we described the distributed architecture of multi-tiered enterprise applications. Among other things, we defined process topologies (see Subsection 3.2.2), discussed the design of process topologies (see Section 3.3), and presented a language of topology patterns for constructing process topologies (see Section 3.4). In this chapter, we first motivate that, for many enterprise applications, simple standard process topologies are not sufficient. We explain that, instead, many demanding enterprise applications need *custom* and *adaptable* process topologies. To illustrate the need and to give an example, we present a detailed case study. The case study describes a tiny system that develops into a large, complex, multi-tiered enterprise application. During its life cycle, its process topology is constantly subject to change to meet new requirements. After presenting the case study, we explain what *cross-process management of data objects* is and why it is important for custom process topologies. Then we identify several key requirements for enterprise application middleware to support custom and adaptable process topologies. However, we also show that existing enterprise application middleware does not fulfill all these requirements. Therefore, it is still hard today to build enterprise applications with custom and adaptable process topologies on top of existing middleware.

This chapter is structured as follows: In Section 4.1, we motivate the need for custom and adaptable process topologies. Section 4.2 presents a case study that illustrates the need for custom and adaptable process topologies. Section 4.3 explains cross-process management of data objects. In Section 4.4, six key requirements for enterprise application middleware are identified and discussed. Section 4.5 explains limitations of existing middleware and their consequences for application developers. A few selected questions are discussed in more detail in Section 4.6. Finally, in Section 4.7, we provide a brief summary of this chapter.

4.1 Motivation for Custom and Adaptable Process Topologies

In Section 3.3, we discussed the design of process topologies: We explained that, traditionally, the design of a suitable process topology for an enterprise application is a task of a software architect. The software architect decides on distribution and other high-level design details early in the development process. We outlined the most important driving forces that have to be taken into account when designing process topologies – e.g., performance, fault tolerance, security, and costs. We also explained why, in general, there are no precise rules for constructing *complete* adequate process topologies for a given set of requirements; among other things, a reasonable trade-off between a large number of conflicting and heavily application-dependent factors has to be found. Finding such a trade-off in the context of a specific project is a complex task that cannot easily be formalized. However, in Section 3.4, we presented a pattern language that addresses typical design problems in *parts* of process topologies.

In the following subsections 4.1.1 and 4.1.2, we motivate the need for custom and adaptable process topologies, respectively.

4.1.1 Custom Process Topologies

For small enterprise applications, simple standard topologies, such as shown in Figure 3-1 and Figure 3-2 in Subsection 3.2.3, are often sufficient. That is especially the case for enterprise applications that have only few users, are not mission-critical, do not experience high load situations, and run in local area networks. However, large, mission-critical enterprise applications are typically much more demanding –

building such applications on top of simple standard topologies would prevent the applications from fully making use of their capabilities.

As already discussed in Section 3.3, different requirements may lead to different process topologies. We call such process topologies *custom* process topologies because they are designed to meet individual requirements of their respective applications and users, i.e., are heavily application-dependent. An example of a custom (non-standard) process topology has been shown in Figure 3-4.

How should a software architect construct a custom process topology? Ideally, he starts with a simple standard topology (e.g., a two-tier or a three-tier structure) and then successively applies process topology patterns to parts of the process topology. Each pattern changes a part of the process topology. Analogous to design patterns, which can be freely combined to form complex, object-oriented designs, topology patterns can be used to form custom process topologies. Step by step, a standard topology can be transformed until it meets the specific requirements of an application.

However, it should be noted that not all topology changes necessarily correspond to topology patterns as defined in Section 3.4. An example is the addition of a new client process that implements a graphical user interface (GUI) to a process topology. Although that action changes the topology (and the definition of the *process replication* pattern in Subsection 3.4.2 could be modified to include that change), we would not consider such a change a software pattern because it does not encapsulate a significant portion of design knowledge (see Subsection 3.5.4).

Custom process topologies can either be designed from scratch (a sequence of simple changes and/or topology patterns is applied to construct an initial process topology) or be the result of system evolution (from time to time, simple changes and/or topology patterns are applied to an existing enterprise application).

4.1.2 Adaptable Process Topologies

A typical custom process topology is heavily application-dependent because the basis for its design is a trade-off between many application-dependent factors (see Section 3.3). However, during the life-cycle of an enterprise application, one or more of these factors are likely to change. For example, a change to the user interface could introduce new access patterns and thus change transaction load. Or, due to its success, a tiny application may evolve into a mission-critical application and require more availability. When relevant factors change, a software architect has to reconsider the trade-off that led to the current process topology and, if necessary, adapt the process topology to better meet the new requirements.

Ideally, a process topology is *adaptable*, i.e., the topology can be changed without having to re-design and/or re-implement large parts of the corresponding enterprise application. Typically, re-design and re-implementation are not only costly but also time-consuming. Therefore, adaptable process topologies are particularly important for organizations that have to quickly react and adapt to changes. (However, in Section 4.5, we will see that enterprise applications with adaptable process topologies are difficult to realize today.)

4.2 Case Study

In this section, we present a case study with an imaginary – but nevertheless realistic – scenario that illustrates the need for custom and adaptable process topologies.

Let us consider an enterprise application that implements a marketplace for accommodations. The enterprise application is developed and is run by a travel agency. Customers of the agency can make, update, withdraw, and query accommodation offers. Also, they can view the location of particular accommodations on a map and make reservations. For the travel agency, a comfortable, sophisticated user

interface is important. Therefore, they have decided to use rich (fat) clients instead of thin HTML clients (web browsers).

We will see that the application starts as a tiny system. Then, step by step (as shown in Figure 4-1), the application grows and develops into a system with a complex custom process topology. At each step, requirements change and the process topology is adapted accordingly:

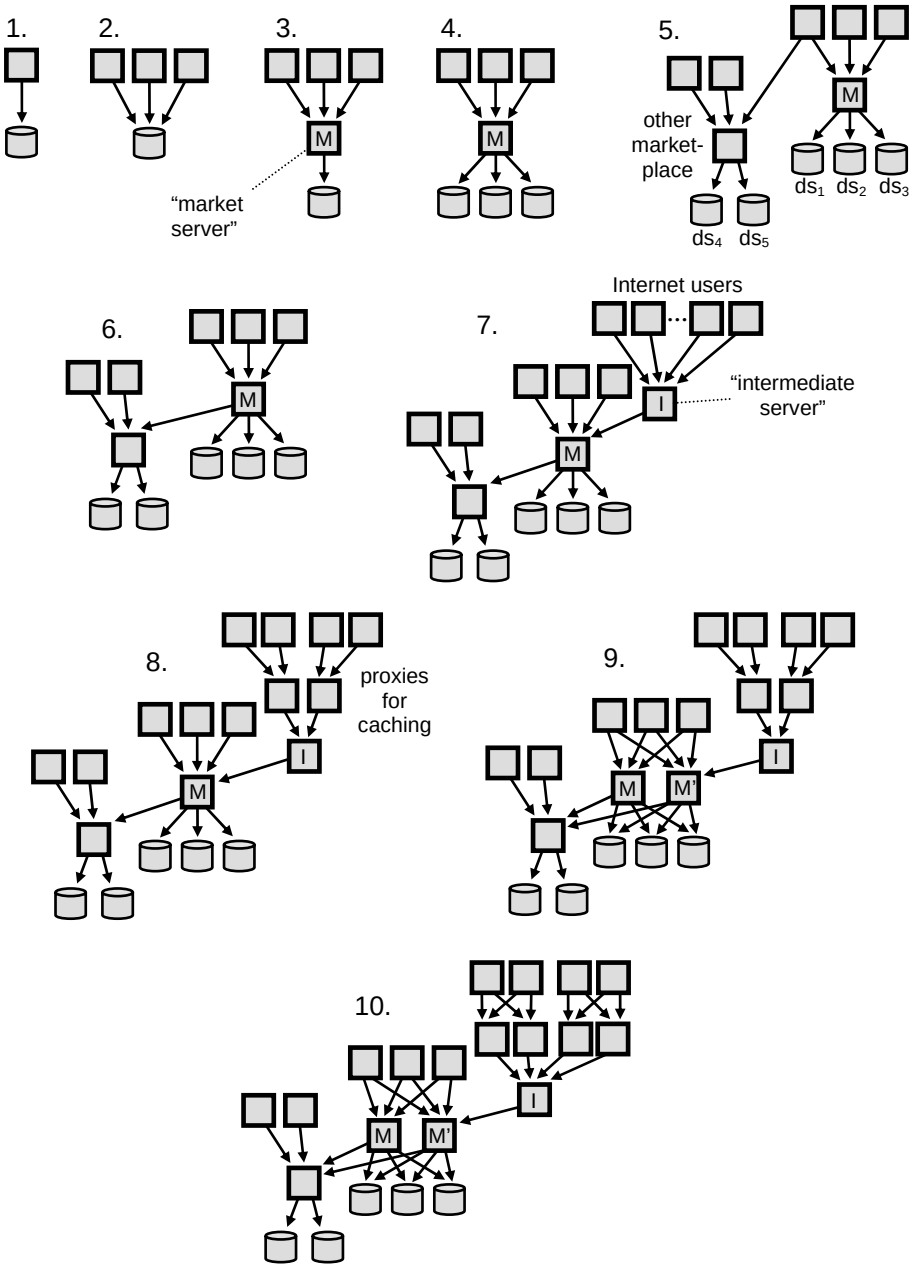

Figure 4-1. Evolution of an enterprise application's process topology.

Step 1: Initial Two-Node Structure

The initial enterprise application is designed with a minimal process topology that simply consists of two nodes: a client process (a rich client with a GUI) and a transactional data store. The data store persistently stores all accommodation offers and reservations and is directly accessed by the client. The client runs on a personal computer and is operated by a single travel agent employed by the travel agency. Customers of

the travel agency that are interested in accommodations do not have direct access to the enterprise application. Instead, they have to contact the travel agent (face-to-face or via fax/email/phone).

Step 2: Multiple Clients

The customer base grows and more travel agents are hired so that multiple customers can be served simultaneously. To allow multi-user access, more client processes are added to the process topology. Each agent is assigned his own personal computer that runs one of the client processes.

Step 3: Three-Tier Topology

The software architect of the enterprise application decides that the current two-tier structure is not flexible and scalable enough. He introduces a three-tier architecture to decouple clients from the central data store and thus to improve modularity and scalability. The software architect applies the *wrapper insertion* pattern (see Subsection 3.4.1) to insert a new process *M* ("market server") between the clients and the central data store.

Step 4: Data Distribution

With a growing number of concurrent users (agents from several call centers at this stage), higher transaction throughput is required. The software architect identifies the central data store as the main bottleneck. He decides to apply the *data distribution* pattern (see Subsection 3.4.4) and inserts additional data stores to the process topology. More specifically, he selects variant (b) of the pattern (i.e., distribution based on a static partitioning criterion) to distribute data objects among the data stores. With the pattern applied, requests – and thus load – are distributed horizontally among the data stores. As a result, the enterprise application's transaction throughput improves significantly.

Step 5: Cooperation with another Marketplace

The travel agency starts to cooperate with another enterprise that runs a similar marketplace. At first, the cooperation is loose – the other enterprise agrees that one of the travel agency's clients can directly access its marketplace (read-only). The software architect orders a developer to prototypically extend one of the rich clients such that the client can access data from both marketplaces. Letting the client access two different systems corresponds to application of variant (a) of the *integration of subsystems* pattern (see Subsection 3.4.6).

Step 6: Integration of the other Marketplace

The other marketplace is more tightly integrated into the travel agency's marketplace: All clients of the travel agency are granted full read and write access to both systems. In principle, that change could be realized by extending all client processes as described in Step 5. However, the software architect decides that managing access to different data sources is a server-side issue and thus should not be implemented as part of client processes. Instead, the changes made in Step 5 are undone, the implementation of the travel agency's middle tier process is extended, and a C1 connector from market server *M* to the other enterprise application is added. Adding the C1 connector again corresponds to variant (a) of the *integration of subsystems* pattern.

Step 7: Access from the Internet

To further expand the travel agency's customer base, customers are allowed to directly access the enterprise application via the Internet (without having to contact a travel agent). The travel agency develops a Java applet that runs inside the customers' web browsers and communicates with the enterprise application. Via the applet, Internet users are offered a subset of the functionality available to travel agents. For instance, unlike agents, Internet users are not allowed to perform system management functions, update maps, or query revenue data. For security reasons, Internet users are never allowed to directly access the enterprise application's central market server process *M*. Instead, they access the

system through a dedicated new intermediate server *I*. The intermediate server evaluates client requests and rejects all requests that go beyond the Internet clients' reduced functionality. It also rejects requests that would produce too much load, e.g., querying all accommodation offers without specifying additional criteria. The process topology is changed as follows: An arbitrary number of new client processes (web browsers with Java applets) are added to the process topology. For each new client, a C1 connector from the client to *M* is added. Then the *wrapper insertion* pattern is applied to insert the intermediate server *I* between the new clients and *M*.

Step 8: Proxies for Internet Clients

The number of Internet clients increases with growing popularity of the marketplace. Soon, both intermediate server *I* and central market server process *M* become overloaded. A thorough analysis shows that the set of data objects accessed by a typical Internet client is closely correlated with the client's geographical location (e.g., France, Scandinavia, North America). In order to better handle the high load produced by Internet users, the software architect decides to insert a set of proxy processes between the Internet clients and intermediate server *I*. Each proxy process serves clients from a specific geographical region and caches data objects frequently accessed by its clients. In this way, many requests can be answered by proxy processes without having to contact processes *I* and *M*. The process topology is adapted as follows: First, the *wrapper insertion* pattern is applied to insert a single proxy process between Internet clients and *I*. Then the *process replication* pattern (see Subsection 3.4.2) is used to replicate the proxy process. As a result, load is first shifted from *I* and *M* to the proxies and then horizontally distributed among the proxies.

Step 9: Replication of the Central Market Server Process

Although the actions taken in Step 8 take much load away from central market server process *M*, that process still remains a bottleneck: During peak Internet usage times, not only Internet users but also all travel agents experience bad response times. Additionally, the travel agency becomes aware that process *M* is a single point of failure since all clients of the travel agency finally depend on *M*. The enterprise application has developed into a mission-critical system by that time and generates a significant part of the travel agency's revenues. Therefore, a single point of failure is not acceptable any more. To address the problems, the software architect replicates *M*. He applies the *process replication* pattern to add a replica *M'* of *M* to the process topology and deploy *M'* on a dedicated new server machine. While clients operated by travel agents are allowed to access both *M* and *M'* (see the *meshing* pattern in Subsection 3.4.3), requests of Internet users are always routed via *M'*. As a result, the enterprise application can provide basic fault tolerance and better response times for the travel agents: First, when either *M* or *M'* fails, the remaining replica keeps the system functional. Second, requests from travel agents are distributed among *M* and *M'* (load balancing). Third, travel agents are less affected by heavy load from Internet users because *M* is not affected by Internet load.

Step 10: Replication of Proxy Processes

Each month, more Internet users access the marketplace and thus put more load on the proxy processes responsible for specific geographical regions. One option to handle the growing load would be to add more proxy processes and to reduce the size of the geographic region handled by each proxy. That would correspond to changing (or: undoing and then re-applying) the *process replication* pattern applied in Step 8. However, since there are also availability issues related to the proxy processes (which are hosted by another company), the software architect selects a different solution: For each existing proxy process, the *process replication* pattern is applied in combination with the *meshing* pattern (see Subsection 3.4.3). The Internet clients of each specific geographical region are now served by a *pair* of proxy processes. Load produced by Internet clients of a region is horizontally distributed among the region's two proxy processes. Since each Internet client needs only one operational proxy process but can access two, this also improves availability.

The case study above describes an enterprise application with a simple process topology that evolves into a complex *custom* topology. We can see that a simple standard topology clearly would not address the travel agency's requirements in an appropriate way. We can also see that the final process topology layout (see Step 10 in Figure 4-1) has not been designed from scratch – instead, it is the result of an iterative process that consists of many evolutionary steps. In each step, parts of the process topology are changed to adapt the application to new requirements. In scenarios similar to the one presented here, *adaptable* process topologies are of particular importance because they significantly simplify topology changes.

4.3 Challenge: Cross-Process Management of Data Objects

Up to this point, enterprise applications and their process topologies have been discussed on a relatively high architectural level. Having motivated the need for custom and adaptable process topologies in the previous section, we now approach the question of how such systems can and should be realized. As a prerequisite, this section describes what *cross-process management of data objects* is and why it plays an important role for custom process topologies.

From a traditional service-oriented perspective, realizing enterprise applications with custom topologies seems straightforward: Each process in a topology offers services to its client processes. Client processes consume these services by invoking functions on server processes using, for instance, remote procedure calls, web service requests, or other messaging mechanisms. A consumer/client of a service can in turn be a server for other processes, which, in principle, allows one to construct arbitrarily complex process topologies.

However, this perspective tends to ignore the data management dimension of the problem, which is often one of the most important issues to be addressed. In object-oriented, data-intensive enterprise applications, services are centered around transactional data objects. Usually, these objects play an important role for *both* the provider and the consumer of a service. For example, in an object-oriented enterprise application that manages customer data and reservation data, both the provider and the consumers of a service require access to *Customer* and *Reservation* data objects. As part of its service, a server usually has to expose data objects *across process boundaries* to its clients. In that case, we say that a server *offers* clients access to a given set of data objects.

For such services, a mechanism for cross-process management of data objects is needed for efficient, transactional, distributed, and shared access to data objects. The mechanism is part of the implementation of C1 connectors and can either be provided by enterprise application middleware (which means more comfort and more transparency) or be part of the application code itself (less comfort and less transparency). Cross-process management of data objects plays an important role in the requirements we present in the subsequent section.

4.4 Requirements for Enterprise Application Middleware

In this section, we identify and discuss key requirements for enterprise application middleware to support custom and adaptable process topologies. We specifically address data management for data-intensive, object-oriented enterprise applications.

Traditionally, aspects like distribution and data access are mostly handled by middleware (see Section 2.4): Application code is developed by application developers and primarily focuses on business logic and/or user interface. For issues that involve distribution and transactional data access, application code typically relies on functions provided by the underlying middleware. Therefore, especially the capabilities of the underlying middleware determine to what extent custom process topologies can be realized and how adaptable these topologies are.

We assume that application code, regardless of its location in the process topology, requires object-oriented access to transactional data objects. Typically, that includes (but is not limited to) transparent navigational access, support for object identity, and type-safety. In principle, enterprise applications can be realized without such an object view on data. For example, an enterprise application implemented in Java may rely on low-level access to a relational database and process data through explicit SQL statements and tabular JDBC result sets. But in practice, an object view is considered very important in many projects – especially when object-oriented analysis (OOA), object-oriented design (OOD), object-oriented programming languages, and object-oriented tools are used. Typically, an object view on data (at least for the business logic) is much more intuitive and provides a higher level of abstraction than low-level access. In fact, often, one of the main reasons for using a middleware (e.g., an object/relational mapping framework or an application server that supports Enterprise JavaBeans) is to realize an object view on transactional data.

In the following subsections, we identify six fundamental requirements (R1 to R6) for enterprise application middleware to support custom and adaptable process topologies. Requirement R1 addresses efficiency, R2 addresses consistency, R3 and R5 address custom topologies, and R4 to R6 address adaptable topologies.

4.4.1 Copying Data Objects across Process Boundaries (Requirement R1)

A service (offered by a server process) usually has data objects associated with it. For instance, a service for managing accommodations might heavily rely on data objects of type *Customer*, *Accommodation*, and *Reservation*. When a client wants to access data objects associated with a service (see cross-process management of data objects in Section 4.3), there are basically three alternatives for handling these data objects:

(1) *Remote data objects.*

Data objects physically remain in the server process and are remotely accessed by the client. This alternative is usually realized by using object middleware such as CORBA or RMI to represent data objects as middleware-level remote objects that are directly accessed by clients. For example, a *Customer* object could be represented as an entity bean using Enterprise JavaBeans or as a CORBA transactional object. While this approach has several advantages, it also leads to significant performance and scalability problems: In many applications, data objects and object-oriented access to them are relatively fine-grained. Remote client access to data objects typically requires a large number of network roundtrips – in the worst case one request per access to an attribute of a remote data object. For example, a client that simply displays a list of customers including all their attribute values could easily produce hundreds of remote invocations just for displaying a single view and hence create an immense overhead. Each invocation requires several actions, such as marshalling of parameters, network communication, encryption, security checks, unmarshalling of parameters, request dispatching, establishing a transaction context, and processing the invoked code. Even with a relatively small number of clients, this approach often results in high bandwidth consumption, high server load, and bad response time. Later, in Section 7.4, we will demonstrate and analyze the adverse effects of fine-grained access to remote data objects for one particular application scenario.

In many cases, an object middleware approach is adequate for coarse- and medium-grained objects/communication only and is less suited for fine-grained objects/communication [WWWK94] [CD96]. For example, benchmark results presented in [CMZ02] show that, with EJB entity beans, only a fraction of the throughput of a solution with less fine-grained remote access can be achieved. The problem is also described in an increasing number of publications that address practitioners, e.g., [SSJ02], [FRF+02], and [Tate02]), and is also mentioned in the

Enterprise JavaBeans specification [Sun01]. For performance reasons, these publications strongly recommend avoiding direct client access to remote objects for fine-grained access patterns.

(2) *Data objects are moved.*

Data objects can be moved (i.e., migrated) from the server process to the client's address space. Systems that support object migration are, for example, Emerald [JLHB88] and COOL [LJP93]. However, data objects are shared objects and often have to be accessed concurrently by different clients. Moving a data object away from a server to one of its clients complicates, delays, or even prevents other clients from accessing it. Therefore, in enterprise applications, it is usually preferred to copy data objects between clients and servers (see below) instead of moving them. Copying also allows for caching at both the client and the server.

(3) *Data objects are copied.*

With the third alternative, data objects (or at least their state) are copied from the server into the client's address space. Depending on the cross-process data management mechanism, only individual data objects or whole collections of data objects (e.g., a query result) can be copied into the client process. Repeated accesses to the same data objects can then be satisfied by the local client cache, thus avoiding a too fine-grained remote communication style. The degree of caching on the client side may range from simple short-term buffering to a full-fledged object-caching solution.

Connections from clients to transactional data stores, e.g., JDBC or ODBC, are typically implemented using the copy approach. While relational database management systems copy query results as flat, tabular data (i.e., not as data objects) to a client process, object database management systems typically copy complete objects.

For copying a data object from an intermediate server process to a client process, there are many possibilities. For example, application code at the server may convert a data object's state into a string or array of bytes and transmit it to a client, where it is converted back to an object again. Today, middleware products typically have basic built-in support for such tasks. For instance, data objects can be copied as SOAP XML messages, with the help of Java serialization [Sun03b], or as CORBA *value type* objects.

Another alternative would be to use mobile code [FPV98] [CHK97]. In principle, mobile code can be sent from a client process to a server process to locally access data objects at the server side. However, as the mobile code approach limits the choice of programming languages, raises new security and interoperability problems, and, in general, has not been accepted in the domain of enterprise applications yet, we do not consider mobile code an option here.

Because of the efficiency problems of the remote data objects approach and the need for shared access, we require that middleware supports a way to *copy* data objects across process boundaries. Not only copying from server processes to client processes must be supported, but also the reverse direction: When clients obtain copies from a server and are allowed to perform updates on them, the updated objects (or updates) must eventually be sent back to the server.

4.4.2 Preserving Object Identity (Requirement R2)

One of the most fundamental object-oriented concepts is the concept of *object identity* [KC86]. With object identity, the existence of an object does not depend on its value. Equality (same object values) is distinguished from identity (same objects). Object identity can be realized, for instance, by assigning unique object identifier values to objects or by associating identity with a physical location in memory.

Object identity also applies to objects with persistent state (i.e., data objects); for example, it has been defined as a mandatory feature in the object-oriented database system manifesto [ABD+89].

In object-oriented systems, there can be multiple (different) references to a single data object. Adequate support for object identity implies that, for read and write operations, such references are interchangeable, which means that it is important which *objects* are accessed but not which *references* are used for access. However, when data objects are copied across process boundaries (see requirement R1), special care must be taken to provide client applications with a consistent object view. Relying solely on low-level mechanisms for copying data objects – such as Java serialization or CORBA value objects – leads to a problem: Multiple requests can produce multiple copies for the *same* entity in a client process. As a result, transaction semantics are violated, e.g., when a transaction updates a copy and subsequently reads a second (now inconsistent) copy representing the same entity. Even existing higher-level mechanisms for copying data – such as ADO.NET [Scep02] and Java RowSets [Sun03c] – do not preserve object identity. Without middleware support for object identity, that problem forces application developers to pay close attention to avoid inconsistencies and also prevents an effective mid- and long-term caching of data objects. Furthermore, when an application processes data from multiple (and possibly overlapping) data sources, application-level object identity management may become highly complex.

To preserve consistency, we require that middleware supports object identity. To adequately support object identity in the presence of copies (see requirement R1), a data management mechanism has to ensure that application code "sees" only a single, consistent state for a data object – independent of the reference(s) used for accessing the data object and independent of the remote request that created a copy in an address space.

4.4.3 Transitive Data Management (Requirement R3)

In custom process topologies, it cannot be assumed that each process that processes data objects has direct access to all data stores that persistently store those data objects. Instead, indirect access via other processes is often required. Indirect access implies that there are one or more intermediate processes that are both client and server. Indirect access has implications for cross-process management of data objects (see Section 4.3). To support indirect access, we require that a data management mechanism has to be *transitive*, i.e., has to support arbitrary chains of client/server relationships. This includes, e.g., copying, caching, and synchronizing data objects along these chains.

More formally, we define transitivity as follows: Given a process topology that contains at least a process *A*, a process *B*, and a server (process or data store) *C*. Also given a path of one or more C1 connectors from *A* to *B* and a path of one or more C1 connectors from *B* to *C*. Let us assume that *B* offers *A* access to a set *S* of data objects through a mechanism *M* for cross-process management of data objects. Similarly, *C* offers *B* access to *S* through the same mechanism *M*. We call *M transitive* if it always follows that *C* implicitly offers *A* access to *S* through *M*. The idea of transitive data management is illustrated in Figure 4-2.

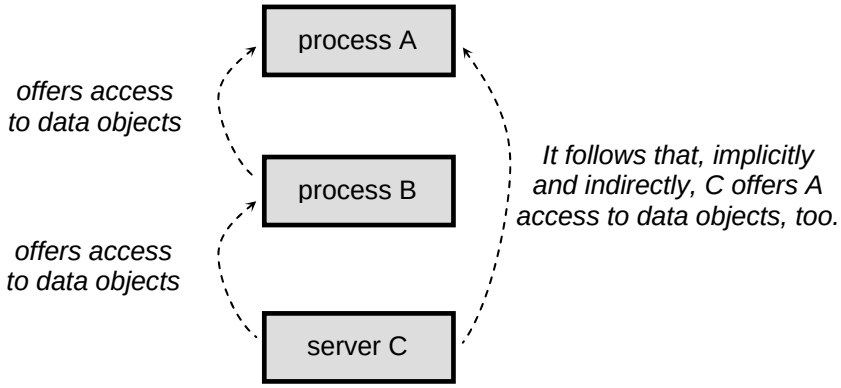

Figure 4-2. The idea of transitive data management.

4.4.4 Limited Visibility (Requirement R4)

For each process, only directly connected clients and servers should be visible. Each process is a data source for its client processes, provides them with a consistent, integrated view of data, and hides the fact that some or all data may in turn originate from other, underlying data sources. A process should neither make assumptions about processes and data stores not directly connected to it nor rely on a specific topology, e.g., on the (non)existence of meshes (which are alternative paths to a node, see Subsection 3.4.3).

This requirement ensures modularity and helps to minimize the impact of changes: Topologies are much easier to adapt because changes primarily affect only the parts that have been modified. Figure 4-3 shows an example process topology and the processes/data stores that are visible to process *P*. When, for instance, *P* is replicated or removed, only those processes to which *P* is visible are directly affected. All other processes are shielded from the change as long as the directly affected processes find a way to keep their service functional.

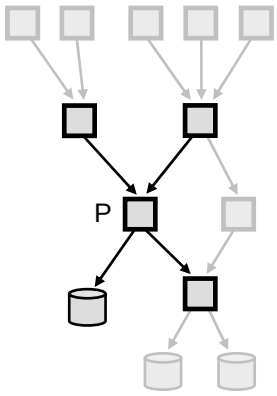

Figure 4-3. Limited visibility: Only directly connected processes and data stores are visible to a process.

4.4.5 Decoupling of Topology, Data Distribution Scheme, and Application (Requirement R5)

In many existing enterprise applications, the following three aspects are tightly coupled:

(1) the process topology,

(2) the data distribution scheme, which defines how data objects are distributed among multiple data stores (see 3.4.4), and

(3) the application code.

Typical reasons for pairwise coupling are:

- (1) ⇔ (2): A given data distribution scheme requires a specific process topology. For example, if a set of objects is replicated among several data stores, a central process with direct access to all these data stores could be required – instead of allowing topologies where all data stores are only reachable via other nodes.
- (1) ⇔ (3): Application code contains logic that explicitly controls server interactions and assumes a specific ("hard-coded") process topology. For example, application logic in a process could assume that there is only a single server or exactly two replicated servers. Or, application code could assume that its process is the only server for its clients (i.e., that clients never load data objects offered by this server from other, alternative servers).
- (2) ⇔ (3): Application code contains logic specific to a particular data distribution scheme. For instance, application code could assume that sets of data objects loaded from different data stores can never overlap (i.e., data objects are never replicated across multiple data stores).

Coupling between the three aspects makes it hard to change one aspect independently of the others. In particular, coupling complicates (or even prevents) changes to the process topology. We require that middleware pairwise decouples all three aspects so that each aspect can be changed with minimum impact on the others. Decoupling (1) from both (2) and (3) is essential for adaptable process topologies. In addition, decoupling (2) from (3) is important for building custom process topologies because explicitly managing data distribution in the application code is likely to rely on a specific process topology.

Note that "decoupling" does not imply that the three aspects can be treated completely independently. A certain amount of coupling is inevitable as, for example, a data distribution scheme that specifies replication across *n* data stores implies that there is a process topology with at least *n* data stores. Or, application code that processes data objects of a given type always requires direct or indirect access to at least one data store that stores data objects of that type. However, it is important to minimize the dependencies between the three aspects as much as possible.

4.4.6 A Mechanism for Object and Query Routing (Requirement R6)

When a process requests execution of a query, the query has to be routed from that process through the process topology to appropriate data store(s). This process is called *query routing*. Similar to query routing is the process of routing newly created data objects or updated copies of data objects from a given process to the appropriate data store(s) to store them persistently. We call that process *object routing*.

In simple topologies where each process has at most one server, object and query routing is not an issue because there is always exactly one path to be followed. But in topologies where processes can connect to multiple servers and/or where meshes exist, a mechanism for object and query routing is required. Such a mechanism has to take both the structure of the topology and the data distribution scheme into account. In addition, when there are multiple valid options for routing (e.g., in the presence of meshes), a routing mechanism might try to optimize the routing according to given criteria (see Section 5.8 and Section 6.9).

In which part of an enterprise application should routing be implemented? As routing is an infrastructure issue, it should be part of the middleware. In principle, it could alternatively be implemented by application developers as part of their application code. However, we believe that application developers should focus on business logic and not on implementing infrastructure functionality. Also, implementing routing as part of the application code bears the risk that the routing mechanism is specific to a particular topology and/or data distribution scheme, which conflicts with requirement R5.

When object and query routing is realized as part of the middleware, how can we assure that routing does not introduce unnecessary coupling between process topology and data distribution scheme (see requirement R5)? To minimize coupling between these two aspects, we require that the routing mechanism supports all reasonable combinations of topologies and data distribution schemes. Only when all combinations are supported, it is possible to change one aspect to a large extent independently of the other aspect, which is important for adaptable process topologies.

Example: Figure 4-4 shows three different process topologies and defines three different data distribution schemes. An appropriate routing mechanism would support all nine combinations. If one of the nine options – e.g., *(D_1, T_2)* – was not supported, that would limit the options for adapting the process topology. For instance, a transition from *(D_1, T_1)* to *(D_1, T_2)* would be difficult.

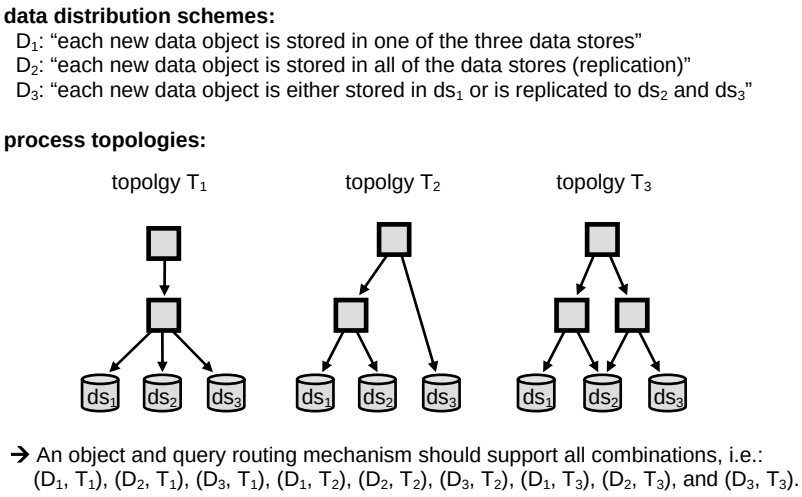

Figure 4-4. A routing mechanism should support all reasonable combinations of process topologies and data distribution schemes.

4.5 Limitations of Existing Middleware

In the previous section, we identified six key requirements for enterprise application middleware to support custom and adaptable process topologies. This section shows that current middleware does not (or only partially) fulfill our requirements. We analyze the limitations and discuss consequences for application developers.

In Section 2.4, we gave an overview of current middleware for enterprise applications. To what extent does existing middleware meet our six requirements for custom and adaptable process topologies? In principle, it is not sufficient to analyze each middleware standard/product separately to answer that question because enterprise applications often employ combinations of different middleware products. For example, an enterprise application may use an object/relational mapping framework for mapping relational data in a data store to objects in a middle tier server. These data objects could be wrapped and exposed as remote objects that are accessed by clients via CORBA. However, analyzing each of the

countless possible middleware combinations separately clearly is not realistic. Instead, we approach the question in three steps:

(1) First, in Subsection 4.5.1, we discuss to what extent existing middleware supports C1 connectors (including object data management) from processes to transactional data stores. In this step, we consider only those requirements that apply to such connectors, i.e., R1, R2, and R6.

(2) Then, in Subsection 4.5.2, we examine how data objects located in a given process can be mapped to direct client processes, i.e., we look at possible realizations of C1 connectors that connect two processes. In that context, requirements R1, R2, and R4 play an important role – we show that application developers often face significant problems here. R3, R5, and R6 are not considered because that would only be useful for middleware that complies with at least R1 and R2 as a prerequisite.

(3) Subsection 4.5.3 discusses various approaches for application developers to circumvent the problems described in the second step. In addition, we explain why these approaches are suboptimal and lead to conflicts with at least some of our six requirements.

4.5.1 C1 Connectors from Processes to Data Stores

How does existing middleware support C1 connectors from processes to transactional data stores? As we focus on object-oriented enterprise applications, we consider only middleware that provides application code with at least a basic object view on data (see Section 4.4). Current object-oriented enterprise applications typically use either object/relational mapping frameworks or (less common) object database management systems as C1 connector implementations. Both approaches can be used in combination with other middleware (e.g., Enterprise JavaBeans or Java Data Objects) and, in general, provide good support for direct object access to persistent data. Typically, data objects are copied across process boundaries (requirement R1) and can be cached at direct client processes of the data stores. Furthermore, despite copying, the identity of data objects is preserved (R2) – application code typically sees only a single instance per data object. Sophisticated middleware products also allow clients to connect to multiple data stores and transparently handle routing of objects and queries to appropriate data stores (R6). All in all, C1 connectors from processes to transactional data stores are well-supported with respect to requirements R1, R2, and R6.

4.5.2 C1 Connectors between Processes

This subsection discusses the use of existing middleware for C1 connectors between processes. Implementing these connectors is much more challenging than connectors between processes and data stores.

Note that object databases and object/relational mapping frameworks cannot be used here because they do not provide ways to realize arbitrary C1 connectors, which prevents construction of most custom topologies. Object databases provide an object view on data only for database servers and their direct clients, i.e., are useful only in the last two tiers of a multi-tier application. Similar restrictions apply to many object/relational mapping frameworks. A couple of advanced mapping products, such as ObJectRelationalBridge (OJB) [OJB03], Caché [Inte03], and OracleAS TopLink [Orac03], can extend their data management to some degree one step further, i.e., from direct clients of a database server to the clients of these clients. Unfortunately, these products are designed for very specific two- and three-tier process topologies – they are not general-purpose.

With RPC-based and message-oriented middleware, it is straightforward to connect two processes. However, only relatively low-level communication is supported; there is no cross-process data management and, if objects are supported, identity of transmitted data objects (R2) is not preserved (see Subsection 4.4.2). Currently, the only middleware-based approach to realizing C1 connectors between processes while preserving object identity is object middleware. With object middleware, data objects in a

process are exposed as middleware remote objects and other processes can access them through remote invocations, e.g., CORBA operation calls or RMI invocations. However, that approach does not meet requirement R1 (see discussion on performance in Subsection 4.4.1). In addition, it does not fulfill requirement R4 because remote data objects have fixed locations. This problem is depicted in Figure 4-5: Consider a remote data object *R* located in process *C*. *C* has a direct client *B* which obtains a reference to *R* and passes the reference to a process *A* (which is a direct client of *B*). When *A* accesses *R*, a direct communication between *A* and *C* is required. This in turn violates R4 because *C* becomes visible to *A* although there is no C1 connector between *A* and *C*. Even worse, direct communication between *A* and *C* is likely to offset all advantages that were the reason for introducing an indirection between processes *A* and *C*. Bypassing *B* may, for instance, break security or prevent an effective load balancing provided by *B*.

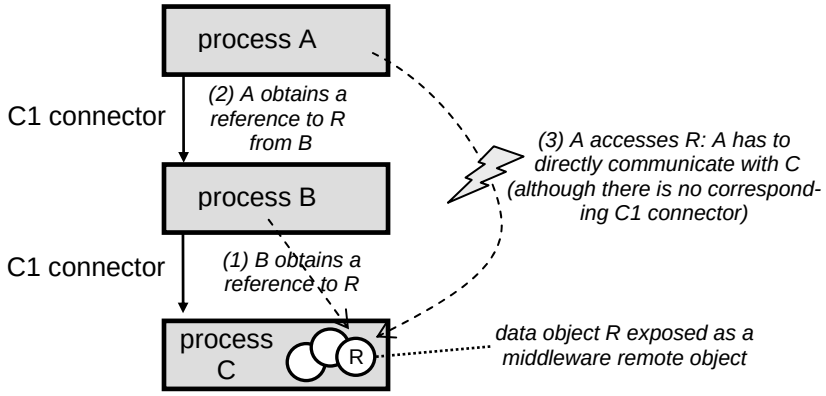

Figure 4-5. Exposing data objects as remote objects conflicts with the limited visibility requirement.

Note that performance penalties of object middleware can partly be offset by using so called *smart proxies* [Bake97] [IONA99] (or *smart stubs*): With object middleware, clients access server-side remote objects through local proxy (stub) objects. Smart proxies can be used to add a client-side caching mechanism to such proxies objects [KAD96] [WT00]. For example, a smart proxy could fetch and cache all attribute values of a remote object on the first access to it. Subsequent read accesses could then be processed locally. Smart proxies for caching can reduce the number of remote accesses from a per-attribute basis to a per-object basis. However, object-oriented, navigational access still requires a significant number of remote invocations. Furthermore, the problems with requirement R4 and indirect access (see above) remain. Besides, smart proxies are not standardized and a caching mechanism burdens application developers with complex infrastructure issues like cache replacement, propagation of changes, dealing with inconsistencies, and many more (see item 4 in the following subsection).

All in all, there is no reasonable middleware support for implementing arbitrary C1 connectors between processes. As a consequence, such connectors have to be implemented by application (see following subsection).

4.5.3 Consequences for Application Developers

The limitations of existing middleware often force application developers to consider suboptimal solutions, e.g.:

(1) *No custom process topologies.*

Application developers restrict themselves to a limited subset of topologies adequately supported by their middleware.

(2) *Thin clients.*

Thin clients (e.g., web browsers) are selected as clients even in situations where fat clients are much more appropriate, e.g., for realizing complex, user-friendly interfaces. Thin clients focus on presentation issues – they do not require an object view on data because all business logic and data access logic are located at the server side. Thus, the implementation of C1 connectors between thin clients and server processes is relatively simple as no cross-process management of data objects is required at all. The problems discussed in Subsection 4.5.2 are avoided – at least for C1 connectors between thin clients and other processes. However, for the rest of the process topology, the problems remain.

(3) *Application-specific facades for data shipping.*

Application developers implement a simple, ad-hoc solution for C1 connectors between processes. The server process exposes an application-specific facade to client processes. Clients cannot directly access data objects on the server; instead they talk to the facade, which implements application-specific data-shipping operations: The object states are extracted, transformed into a format for shipping, and then copied to the client (e.g., using one of the copy mechanisms outlined in Subsection 4.4.1 and Subsection 4.4.2).

Listing 4 shows an excerpt of an example interface (RMI) for such an application-specific facade. A facade often implements application-specific operations for inserting, updating, and deleting data objects in addition to its data-shipping operations (as shown in the listing). For data shipping, either generic data structures (e.g., strings or Java hash tables) or type-specific container classes can be used. While generic data structures are not type-safe, type-specific container classes are effectively duplicates of all server-side data object types. In Listing 4, type-specific container classes are used.

```
// container class for shipping values of a Customer instance
public class CustomerRecord implements java.io.Serializable {
   public CustomerRecord( Customer c ) { ... }
   ...
}
...
public interface OrderSystemFacade extends java.rmi.Remote {
   CustomerRecord[] getCustomersByName( String pattern, int maxHits )
                throws RemoteException, DatastoreEx;
   OrderRecord[] getOrdersByCustomer( String customerId )
                throws RemoteException, DatastoreEx;
   OrderRecord[] getOrdersByDate(Date from, Date to, int page, int ordersPerPage)
                throws RemoteException, DatastoreEx;
   void updateCustomer( CustomerRecord cr )
                throws RemoteException, NotFoundEx, UpdateConflictEx, DatastoreEx;
   void insertCustomer( CustomerRecord cr )
                throws RemoteException, NotFoundEx, AlreadyExistsEx, DatastoreEx;
   void deleteCustomer( CustomerRecord cr ) ...
   ... (other methods) ...
}
```

Listing 4. Example of an application-specific facade for data shipping.

Typically, such ad-hoc solutions do not preserve object identity (R2) on the client side and provide only a very limited object view on data. For instance, transparent, navigational access is difficult to realize and is rarely supported. In addition, developers have to create and maintain application-specific facades, container classes, and access operations, which can be a tedious and error-prone task even for object models of medium complexity. Finally, client application code explicitly manages communication with servers (by invoking functions of the facade), which conflicts with requirements R5 and R6.

(4) *Development of a cross-process data management mechanism.*

Application developers implement a full-fledged cross-process data management mechanism as part of their application code. Such solutions tend to become very complex since application developers have to deal with many aspects that are generally regarded as infrastructure issues, for instance:

- client side caching of objects,
- managing identity of objects on the client side,
- integration of cached data and client access operations into the server side transaction management,
- synchronization of stale data,
- load balancing, and
- error handling.

Most application developers want to focus on business logic and are not prepared to handle these infrastructure issues, which should be part of the middleware. Even with skilled developers, this approach is costly, error-prone, and time-consuming, and thus a risk for many projects. In addition, as the data management is part of the application code, requirements R5 and R6 are typically not fulfilled.

In this section, we have shown that application developers have several options for dealing with the limitations of existing middleware. In practice, often a combination of these options is employed, e.g., thin clients, simple process topologies, and application-specific facades. In our opinion, neither option (nor combination of options) represents an adequate solution. Moreover, it should be noted that the problems discussed in this subsection and the previous subsection do not apply exclusively to enterprise applications with exceptionally complex custom topologies. Even simple standard topologies, like three-tier structures with fat clients, experience such problems (especially with respect to R1 and R2).

4.6 Discussion

In this section, we discuss two aspects in more detail:

- In Section 4.4, we identified requirements for middleware to support custom and adaptable process topologies. Fulfilling all requirements is a key factor for realizing custom and adaptable process topologies – however, it is not a sufficient condition. We chose the given six requirements given because they address crucial, high-level design issues. In addition to the requirements, a broad range of design and implementation details of a middleware usually play an important role. Also, each of the process topology patterns described in Section 3.4 requires at least some support by a middleware implementation: For instance, when the *wrapper insertion* pattern (see Subsection 3.4.1) is applied for vertical load distribution, a caching mechanism could be needed. Or, when the *meshing* pattern (see Subsection 3.4.3) is applied to improve fault tolerance, a mechanism for fail over is important.
- In Subsection 4.1.2, we stated that, because of changing requirements, it is often necessary to adapt an existing process topology. It should be noted that not all possible changes in requirements can (and have to) be addressed by topology changes. For instance, when functional requirements change, usually this cannot be addressed by topology changes (alone). Also, as discussed in Section 3.3, the design of a process topology typically is a trade-off between many application-dependent factors. Adapting a process topology is only helpful and required if factors change in such a way that the result of the new trade-off differs from the previous topology.

4.7 Summary

In this chapter, we first motivated the importance of custom and adaptable process topologies. *Custom* processes are needed to address individual requirements of demanding enterprise applications – often, simple standard topologies are not sufficient. *Adaptable* process topologies address evolution of enterprise applications and their process topologies. We presented a case study describing the evolution of a tiny enterprise application into a large, mission-critical system with a complex topology. The case study illustrated the use and advantages of custom and adaptable process topologies.

Then we explained that support for custom and adaptable process topologies is a matter of enterprise application middleware. We identified six key requirements for middleware and showed that existing middleware do not support all of these requirements. This makes it difficult to realize enterprise applications with custom and adaptable process topologies on top of current middleware products.

The six key requirements identified in this chapter are a basis for the following Chapter 5. In that chapter, we present concepts for middleware to fulfill all six requirements and explicitly support custom and adaptable process topologies.