# Flexible Distributed Process Topologies for Enterprise Applications

*Christoph Hartwich*[*]
Freie Universitaet Berlin
Takustrasse 9
D-14195 Berlin, Germany
hartwich@inf.fu-berlin.de

## Abstract

Enterprise applications can be viewed as topologies of distributed processes that access business data objects stored in one or more transactional datastores. There are several well-known topology patterns that help to integrate different subsystems or to improve nonfunctional properties like scalability, fault tolerance, or response time. Combinations of multiple patterns lead to custom topologies with the shape of a directed acyclic graph (DAG). These topologies are hard to build on top of existing middleware and even harder to adapt to changing requirements. In this paper we present the principles of an enterprise application architecture that supports a wide range of custom topologies. The architecture decouples application code, process topology, and data distribution scheme and thus allows for an easy adaptation of existing topologies. We introduce RI-trees for specifying a data distribution scheme and present rules for RI-tree-based object routing in DAG topologies.

## 1. Introduction

Enterprise applications are transactional, distributed multi-user applications that are employed by organizations to control, support, and execute business processes. Traditionally, data-intensive enterprise applications have been built on top of centralized transaction processing monitors. Nowadays, these TP monitors are replaced by object-oriented multi-tier architectures where entities of the business domain are typically represented as *business objects*. To differentiate between process-centric and data-centric business objects we use the terms *business process object* and *business data object*, respectively. In this paper we take a data-centric view and thus focus on business data objects, which constitute the application's object-oriented data model. A business data object represents an entity of persistent data that is to be accessed transactionally by processes of the enterprise application, e.g., a `Customer` or `Account` object. In typical enterprise applications business data objects (or short: *data objects*) reside in and are managed by the second last tier while their persistent state is stored in the last tier. The last tier consists of one or more transactional datastores, for example, relational database management systems.

Current platforms for object-oriented enterprise applications (like CORBA [5] or Sun's J2EE [9]) provide excellent support for two-tier architectures and three-tier architectures

[*] This research was supported by the German Research Society, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316).

with ultra-thin clients, e.g., web browsers. However, many large-scale applications need more advanced structures that differ from these simple architectures, for example, to meet specific scalability or fault tolerance requirements.

In Section 2 we introduce *process topologies,* which provide an appropriate view on the distributed structure of an enterprise application. In addition, we present several well-known patterns that are used in many topologies and motivate the need for flexible DAG topologies. Section 3 discusses types of connections for building process topologies and difficulties developers typically face when custom topologies are required. An enterprise application architecture for flexible topologies that decouples application code, process topology, and data distribution scheme is outlined in Section 4. Two key aspects of the architecture are discussed in the following two sections: In Section 5 we present RI-trees for specifying a data distribution scheme. Section 6 proposes rules for RI-tree-based object routing in DAG topologies. We discuss related work in Section 7 and finally give a brief summary in Section 8.

## 2. Process Topologies

Enterprise applications can be viewed as topologies of distributed processes. Formally, a process topology is a directed acyclic graph (DAG). Nodes of the DAG represent distributed, heavyweight, operating system level processes (address spaces) which are either transactional datastores (leaf nodes) or application processes (inner nodes). Two processes may but are not required to be located on the same machine. Each edge in the DAG represents a (potential) client/server communication relationship. The concrete communication mechanism, e.g., RPC-style or message-oriented, is not important at this level of abstraction. Figure 1 shows an example of such a process topology.

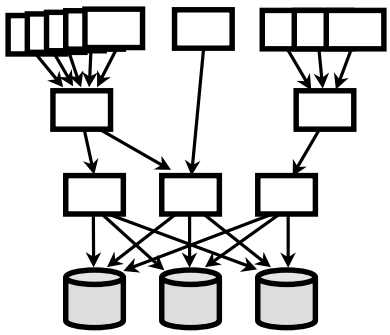

Figure 1. Example of a topology of distributed processes.

Nodes that are not connected with an edge can never communicate directly. In enterprise applications there are many reasons for restricting communication and not allowing processes to directly communicate with arbitrary other processes, for instance:

- *Security* – A sub-system (nodes of a sub-graph) is shielded by a firewall which allows access to processes of the subsystem only via one or more dedicated processes.

- *Scalability* – Highly scalable systems often require a sophisticated topology to employ services for caching, load balancing, replication, or concentration of connections. For example, allowing client processes to directly connect to datastores reduces communication overhead, but then the overall system cannot scale better than two-tier architectures, which are well-known for their restricted scalability.
- *Decoupling* – The concrete structure and complexity of a sub-system is to be hidden by one or more processes that act as a facade to the subsystem. For instance, in a three-tier structure tier two can shield the client tier from the complexities of tier three.

### 2.1. Process Topology Patterns

There are a number of patterns that can be found in many enterprise applications and that are directly related to process topology. These *topology patterns* can be viewed as high-level design patterns [2], [4], where distributed processes represent coarse-grained objects. In Figure 2 six topology patterns are depicted:

1. *Process replication* – An application process is replicated and the load produced by its clients is horizontally distributed among the replicated processes.
2. *Distributed data* – Instead of using a single datastore, data objects are distributed (and possibly replicated) among multiple datastores. This pattern facilitates load distribution and basic fault tolerance.
3. *Proxy process* – A proxy process is placed between the proxified process and its clients. Tasks are shifted from the proxified process to the proxy to achieve vertical load distribution. For instance, the proxy can cache data and process a subset of client requests without having to contact the proxified process. In addition, this pattern allows to add functionality to the services provided by the proxified process without having to modify the proxified process.
4. *Group of proxy processes* – This is a common combination of Pattern 1 and Pattern 3. By introducing a new tier of proxy processes, load is first shifted vertically and then distributed horizontally among the replicated proxy processes. This pattern is typically employed for a concentration of connections when handling too many client connections would overload a server. The replicated proxies receive client requests, possibly perform some pre-processing (e.g., pre-evaluation) and forward the requests to the server using only a view “concentrated” connections.
5. *Integration of subsystems* – Often existing data and/or applications have to be integrated. This pattern has two variants: a) Equal integration by introducing a federation layer/tier on top of the existing subsystems. b) Integration of subsystems into another (dominant) system.
6. *Mesh* – Redundant connections are added to create alternative paths of communication relationships between processes. Usually, this pattern is employed in conjunction with Pattern 1 and facilitates fault tolerance and horizontal load distribution.

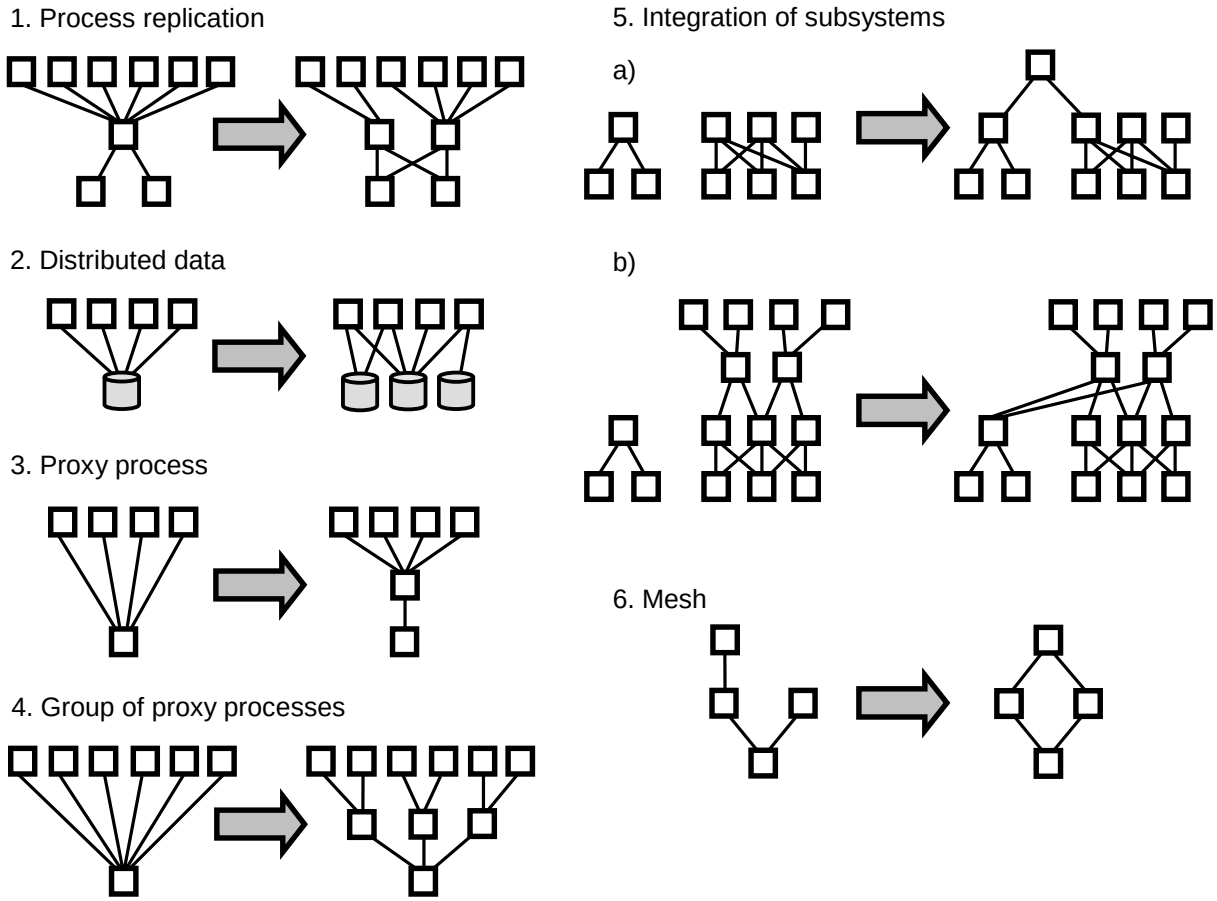


Figure 2. Typical topology patterns for large-scale enterprise applications.

## 2.2. Construction of Custom Process Topologies

The patterns presented above are used to *describe* parts of a given process topology at a high level of abstraction. In addition, a pattern can be *applied* to a process topology and hence define a transformation step. Ideally, a concrete process topology is constructed by starting with a standard topology (e.g., simple three-tier) and successively applying combinations of patterns to parts of the topology as required by the enterprise application.

What an adequate process topology looks like is highly application-specific and depends on many factors, including organizational and legal requirements, existing hardware and operating systems, underlying network structure, existing systems to be integrated, number and location of clients, typical usage patterns, and performance/scalability/fault tolerance requirements.

Many of these factors tend to change over time. For example, a successful enterprise application might have to handle a rapidly growing number of clients, or new functionality is introduced that requires a higher level of fault tolerance. To adapt a custom process topology to changing requirements, developers have to transform it by applying, removing,

and modifying topology patterns. Thus, it is desirable to have a *flexible process topology* that can easily be adapted, for example by changing configuration data. With application code designed for a specific topology, this is extremely difficult. Therefore, a flexible process topology requires that application and underlying process topology are decoupled as much as possible.

# 3. Connection Types in Process Topologies

Communication relationships in a process topology are implemented by connections. In this section we examine connections offered by current middleware and discuss problems that occur when custom topologies are built on top of them.

In general, connections can be categorized along various dimensions, for example, the degree of coupling, available bandwidth, or underlying communication protocols. For our problem analysis it is useful to define categories with regard to how business data objects are remotely accessed and represented. Based on that dimension, most connections used in current enterprise applications can be categorized into four types:

## 3.1. Type D – Datastore Connections

A Type D connection connects an application process (client, inner node) with a datastore (server, leaf node). The client accesses data objects persistently stored in the datastore. Often generic access is supported via a query interface. Type D connections are typically provided by datastore vendors, examples are ODBC, JDBC, and SQL/J. For many non–object-oriented datastores, there are adapters that provide (local) object-oriented access to Type D connections, for instance, object-relational mapping frameworks.

## 3.2. Type P – Presentation-oriented

A Type P connection connects two application processes. The client process is specialized on presentation and has no direct access to business data. Instead, the server transforms data objects into a presentation format before it sends data to the client. The transformation effectively removes object structure and identity. The client can update pre-defined fields of the presentation and post their values to the server, which has to associate the presentation-oriented changes with data objects again. Examples are HTML(-forms) over HTTP and terminal protocols.

## 3.3. Type R – Remote Data Objects

The server process of a Type R connection represents data objects as remote objects. Remote objects have a fixed location and clients can access their attribute values via remote invocations. Examples are (pure) CORBA and RMI access to EJB entity beans [8], [7].

## 3.4. Type A – Application-specific Facades for Data-shipping

A Type A connection connects two application processes. The server exposes an application-specific facade to the client. Clients cannot directly access data objects on the server (and thus the original object-oriented data model), instead they talk to the facade which implements application-specific data-shipping operations: The object states are extracted, transformed into a format for shipping (often proprietary or XML), and then

copied to the client. The following listing shows an example interface (RMI) for such an application-specific facade.

```
// for shipping values of a Customer instance
public class CustomerRecord implements java.io.Serializable {
   ...
}
...

public interface OrderSystemFacade extends Remote {
   CustomerRecord[] getCustomersByName(String pattern, int maxHits)
                  throws RemoteException, DatastoreEx;
   OrderRecord[] getOrdersByCustomer(String customerId)
                  throws RemoteException, DatastoreEx;
   OrderRecord[] getOrdersByDate(Date from, Date to, int page, int
                                 ordersPerPage)
                  throws RemoteException, DatastoreEx;
   void updateCustomer(CustomerRecord c)
                  throws RemoteException, NotFoundEx, UpdateConflictEx,
                         DatastoreEx;
   void insertCustomer(CustomerRecord c)
                  throws RemoteException, NotFoundEx, AlreadyExistsEx,
                         DatastoreEx;
   void deleteCustomer(CustomerRecord c) ...
   ... (other methods) ...
}
```

In addition to data-shipping operations the facade usually implements application-specific operations for inserting, updating, and deleting data objects. Examples of Type A connections are CORBA facades to local data objects, RMI + EJB session beans, application-specific messages sent via messaging systems, and low-level, application-specific communication via sockets.

There is a subtle but important difference between types R and A. Both can be implemented on top of distributed object middleware. However, while Type R exposes data objects as remote objects, Type A treats them as local objects and copies their state by value.

### 3.5. Limitations of Existing Connections

In general, connections of Type D and P are well-understood and have mature implementations. In fact, three-tier architectures based on P/D combinations are the first choice in many projects because of the simplicity of the approach. Unfortunately, Type P connections and their thin clients (web browsers, terminals) are not an option for many applications that need complex, sophisticated, user-friendly, and highly interactive GUIs. Also, P/D combinations are not a solution for applications that employ topology patterns and require custom topologies.

Type R connections are comfortable – but object-oriented, navigational client access to remote data objects typically leads to communication that is too fine-grained. For instance, a fat client that displays a set of data objects as a table can easily perform hundreds of

remote invocations just for displaying a single view to a single user. This results in high network traffic, bad response time, high server load, and thus limited scalability [10].

Often, this leaves only Type A for connections that cannot be covered by D or P. Unfortunately, efficient implementations of Type A connections tend to get very complex since application developers have to deal with many aspects that are generally regarded as infrastructure issues, for instance:

- Client side caching of objects,
- managing identity of objects on the client side: Objects copied by value from the server should be mapped to the same entity if they represent the same data object,
- integration of client data and client operations into the server side transaction management,
- mechanisms for handling large query results that avoid copying the complete result,
- synchronization of stale data.

Most application developers are not prepared to handle these infrastructure issues, which makes Type A connections a risk for many projects. An additional drawback is that developers have to maintain application-specific facades and keep them in sync with the server's object-oriented data model.

In principle, Types R and A can be combined, which results in a data model that consists of first class remote objects and second class value objects, which depend on their first class objects. This approach is a compromise, but it combines both the advantages and disadvantages.

While the limitations of Type A connections make it hard to build custom topologies on top of them, it is even harder to adapt such topologies to changing requirements. The insufficient separation of application and infrastructure concerns usually makes it necessary to modify and redesign large parts of an enterprise application when topology patterns have to be applied, modified, or removed.

# 4. Overview of the FPT Architecture

In this section, we outline principles of an enterprise application architecture that specifically addresses the problems motivated above.

Our *flexible process topology (FPT) architecture* specifies a data-centric infrastructure for object-oriented enterprise applications. The architecture has the following two main goals:

(a) Support for arbitrary DAGs of distributed processes as underlying topologies. For each enterprise application a custom-made topology can be designed that exactly matches application-specific requirements.

(b) Flexible topologies – topologies are easy to adapt, because application code, process topology, and data distribution scheme are decoupled.

### 4.1. Approach

The basic idea behind the FPT architecture is to place a generic *object manager* component in each application process of the topology as depicted in Figure 3. An object manager

- maintains and transparently manages connections to object managers of connected client and server processes (these connections are called *object manager connections*, or *Type O* connections),
- offers to local application code an interface for transactional, object-oriented access to business data objects, including object queries and navigational access,
- internally represents business data objects as value objects that are copied (not moved) across process boundaries,
- transparently loads, stores, and synchronizes business data objects from/with connected object managers, and
- acts as a cache for business data objects that services local application code and client object managers.

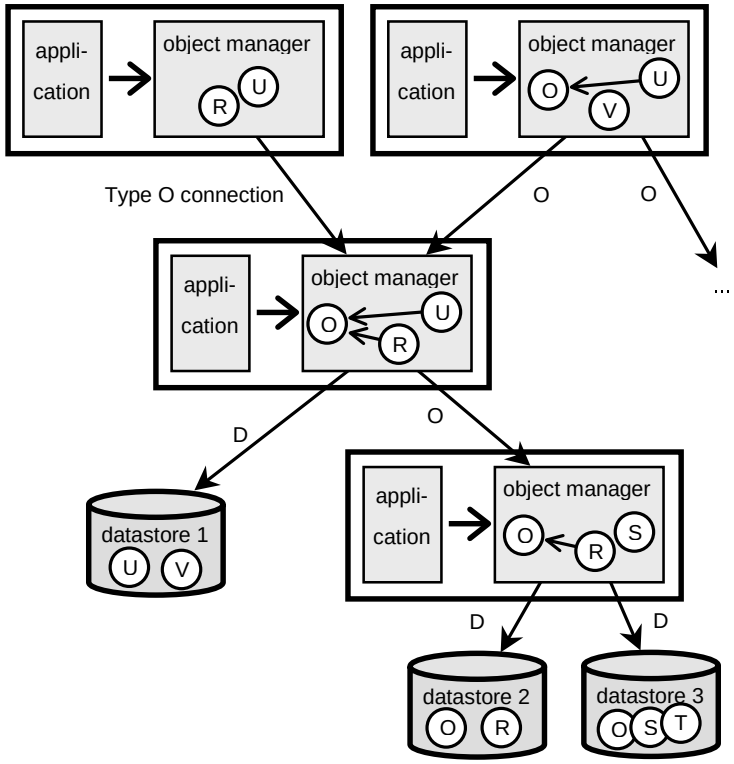


Figure 3. A process topology based on our FPT architecture: In each application process resides an object manager component that manages business data objects.

### 4.2. Decoupling

Application code is decoupled from process topology and data distribution scheme because it accesses data objects exclusively through the local object manager that transparently loads objects by value via Type O connections. Type O connections are a matter of object manager configuration and hidden from the application. Optimizations, like bulk transfer of

object state, object caching [3], query caching, or pre-fetching [1], are also handled by object managers, to clearly separate application and infrastructure issues.

The DAG of processes of an enterprise application corresponds to a DAG of object managers, which cooperate to access data objects persistently stored in datastores. A distribution scheme for data objects in datastores, including replication, is defined by *RI-trees*, which are discussed in the Section 5. Object managers are responsible for routing data objects and queries via other object managers to the appropriate datastores. To decouple data distribution from topology, object managers use a routing mechanism that takes topology and distribution scheme as parameters and produces a correct routing for all combinations.

### 4.3. FPT Transactions

Application code can set boundaries of *FPT transactions* and transactionally access data objects, i.e. perform insert, update, delete, and query operations. The local object manager fetches all objects accessed (and not present in the local cache) from connected server processes, which in turn can fetch them from their servers. Internally, each object has a version number, which is incremented whenever a new version is made persistent. Cached data objects may become stall, i.e. their version number is smaller than the version number of the corresponding entry in the datastore. Object managers keep track of each object's version number and, in addition, of one or more *home datastores* that store the (possibly replicated) object. For replicated objects, which are accessed according to a "read-one-write-all" (ROWA) scheme, it is possible that only a subset of home datastores is known to an object manager, unless it explicitly queries other datastores that may store the object. For efficient navigational access, one, a subset, or all of the home datastores can be stored as part of object references.

When a transaction changes data objects, the local object manager transparently creates new private versions in the local cache. Transaction termination is initiated by a *commit* call and consists of two phases: (1) push-down and (2) distributed commit. In phase 1 all private versions are propagated "down" the DAG topology via Type O connections to application processes that can directly access the corresponding datastores. In phase 2, when all involved datastores have been determined, a (low-level) distributed database transaction is initiated, all propagated object versions are stored to their home datastores, and a distributed commit protocol is executed, for example two-phase commit.

To exploit caching and to relieve datastores and connections of fine-grained lock requests, an optimistic concurrency control scheme is used: In phase 2, before new versions are stored, the version number of each new version is checked against the version number stored with the corresponding datastore entry to detect conflicts with other transactions.

Due to space restrictions, we can only outline the FPT architecture – many aspects that deserve and require a detailed discussion, like isolation properties, vertical distribution of business logic, cache synchronization, "hot spot" data objects, fault tolerance, and various optimization techniques cannot be addressed in this paper. Instead, we focus on our approach to decouple process topology and data distribution scheme. The following section introduces the *RI-tree,* which is the basis for separating these two concerns.

## 5. RI-Trees

We use RI-trees for specifying a replication and distribution scheme for data objects. We assume that the set of all possible data objects is partitioned into one or more disjoint *domains*, for instance

*Domain 1*: All data objects of type `Customer`
*Domain 2*: All `Order`s with date < 1/1/2002
*Domain 3*: All `Order`s with date ≥ 1/1/2002.

For each domain a separate RI-tree is specified that describes where data objects of that domain can be stored. For simplicity we will focus on a single domain and the corresponding tree only.

An RI-tree is a tree whose leaf nodes represent different transactional datastores 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. An I-node means that an object is placed *in exactly one sub-tree* of the node. Please note that R-nodes and I-nodes do not correspond to processes in a topology. In fact, RI-trees and process topology are orthogonal concepts.

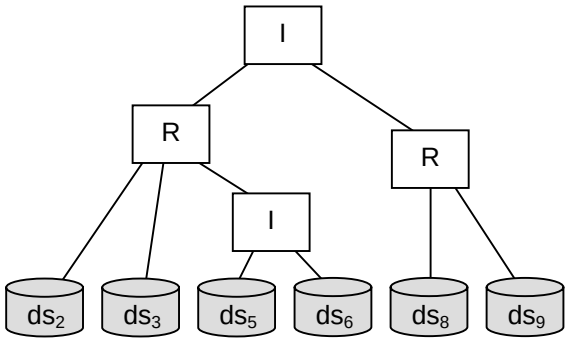


Figure 4. Example of an RI-tree.

Figure 4 shows an example of an RI-tree that specifies replication and distribution among six datastores labeled $ds_2$, $ds_3$, $ds_5$, $ds_6$, $ds_8$, and $ds_9$. The given tree specifies that there are three *options* for storing an object, each option defines a possible set of *home datastores*:

*Option 1*: home datastores 2, 3, and 5
*Option 2*: home datastores 2, 3, and 6
*Option 3*: home datastores 8 and 9

For each newly created object one option, which defines the object's home datastores, has to be selected. Then the object has to be stored in *all* these home datastores (replication). Once an option has been selected for an object, its home datastores cannot be changed. Note that partitioning of objects can be achieved either by creating domains or by using I-nodes (or both). While domains need a pre-defined criterion based on object type and/or attribute values, I-nodes are more flexible and allow new objects to be inserted, e.g., following a load balancing, round robin, random, or fill ratio based scheme.

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 datastores $DS_{all} = \{ds_1, ds_2, ..., ds_{maxds}\}$. T's inner nodes are $RNODES \cup INODES$, its leaf nodes are DATASTORES. Now we introduce a function options that maps each element of $V$ to a subset of $2^{DS_{all}}$. For each RI-tree $T$ its options are the result of a function options(root(T)), or short: options(T). options is recursively defined as follows:

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

The following example shows the formal representation of and options for the RI-tree from Figure 4:

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

# 6. Object Routing

Having introduced the RI-tree formalism in the previous section, we can now discuss how RI-trees and process topology are related and how objects can be routed.

## 6.1. Imports and Exports

For each data domain $dom_i$ a separate RI-tree $T_i$ defines which datastores are potential home datastores for objects of the given domain. We propose a simple import/export scheme for process topologies that helps us to determine which domains and datastores can be accessed by which processes.

Each process in the DAG is assigned an attribute labeled *exports* and each communication relationship (edge) is assigned an attribute labeled *imports*. Both attributes contain a set of tuples of the form (*domain*, *datastore*) as values. Each tuple represents the ability to access objects of a *domain*, which are stored in a specific *datastore*. Access can be either direct or indirect via other processes. Figure 5 illustrates an example of an import/export scheme for

three domains in a process topology with two datastores. In an FPT enterprise application import/export rules for a process and its connections to server processes are part of the configuration data of that process.

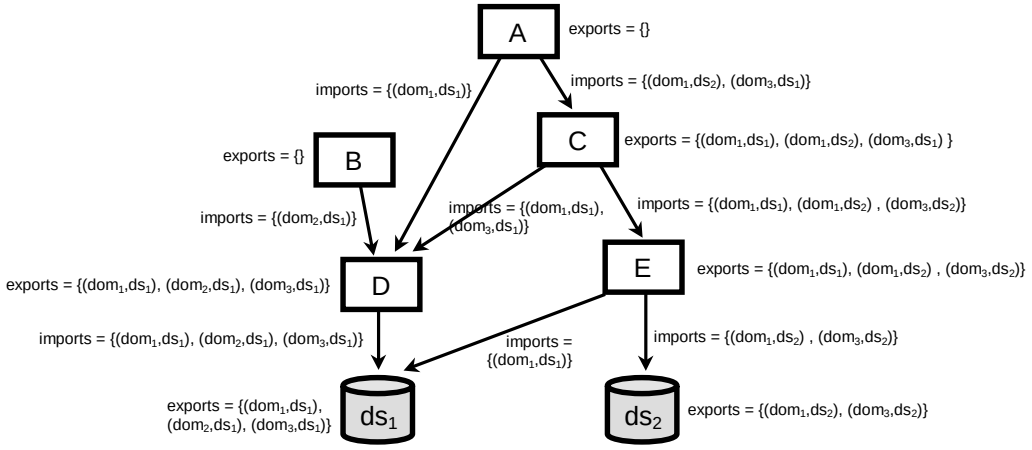


Figure 5. Example of an import/export scheme for process topologies.

For each application process p and domain $dom_i$ we define canAccess(p, $dom_i$) := { $ds_k$ | ($ds_k$, $dom_i$) is element of the union of all imports of p's outgoing edges}. canAccess is the set of datastores that can be accessed (directly or indirectly) by p for a given domain $dom_i$.

### 6.2. Formal Definition of Object Routing

When an application process root transactionally accesses a set of objects, i.e. performs insert, update, delete, and query operations, all objects in the set are copied by value to root's cache and new versions are created locally for changed objects. On commit all new versions have to be propagated "down" the DAG topology via client/server connections and applied to datastores. A changed object o is either

- *new*, i.e. it has been inserted by the application but not yet persistently stored, or
- *existing*, i.e. an option home(o) has already been selected and a previous version of the object has been stored in the corresponding home datastores.

New objects have to be assigned an option home(o) and have to be stored in all home datastores defined by that option. For existing objects (updated or deleted) all home datastores have to be identified and accessed. The process of propagating changed objects through the topology is called *object routing*.

Formally, an object routing R for an object o of domain $dom_i$ is defined by an application process root, a set of datastores rtargets and a set of paths rpaths. For each datastore ds ∈ rtargets there is a path in rpaths that starts at root and ends at ds. Each path in rpaths ends at a datastore ds ∈ rtargets and all edges in the path must contain ($dom_i$, ds) in their *imports*.

For an existing object o we call an object routing *correct* iff rtargets = home(o), i.e. the object is routed to all its home datastores. For a new object o we call an object routing correct iff rtargets $\in$ options($T_i$), i.e. a valid option is selected and the object is routed to all home datastores of that option.

### 6.3. A Conceptual Framework for Object Routing

A simple approach for object routing would let the root process locally pre-calculate an object routing R on commit. This simple approach has two important disadvantages:

- Instead of knowing only processes that are directly connected, the root process would have to be aware of the structure of the complete underlying topology.
- An object routing can be optimized with regard to various parameters, for example, current and maximum load of processes and datastores, clustering of objects in datastores, bandwidth of connections, or fill ratio of datastores. Each potential root process would have to obtain and track these parameters of other processes, which does not scale for many client processes.

Instead of suggesting one of countless possible optimization techniques for object routing we present a conceptual framework that is independent of specific optimization parameters and decisions. To preserve the autonomy of subsystems we only require each process to know its direct server processes and all RI-trees. We propose that an object routing is calculated incrementally during the propagation process and decisions are deferred for as long as possible/necessary. Each process recursively delegates routing decisions to its server process(es) unless RI-tree, import/export schema, and topology require a local decision.

Each application process p involved in an object routing performs a *local routing* (see Figure 6) as follows:

1. With a *routing message* a client c propagates an object o of domain $dom_i$ together with a set of datastores (represented by ids) homeCandidatesIn to p.
2. Let p have s servers $child_1$..$child_s$. p uses a *local routing function* to calculate s sets of datastores $homeCandidatesOut_1$..$homeCandidatesOut_s$, one for each server of p. Each set $homeCandidatesOut_k$ must be a subset of homeCandidatesIn. In addition, for each datastore ds in $homeCandidatesOut_k$ there has to be an element ($dom_i$, ds) in the *imports* of edge (p, $child_k$).
3. For each non-empty set $homeCandidatesOut_k$ the following is done:
   - if $child_k$ is an application process then p propagates o and $homeCandidatesOut_k$ to its server $child_k$ with a routing message. Each $child_k$ in turn performs a local routing for o and takes $homeCandidatesOut_k$ as its homeCandidatesIn input parameter.
   - if $child_k$ is a datastore then p temporarily stores o and $child_k$ for phase 2 (in which o's version number is checked and o is stored to $child_k$ – see Subsection 4.3).
4. Replies for routing messages sent in 3. are collected and a reply message is sent to client c.

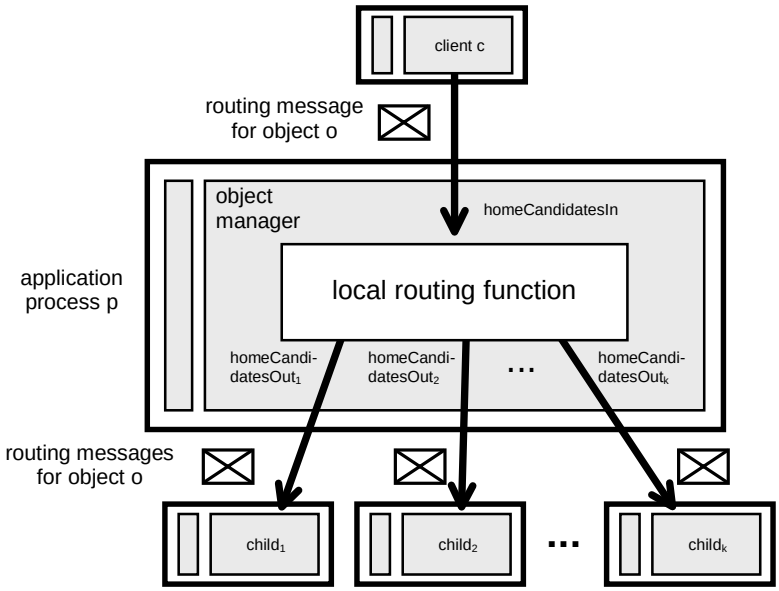


Figure 6. Each application process performs a local routing using a local routing

The local routing function can be application-specific and even process-specific as it is independent of routing functions of other processes. A root process skips step 1: o is determined by the current transaction and homeCandidatesIn is set to {ds | there is an e ∈ options($T_i$) with ds ∈ e ∧ e ⊆ canAccess(p, $dom_i$)}. Constraints for root processes and rules for local routing functions depend on whether o is new or existing:

**Routing a New Object**

A root process requires that options($T_i$) contains an element e with e ⊆ canAccess(p, $dom_i$). A correct routing can be guaranteed when the local routing functions of all involved processes observe the following rules:

(a) $homeCandidatesOut_1$..$homeCandidatesOut_s$ are pairwise disjoint.

(b) For each pair (x, y) of datastores (x ∈ $homeCandidatesOut_m$, y ∈ $homeCandidatesOut_n$, and m≠n) their lowest (deepest) common ancestor in the RI-tree $T_i$ is an R-node.

(c) Let del := { ds | ds ∈ homeCandidatesIn, but ds is not included in the union of all $homeCandidatesOut_1$..$homeCandidatesOut_s$}. For each ds in del there must be a node x in the RI-tree $T_i$ so that
- x is ancestor of ds or x = ds,
- x is child of an I-node y,
- all descendants of x (including x itself) that are included in homeCandidatesIn are also included in del,
- and there is at least one element in homeCandidatesIn that is not included in del and is a descendant of y in $T_i$.

The example in Figure 7 shows an RI-tree and two different correct routings (both with process A as root) in the same topology. For simplicity we assume a *maximum import/export scheme*, i.e. all datastores export $dom_i$, everything an application process can export is exported, and everything an application process can import is imported. Solid arrows between application processes indicate the path of routing messages in transaction phase 1. A solid arrow from an application process to a datastore indicates access to that datastore in phase 2. Dotted arrows represent edges not used in the routing. The value of the parameter homeCandidatesIn for each involved application process is shown above the rectangle that represents the corresponding process.

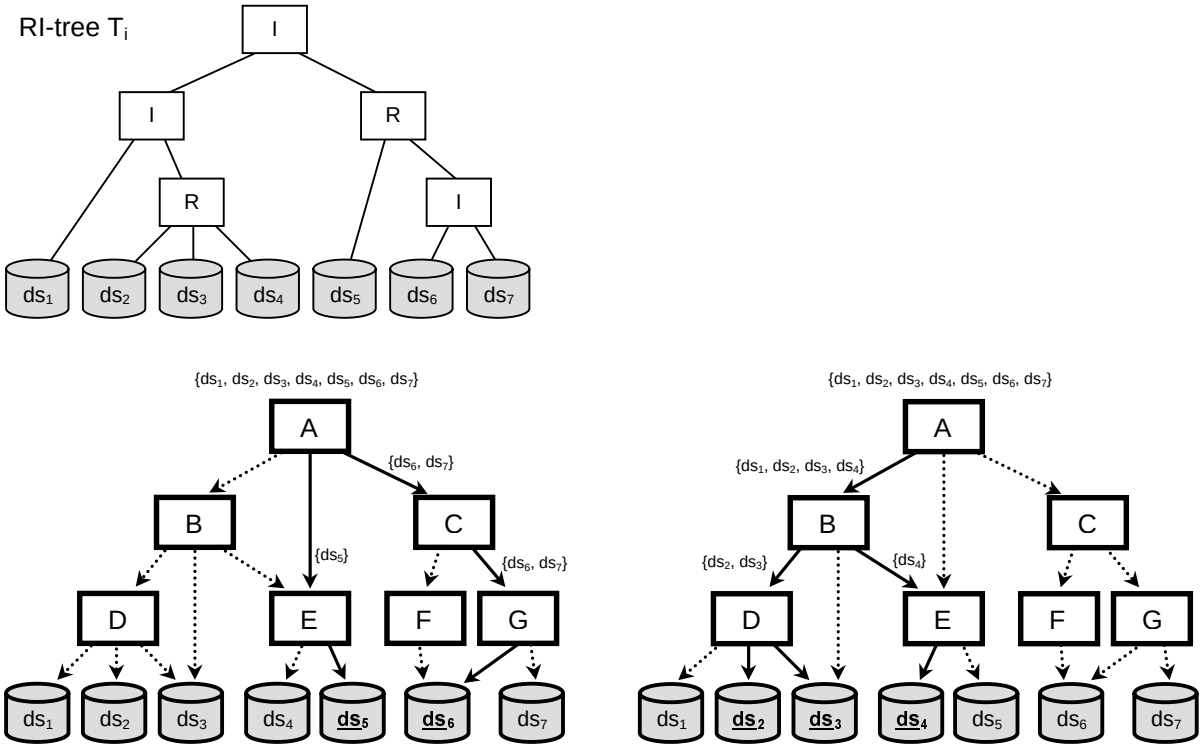


Figure 7. Two different correct routings for the same newly created data object.

Please note that the examples in Figure 7 and Figure 8 are not real world examples. Their RI-trees are complex and intentionally do not match the topology so that they clearly demonstrate how object routing works. In realistic scenarios we expect typical RI-trees to have only one or two levels of inner nodes.

#### Routing an Existing Object

A root process requires that $home(o) \in canAccess(p, dom_i)$, although $home(o)$ itself may not be known to the root process and other processes. We suppose that $p$ knows a set $homeConfirmed \subseteq homeCandidatesIn$, which contains a subset of $o$'s home datastores (see Subsection 4.3). homeConfirmed may but is not required to be included as a

parameter in routing messages to server processes. A correct routing can be guaranteed when the local routing functions observe the following rule – in addition to rules (a) and (b) given for new objects:

(c) Let del := { ds | ds ∈ homeCandidatesIn, but ds is not included in the union of all $homeCandidatesOut_1$..$homeCandidatesOut_s$}. For each ds in del there is no element e ∈ options($T_i$) so that ds ∈ e ∧ homeConfirmed ⊆ e.

In some cases, especially when o is a replicated object and $T_i$ contains I-nodes, an application process may face the situation that it cannot immediately produce a correct routing. In that case, one or more *probe queries* are sent to server processes (sequentially or in parallel) to confirm or rule out that datastores from homeCandidatesIn \ homeConfirmed are home datastores of o:

- Each confirmed datastore $ds_{conf}$ is added to homeConfirmed.
- For each datastore $ds_{notfound}$ that is reported not to be a home datastore, $ds_{notfound}$ and all elements of { ds | ds ∈ homeCandidatesIn, ds≠$ds_{notfound}$, the lowest (deepest) common ancestor x of $ds_{notfound}$ and ds in $T_i$ is an R-node, and all nodes between $ds_{notfound}$ and x are either I-nodes with only one child or R-nodes} are removed from homeCandidatesIn.

Eventually, a correct routing can be produced – at the latest when homeCandidatesIn = homeConfirmed. Probe queries can be expensive, especially when options($T_i$) contains many options and each of these options contains many datastores.

Figure 8 illustrates a scenario for routing an existing object where A, the root process, has to perform a probe query first to produce a correct routing. The existing object's home datastores are $ds_2$, $ds_3$, $ds_5$, $ds_6$, but only $ds_3$ is initially known. Again, we assume a maximum import/export scheme.

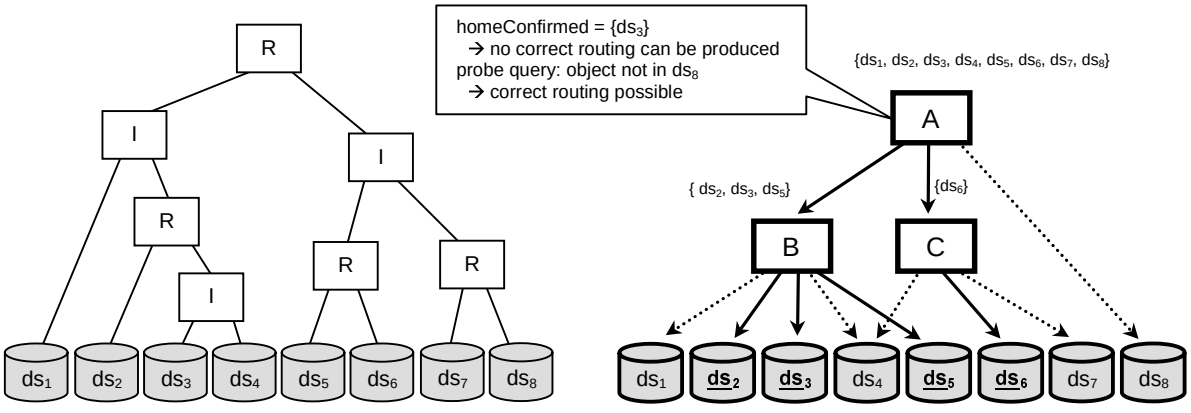


Figure 8. A correct routing for an existing data object.

So far, only routing of a single object has been discussed. When a transaction accesses multiple objects then complete sets of objects are to be routed. Often it is possible to route

them together using only a few coarse-grained messages for inter-process communication. On the way "down" the DAG topology sets may be split into smaller subsets that are treated differently and routed individually. Ideally, all objects can be routed as one set to a single datastore – in that case no distributed commit protocol is required.

When Pattern 6 (mesh, see Subsection 2.1) has been applied then a process may receive two or more routing messages for the same data object under certain circumstances. This is not a problem, since all these messages can be processed independently. To integrate two systems with Pattern 5, the data distribution schemes of both systems have to be merged: The new scheme simply consists of all domains and RI-trees of both data distribution schemes (union). Special treatment is needed for domains that exist in both systems (for example, each system stores `Customer` objects and both sets have to be integrated). Provided there are no duplicates or these are removed first, a new RI-tree is constructed by placing an I-node as a new root on top of both previous RI-trees for the domain.

## 7. Related Work

To our knowledge, neither data-centric custom process topologies nor the flexibility aspect of these topologies have been discussed in the context of enterprise applications before.

The distributed objects paradigm, especially CORBA [5], offers the powerful concept of transparencies, including access and location. But it does not sufficiently address the fact that in many topologies inter-process access to data objects is restricted and most processes are only allowed to directly communicate with a small subset of other processes in the topology (see Section 2).

TopLink [11], an object-relational mapping framework, supports so called *remote sessions* that are similar to Type O connections (see Subsection 4.1) but can only be employed to connect TopLink clients to TopLink servers. Since client and server roles are fixed and only one server per client is supported, remote sessions are rather limited and cannot be used to build arbitrary process topologies.

Research in the context of peer-to-peer networks focuses on flexible topologies and routing (for example [6]), but usually objects are coarse-grained and access is both read-only and non-transactional.

## 8. Summary and Conclusion

We view enterprise applications as topologies of distributed processes that access business data objects persistently stored in transactional datastores. We explained why many large-scale applications need custom topologies to address their application-specific requirements, e.g., regarding scalability and fault tolerance. There are several well-known topology patterns for custom topologies, which, when combined, lead to arbitrary DAG topologies. We categorized connections offered by current middleware and explained why it is difficult to build DAG topologies on top of them.

Then we outlined principles of our FPT architecture for object-oriented, data-centric enterprise applications with arbitrary DAGs as underlying process topologies. The

architecture is based on a network of object manager components which cooperate to access data objects. In contrast to existing middleware topologies are *flexible*, i.e. easy to adapt to changing requirements, because application, topology, and data distribution scheme are decoupled and can be specified independently. We introduced RI-trees for specifying a data distribution scheme (including replication) and a conceptual framework for RI-tree–based object routing in DAG topologies.

The framework does not define a specific routing strategy, instead only the general approach and constraints for correct object routing are given. A local routing function can be specified separately for each node in the topology. These functions typically have a large set of routing possibilities from which they can select one according to their (private) optimization criteria. This allows a broad range of application-specific optimization techniques to be integrated. For all topologies and RI-trees a correct routing can be produced, provided that the corresponding domains/datastores can be reached by a root process. The enterprise application can even tolerate the loss of redundant processes and connections at runtime when the corresponding tuples are removed from the import/export scheme.

The fact that arbitrary topologies and RI-trees are supported does not mean that all combinations necessarily lead to efficient systems. For example, extensive use of replication always has an impact on performance. Selecting *efficient* topologies and distribution schemes for an application is still a challenging task and up to software architects. But once these decisions are made, our architecture significantly simplifies the development of an enterprise application and improves maintainability by factoring out the topology aspect.

We view the FPT architecture as a set of concepts that can either be used for extending existing enterprise application frameworks or be used as a basis for new frameworks. Currently, we follow the latter approach and are working on a framework based on Java, RMI, relational databases as datastores, and xa transactions. A first prototype has already been completed. Future work includes a detailed performance and scalability analysis, case studies, and a broad range of optimizations for queries, routing, and synchronization in flexible DAG topologies.

## References


[1] P.A. Bernstein, S. Pal, D. Shutt, “Context-based prefetch - an optimization for implementing objects on relations“, VLDB Journal vol 9, no. 3, pag 177-189, 2000

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, “Pattern-Oriented Software Architecture - A System of Patterns“, Wiley and Sons, 1996

[3] M. Franklin, M. Carey, M. Livny, “Transactional Client-Server Cache Consistency: Alternatives and Performance”, ACM Transactions on Database Systems, vol 22, no. 3, pag 315-363. September 1997

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, ‘‘Design Patterns, Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995

[5] Object Management Group. “The Common Object Request Broker: Architecture and Specification”, Rev. 2.6, December 2001, http://www.omg.org

[6] A. Rowstron, P. Druschel, “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems”, in Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), November 2001

[7] Sun Microsystems. “Enterprise JavaBeans Specification, Version 2.0”, Final Release, August 2001, http://java.sun.com/products/ejb/2.0.html

[8] Sun Microsystems. “Java Remote Method Invocation”, http://java.sun.com/products/jdk/rmi/

[9] Sun Microsystems. “Java 2 Platform Enterprise Edition Specification, v1.3”, July 2001, http://java.sun.com/j2ee/

[10] J. Waldo, G. Wyant, A. Wollrath, S. Kendall, “A Note on Distributed Computing”, Sun Microsystems. Technical Report 94-29, November 1994, http://www.sun.com/research/techrep/1994/abstract-29.html

[11] WebGain. “TopLink 4.0”, 2001, http://www.webgain.com/products/toplink/