Chapter 8: Conclusions

In this thesis, we examined a family of applications that are commonly referred to as *enterprise applications*. Enterprise applications are transactional, distributed multi-user applications. Today, they are typically built on top of object-oriented multi-tier architectures. In practice, enterprise applications are important because they play a crucial role in many organizations.

In our thesis, we focused on distribution and data management of enterprise applications from a middleware perspective. We explained that, in many cases, the distributed structure of an enterprise application directly influences important non-functional properties such as performance, scalability, availability, and security. We motivated that, for demanding enterprise applications, we need *custom* distributed structures to address application-specific requirements. In addition, the distributed structure should be easy to *adapt* during the life cycle of an enterprise application because requirements often change. However, with current enterprise application middleware, neither custom distributed structures nor adaptation are well supported. Typically, distribution is decided at an early design stage of a software development project, is constrained or determined by the middleware selected for an enterprise application, and is difficult and expensive to change later on.

We proposed a new architecture for enterprise application middleware. With our approach, software architects can design the distributed structure of enterprise applications at an architectural level, based on topology patterns, and purely according to application-specific requirements. In addition, architects can use our middleware as a basis for implementing their designs. Moreover, with our middleware, the distributed structure of an enterprise application can easily be adapted at a later stage (again based on topology patterns) to address changing requirements. Instead of limiting choices for distribution, our middleware architecture gives software architects much more flexibility in the initial design of distribution and its future evolution. Among other things, this property makes our approach particularly suited for those development projects where enterprise applications start as tiny systems and then, successively, are scaled up to larger systems.

Note that good support for custom and adaptable process topologies is a feature that cannot easily be added to existing enterprise application middleware products, e.g., J2EE-compliant application servers. An efficient implementation with a convenient level of transparency requires a consistent set of carefully adjusted design decisions – from the highest architectural level to low-level implementation details. Therefore, we did not adapt an existing middleware but developed a solution following a strict top-down approach. First, we defined an architectural style as a solid conceptual foundation. Then, successively, we identified key requirements, developed our FPT architecture, and finally created an implementation. We hope that the concepts that constitute the FPT architecture will inspire future middleware approaches, standards, and products.

Our architecture and our middleware have been developed in particular for object-oriented enterprise applications. However, of all object-oriented concepts, only *object identity* is really essential from an architectural point of view. While our proof-of-concept middleware framework is well-integrated into an object-oriented platform (Java), our architecture could, in principle, also be utilized in non-object-oriented environments – provided that shared data structures and network communication are supported.

The following Section 8.1 gives a summary of the main contributions of this thesis. Then, in Section 8.2, we outline future work.

8.1 Summary of Contributions

In this section, we summarize the three main contributions of this thesis.

Contribution 1: Architectural Style and Pattern Language

We defined a new architectural style for multi-tiered enterprise applications. The new style focuses on an enterprise application's distributed structure and combines elements of three existing architectural styles. The architectural style constitutes the foundation of this thesis because it defines relevant terms and provides a well-structured, coarse-grained view on the distributed structure of enterprise applications. Moreover, it defines a design space for enterprise applications from the viewpoint of distribution. In particular, the architectural style is a conceptual basis for our FPT architecture and our proof-of-concept implementation.

An important element of our style is an enterprise application's process topology, which we defined as a directed acyclic graph of processes, datastores, and client/server communication relationships. We explained that the design of an appropriate process topology for an enterprise application is a complex design task because it typically involves many conflicting driving forces that, in addition, are not easy to quantify in many cases. In practice, such complex design decisions are made by software architects. To help software architects to design appropriate process topologies, we provided a catalogue of design patterns (a pattern language) for constructing and adapting process topologies.

Contribution 2: Requirements for Enterprise Application Middleware

For realizing object-oriented enterprise applications with custom and adaptable process topologies, middleware support is essential. We identified and discussed six key requirements for enterprise application middleware. The requirements address distributed management of data objects, routing, and visibility constraints in process topologies. Also, it is an important requirement to pairwise decouple process topology, application code, and data distribution scheme so that each aspect can be changed mostly independently of the other two. We explained why existing enterprise application middleware does not fulfill our requirements. We pointed out that not only the development of enterprise applications with large, complex process topologies suffers from those limitations of existing middleware. Non-compliance with our requirements also has a negative impact on many projects that rely on much simpler distributed structures.

Contribution 3: FPT Architecture including Proof-of-Concept Implementation and Evaluation

We developed the *Flexible Process Topology* (FPT) architecture, which consists of concepts for enterprise application middleware that supports custom and adaptable process topologies. The architecture is based on a network of object manager components that are deployed in the processes of an enterprise application and collaboratively provide data management services to the application code. We discussed details of the architecture, including data distribution, decoupling through configuration, object caching, optimistic concurrency control, and routing. Our architecture addresses all six requirements for enterprise application middleware we previously identified. With our architecture, the distributed structure of an enterprise application can be defined and adapted through (re)configuration and without affecting its implementation. This gives developers more flexibility in constructing and customers more flexibility in deploying their enterprise applications.

As proof-of-concept, we implemented a middleware framework based on the concepts of our FPT architecture. The prototype supports arbitrary DAG process topologies and all process topology patterns we presented. The middleware framework serves as an example of how the concepts of the FPT architecture fit together and map to a concrete middleware implementation. Based on our implementation, we provided a detailed evaluation. We implemented a sample enterprise application on top of our framework and demonstrated that constructing custom process topologies and then adapting them in several evolutionary steps later on is straightforward. Also, we provided a performance evaluation by means of several different scenarios that cover various access patterns and process topologies. We showed that our middleware framework introduces only little overhead, provides good performance (even with hotspot data), and that an object manager on a single server machine can efficiently handle hundreds of concurrent clients.

8.2 Future Work

In this thesis, we have made several contributions regarding the distributed structure of enterprise applications from a middleware point of view. However, in many areas, further research is required. In this section, we outline the most important directions for future work.

Empirical Study of Process Topologies and Topology Patterns in Existing Enterprise Applications

In practice, enterprise applications with many different process topologies are used. In some cases, developers restrict themselves to process topologies directly supported by the underlying enterprise application middleware. In other cases – especially in the area of enterprise application integration – complex custom process topologies with many different in-house developed connectors are used.

How many enterprise applications use simple standard process topologies; how many use complex custom topologies? How many and what evolutionary steps can be observed in what types of enterprise applications? In what situations are process topologies typically adapted and what are the estimated costs? How does the age or success of an enterprise application affect the complexity of its process topology? How many and what process topology patterns can be identified in existing enterprise applications and what combinations of patterns are typically used? It would be insightful to make a *quantitative* analysis of enterprise applications, process topologies, patterns, and evolutionary steps. An empirical study based on questionnaires and/or interviews could yield concrete numbers. In such a survey, a large number of software architects from different projects should be interviewed.

Cyclic Process Topologies

In this thesis, we restricted the discussion to cyclic process topologies (see Subsection 3.2.2). We defined our "multi-tiered enterprise applications" architectural style explicitly for acyclic process topologies because cyclic topologies increase complexity and many typical enterprise applications do not require them. However, in some cases, cycles might be desired. For example, cycles may occur when two different systems are integrated in such a way that two processes mutually exchange data.

In principle, cyclic process topologies are not incompatible with our approach. To integrate them into the approach, changes have to be made to the architectural basis, to concepts of the FPT architecture, and, consequently, also to our implementation. Among other things, a new architectural style has to be defined – possibly as a variant of our "multi-tiered enterprise application" style. A straightforward approach would be to still model remote communication as client/server communication relationships (C1 connectors), i.e., with directed edges, but to allow cycles. At the level of the FPT architecture, there are two options for incorporating cycles:

(1) In import/export schemes (see Section 5.3), cyclic import/export relationships are permitted (along C1 connectors) but only if there is no cycle with respect to the same domain. That would, for example, permit a situation where a process *A* grants process *B* access to customer data objects and *B* grants *A* access to order data objects. This option permits cycles to a limited extent.

(2) In import/export schemes, all kinds of cycles are permitted, except for cycles with respect to the same (*domain*, *data store*) tuples. This option provides more freedom than option (1) because it would even allow two processes to mutually import/export data from the same domain. For example, both processes could share customer data objects without partitioning them into different disjoint domains first.

Option (1) is straightforward to integrate into the concepts of the FPT architecture and into our implementation. Option (2) requires slightly more work because it is necessary to adapt the routing mechanism to prevent objects and queries from being indefinitely routed and so never arriving at datastores.

Extension of the FPT Architecture and Implementation

In this thesis, we proposed our FPT architecture as a basis for building enterprise application middleware. We also presented a proof-of-concept implementation. In both cases, we focused on a few core concepts to realize custom and adaptable process topologies. However, there are several features that are important in practice and have not been covered (in detail) in our work:

- *Transactional RPCs.* We focused on data-centric enterprise applications with business logic that directly accesses transactional data objects. However, there are cases where developers might want to combine direct access to data objects with an RPC-based communication style. For example, a client could query item data objects from a server process *S* and then invoke a remote procedure *createOrder* implemented by *S* with one or more item data objects as parameters. Thereby, the client could delegate the task of creating a new order data object instance to *S*. Application-specific RPCs between processes in a process topology and data object references as parameters are possible with the FPT architecture and our implementation, but RPCs are not transactional, i.e., FPT transactions are not propagated as is the case, for example, with CORBA's object transaction service.

  To fully integrate transactional RPCs into our architecture, cross-process management of data objects (see sections 4.3 and 4.4), currently applied to communication between object managers only, has to be extended to RPC communication. A trivial solution for realizing a transactional RPC from client *C* to server *S* is to copy to *S* the context of the current transaction and all public and private versions of data objects accessed by *C* within the transaction. When the RPC returns, all new public versions and all new or changed private versions are copied back from *S* to *C* and *S* immediately forgets about the transaction. However, for transactions that access many data objects and perform many RPCs, this solution would introduce a significant overhead because many versions are copied over the network. Possible optimizations include (a) lazy synchronization of versions between *C* and *S* and (b) maintaining transaction state in *S* to optimize for repetitive RPCs from *C* to *S* performed within the same transaction.

- *Fault tolerance and Recovery.* For several process topology patterns (process replication, distributed data, meshing) presented in Section 3.4, fault tolerance is one possible motivation. In this thesis, we did not focus on fault tolerance. However, in many projects, an effective fault tolerance mechanism is required. For that purpose, a detailed failure model and a recovery protocol have to be developed for the FPT architecture and our implementation.

  A simple failure model could assume that each element of a process topology – whether a process, a data store, or a connector – can fail. (Also, message loss or message duplication might occur.) When a data store or a process fails, all attached connectors fail, too. Object managers in processes need some fault tolerance logic for detecting that a connector to a server failed. On detection of a failed connector, an object manager could dynamically reconfigure itself by removing the corresponding *SERVER CONNECTION* section (see Section 6.2) from its current configuration. As long as there are redundant connections with appropriate imports alive, an object manager can proceed with transaction processing. When a failed connector recovers (or is successfully reinitiated), it has to be added to the configuration again. In addition to such automatic reconfiguration, it is necessary to propagate changed import/export sets (see Section 5.3) to all clients (recursively). Currently committing transactions that are affected by a failure could be aborted (and possibly retried). In the worst case, when a data store has been prepared as part of the 2PC protocol but has not been committed yet, a distributed transaction remains pending until all processes, datastores, and connectors required for communication between the

2PC coordinator and the data store are available again. When a failed process recovers, a recovery protocol has to be executed to terminate all pending distributed transactions that were coordinated by the object manager of that process. For that purpose, coordinators (force) write entries to a transaction log, which they consult after a failure. Our *Free Data Objects* framework already implements such a transaction log, but no recovery protocol is executed yet.

- *Query caching.* Our FPT architecture permits query caching but treats it as an implementation issue and defines only basic consistency requirements that have to be met. In our proof-of-concept implementation, we implemented caching only for individual objects. Since query caching may significantly improve transaction throughput and client response time, it is desirable to incorporate it into our middleware framework. Approaches to query caching are, for example, described in [ACPS96] and [KB96].

Note that the features proposed above can be realized by *extending* the concepts of our FPT architecture. In particular, it is not necessary to apply significant changes to the core concepts.

Auto-Adaptive Process Topologies

In Chapter 3, we explained that designing adequate process topologies is a complex task because many driving forces are involved. We argued that the task of designing and changing topologies is not easy to automate and therefore should be performed by a software architect. Nevertheless, there are situations where automatic (or at least semi-automatic) adaptation is a realistic goal. One particular form of auto-adaptation – reconfiguration for achieving fault tolerance – has already been outlined above. Another promising (semi-automatic) approach is to give administrators or software architects *hints* for adapting process topologies. Based on heuristics and runtime statistics, the recommendations could be generated by management tools. After explicit confirmation, the process topology could be automatically adapted according to the recommendation. In certain well-defined scenarios, even auto-adaptive enterprise applications that autonomously adapt their topologies are realistic. We do not expect that many organizations will let their enterprise applications automatically determine a suitable process topology. However, given runtime statistics and a set of well-defined rules (e.g., for automatically adding and removing replicas to/from a specific tier or for creating a topology with the shape of a reverse tree), auto-adaptation can be a convenient and powerful feature.

Analysis of Large-Scale Process Topologies

The concepts proposed in this thesis provide a powerful basis for constructing enterprise applications with arbitrary custom process topologies. In Chapter 7, we examined and evaluated our proof-of-concept implementation with several different process topologies. Unfortunately, since setting up and benchmarking scenarios with larger process topologies is time consuming and expensive in terms of hardware, we had to limit ourselves to relatively simple process topologies. The next logical step is to analyze larger scale process topologies, for instance, topologies with extensive replication of processes and/or datastores.

Large-scale process topologies can be analyzed using real hardware or, alternatively, by means of *simulation*. In the networking and peer-to-peer communities, it is common practice to simulate behavior of large-scale networked systems using advanced network simulator software, such as OMNeT++ [OMNe03]. An interesting approach would be to use such tools to simulate transaction processing in process topologies. In a simulation environment, datastores should be replaced with stubs that emulate datastores in main memory. A particular challenge would be to develop an adequate model for load, latency of processes, and data store performance. The advantages of a simulation-based approach would be that, within a reasonable time, large and complex process topologies could be analyzed with varying parameters, including bandwidth of connections, processing power of machines, various wire protocols, and different mappings of process topologies to physical network nodes and connections. Particularly, the effect of failures and (combinations of) different routing functions could then be conveniently investigated.