Chapter 5: The FPT Architecture

In the previous chapter, we showed that many demanding enterprise applications need custom and adaptable process topologies. We identified several key requirements for enterprise application middleware to support custom and adaptable process topologies. We also showed that existing enterprise application middleware does not fulfill all these requirements. Therefore, it is currently hard to build enterprise applications with custom and adaptable process topologies on top of existing middleware.

In this chapter, we present our *Flexible Process Topology* (FPT) architecture, which specifies principles of a data-centric infrastructure for object-oriented enterprise applications. The architecture consists of a set of key concepts for enterprise application middleware. The architecture is a basis for building middleware that explicitly supports both custom and adaptable process topologies. We show that it addresses all requirements we identified in Chapter 4.

Note that we decided to separate the FPT architecture from concrete middleware implementation details: In principle, there are many ways to realize enterprise applications with custom and adaptable process topologies, i.e., there is a large space of possible design solutions. The FPT architecture presented in this chapter does not define a concrete solution, i.e., a point in the design space. Instead, the architecture prescribes several fundamental, high-level design decisions and thus *narrows* the design space. Separating our FPT architecture from concrete implementation details has the following advantages:

- This chapter can focus on key concepts that are essential for realizing middleware to support custom and adaptable process topologies. Discussing the concepts together with all relevant implementation details (such as communication protocols, multi-threading, and cache replacement strategies) here would divert attention from the key concepts.
- The FPT architecture as a separate entity is much more useful for developers and researchers because it is not tied to a specific implementation. This separation makes it much easier to compare the FPT architecture to other architectures, to provide specialized implementations, or to reason how FPT concepts can be introduced into existing enterprise application middleware.

Our FPT architecture is not meant to be a complete specification like, for instance, the Common Object Request Broker Architecture. It was not our goal to ensure interoperability of different implementations or to capture all design aspects. There is still considerable freedom for designing and implementing concrete, FPT-based middleware – one particular solution will be described in Chapter 6.

The FPT architecture covers many aspects. To provide a clear structure for this chapter, we group these aspects into seven parts. While the first four parts focus on static aspects, the other three parts concentrate on dynamic/runtime aspects.

The following parts focus on static aspects:

Part 1	**"Data Distribution Scheme"** This part describes how data objects are distributed among multiple data stores.
Part 2	**"Import/Export Scheme"** This part describes how sets of data objects are imported and exported by components of the process topology.
Part 3	**"Process Topology and Configuration"** This part explains why configuration plays an important role in the FPT architecture and how the process topology and other details of an enterprise application are defined through configuration.
Part 4	**"Decoupling"** This part discusses how application code, process topology, and data distribution scheme are decoupled so that one aspect can be changed mostly independent of the others.

The following parts focus on runtime aspects:

Part 5	**"Copies and Caching"** This part outlines how copies of data objects are cached in object managers.
Part 6	**"Transaction Management and Concurrency Control"** This part describes how transactions are managed and how concurrent transactions are isolated from each other.
Part 7	**"Object and Query Routing"** This part describes how objects and queries are routed through a process topology.

This chapter is structured as follows: First, in Section 5.1, we give a brief overview of the FPT architecture. Then, the following sections, from 5.2 to 5.8, discuss the seven parts of the FPT architecture. Section 5.9 discusses several selected aspects in more detail. Finally, in Section 5.10, we give a summary of this chapter.

5.1 Overview of the FPT Architecture

The FPT architecture addresses enterprise applications that follow the "multi-tiered enterprise application" architectural style described in Chapter 3. All valid architectural configurations, i.e., arbitrary DAG process topologies, are supported. The FPT architecture is based on a network of generic *object manager* components that collaboratively perform communication and data management functions for an enterprise application. As depicted in Figure 5-1, a local object manager component is placed in each process of the enterprise application's process topology. An object manager provides local application code with an API for transactional access to data objects. Conceptually, the API is comparable to APIs of object databases and O/R mapping frameworks, i.e., the application can set transaction boundaries, perform object queries, has an object view of data (including object identity; see requirement R2 in Subsection 4.4.2), and can navigate through graphs of data objects. Application code in a process accesses data objects exclusively through the local object manager, which transparently handles data management and remote communication.

Via the network, an object manager in a process *P* can directly communicate with three types of components:

- *Transactional data stores.* All transactional data stores connected to *P* via C1 connectors (see Subsection 3.2.2) can be accessed.
- *Client object managers.* All object managers located in processes that are connected to *P* via C1 connectors and are clients of *P* can be contacted.
- *Server object managers.* All object managers located in processes that are connected to *P* via C1 connectors and are servers of *P* can be accessed.

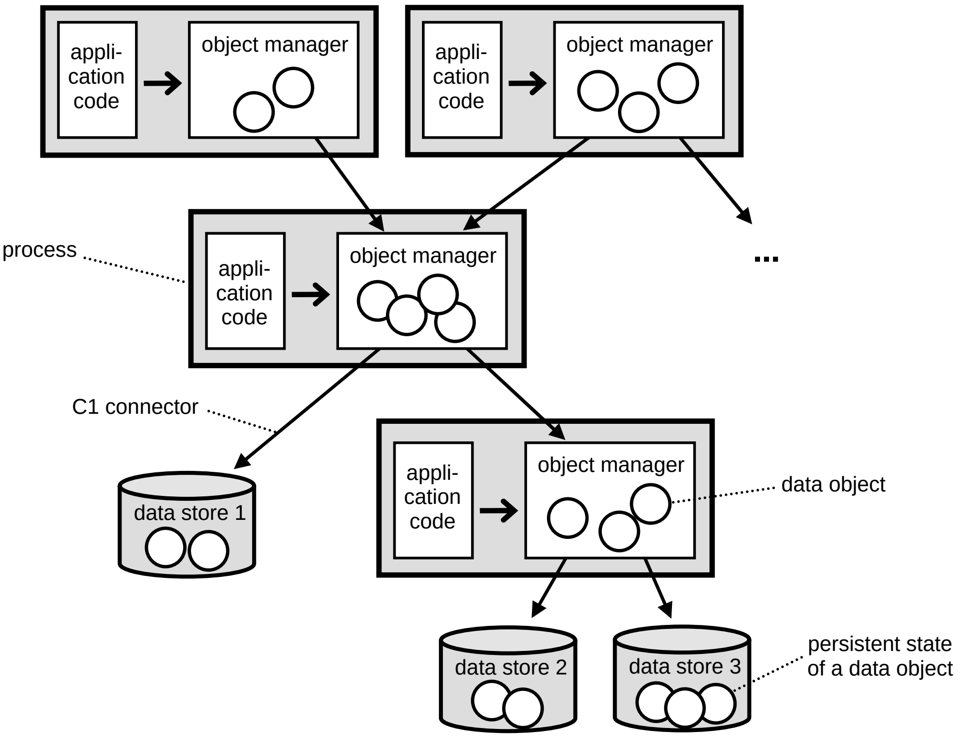


Figure 5-1. A process topology based on our FPT architecture: In each application process resides an object manager component that manages business data objects.

For interaction with transactional data stores, standard access protocols like JDBC or ODBC are employed. In principle, any number and all types of data stores can be accessed, including combinations of different types. An object manager needs an adapter for each kind of data store to be accessed.

An object manager considers all of its servers (data stores and object managers in server processes) *data sources* that provide read and write access to data objects. Along C1 connectors, data objects can be copied *by value* from one object manager to another (see requirement R1 in Subsection 4.4.1). This helps to avoid fine-grained remote access to data objects and, consequently, many performance problems. Data objects can be copied in both directions: For example, insert operations typically require propagating copies “down” the process topology to appropriate data stores whereas a query typically requires propagating copies “up” the process topology to the process that issued the query.

An object manager acts as a cache for business data objects and services local application code as well as client object managers. The object managers in a process topology, thus, form a hierarchy of transactional caches for data objects persistently stored in data stores. For all data objects that are used by application code in a process, a copy must be present in the local object manager’s cache. When application code attempts to access a copy that is not yet cached locally, a *cache miss* occurs. In that case, the local object manager transparently loads the copy from one of its servers (data sources), which in turn may (recursively) load it from other data sources (see requirement R3 in Subsection 4.4.3).

The FPT architecture has been designed to support object-oriented enterprise applications because object-orientation is state of the art in software engineering and current enterprise applications are typically developed with a strong emphasis on data objects. However, not all object-oriented concepts are of fundamental importance to the architecture. In fact, only object identity plays a major role (because of consistency). Other object-oriented concepts, like inheritance or polymorphism, which are convenient for application developers, are compatible with our approach but they are not essential.

In this section, we gave a brief overview of the FPT architecture. The following sections 5.2 to 5.8 discuss specific aspects of the architecture in more detail.

5.2 Data Distribution Scheme (Part 1)

For a variety of reasons (which have been discussed in Chapter 3), large scale enterprise applications often do not store their data objects in a single data store but distribute them among multiple data stores. A *data distribution scheme* defines rules that describe how existing data objects are distributed among data stores and according to which new data objects have to be distributed among data stores. For the FPT architecture, the data distribution aspect is important because the design of a process topology often influences and constrains the design of a data distribution scheme and vice versa. For example:

- When the layout of a process topology is given, a suitable data distribution scheme has to be defined such that each process in the topology has access to the (sub)set of data objects required by it.
- When a software architect adapts an existing process topology by applying topology patterns (e.g., the *integration of subsystems* pattern described in Subsection 3.4.6), it might also be desired or necessary to change the data distribution scheme.
- When a software architect first designs a specific data distribution scheme to achieve good performance, scalability, or fault tolerance, he has to construct an appropriate process topology on top of the given data stores.
- When the data distribution of an existing enterprise application is changed (e.g., by applying the *data distribution* topology pattern described in Subsection 3.4.4), a software architect may want to apply additional changes to the process topology, e.g., to achieve maximum performance.

For enterprise applications with custom and adaptable process topologies, it is essential that sufficiently powerful data distribution schemes can be employed. With limited support for data distribution, the advantages of many custom topologies could not be fully exploited. Furthermore, flexibility of combining arbitrary process topology patterns into custom topologies would be severely restricted if limited support for data distribution would introduce additional constraints for applying topology patterns.

In the following subsections, we present our approach to data distribution. The approach has been published in [Hart03a]; it allows different variants of data distribution to be used and also allows these variants to be nested.

5.2.1 Nesting of Data Distribution Variants

When we presented the *data distribution* process topology pattern in Subsection 3.4.4, we distinguished three variants of the pattern: (a) replication, (b) distribution based on a static partitioning criterion, and (c) ad hoc distribution. Each of the variants stands for a different approach to distribution of persistent data among multiple data stores and has specific advantages and drawbacks. For instance, variant (a) may improve availability but requires additional effort to keep all replicas consistent when data is written.

One of the main goals of the FPT architecture is to enable architects to create custom topologies by allowing them to freely combine topology patterns. Therefore, the architecture supports not only one specific variant of the *data distribution* topology pattern, but all three variants. However, in many situations, this is still not sufficient; for example:

(1) Often there is no single data distribution scheme that is equally appropriate for all data processed by an enterprise application. Instead, different sets of data objects have different characteristics and access patterns and therefore it makes sense to treat them differently. For example, when *Department* data objects are frequently read but rarely written, it might make sense to replicate them (pattern variant (a)). In contrast to that, data objects of type *OrderStatus*, which are frequently written, might be better distributed with pattern variant (b) or (c).

(2) When two (or more) subsystems are integrated using the *integration of subsystems* process topology pattern (see Subsection 3.4.6), each of the subsystems brings in not only its process topology but also its data distribution scheme. For example, assume that one subsystem stores *Customer* data objects according to pattern variant (a) and the other stores *Customer* data objects according to pattern variant (c). Such a situation could occur, for instance, in step 5 or in step 6 of the case study shown in Figure 4-1. Then it would be difficult to integrate both subsystems because either variant (a) or variant (c) had to be used in the integrated subsystem – but not both together. As a consequence, it would be necessary to change the data distribution scheme(s) as well as to reorganize the contents of the data stores. However, to simplify integration, it is often desirable to let both data distribution schemes co-exist in the integrated system.

(3) In some cases, advanced data distribution schemes that combine the characteristics of different variants of the data distribution pattern are required. For example, in a large, widely distributed enterprise application, it could be necessary to replicate a set of data objects to different sites (pattern variant (a)) so that each site remains relatively autonomous (with respect to read access) even when its wide area network connections are down. Internally, each site could decide to distribute the data among several of its data stores (pattern variant (b) or (c)) in order to distribute load. The result is a hierarchical composition of variants of the *data distribution* pattern.

To address these issues, our FPT architecture also supports *combinations* of different data distribution approaches in the sense that it permits software architects to *nest* different variants of the *data distribution* pattern and thus create a hierarchy. For example, it is possible to partition data objects into different sets with pattern variant (b) and then apply a separate distribution strategy to each partition, as shown in Figure 5-2.

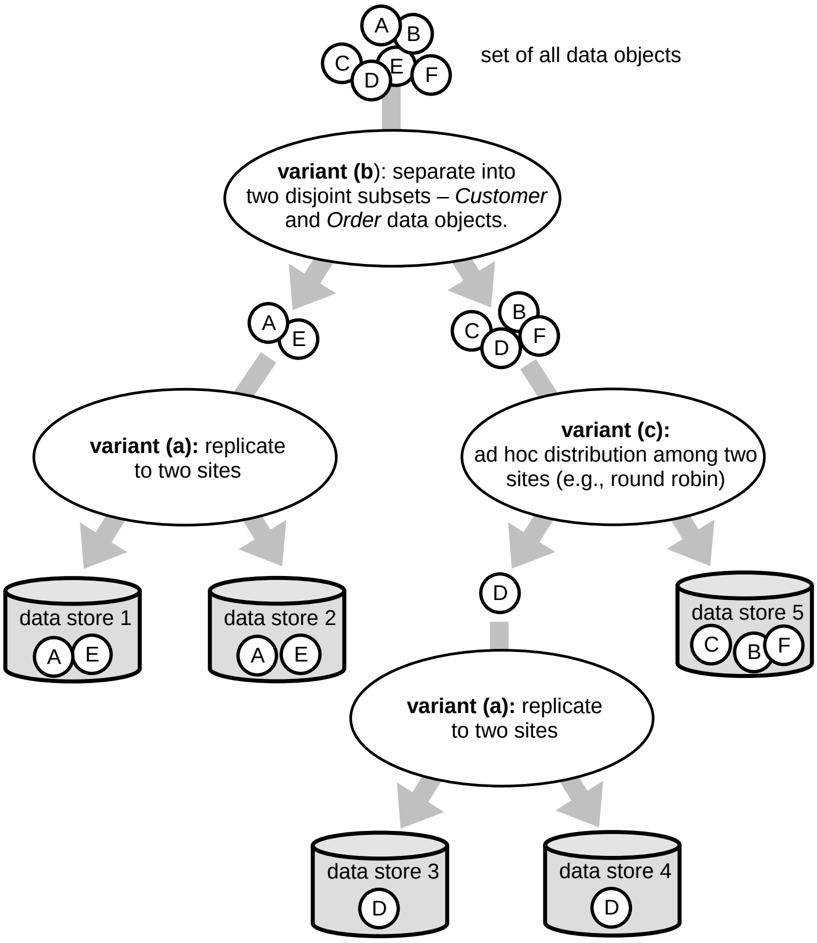

A
B
C
E
D
F
set of all data objects
variant (b): separate into two disjoint subsets – *Customer* and *Order* data objects.
A
E
B
C
F
D
variant (a): replicate to two sites
variant (c): ad hoc distribution among two sites (e.g., round robin)
data store 1
A
E
data store 2
A
E
D
data store 5
C
B
F
variant (a): replicate to two sites
data store 3
D
data store 4
D

Figure 5-2. Example of a hierarchical composition (nesting) of variants of the *data distribution* pattern.

In the following subsections, we describe how such data distribution schemes are defined for systems based on our FPT architecture.

5.2.2 Defining a Data Distribution Scheme in Three Steps

Although it would have been possible to allow arbitrary (recursive) nesting of all data distribution variants in the FPT architecture, we chose a slightly less complex approach: When variants are nested, variant (b) may occur only at the root of the hierarchy whereas variants (a) and (c) can be arbitrarily nested below that root.

Using the FPT architecture, a software architect defines a data distribution scheme for an enterprise application in three steps:

(1) **Partitioning.** The set of all data objects (all possible extensions) is partitioned into one or more disjoint domains (subsets) according to given static partitioning criteria. We name these domains $dom_1..dom_{maxdom}$. This partitioning step corresponds to variant (b) of the *data distribution* pattern (see Subsection 3.4.4).

Example: A simple enterprise application for online shopping defines and processes data objects of three different types: *Item*, *Customer*, and *Order*. The software architect decides to partition the data objects into three domains:

domain dom_1 contains all *Item* and *Customer* data objects,
domain dom_2 contains all *Order* data objects with *acceptTime* < 01/01/2002,
domain dom_3 contains all *Order* data objects with *acceptTime* ≥ 01/01/2002.

(2) **Selection of data stores.** For each domain dom_i, the software architect selects a non-empty subset $DSSet_i$ of all data stores that exist in the topology. These data stores are responsible for persistently storing data objects of the corresponding domain.

Example (continued): The process topology of the online shopping enterprise application contains three data stores: ds_1, ds_2, and ds_3. For the three domains defined in the previous step, the software architect selects data stores as follows:

$DSSet_1 = \{ds_1, ds_2\}$, i.e., data objects of domain dom_1 are stored in ds_1 and/or ds_2,
$DSSet_2 = \{ds_1, ds_3\}$, i.e., data objects of domain dom_2 are stored in ds_1 and/or ds_3,
$DSSet_3 = \{ds_3\}$, i.e., data objects of domain dom_3 are stored in ds_3.

(3) **RI-trees.** For each domain dom_i, the software architect specifies an RI-tree RI_i with a leaf node set $DSSet_i$. The tree defines how data objects are distributed among the data stores in $DSSet_i$ and allows arbitrary nesting of variants (a) and (c) of the *data distribution* pattern. In the following subsection, we describe RI-trees in more detail.

5.2.3 RI-Trees

In this subsection, we introduce *RI-trees*[1], which are used to define rules that determine how data objects are distributed among one or more data stores. For each domain (see previous subsection), a software architect defines a separate RI-tree to determine how data objects are distributed among the data stores selected for that domain.

An RI-tree is a tree whose leaf nodes represent different transactional data stores and whose inner nodes are either R-nodes ("Replication") or I-nodes ("Integration"). Intuitively and informally, an R-node means that an object is replicated and placed in *all* sub-trees of the node. We assume that an eager, read-once/write-all (ROWA) approach to replication [BHG87] [ÖV99] is employed to guarantee one-copy serializability [BG86] for transactions that access replicated data. An I-node means that an object is placed in *exactly one* sub-tree of the node.

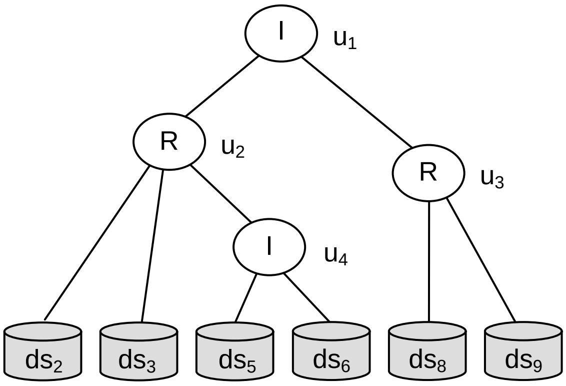


Figure 5-3. Example of an RI-tree.

[1] Our RI-tree is not to be confused with the *Relational Interval Tree* described in [KPS00]

An example of an RI-tree is given in Figure 5-3. For illustration purposes, we show a relatively complex tree – in typical systems, most RI-trees will be much simpler, e.g., have fewer inner nodes or consist only of a single data store node. The tree presented in the figure specifies distribution among six data stores labeled ds_2, ds_3, ds_5, ds_6, ds_8, and ds_9. The tree gives us three *insert options* for storing a newly created data object – each option defines a possible set of *home data stores* for an object:

Insert option 1: select home data stores ds_2, ds_3, and ds_5
Insert option 2: select home data stores ds_2, ds_3, and ds_6
Insert option 3: select home data stores ds_8 and ds_9

At runtime, the enterprise application middleware has to select one insert option for each newly created data object. Then the data object has to be stored in *all* these home data stores (ROWA replication). Different data objects may be stored according to different insert options. Once an insert option has been selected for a data object, its home data stores cannot be changed any more. We will refer to the set of home data stores of an existing data object *o* as *home(o)*. When a data object is updated or deleted, the middleware has to update or delete entries in all of the object's home data stores (ROWA). To load a specific data object, it is sufficient to access one of its home data stores. To query all data objects of a domain that satisfy a given condition, it may be necessary to access one, some, or all data stores of the domain's RI-tree (assumed that no caching is employed). More details on processing and routing of such queries can be found in Section 5.8.

Note that R-nodes and I-nodes do not (necessarily) correspond to processes in a topology. In fact, in Section 5.5, we will see that RI-trees and process topologies are orthogonal concepts.

Having introduced the concept of RI-trees rather informally, we now define RI-trees more formally: An RI-tree T is a rooted tree (V, E); V is the vertex set and E the edge set. $V = RNODES \cup INODES \cup DATASTORES$. The three sets *RNODES*, *INODES*, and *DATASTORES* are pairwise disjoint. *DATASTORES* is a non-empty subset of the set of all data stores $DSall = \{ds_1, ds_2, ..., ds_{maxds}\}$. T's inner nodes are $RNODES \cup INODES$; its leaf nodes are *DATASTORES*. As an example, a formal representation of the RI-tree shown in Figure 5-3 can be found below.

$T = (V, E)$
$V = \{ u_1, u_2, u_3, u_4, ds_2, ds_3, ds_5, ds_6, ds_8, ds_9 \}$
$INODES = \{ u_1, u_4 \}$
$RNODES = \{ u_2, u_3 \}$
$DATASTORES = \{ ds_2, ds_3, ds_5, ds_6, ds_8, ds_9 \}$
$E = \{ (u_1, u_2), (u_2, u_4), (u_1, u_3), (u_2, ds_2), (u_2, ds_3), (u_4, ds_5), (u_4, ds_6), (u_3, ds_8), (u_3, ds_9) \}$
$root(T) = u_1$

As already mentioned, an RI-tree defines different insert options for storing data objects. Formally, these insert options can be described by a function *insertOptions* that maps each element of V to a subset of 2^{DSall}. For each RI-tree RI_i, its *insertOptions* are the result of a function $insertOptions(root(RI_i))$, or shorter: $insertOptions(RI_i)$. The *insertOptions* function is recursively defined as follows:

$$insertOptions(u) = \begin{cases} \{M_1 \cup M_2 \cup \ldots \cup M_m \mid (M_1, M_2, \ldots, M_m) & \\ \quad \in insertOptions(v_1) \times \ldots \times insertOptions(v_m), & \\ \quad where\ \{v_1, v_2, \ldots, v_m\} = \{v \mid (u,v) \in E\}\ \} & \text{for } u \in RNODES \\ \bigcup_{w \in \{v \mid (u,v) \in E\}} insertOptions(w) & \text{for } u \in INODES \\ \{\{u\}\} & \text{for } u \in DATASTORES \end{cases}$$

Each element in $insertOptions(RI_i)$ is a set of data stores and represents one option for storing data objects contained in the domain dom_i. As an example, all insert options defined by the RI-tree shown in Figure 5-3 are given below. First, the *insertOptions* of the lower level inner nodes are calculated; then, bottom-up-style, the *insertOptions* of the higher level inner nodes, and finally the *insertOptions* of the complete RI-tree.

$insertOptions(ds_i) = \{\{ds_i\}\}, for\ i=2,3,5,6,8,9$
$insertOptions(u_4) = \{\{ds_5\}, \{ds_6\}\}$
$insertOptions(u_3) = \{\{ds_8, ds_9\}\}$
$insertOptions(u_2) = \{\{ds_2, ds_3, ds_5\}, \{ds_2, ds_3, ds_6\}\}$
$insertOptions(T) := insertOptions(u_1) = \{\{ds_2, ds_3, ds_5\}, \{ds_2, ds_3, ds_6\}, \{ds_8, ds_9\}\}$

In the example, only the results of the *insertOptions* function for nodes of the RI-tree are given. The following example shows, step by step, how the result of the *insertOptions* function is calculated for the RI-tree node u_2 (which is an R-node):

(1) The set of all children of u_2 is calculated: $\{v_1, v_2, ..., v_m\} = \{ds_2, ds_3, u_4\}$

(2) The cross product of all insert options of the children is calculated:

$insertOptions(v_1) \times ... \times insertOptions(v_m)$
$= insertOptions(ds_2) \times insertOptions(ds_3) \times insertOptions(u_4)$
$= \{\{ds_2\}\}\times\{\{ds_3\}\}\times\{\{ds_5\}, \{ds_6\}\}$
$= \{(\{ds_2\}, \{ds_3\}, \{ds_5\}), (\{ds_2\}, \{ds_3\}, \{ds_6\})\}$

(3) There are two elements of the form $(M_1, M_2, ..., M_m)$ in the cross product: $(\{ds_2\}, \{ds_3\}, \{ds_5\})$ and $(\{ds_2\}, \{ds_3\}, \{ds_6\})$

(4) For each of the two elements described above, $M_1 \cup M_2 \cup ... \cup M_m$ is calculated. For the first element, the result is $\{ds_2, ds_3, ds_5\}$, and for the second element, the result is $\{ds_2, ds_3, ds_6\}$.

(5) Both results are included as elements into the result of $insertOptions(u_2)$.

5.2.4 Integration of Data Distribution Schemes

How does our approach to data distribution help to integrate different subsystems, as motivated in Subsection 5.2.1, item (2)? Let us assume that two subsystems are to be integrated and each of them defines its own data distribution scheme, i.e., defines multiple domains (see Subsection 5.2.2) and an RI-tree (see Subsection 5.2.3) for each domain.

- In the simplest of cases, the domains defined as part of both subsystems' data distribution schemes are pairwise disjoint. For instance, a subsystem *A* defines domains for *Item* and *Customer* data objects and a subsystem *B* defines a domain for *Order* data objects. Then a data distribution scheme for the integrated enterprise application is constructed by simply adding the two initial data distribution schemes as depicted in the example in Figure 5-4.
- More interesting are cases where a given domain is defined by *both* enterprise applications to be integrated (e.g., both applications define a domain for *Item* data objects), but each application defines a different RI-tree for that domain. In those cases, the two initial RI-trees can be merged into a single one by adding a new root node on top of both initial trees as illustrated in Figure 5-5.

  The advantage of that approach is that both initial data distribution schemes are preserved and become part of a new, integrated data distribution scheme. Without the possibility to nest different data distribution variants, that would not be possible. If desired, the resulting integrated system can be integrated again with other systems (recursively) using the same approach. A new root node placed on top of two existing RI-trees can be either an I-node or an R-node. In both cases, it may be necessary to first eliminate or merge data objects that are duplicates, i.e., exist in both subsystems and represent the same real world entity. In addition, when a software architect selects an R-node as the new root node, it may be necessary to copy data from the first subsystem to data stores of the second subsystem and vice versa, in order to comply with the new R-node.

- An additional intermediate step is required when domains defined by the two initial data distribution schemes *overlap*, i.e., there is a domain dom_i defined in subsystem *A* and a domain dom_k defined in subsystem *B* with $dom_i \neq dom_k$ and $dom_i \cap dom_k \neq \emptyset$. In such cases, it is necessary to first split such domains in such a way that no overlap remains.

  Example:

  Subsystem *A* defines a domain that contains all *Item* and *Customer* data objects,
  Subsystem *B* defines a domain that contains all *Item* and *Vendor* data objects.

  Before *A* and *B* can be integrated, the domains are split as follows:

  Subsystem *A* defines a domain that contains all *Item* data objects and another domain that contains all *Customer* data objects,
  Subsystem *B* defines a domain that contains all *Item* data objects and another domain that contains all *Vendor* data objects.

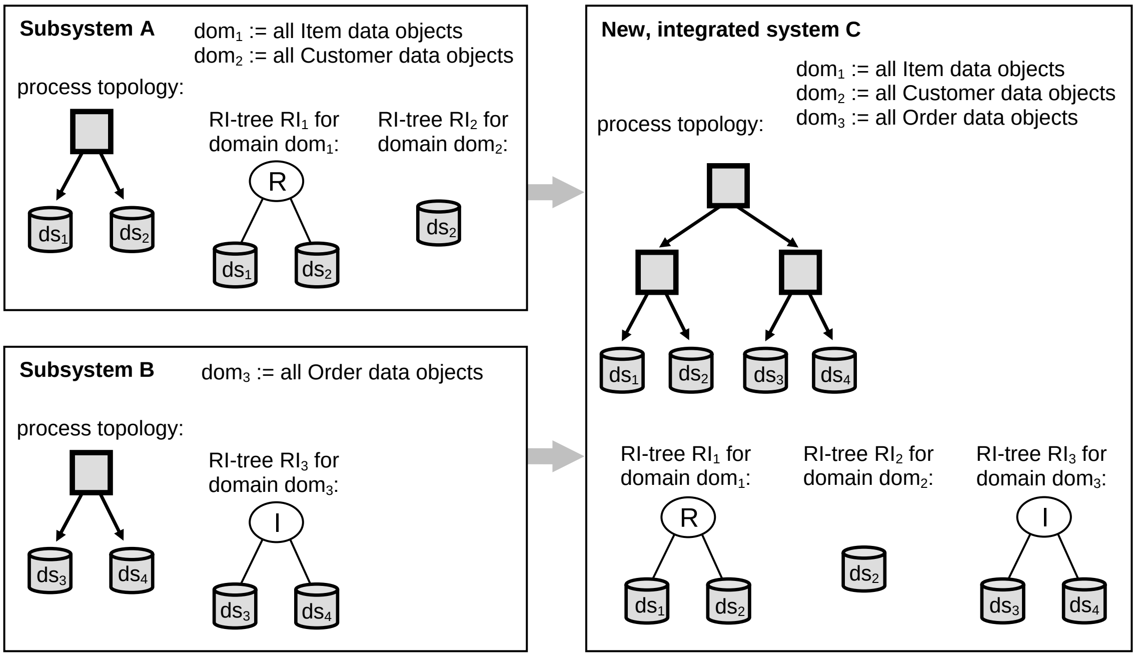

Subsystem A
dom1 := all Item data objects
dom2 := all Customer data objects
process topology:
RI-tree RI1 for domain dom1:
RI-tree RI2 for domain dom2:
R
ds1
ds2
Subsystem B
dom3 := all Order data objects
process topology:
RI-tree RI3 for domain dom3:
I
ds3
ds4
New, integrated system C
dom1 := all Item data objects
dom2 := all Customer data objects
dom3 := all Order data objects
process topology:
RI-tree RI1 for domain dom1:
RI-tree RI2 for domain dom2:
RI-tree RI3 for domain dom3:

Figure 5-4. Example of integrating two subsystems A and B, including their data distribution schemes, into a new system C (disjoint domains, pattern variant (b) used).

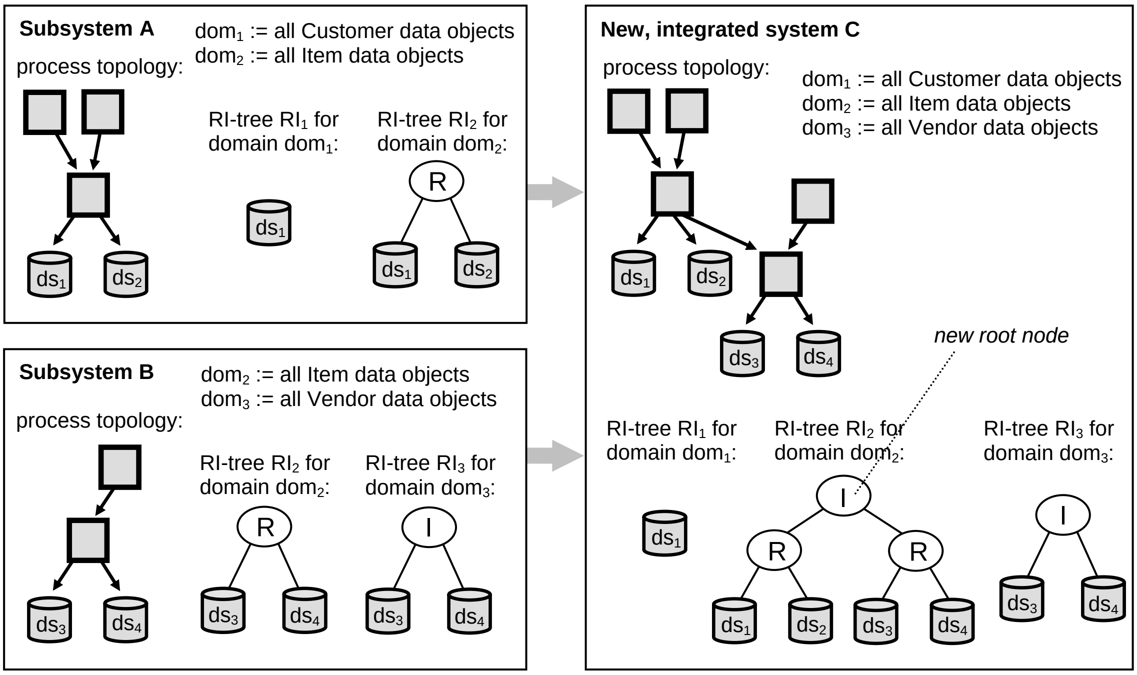

Subsystem A
dom₁ := all Customer data objects
dom₂ := all Item data objects
process topology:
RI-tree RI₁ for domain dom₁:
RI-tree RI₂ for domain dom₂:
R
ds₁
ds₂
Subsystem B
dom₂ := all Item data objects
dom₃ := all Vendor data objects
process topology:
RI-tree RI₂ for domain dom₂:
RI-tree RI₃ for domain dom₃:
I
ds₃
ds₄
New, integrated system C
process topology:
dom₁ := all Customer data objects
dom₂ := all Item data objects
dom₃ := all Vendor data objects
new root node
RI-tree RI₁ for domain dom₁:
RI-tree RI₂ for domain dom₂:
RI-tree RI₃ for domain dom₃:

Figure 5-5. Example of integrating two subsystems A and B, including their data distribution schemes, into a new system C (common domain dom$_2$, pattern variant (a) used).

5.2.5 Data Distribution Variants and Nesting in Existing Systems

Existing systems cover only a subset of the features of our data distribution scheme because they do not support all variants of data distribution described in Subsection 3.4.4 and/or do not permit recursive nesting. Also, with our approach, data distribution is handled by the middleware (i.e., the network of object manager components). In contrast to that, in existing systems, data distribution is, typically, either handled entirely at the data store level or is addressed at the application level:

- When data distribution is handled at the *data store level* (especially in clusters of data stores, see Subsection 3.5.6), the mechanism is typically transparent to application code. Distributed databases [ÖV99] usually support variant (a) to replicate and variant (b) to partition data items among sites. Many databases also support partial replication, which corresponds to nesting variant (a) under variant (b). Variant (c) is less common and can be found, for example, in the Informix database system, where a mechanism called “round robin fragmentation” allows rows of a table to be distributed round-robin-style among multiple DBspaces/disks [IBM03a]. Round robin fragmentation corresponds to nesting variant (c) under variant (b).

- When data distribution is addressed at the *application level*, all data distribution logic has to be implemented as part of the application code running in different nodes. For example, a server that receives an *insertCustomer* remote operation call could explicitly insert a customer record into three different databases to realize data distribution variant (a). Or, the server could insert a record into the data store with the least load to realize an instance of variant (c). In principle, all variants and arbitrary nesting can be achieved at the application level. However, as data distribution is an infrastructure issue and explicitly coded data distribution mechanisms can easily get complex, error-prone, and hard to maintain, only simple data distribution schemes are realized that way, if at all. Moreover, handling data distribution as part of the application code conflicts with requirement R5 (see Subsection 4.4.5).

A remarkable exception is the RAIDb (redundant array of inexpensive databases) concept [CMZ03], which manages data distribution at the middleware level, as our approach does. RAIDb applies the *redundant array of inexpensive disks* concept (RAID, also see Subsection 5.9.5) to data stores. Several RAIDb levels, which correspond to different data distribution approaches, are defined – it is also possible to nest them recursively. The RAIDb concept has been implemented in the C-JDBC project [OWC02]. However, in contrast to our approach, RAIDb and C-JDBC address relational data instead of objects. In addition, there is no RAIDb level that covers fine-grained distribution per data item, which our data distribution variant (c) and also RAID level 0/striping do.

Having described our approach to data distribution, we now describe how an *import/export scheme* helps to keep track of which data objects can be accessed via which processes and data stores in a distributed process topology.

5.3 Import/Export Scheme (Part 2)

As already mentioned in Section 5.1, an object manager considers all of its direct servers in the process topology as data sources that provide read and write access to data objects. When data objects are partitioned into different domains and distributed among multiple data stores (as described in Section 5.2), it is essential for an object manager to know which of its data sources provide access to which data objects. Without that knowledge, an object manager would not be able to execute queries or to insert, update, and delete data objects in compliance with the data distribution scheme defined for the enterprise application.

Our FPT architecture requires a definition of an *import/export scheme* for an enterprise application. Usually, such a scheme will be defined by a software architect along with the application's process topology. An import/export scheme consists of definitions of (a) *imports* for each C1 connector and (b) *exports* for each process and each data store of a process topology (see Subsection 3.2.2 for details on connectors). Both *imports* and *exports* are sets of (*domain*, *data store*) tuples:

- Each (*domain*, *data store*) tuple in the *imports* of a C1 connector that connects *C* to *S* means that *C* may – via *S* – access data objects of the given *domain* that are persistently stored in *data store*.
- Each (*domain*, *data store*) tuple in the *exports* of a process *P* means that *P* offers direct client processes access to data objects that belong to the given *domain* and are persistently stored in *data store*. The data objects are accessed via *P*, which in turn relies on its underlying data sources.
- Each (*domain*, *data store*) tuple in the *exports* of a data store ds_i means that data objects of the given *domain* are stored in ds_i.

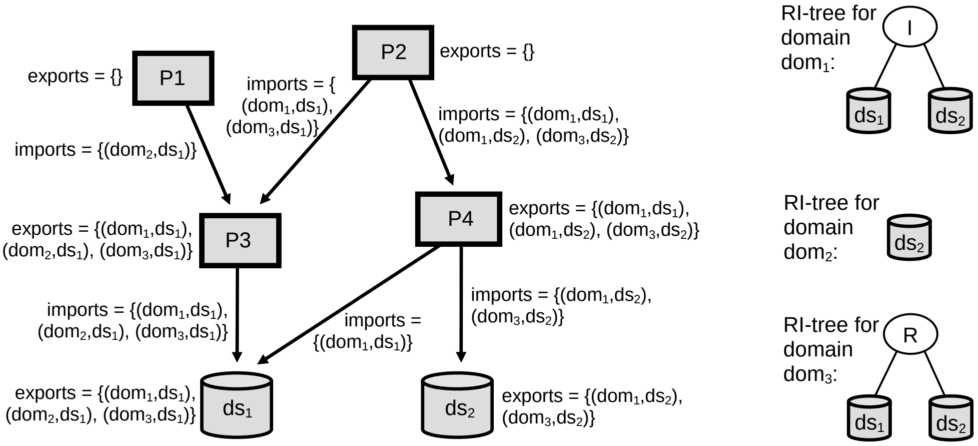

exports = {}
P1
imports = {(dom$_2$,ds$_1$)}
P2
exports = {}
imports = {
(dom$_1$,ds$_1$),
(dom$_3$,ds$_1$)}
imports = {(dom$_1$,ds$_1$),
(dom$_1$,ds$_2$), (dom$_3$,ds$_2$)}
exports = {(dom$_1$,ds$_1$),
(dom$_2$,ds$_1$), (dom$_3$,ds$_1$)}
P3
P4
exports = {(dom$_1$,ds$_1$),
(dom$_1$,ds$_2$), (dom$_3$,ds$_2$)}
imports = {(dom$_1$,ds$_1$),
(dom$_2$,ds$_1$), (dom$_3$,ds$_1$)}
imports =
{(dom$_1$,ds$_1$)}
imports = {(dom$_1$,ds$_2$),
(dom$_3$,ds$_2$)}
exports = {(dom$_1$,ds$_1$),
(dom$_2$,ds$_1$), (dom$_3$,ds$_1$)}
ds$_1$
ds$_2$
exports = {(dom$_1$,ds$_2$),
(dom$_3$,ds$_2$)}
RI-tree for
domain
dom$_1$:
I
ds$_1$
ds$_2$
RI-tree for
domain
dom$_2$:
ds$_2$
RI-tree for
domain
dom$_3$:
R
ds$_1$
ds$_2$

Figure 5-6. Example of a process topology, annotated with import/export definitions (the RI-trees of the data distribution scheme are also shown).

Figure 5-6 shows an example process topology annotated with import/export definitions. The example uses two data stores and three domains; respective RI-trees for the domains are also shown. For instance, we can see that, in this particular example, the application architect made process *P4* import (dom_1, ds_1) from ds_1, and (dom_1, ds_2) as well as (dom_3, ds_2) from ds_2. That means that *P4* imports all data objects of domain dom_1 and also those data objects of domain dom_3 that are stored in data store ds_2. The imported data objects become visible and accessible for the object manager in process *P4* and, consequently, also for the application code in *P4*.

In an import/export scheme, the following constraints apply:

- A data store may only export data objects that are stored in it, which means that all elements in the *exports* of a data store ds_i have to be of the form $(?, ds_i)$.
- A process may only export data objects that are imported by it, which means that the *exports* of a process *P* have to be a subset ($\subseteq$) of the union of the *imports* of all C1 connectors with *P* as client.
- A process may only import those data objects from a data source that are exported by the data source, which means that the *imports* for a C1 connector with *P* as server have to be a subset ($\subseteq$) of *P*'s *exports*.

At runtime, the import/export scheme is used for routing objects and queries as well as for determining whether an object manager has access to a sufficient set of data sources to execute a given operation. Both aspects are discussed in more detail in Section 5.8.

In many situations, a *full* import/export scheme will be used, i.e., each process imports all data exported by all of its data sources and exports all data that has been imported. However, an architect may want to restrict imports and/or exports, e.g., for security reasons, when some data is not needed by all processes, or to enforce or prevent a specific routing through the process topology.

5.4 Process Topology and Configuration (Part 3)

With our FPT architecture, the process topology of an enterprise application is not explicitly defined ("hard-coded") in the application code – instead, it is a matter of *configuration*. After a software architect has selected an appropriate process topology for an enterprise application, the topology's shape has to be described in the configuration data of the enterprise application. The FPT architecture assumes that each object manager is given a separate set of configuration data. From its configuration data, an object manager in process *P* learns details about C1 connectors where *P* takes part either as client or server. Details for a C1 connector may include, for example, an IP address and a protocol to be used – they enable the object manager to communicate with directly connected processes and data stores in the process topology. Depending on the implementation, a C1 connector can be described in the client's configuration, in the server's configuration, or in both configurations.

Besides the relevant C1 connectors and communication details for them, the configuration of an object manager *OM* in a process *P* contains the following information:

- a definition of domains (see Subsection 5.2.2) that are relevant for *OM*, i.e., are imported by it,
- an RI-tree (see Subsection 5.2.3) for each domain above,
- the corresponding *imports* (see Section 5.3) for each C1 connector from *P* to a server of *P*, and
- *P*'s *exports* (see Section 5.3)

Essentially, the configuration of an object manager describes three subsets: A subset of the enterprise application's process topology, a subset of the data distribution scheme, and a subset of the import/export scheme. Only those subsets that are relevant – i.e., visible – to an object manager are described. For instance, only those domain definitions and RI-trees that are used by an object manager have to be included in its configuration. Furthermore, only its direct clients and servers are visible to an object manager, which limits the impact of local changes on a process topology (see requirement R4 in Subsection 4.4.4).

The configuration approach described above is important for supporting custom and adaptable process topologies: Arbitrary custom DAG process topologies and thus all valid architectural configurations for our "multi-tiered enterprise applications" architectural style can be configured. In addition, when requirements change, an existing process topology can be easily adapted through *reconfiguration*.

Not only the process topology but also the data distribution scheme can also be configured. However, it should be noted that a reconfiguration of an existing data distribution scheme – e.g., major changes to an RI-tree – may require a reorganization of affected existing data in data stores so that data placement remains consistent with the new data distribution scheme.

5.5 Decoupling (Part 4)

As per by requirement R5 (see Subsection 4.4.5), our FPT architecture pairwise decouples application code, process topology, and data distribution scheme.

To decouple *application code* from the other two aspects, a strict interface is introduced between application code and its local object manager. The API hides details regarding topology and data distribution. In fact, for application code, the concrete topology, data distribution scheme, and import/export scheme are completely transparent. Application code exclusively uses the API offered by the local object manager to access data objects – the local object manager is responsible for data management and remote communication with other components in the process topology. This prevents application code from making assumptions about the enterprise application's process topology and how data is distributed. This is important because such assumptions would severely limit the ability to adapt a process topology (and/or a data distribution scheme) through reconfiguration.

How does the FPT architecture decouple the *process topology* of an enterprise application from its data *distribution scheme*? In our FPT architecture, both process topology and data distribution scheme are defined in the configuration of object managers. To decouple them, the FPT architecture treats them as orthogonal aspects: All valid combinations of topology and data distribution scheme are allowed and supported. Figure 5-7 shows an example data distribution scheme and five (of many) different process topologies that can be used in combination with it. We can clearly see that the shape of RI-trees is not required to match the shape of (parts of) the process topology. The only strict requirement is that a process topology contains all data stores defined in the RI-trees of the data distribution scheme.

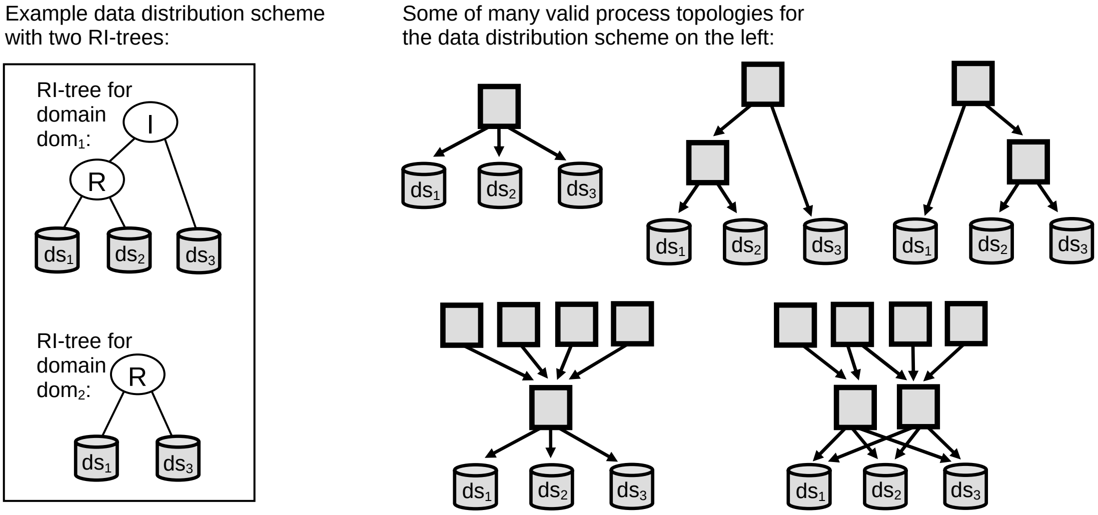

Example data distribution scheme with two RI-trees:
Some of many valid process topologies for the data distribution scheme on the left:
RI-tree for domain dom₁:
I
R
ds1
ds2
ds3
RI-tree for domain dom₂:
R
ds1
ds3

Figure 5-7. The FPT architecture supports arbitrary combinations of process topologies and data distribution schemes.

For the execution of most operations on data objects, both the topology and the data distribution scheme have to be taken into account. In order to support arbitrary combinations, object managers contain a mechanism for object and query routing that works for all valid combinations of topology and data distribution scheme. The mechanism is described in Section 5.8.

By decoupling application code, process topology, and data distribution scheme, the FPT architecture allows software architects and application developers to change one aspect mostly independently of the other two. As a consequence, the topology can easily be reconfigured to reflect changing requirements, without having to change the application code. Without decoupling, even minor changes to a process topology could have a major impact on many parts of the corresponding enterprise application and require a costly re-design and/or re-implementation. Therefore, decoupling is a key concept in the FPT architecture for supporting adaptable process topologies.

Another positive aspect of decoupling as proposed by our FPT architecture is that business logic and infrastructure concerns are strictly separated. Application developers who implement the application code can focus on the business logic part of the enterprise application because the network of object managers is responsible for the infrastructure concerns.

Up to this point, we have described the first four parts of the FPT architecture. These parts have covered mainly static aspects. The remaining three parts, which focus on runtime aspects, are described in the following three subsections.

5.6 Overview of Copies and Caching (Part 5)

In this section, we outline which elements constitute a data object and how copies of data objects are cached in object managers.

In the FPT architecture, a business data object is an abstraction that consists of the following entities:

- A unique *object identifier* for distinguishing different data object instances.
- A *type* that has a name (e.g., “Employee”) and defines a collection of attributes. Each attribute has

a name (e.g., “salary”) and a domain of valid values (e.g., all integer values between 0 and 9,999 and the *null* value).

- *Persistent state* stored in (i.e., replicated to) one or more home data stores (see Section 5.2.3). The persistent state of a data object includes the object’s unique identifier and values for all attributes defined by the data object’s type. How many and which data stores can contain state information is defined by the data distribution scheme.
- Zero or more *copies* cached in object manager components in processes of the process topology. A copy contains state information similar to the persistent state stored in a data store, but it is transient. Application code can access data objects only through copy instances in the local object manager.

In this thesis, we use the terms *business data object* or simply *data object* (e.g., “a data object is written”) whenever we want to ignore details and/or the context is clear. However, in cases where it is important to distinguish between persistent state and copies, we explicitly use either of the respective terms for more clarity.

As already mentioned in the overview section of this chapter, an object manager acts as a cache for business data objects and services local application code as well as client object managers. Object managers offer transactional access to data objects, and thus, all object managers form a multi-level hierarchy of transactional object caches for data objects that are persistently stored in data stores. A write-through approach to caching is used, i.e., (committed) writes are always propagated “down” the hierarchy into transactional data stores. Data objects are not remotely accessed like, for instance, in CORBA- or RMI-based solutions. Instead, copies of data objects are copied by value along C1 connectors of the process topology in accordance with requirement R1 in Subsection 4.4.1. This approach avoids, in many cases, fine-grained remote access and thus the system does not suffer from many of the performance problems that arise in systems based on the distributed objects paradigm.

Copies can be transmitted in both directions: For example, insert and update operations typically require propagating copies “down” the process topology (i.e., from clients to servers) to appropriate data stores whereas a query typically requires propagating copies “up” the process topology (i.e., from servers to clients) to the process that issued the query. For C1 connectors between processes and data stores, a data store adapter component is responsible for converting copies into persistent state and vice versa.

There are two ways for a copy to enter an object manager: Either it is newly created on request of local application code (e.g., when a user enters data of a new customer) or the object manager receives the copy via a C1 connector (e.g., as a result of a query). To prevent the set of cached copies from growing indefinitely, an object manager may discard copies according to some (implementation-defined) cache replacement strategy.

For all data objects that are used by application code in a process, a copy must be present in the local object manager’s cache. When application code attempts to access a copy that is not yet locally cached, i.e., when a *cache miss* occurs, the local object manager transparently loads the copy from one of its servers (data sources), which in turn may (recursively) load it from other servers (see requirement R3 in Subsection 4.4.3). Cached copies can be used by multiple transactions, i.e., an *inter-transaction caching* mechanism is employed (as opposed to *intra-transaction caching* where copies are cached only for a single transaction).

An object manager is responsible for preserving the identity of data objects as demanded by requirement R2 in Subsection 4.4.2. Each object manager contains at most one copy per data object, even if several copies of the same data object are received via the same or different C1 connectors over time. This prevents application code from ever seeing multiple instances per data object, which would otherwise violate transaction semantics.

At the same time, multiple object managers in an enterprise application may cache (i.e., contain a copy of) the same data object. Note that the FPT architecture permits some or all of these copies to become *stale*. We call a copy *stale* if and only if it does not contain the most recent committed values persistently stored in the data object's home data store(s). In many systems that use caching, some mechanisms are employed that ensure that a cache's content is always current or that at least to prevent it from becoming "too" outdated according to some measure. One way to keep caches current in a transactional system is to let a server send notifications to clients whenever changes are committed to a transactional data store [FCL97]. Notified clients can invalidate affected cache entries (with protocols like *Cache Locks* [WN90] or *Optimistic Two-Phase Locking – Invalidate* [CL91]), or update them (with protocols like *No-Wait Locking with Notification* [WR91] or *Optimistic Two-Phase Locking – Propagate* [CL91]). Keeping caches current is also closely related to the problem of materialized view maintenance, which is discussed, for example, in [BLT86], [GMS93], and [ZGHW95]. Employing mechanisms for keeping caches fresh makes also sense for FPT-based middleware. However, the FPT architecture does not prescribe a particular mechanism and treats that as an implementation issue.

An object manager may store not only copies of individual data objects but also *query results*, which are ordered sets (i.e., sequences) of copies. While caching of individual copies improves performance especially for navigational access, caching of complete query results can speed up repeated queries for the same (or a similar) set of data objects. However, except for basic consistency rules (see 5.7.5), our FPT architecture does not prescribe whether and how query caching is realized – this is considered an implementation issue.

Note that we used the term *copy* (of a data object) informally in this section. In the following section, we provide more details on copies, including public and private *versions* that may be stored in a copy instance.

5.7 Transaction Management and Concurrency Control (Part 6)

In this section, we describe the approach to transaction management and concurrency control of our FPT architecture. Concurrency control does not directly address the six key requirements identified in Chapter 4 and, at first glance, might appear as one of many implementation details. Nevertheless, there are two reasons why we consider it an important part of our FPT architecture: First, concurrency control is an essential part of any transaction processing system. Second, the approach to concurrency control determines many properties of enterprise applications and heavily influences many other parts of the architecture. Without a clear and relatively detailed description of concurrency control, many other parts of the architecture and also the FPT architecture as a whole would remain too abstract to be useful. Therefore, we treat concurrency control as a part of the architecture and, in addition, present it in more detail than the other parts.

In FPT-based systems, application code is responsible for explicitly setting transaction boundaries. For that purpose, object managers offer an API for transaction demarcation to local application code. The API contains methods for starting, committing, and rolling back transactions[1]. When a new transaction is started, it is implicitly associated with the current thread of control. All further accesses to data objects of that thread are performed as part of that transaction until the transaction is terminated. Multiple application threads may concurrently access the same object manager; each of them may start and execute a separate transaction. An object manager is responsible for intercepting all accesses to data objects (or more precisely: to copies cached in the object manager) in order to provide application code with a consistent, transactional view on data objects (see ACID properties outlined in Section 2.2).

[1] Support for suspending and resuming transactions is optional.

5.7.1 Approach to Concurrency Control

In transaction processing systems, a concurrency control mechanism is necessary to guarantee isolation properties of transactions. A standard approach to concurrency control is to use pessimistic locking [WV01] [LBK02]: Before a transaction is allowed to access a data item, it has to request and obtain a *lock* on that data item from a lock manager. The lock manager detects potential conflicts between concurrent transactions and, if necessary, resolves such conflicts by rejecting or delaying (blocking) lock requests. As a result, it can be guaranteed that transaction schedules are serializable, i.e., do not violate transaction semantics. A particularly important pessimistic locking protocol is the two-phase locking (2PL) protocol proposed in [EGLT76].

An alternative to pessimistic locking is to use an *optimistic* approach to concurrency control [KR81]. A simple model of optimistic transactions divides them into three phases:

(1) In the *read phase*, the transaction is executed. Writes performed by the transaction are stored in a buffer that is not visible to other transactions.

(2) In the *validation phase*, conflicts with other transactions are detected. If committing a transaction would lead to a transaction schedule that is not (conflict) serializable, then the transaction is aborted.

(3) In the *write phase*, all writes are copied from the buffer to the database and become globally visible.

All operations of the validation phase and write phase have to be executed as a single, indivisible unit. While pessimistic locking prevents conflicting operations, optimistic approaches allow transactions to proceed; conflicts are detected at a later time and, if necessary, trigger transaction aborts to guarantee serializable transaction schedules.

In a distributed transaction processing system where clients are allowed to cache transactional data, a concurrency control mechanism also has to ensure that caching effects do not lead to violation of transaction semantics. For that purpose, protocols are employed that are commonly referred to as *cache consistency* (or *cache coherency*) protocols. In [FCL97], a taxonomy of transactional client/server cache consistency protocols is presented[1]. In terms of that taxonomy, the FPT architecture requires a *detection-based* protocol (where caches are allowed to contain stale data) with validity checks deferred until commit. This represents a relatively optimistic approach. In particular,

- it is not necessary for clients to request locks from servers during a transaction and,
- on commit, it is not required to eagerly update all caches that contain data that was changed.

The advantage of that approach is that concurrency control requires only few and relatively coarse-grained remote interactions; fine-grained lock requests are avoided. This is important since fine-grained remote interactions could become a severe bottleneck (especially when application code performs many navigational accesses to data objects, cf. Subsection 4.4.1).

5.7.2 Isolation through Multiple Versions

When multiple transactions can concurrently access the same data it is important to guarantee a certain, well-defined level of consistency [GLPT76]. Our FPT architecture requires that reads of a transaction are repeatable and transactions see only committed values – even when multiple transactions concurrently access the same data objects within the same object manager. That means that dirty and non-repeatable reads as defined in [ANSI92] must be prevented. In the FPT architecture this is achieved by allowing a copy of a data object (cached in an object manager) to contain different *versions*. In particular, we employ several concepts that have been proposed for the read phase of an optimistic transaction in [KR81].

[1] Protocols are discussed for pure client/server architectures but, in principle, can also be employed in multi-tier architectures.

As a basis for distinguishing different versions, the persistent state information for each data object stored in a home data store contains a *persistent counter* field. When a new data object is created, its persistent counter field is set to 1 on commit. Each transaction that commits changes to an existing data object does not only write changed attribute values to the object's home data store(s) but also increments its persistent counter field(s) by one.

A copy of an existing data object cached in an object manager contains one or more *public versions* and zero or more *private versions*:

- A **public version** is a transient snapshot of a data object's committed state. A public version also contains a *counter* field, which, when compared to the persistent counter field of the current committed persistent state in a data store, tells us whether the public version is *current* (both counters are equal) or *stale* (the counter of the public version is smaller than the persistent counter). For new transactions, only the *most recent* public version (the public version with the highest counter in a copy, not necessarily the current version) is visible. For running transactions, the most recent or an older public version may be visible. A copy is stale if its most recent public version is not current.
- A **private version** stores the changes to a data object performed by a transaction that has not committed yet. A private version is based on a specific public version (optimistically locked by the transaction) that reflects the snapshot visible to the transaction on its first access to the copy. A private version is visible only to the writing transaction; this ensures that concurrent transactions never see uncommitted values. On successful commit, the private version is removed from the copy and a new public version is added to the copy. The new public version reflects the changes performed and committed by the transaction. When a transaction is rolled back, all private versions created for it are simply discarded.

The local object manager transparently intercepts all invocations on data objects and provides application code with a transactional view on data. In the following, we describe the basic mechanism for handling transactional read and write accesses to data objects. We assume that, for each active transaction, a copy is in a defined state, which is *STATE_NOT_ACCESSED* or *STATE_READ* or *STATE_WRITTEN*. For a newly created transaction, all copies are in state *STATE_NOT_ACCESSED*. The effects of read and write operations and the corresponding state transitions are defined in Table 5. In addition to version management, the table also gives information on when optimistic locks are created.

	STATE_NOT_ACCESSED	STATE_READ	STATE_WRITTEN
transaction T reads attribute A of copy C	(1) select the most recent public version V in C (2) place an optimistic lock for T on V (3) read A's value from V (4) move C to STATE_READ	(1) select the public version V in C that has been optimistically locked by T (2) read A's value from V	(1) select T's private version V in C (2) if V contains a value for A then read it (3) if V does not contain a value for A then read A's value from the public version in C that has been optimistically locked for T
transaction T writes value Val to attribute A of copy C	(1) select the most recent public version V1 in C (2) place an optimistic lock for T on V1 (3) create a new private version V2 (based on V1) for T in C (4) write Val to A in V2 (5) move C to STATE_WRITTEN	(1) select the public version V1 in C that has been optimistically locked by T (2) create a new private version V2 (based on V1) for T in C (3) write Val to A in V2 (4) move C to STATE_WRITTEN	(1) select T's private version V in C (2) write Val to A in V

Table 5. Effects of transactional read and write accesses to a copy of a data object.

In the following, we present an example that demonstrates, step by step, how concurrent transactions are isolated and how public and private versions are managed. Let us consider the following situation: A data object *E* of type *Employee* with object identifier 4C0B724E has not been recently accessed by any transaction. Therefore, no copies of *E* are present in any object manager's cache. *E*'s persistent state with attribute values (*name*="Meyer", *salary*=4500) is stored in a single data store *Y*. As *E* has been updated 41 times since its creation, its persistent counter in *Y* is set to 42. Now, let us assume that four transactions concurrently1 running within the same process access data object *E*. For each step, we describe the actions of a transaction and how the object manager internally manages public and private versions:

Step 1 - Transaction #81 starts, reads *E*'s *salary*, and writes 4800 to *E*'s *salary*.

> While executing the read operation, the object manager does not find a copy instance for *E* in its object cache. Therefore, the object manager reads *E*'s persistent state from the data store *Y* (directly or indirectly via other object managers). The object manager creates a copy *C* for data object *E* in its cache and inserts a public version 42 (i.e., with *counter*=42) with values (*name*="Meyer", *salary*=4500) into *C*. Then the object manager reads *salary*=4500 from the public version 42 and returns that value to the transaction.
>
> To execute the following write operation, the object manager creates a new private version (based on the public version 42) for transaction #81 and writes the new *salary* value 4800 to that private version. A snapshot of the current situation is depicted in Figure 5-8 (a).

Step 2 - Transaction #82 starts and reads *E*'s *salary*.

> The object manager reads *salary*=4500 from *C*'s public version 42 and returns that value to the transaction. The private version created for transaction #81 in Step 1 (an uncommitted write) is not visible to transaction #82.

[1] Transactions run in parallel but we assume that an object manager synchronizes (i.e., serializes) access operations on individual copies to protect consistency of internal data structures.

Step 3 - Transaction #83 starts, writes 5000 to *E*’s *salary*, and commits.

> First, the object manager creates a new private version (based on public version 42) for transaction #83 in *C* and then writes the new *salary* value to that private version.
>
> On commit, the object manager stores the new *salary* value from the private version into *E*’s persistent state in data store *Y*. Also, the persistent counter for *E* in *Y* is incremented to 43. Finally, the object manager removes the private version for transaction #83 and adds a new public version 43 with values (*name*=”Meyer”, *salary*=5000) to *C*.

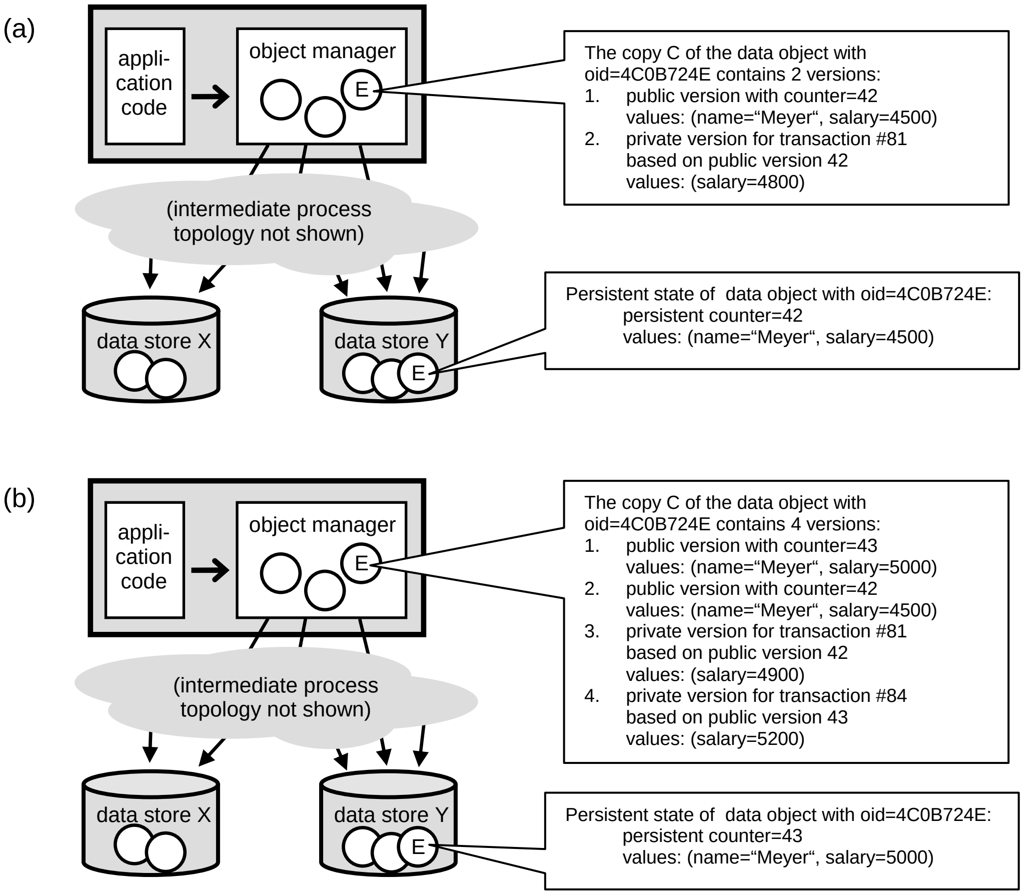
Step 4 - Transaction #84 starts, reads *E*’s salary, and writes 5200 to *E*’s salary.

> The object manager reads *salary*=5000 from the public version 43 (which is now the most recent version) in *C* and returns that value to the transaction.
>
> Then the object manager creates a new private version (based on public version 43) for transaction #84 in *C* and writes the new *salary* value 5200 to that private version.

Step 5 - Transaction #81 reads *E*’s *salary* and then writes 4900 to *E*’s *salary*.

> The object manager reads *salary*=4800 from the private version created for transaction #81 in Step 1. The new public version that has been created as a result of the commit in Step 3 is not visible to transaction #81. Instead, the transaction still sees the (uncommitted) value it has written in Step 1.
>
> Then the object manager writes the new *salary* value 4900 to the transaction’s existing private version.

Figure 5-8 (b) illustrates the situation right after Step 5 has been executed. Note that transactions #81 and #82, which are still running, are isolated from the effects of transaction #83 but now work on stale data (transaction #81 works on its private version, transaction #82 works on stale public version 42). As a consequence, both transactions will not be able to commit because their optimistic locks cannot be successfully validated (see Subsection 5.7.4). Transaction #84 is still running, too, and – if no further conflicts occur – will be able to commit.



(a)

The copy C of the data object with oid=4C0B724E contains 2 versions:
1. public version with counter=42
   values: (name="Meyer", salary=4500)
2. private version for transaction #81
   based on public version 42
   values: (salary=4800)

Persistent state of data object with oid=4C0B724E:
persistent counter=42
values: (name="Meyer", salary=4500)

(b)

The copy C of the data object with oid=4C0B724E contains 4 versions:
1. public version with counter=43
   values: (name="Meyer", salary=5000)
2. public version with counter=42
   values: (name="Meyer", salary=4500)
3. private version for transaction #81
   based on public version 42
   values: (salary=4900)
4. private version for transaction #84
   based on public version 43
   values: (salary=5200)

Persistent state of data object with oid=4C0B724E:
persistent counter=43
values: (name="Meyer", salary=5000)

Figure 5-8. Each copy of a data object contains one or more public versions and zero or more private versions. Two snapshots of a data object *E* are shown.

In the interest of brevity, we mainly focused on reading and updating *existing* data objects in this subsection. Creating new data objects and deleting existing data objects follows a similar scheme: Copies of newly created data objects contain only a single private version and no public version before they are committed. To delete an existing data object, a private version that is marked as deleted is added to a copy.

The versioning approach described in this subsection prevents dirty reads because uncommitted changes are made to private versions, which are only visible to their respective writing transactions. Non-repeatable reads are prevented because, when a transaction accesses a data object for the first time, an optimistic lock is granted for a specific public version. Other transactions that commit changes may lead to insertion of newer public versions into the copy but cannot affect the view of running transactions that already obtained an optimistic lock.

Our versioning approach isolates transactions from concurrent transactions that run within the same process. In addition, the same mechanism isolates transactions from concurrent transactions that run in other processes of the process topology. Each object manager maintains its versions independently of other object managers. The only way for a transaction T_1 running in one process to influence a transaction T_2 running in another process is to commit changes to the data stores. The changes may lead to insertion of newer public versions into copies cached in T_2's object manager. The changes may also prevent T_2 from committing successfully (see Subsection 5.7.4). Nevertheless, in any case, isolation is guaranteed.

We note that the overhead for storing multiple versions (instead of only one version) in an object manager is relatively low. For copies that have not been accessed by a currently running transaction, only one version – the most recent public version – has to be stored. When *n* transactions have accessed a given copy and are still running, the object manager has to maintain, in the worst case, 2*n*+1 versions for that copy (*n* private versions for buffering writes, *n* stale public versions, and one most recent version) and only one most recent version in the best case (all running transactions are read-only and “see” the same public version). An object manager can silently discard all versions that are older than the most recent version and, at the same time, are not locked by running transactions any more.

5.7.3 Concurrency Control and Caching

In Section 5.6, we have described caching of copies of data objects. Then, in Subsection 5.7.2, we provided more details on versions contained in a cached copy. In this subsection, we discuss the relationship between caching and concurrency control.

The versioning approach described in the previous subsection can be regarded as a variant of a *multi-version concurrency control protocol* (see [Reed83] and [BG83] for such protocols). However, compared to standard multi-version concurrency control protocols, our approach differs in the selection of the public version that is seen by a transaction on its first access to a copy of an existing data object. Ideally, the first access always goes to a public version that reflects the data object’s current persistent state (and not to a stale public version). However, since the FPT architecture uses a detection based approach and thus permits the use of stale cache entries (see Subsection 5.7.1), that cannot be guaranteed.

With the FPT architecture, the first access of a transaction *T* to a data object *E* is carried out as follows:

1. If the local object manager does not find a copy instance *C* for *E* in its local cache, then it inserts a new copy instance *C* into the cache, queries data for a public version *V* from underlying data sources, and inserts *V* into *C*. Ideally, *V* reflects *E*’s current persistent state – but that is not guaranteed.
2. The local object manager may attempt (but is not required) to load data for a public version that is more recent than the most recent public version currently contained in *C*.
3. The local object manager selects the most recent public version currently contained in *C* for *T* (see Table 7). Note that there might exist stale public versions in *C*; either because these versions have not been discarded yet or because they are needed to guarantee repeatable reads to concurrent transactions that have not committed yet.

In the second step, an object manager has a chance to improve freshness of cached data. However, even if an object manager always queries fresh data for *E* from data stores, it can never be sure to work on fresh data because concurrent transactions might be able to commit additional changes to *E* to data stores before *T*’s access operation to *C* is completed.

5.7.4 Three-Phase FPT Transactions

In FPT-based systems, transactions are employed at two different levels; we call transactions on the higher level *FPT transactions* and transactions on the lower level *data store transactions*. Only FPT transactions are visible to application code, which explicitly starts and terminates these transactions. Internally, an FPT transaction uses zero or more data store transactions to access persistent state in transactional data stores, e.g., to query data objects or write new values. While FPT transaction management is implemented as part of an object manager, data store transaction management is implemented by transactional data stores and/or their drivers.

FPT transactions are processed in three consecutive phases, which are depicted in Figure 5-9.

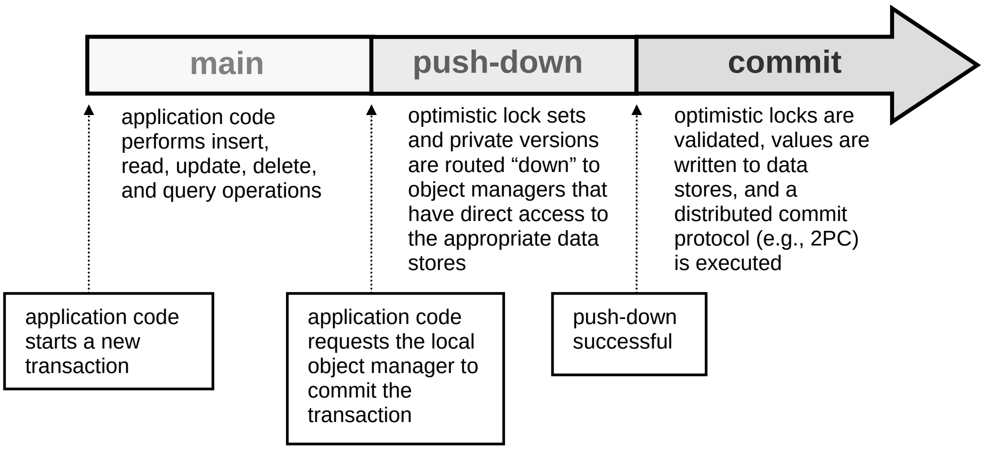

main
push-down
commit
application code performs insert, read, update, delete, and query operations
optimistic lock sets and private versions are routed “down” to object managers that have direct access to the appropriate data stores
optimistic locks are validated, values are written to data stores, and a distributed commit protocol (e.g., 2PC) is executed
application code starts a new transaction
application code requests the local object manager to commit the transaction
push-down successful

Figure 5-9. FPT transactions consist of three phases: main, push-down, and commit.

(1) In the **main phase**, an FPT transaction runs only within a single process of the process topology: The phase starts when application code in a process requests the local object manager to start a new FPT transaction. (We will call that object manager the *root object manager* for that transaction.) Then the application code performs arbitrary insert, read, update, query, and delete operations on data objects within the FPT transaction.

Write operations are buffered locally with the help of private versions as described in Subsection 5.7.2. If necessary, i.e., on cache misses, the root object manager transparently fetches data objects from underlying data sources (which in turn may consult their data sources). A fetch is performed on behalf of the root object manager and not on behalf of the FPT transactions that triggered the fetch. Therefore, no transaction context has to be propagated to remote sites at this stage. For fetching data from data sources that are transactional data stores (e.g., with SQL queries to RDBMS), an object manager uses separate, non-distributed data store transactions.

For each locally running FPT transaction, the root object manager maintains a set of locks granted to the transaction and a set of private versions implicitly created by the transaction (see Subsection 5.7.2). Locks are always optimistic – the root object manager grants them immediately without having to contact remote sites. In the main phase, a thread/transaction is only blocked while its root object manager fetches data for it.

(2) The **push-down phase** starts when application code explicitly requests the root object manager to commit the FPT transaction. Along C1 connectors, the lock set and the private version set are propagated “down” the process topology. This activity is called *object routing* and is described in more detail in Section 5.8. When the object routing process is complete, each lock and each private version has been propagated to one or more object managers with direct access to transactional data store(s). We will refer to such object managers as *routing endpoints*. Routing endpoints temporarily store locks and private versions until that data is used in the commit phase. The root object manager may also be a routing endpoint.

(3) The **commit phase** starts when the root object manager learns that the object routing has been successfully completed. First, the root object manager starts a new, distributed data store transaction, e.g., an X/Open DTP transaction (see Section 2.2). All accesses to transactional data stores described below are performed as part of that single distributed data store transaction.

Then all object managers that are routing endpoints validate all optimistic locks that have been routed to them during the push-down phase. In principle, a committing optimistic transaction can be validated against already committed transactions (*backward validation*) or against running transactions (*forward validation*) [Härd84]. Obtaining all relevant running transactions including their lock sets in a process topology can be difficult and costly because there is no centralized transaction management. Therefore, FPT transactions employ a backward validation scheme; all validations are performed against committed values in transactional data stores. More specifically, the counters of all locked public versions are tested for equality against the respective persistent counters in transactional data stores. A validation could, for instance, be "test if the persistent counter of the employee with *oid*=4C0B724E is still 42 and the persistent counter of the department with *oid*=627F7939 is still 1". When the FPT transaction has worked on stale data, the validity check fails and prevents the transaction from committing successfully.

After successfully validating its optimistic locks, each routing endpoint object manager writes private versions first propagated to it to transactional data stores. As described in Subsection 5.7.2, all updates to an existing data object require that its persistent counter is incremented by one. For read-only transactions, optimistic locks are validated but no data is written to transactional data stores.

Finally, a distributed commit protocol – such as the two-phase commit protocol (2PC) proposed in [Gray78] and [LS79] or one of its variants – is executed to terminate the distributed data store transaction. The root object manager plays the role of the transaction coordinator. When it has no direct access to participating transactional data stores, messages are routed via intermediate object managers in the topology, including all object managers that were routing endpoints during the push-down phase.

The description above covers only the case that a transaction commits successfully. However, there are several other cases, for instance:

- When application code requests to rollback an FPT transaction in the main phase, the root object manager immediately discards the transaction, its lock set, and its private version set. No remote interaction is necessary for a rollback.
- When a system error or a routing error occurs during the push-down phase, the root object manager aborts the FPT transaction. Object managers that are routing endpoints of the aborted transaction can silently discard temporarily stored data (locks and private versions) after a timeout.
- Errors during the commit phase of an FPT transaction are handled by the distributed commit protocol. When a validity check is not successful, e.g., because an FPT transaction worked on stale data, then the routing endpoint object manager that performed the check aborts the distributed data store transaction and thus the FPT transaction. This can be done, for example, by voting to abort the distributed data store transaction in the prepare phase of the 2PC protocol.

In principle, our approach to concurrency control corresponds to the original optimistic scheme described in [KR81]. The three-phase model (1. read, 2. validation, 3. commit) for optimistic transactions also applies to FPT transactions: Phase 1 is part of the FPT main phase and phases 2 and 3 are part of the FPT commit phase. Compared to the three-phase model, we arrange and name phases in a different way for FPT transactions and introduce an additional phase, the push-down phase. We decided to redefine the phases because, that way, the three phases better map to elements of our architecture. Nevertheless, the basic concurrency control mechanism remains the same.

The issue of correctness of optimistic concurrency control and forward/backward validation is discussed, for example, in [WV01]. As is shown there, both forward and backward validation (see *commit phase* described above) always produce conflict serializable histories. Note that caching of copies in object managers (see Section 5.6 and subsections 5.7.2 and 5.7.3), does not impair correctness. If an FPT transaction accesses a current version in a cache, the situation is equivalent to a non-caching scenario. If an FPT transaction accesses a stale version in a cache, the transaction will not be able to commit because it cannot be validated in the commit phase.

5.7.5 Guarantees of Consistency

What level of consistency has to be supported by FPT-based enterprise application middleware? For *individual data objects*, the versioning approach described in Subsection 5.7.2 guarantees repeatable reads and that an FPT transaction reads committed values only. FPT transactions may read stale values, e.g., when the contents of a cache is out of date, but the validation part in the commit phase of an FPT transaction (see Subsection 5.7.4) prevents such transactions from committing. Therefore, the ANSI *repeatable read* isolation level [ANSI92] is supported. The *serializable* isolation level, which prevents phantoms, is not supported because different queries performed within the same FPT transaction may be executed on data stores using different data store transactions. Consequently, even with serializable data store transactions, phantoms are not prevented within FPT transactions. In our opinion, this is not a major limitation because, in many cases, the *serializable* level is either not required or it is not used because of performance concerns. In addition, for many scenarios where the *serializable* level seems appropriate at first glance, there are alternative solutions that do not require the *serializable* level but guarantee an equivalent level of consistency. An example can be found in the case study described in Section 7.1.

For queries, consistency guarantees are much weaker than for accessing individual data objects. In FPT-based systems, data objects may be distributed among multiple data stores and may be cached in intermediate object managers. In particular, the following actions are permitted:

- Queries may be evaluated on (potentially) stale data cached within object managers.
- A query result may be constructed by combining the results of several other queries. For example, an object manager may have to query three of its data sources to answer a given query and merge the three result sets into a single query result. Each of those queries may be executed at a slightly different time and (if performed on a data store) within a separate data store transaction. However, when different query results are combined, it has to be guaranteed that each data object occurs at most once in the combined query result.

In principle, a much higher level of consistency could be realized, for example, by assuring that caches never contain stale data and/or by performing all queries within an FPT transaction directly on data stores and in the context of one large distributed data store transaction. However, a high level of consistency also tends to be expensive and often conflicts with performance and/or availability requirements [DGS85]. In many cases, transactions can live with a lower level of consistency [GW82] – especially when this leads to better performance. An example is a user of an online shopping enterprise application who enters a new address and has to select a country from a list. Usually, it is not required that the list of countries is strictly up to date in a transactional sense. In most cases, it would be sufficient to use a cached list that is updated, e.g., once per hour or once per day.

We do not impose unnecessary restrictions on FPT-based middleware or assume a specific implementation. Therefore, the FPT architecture defines only relatively weak consistency requirements – especially for queries. Implementations are free to define additional consistency guarantees, for example, serializability or timed consistency [TAR99]. Also, implementations are allowed to offer additional modes with weaker consistency guarantees, e.g., read operations that may return uncommitted changes of concurrent transactions or read operations without placing optimistic locks on the data objects read. Especially the latter option has several advantages (e.g., reduced risk of transaction aborts due to failed validation during the commit phase) and therefore is included in our proof-of-concept implementation (see Section 6.5).

5.8 Object and Query Routing (Part 7)

One of the key requirements for middleware to support adaptable process topologies is a mechanism for *routing* (as argued in requirement R6 in Subsection 4.4.6). Such a mechanism is needed to decouple process topology, data distribution scheme, and application code as described in Section 5.5, which is essential for realizing adaptable process topologies. A routing mechanism has to be implemented as part of an object manager since topology and data distribution scheme should be transparent to application code. An appropriate mechanism has to support all reasonable combinations of process topology and data distribution scheme, as described in Section 5.5. In this section, we discuss routing in more detail.

We distinguish two kinds of routings, *query routing* and *object routing*. In the following two subsections, we describe each kind of routing:

5.8.1 Query Routing

Query routing is the process of propagating a query to appropriate data stores in the main phase (see Figure 5-9) of an FPT transaction. Here we consider only two types of queries:

- An *instance query* looks up a data object by its unique object identifier. Instance queries are especially important for realizing navigational access to data objects, e.g., a *getCustomer* call performed on an *Order* data object.
- A *set query* returns all data objects that fulfill a given condition. We assume that a set query addresses only data objects that belong to one specific domain (see Subsection 5.2.2 for domains). This is not a significant limitation because a set query over *n* domains can be split into *n* disjoint set queries such that each of them addresses a single domain only.

When application code requests the local object manager (the root object manager, see Subsection 5.7.4) to execute a query that cannot be answered from the local cache, the object manager propagates the query "down" the process topology to the appropriate *target data stores*. On the way to the target data stores, intermediate object managers also have a chance to answer the query from their caches. In some situations, it is sufficient to route the query to a single target data store only. In other cases, when the data objects to be queried are distributed among multiple data stores (see Section 5.2), it might be necessary to route a query to more than one data store. In that case, multiple partial query results have to be combined into a single query result.

5.8.2 Object Routing

Object routing is the process of propagating objects (private versions of data objects and optimistic locks) to appropriate data stores in the push-down phase (see Figure 5-9) of an FPT transaction. When application code requests the local object manager (the root object manger) to commit an FPT transaction, private versions and locks are propagated from the root object manager "down" the process topology to the appropriate *target data stores*[1]. Object routing is necessary for data objects that have been

- inserted (a private version with initial values is to be routed),
- updated (a private version with changed values and an optimistic lock on a public version are to be routed),
- deleted (a private version marked as deleted and an optimistic lock on a public version are to be routed), or
- only read (an optimistic lock on a public version is to be routed)

by an FPT transaction.

Figure 5-10 depicts an example of object routing: A private version of a newly created data object is routed from the root process, which contains the root object manager, to three target data stores – ds_2, ds_3,

[1] An implementation may also choose not to directly hand over private versions and optimistic locks to target data stores in the main phase of an FPT transaction. Instead, these items can be propagated to object managers with direct access to the target data stores. Only during the commit phase of the FPT transaction are these data stores actually accessed. However, for the sake of simplicity, we will assume in this section that private versions and locks are propagated directly to target data stores.

and ds_5. Only a part of the process topology is shown in the figure. C1 connectors, processes, and data stores along which the item is routed are shown in normal style; all elements not used in the routing are shown in grey. We assume a full import/export schema (see Section 5.3). The routing shown in the figure is compliant with the RI-tree given in Figure 5-3 because the three target data stores correspond to one of the insert options defined by that RI-tree (see Subsection 5.2.3 for RI-trees). The figure shows one option for a valid routing since there are alternative paths to data stores ds_3 and ds_5. Note that we selected a relatively complex process topology and RI-tree for the example (rather than a simpler, more likely scenario) to better illustrate the main idea of routing.

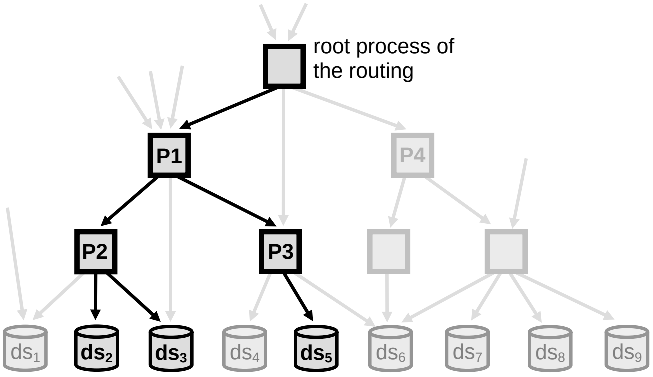


Figure 5-10. Example of object routing: A private version of a newly created data object is routed to three target data stores.

In many ways, object routing is similar to query routing – but one important difference is that, while query routing propagates a *single* item (a query) to one or more target data stores, object routing usually propagates *multiple* items (private versions and locks) to one or more target data stores. For efficiency reasons, implementations should not route each item in a separate message. Instead, items should be grouped and routed together wherever possible because, this way, fewer and more coarse-grained messages are sent. Keeping this in mind, we will focus on routing of a single item in the remainder of this section.

5.8.3 Selection of Target Data Stores and Paths

In order to route an item from a root object manager through the process topology to appropriate data stores, two main decisions have to be made: (a) A set of target data stores has to be selected and (b) paths from the root object manager to that target data stores have to be selected. Depending on the process topology, data distribution scheme, and import/export scheme, there may be no option, one option, or several options for selecting target data stores and paths through the topology. When there is no option, a routing error occurs and the query or commit request cannot be executed. When there are multiple options, the middleware has to select an “appropriate” one. Optimization criteria and mechanisms for selecting appropriate routings are implementation-dependent.

Below, we describe constraints for the selection of valid target data stores and valid paths. The constraints are described using the formal basis we developed in subsections 5.2.2 and 5.2.3.

Selection of Valid Target Data Stores for a Routing

Which set *TDS* of target data stores may be (or have to be) selected, depends on the item to be routed. In Section 5.8.1, we presented two items (instance query and set query) subject to query routing. In Section 5.8.2, we presented three items (private version of an existing data object, private version of a newly created data object, and optimistic lock) subject to object routing. For each of the five items, we give rules for selecting valid target data stores:

(a) **Instance query.** If the item to be routed is an instance query for a specific data object *o*, then it is sufficient to route the item to one of *o*'s home data stores, i.e., $TDS=\{ds_i\}$, where $ds_i \in home(o)$.

(b) **Set query.** Analogous to the *insertOptions* function given in Subsection 5.2.3, which defines all valid options of selecting home data stores for a newly created data object, we now provide a definition of a function *queryOptions*, which defines all valid options of selecting target data stores to be queried.

queryOptions is a function that maps each node of an RI-tree (i.e., each element of *V*) to a subset of 2^{DSall}. For each RI-tree RI_i, its *queryOptions* are the result of a function $queryOptions(root(RI_i))$, or short: $queryOptions(RI_i)$. *queryOptions* is recursively defined as follows:

$$queryOptions(u) = \begin{cases} \{M_1 \cup M_2 \cup \ldots \cup M_m \mid (M_1, M_2, \ldots, M_m) \\ \quad \in queryOptions(v_1) \times \ldots \times queryOptions(v_m), \\ \quad where\ \{v_1, v_2, \ldots, v_m\} = \{v \mid (u,v) \in E\}\ \} & for\ u \in INODES \\ \bigcup_{w \in \{v \mid (u,v) \in E\}} queryOptions(w) & for\ u \in RNODES \\ \{\{u\}\} & for\ u \in DATASTORES \end{cases}$$

Let us assume that a set query addresses data objects of domain dom_i, for which the RI-tree RI_i has been defined. To route the set query, a set of target data stores *TDS* must be selected such that $TDS \in queryOptions(RI_i)$. For example, for the RI-tree given in Figure 5-3, the *queryOptions* function would return $\{\{ds_2, ds_8\}, \{ds_2, ds_9\}, \{ds_3, ds_8\}, \{ds_3, ds_9\}, \{ds_5, ds_6, ds_8\}, \{ds_5, ds_6, ds_9\}\}$. That means that there are six options and the middleware could, for example, select ds_3 and ds_9 as target data stores for the set query.

Interestingly enough, *queryOptions(RI)* is always equivalent to *insertOptions(invert(RI))*, where *invert* is a function that transforms an arbitrary RI-tree into another RI-tree by exchanging all of its R-nodes for I-nodes and vice versa. This is possible because the behavior of insert operations and set query operations is symmetric with respect to RI-tree nodes: While a newly created data object has to be written to all subtrees of an R-node and to one subtree of an I-node (see Subsection 5.2.3), a set query has to be executed on all subtrees of an I-node and on one subtree of an R-node.

(c) **Private version of an existing data object.** A private version that contains new values of an existing data object *o* has to be routed to all of its home data stores (to comply with the ROWA replication scheme), i.e., $TDS=home(o)$.

(d) **Private version of a newly created data object.** To route a private version of a newly created data object that belongs to domain dom_i, one of the insert options defined by the corresponding RI-tree RI_i has to be selected by the middleware (as described in Subsection 5.2.3). That means that a set of target data stores *TDS* must be selected such that $TDS \in insertOptions(RI_i)$.

(e) **Optimistic lock.** Like an instance queries, it is sufficient to route an optimistic lock on an existing data object *o* to one of *o*'s home data stores, i.e., $TDS=\{ds_i\}$, where $ds_i \in home(o)$.

For (a), (c), and (e), we assume that the home data stores *home(o)* of an existing data object are known to the root object manager, e.g., because the implementation stores that information as part of copies, persistent state, and object references. How this is handled in our proof-of-concept implementation is described in Subsection 6.4.3.

Selection of Valid Paths to the Target Data Stores

Let us assume that a set of target data stores *TDS={ds$_1$, ..., ds$_n$}* has been selected for a routing in compliance with the rules above. Then the middleware has to choose paths for the routing. For each target data store, a continuous path of C1 connectors $c_1..c_m$ ($m\geq 1$) "down" the process topology from the root object manager to that target data store has to be selected. Let *dom$_i$* be the domain that is associated with the item to be routed. (For a set query, the domain addressed by the query is used. In all other cases, the corresponding data object's domain is used.) Which paths through the topology may be or have to be selected is restricted by the import/export scheme of the enterprise application (see Section 5.3): A path $c_1..c_m$ is only valid if an element *(dom$_i$, ds$_k$)* is included in the *imports* of each C1 connector in the path. This restriction ensures that items are only routed via C1 connectors that import appropriate data.

A simple approach to object and query routing could be realized as follows: First, the root object manager locally selects a set of target data stores and then selects valid paths through the process topology to that target data stores. After that, the root object manager initiates the routing of items along the selected paths. However, this simple approach has two important disadvantages:

- To select valid paths, a root object manager would have to be aware of the complete structure of the part of topology between its process and all target data stores. This would conflict with the limited visibility requirement (requirement R4, see Subsection 4.4.4).
- When there are multiple valid paths for an item to be routed, a routing can be optimized with regard to various parameters, for example, current and maximum load of processes and data stores, clustering of objects in data stores, bandwidth of connections, or fill ratio of data stores. To be able to optimize, each potential root process would have to obtain and track these parameters for all processes and data stores which may take part in a routing, However, that approach is not likely to scale to many client processes.

In the next chapter (in Section 6.9) we present a more sophisticated routing mechanism that is employed in our proof-of-concept implementation of the FPT architecture. There, we show how our routing mechanism avoids the disadvantages described above by selecting target data stores and paths *incrementally* and during the routing process.

5.9 Discussion

In this section, we discuss a selection of additional aspects of the FPT architecture in more detail.

5.9.1 Optimistic Concurrency Control and Contention

Our assumption is that, in typical enterprise applications, conflicts between transactions are relatively rare. Because of this assumption and in order to avoid too fine-grained communication for lock requests, we decided to use an optimistic approach to concurrency control. However, in scenarios with high contention, optimistic approaches often do not perform well, as is shown, for example, in [HSRT91]. This is especially the case when transactions access "hotspot" data, i.e., data that is frequently read and written by concurrent transactions. In the presence of many transaction conflicts, optimistic concurrency control schemes typically lead to many transaction aborts. As a consequence, transaction throughput decreases because aborted transactions typically have to be restarted/retried some time later and thus to be processed again. Frequent aborts can also be a burden for application developers, who may have to implement logic for gracefully handling aborts and restarts, and users of the enterprise application, who may have to resubmit their requests.

In enterprise applications with high contention, other or additional concurrency control mechanisms should be employed – either for all transactional data or exclusively for "hotspot" data. For example, pessimistic locking can be used. Alternatively (or additionally), semantics-based concurrency control [Garc83] [BR92], which uses semantic knowledge about access operations, can be employed.

Our proof-of-concept implementation presented in the following chapter uses the FPT architecture's standard optimistic concurrency control mechanism by default. However, in addition to the default mechanism, our implementation realizes one particular semantics-based approach (see Subsection 6.5.2). In those (exceptional) cases where hotspot data has to be accessed, application developers can explicitly request semantics-based concurrency control.

5.9.2 Simple Implementation as Design Rationale

One of the main design rationales for the FPT architecture was to allow for a simple implementation. For example:

- It is possible (but not required) to implement object managers as *stateless servers*: Except for the commit phase of an FPT transaction, an object manager never has to keep state information for remote clients between two client requests. In particular, an object manager is not required to keep track which copies are cached in or locked by which client. This simplifies state management and also makes it much easier for clients to switch from one server to another, e.g., for balancing load or when a server crashes.
- With our optimistic approach to caching and concurrency control, there is no need to ever send asynchronous messages from servers to clients. This reduces complexity and is especially suited for loosely coupled systems. Implementations may choose to use asynchronous messages from servers to clients in some situations. For example, a server process may want to notify client caches of changes to data objects. However, it is also possible to keep caches fresh using other approaches like client-initiated reloading of caches or polling changes from a server.
- Transaction management and concurrency control (especially management of versions and the backward validation on commit) can be implemented as a thin layer on top of the standard transaction processing capabilities of underlying transactional data stores. That layer is implemented as part of an object manager – there is no need to change the behavior of data stores. Even other, non-FPT-based clients can concurrently access the same data as FPT-based clients – as long as they observe to the convention that persistent counters in data stores have to be incremented on each update to a data object.

5.9.3 Fixed Home Data Stores per Data Object

As already mentioned in Subsection 5.2.3, the home data stores of a data object are selected by the middleware when the object is created. After the creating FPT transaction has committed, the data object's home data stores cannot be changed any more. That also implies that, once created, a data object cannot change its domain. For instance, in the example in Subsection 5.2.2, the date of an existing *Order* data object could not be changed from 06/12/1998 to 06/12/2003 because that would move the object from domain dom_2 to dom_3. To permit such changes, domains have to be defined such that data objects of the same type always belong to the same domain. Alternatively, a data object can be deleted and re-inserted instead of updated.

5.9.4 Import/Export Scheme versus Limited Visibility

With an import/export scheme as presented in Section 5.3, the imports of each process are described by a set of *(domain, data store)* tuples. Arguably, this conflicts with the limited visibility requirement (requirement R4, see Subsection 4.4.4) because data stores become (indirectly) visible to all processes that import their data, not only to directly connected processes. However, we believe that giving a process the names of underlying data stores does not harm modularity and adaptability much as long as the names are used only to route queries and objects at runtime. Making data store names completely transparent to all processes would be difficult because that information is needed for object and query routing.

We think that it is far more important to hide the shape of all parts of the process topology (i.e., processes and C1 connectors) from a process that are not directly connected to that process. With our approach, only the names of data stores are revealed but not the structure of a subset of the process topology.

Note that, as a consequence, changes to the process topology and/or data distribution scheme might affect not only directly connected processes but (in the worst case) all indirectly connected client processes. However, when an indirectly connected process is affected, only the import/export part in the configuration file of that process has to be adapted.

5.9.5 Data Distribution in RAID Systems

In Section 5.2, we presented our approach to data distribution. There are some similarities between data distribution in *redundant array of inexpensive disks* (RAID) systems [PGK88] [CLG+94] and our approach: RAID level 0 (striping) is a special case of our data distribution variant (c) – see the *data distribution* topology pattern described in Subsection 3.4.4. RAID level 1 (mirroring) corresponds to our data distribution variant (b). To some degree, many RAID systems also support nesting of different levels to some degree (e.g., RAID levels 10 and 0+1).

However, while there are similarities, both approaches address completely different data items at different levels: RAID systems address the distribution of low-level data blocks among multiple (usually local) disk drives. In contrast to that, our data distribution scheme addresses a much higher level by distributing data objects among different data stores. When persistently stored, a data object may be stored in one or more data blocks on disk. Also, a data block on a disk may contain data of zero or more data objects. Additionally, our approach focuses on transactional access to data, which is typically not handled at the disk/RAID level.

In fact, FPT distribution and RAID distribution are orthogonal concepts – independent of the FPT data distribution mechanism (and transparent to it), each data store server may store persistent data objects with the help of a RAID array.

5.10 Summary

In this chapter, we presented our *Flexible Process Topology* architecture, which consists of a set of concepts for realizing enterprise application middleware that supports custom and adaptable process topologies. In particular, we showed how the FPT architecture addresses all six key requirements we identified in Chapter 4. The FPT architecture is based on a network of object manager components that are responsible for infrastructure issues like distributed data management, caching, and remote communication. In an FPT-based system, construction of custom topologies is a matter of configuration; all architectural configurations of the "multi-tiered enterprise application" architectural style defined in Chapter 3 (i.e., arbitrary DAG process topologies) can be constructed through configuration. When requirements change, a software architect can easily adapt the topology of an existing enterprise application through reconfiguration. This is possible because the FPT architecture decouples process topology, application code, and data distribution scheme – each of the three aspects can be changed mostly independently of the other two. Together with the advanced data distribution scheme we proposed, (re)configurable process topologies are excellent means of constructing and adapting process topologies based on process topology patterns as described in Chapter 3 and Chapter 4.

Our FPT architecture prescribes some main concepts for achieving custom and adaptable process topologies. We restricted the FPT architecture to the very basics – in many cases it makes sense for an implementation to extend these concepts, e.g., to provide more consistency guarantees or introduce additional lock types that would allow for more concurrency between different transactions.

The FPT architecture is not a complete middleware specification because many design and implementation details are left to implementations, for example, concrete APIs, communication protocols, message formats, cache refreshment approach, query language, and additional consistency guarantees. Having discussed our middleware architecture in this chapter, we proceed to present a concrete middleware implementation based on the FPT architecture in the next chapter.