Why It Is So Difficult to Build N-Tiered Enterprise Applications

Technical Report B 01-05, Nov. 2001

Christoph Hartwich*
Freie Universitaet Berlin
Institut für Informatik
Takustrasse 9
D-14195 Berlin, Germany

hartwich@inf.fu-berlin.de

ABSTRACT

Structuring enterprise applications into multiple tiers is a well-tried and successful approach to improve scalability and achieve a better separation of concerns. Unfortunately, state-of-the-art object-oriented middleware standards and component models, like CORBA, RMI, Enterprise JavaBeans, and the CORBA Component Model, do not support well-structured applications with an arbitrary number of tiers. Attempts to build n-tier applications on top of them often result in bad performance, unexpected complexity, and severe architectural problems. This paper provides a definition of n-tier structures, motivates their use for large-scale enterprise applications, and analyzes problems that occur when n-tiered enterprise applications are developed based on object-oriented middleware standards or component models. In addition, an approach to solving these problems is outlined.

Keywords

n-tier, layer, enterprise application, middleware, distributed systems

1. INTRODUCTION

Enterprise applications are distributed, transactional multi-user applications; they play a key role in many organizations. For years and until today, large-scale enterprise applications have been successfully built with a special emphasis on multi-tier structures. This approach has proven to lead to scalable, flexible, and modular designs with a good separation of concerns. Despite their success and widespread use, multi-tier structures receive relatively little attention in the research community. In this paper we take a fresh look at them in the context of state-of-the-art object-oriented middleware standards and component models, like CORBA, RMI, Enterprise JavaBeans, and the CORBA Component Model [5, 8, 6, 9]. The main contributions of this paper are (1) the definitions it provides, (2) an analysis of typical problems that occur when n-tiered enterprise applications are developed based on object-oriented middleware standards or component models, and (3) an approach for overcoming these problems.

In Sections 2 and 3 we define layers, tiers, and n-tier structures. Section 4 introduces enterprise applications, motivates the use of n-tier structures for them, and discusses performance aspects of n-tiered enterprise applications. Section 5 identifies and analyzes problems that are likely to occur when n-tiered enterprise applications are built on top of object-oriented middleware and component models. Section 6 outlines ideas and basic concepts of a framework that provides explicit and efficient support for n-tiered enterprise applications. Finally, in Section 7, we provide a brief summary.

2. LAYERS

To lay a solid foundation for a discussion of n-tiered enterprise applications we start with some basic definitions. This section defines and describes the concept of *layers* and *multi-layer structures*.

"Layer" is a term that refers to an entity and its geometric relationships to other entities. When we talk about *multiple* layers that implies that there is a total ordering of the given entities with each entity having "contact" with at most two neighboring entities, as depicted in Figure 1.

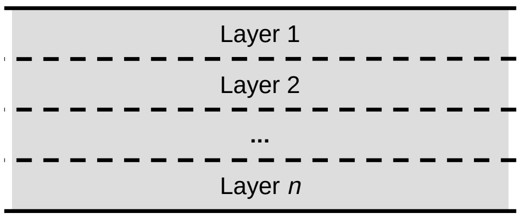

Figure 1. A layered structure.

The term "layer" is also used as a metaphor to describe the architecture of a software system [3, 7]. In that context a layer is a category that contains artifacts of the given software system (functions, classes, object instances, components, files, configuration data, ...).

A multi-layer structure requires a *layering criterion* that defines decomposition rules and thus the association of artifacts with

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

categories. A layering criterion is based on one or more *layering styles* that define an abstract way of decomposing a system into layers. Examples of layering styles are:

- Physical distribution – Layers are aligned with machine boundaries in a distributed system.
- Levels of abstraction – Each layer *i* provides services for layer *i-1* with the help of lower level services consumed from layer *i+1*.
- Separation of application-specific aspects, for instance, logging, presentation, access control, request preprocessing, security, constraint checking, persistence, or error handling.
- Implementation language or runtime environment – Artifacts written in the same programming language or that require the same runtime environment are grouped together. For example, Java classes are placed in layer *i*, and the C++ implementation of their native methods is placed in layer *i*+1. This is done regardless of the level of abstraction provided by the Java and C++ method implementations.

Usually a combination of layering styles is selected to constitute a layering criterion. Also, some styles may apply only to a subset of layers. For example, a layering style may help to clearly separate two specific layers but is less important for the definition of the other layers. In theory, most layering styles are orthogonal to each other. In practice, they are not, because developers create artifacts with respect to the layering criterion, i.e. in such a way that the layering styles used do not conflict. Precisely defining a suitable layering criterion and designing appropriate artifacts for it is one of the main contributions of a software architect to a layered software system.

A multi-layer structure covers a complete software system or only a part of it. Artifacts are usually associated with a single layer, but sometimes it makes sense to assign them to multiple layers. A common example is a class `Message` that is used in one layer for sending, and in another for receiving a message.

There has to be a total ordering of layers. Ideally, each layer communicates with neighboring layers only. This (voluntary) restriction significantly reduces the number of potential dependencies developers have to be aware of. Additionally, communication can be further restricted by allowing only a higher layer (client) to initiate communication with a lower layer (server). Restricting communication in a complex software system and decomposing it into layers that are clearly separated subsystems significantly reduces complexity and provides a good separation of concerns.

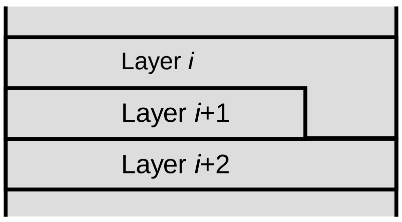

Figure 2. Relaxed variant of a layered structure.

There are several variants of and modifications to layered structures to meet specific requirements of individual software projects. For example, a layer *i* may be allowed to directly access both layers *i*+1 and *i*+2, as shown in Figure 2.

This introduces more complexity but may be necessary for performance reasons or because of a black-box component in layer *i*+1 lacking some desired functionality. Such modifications are reasonable, provided their scope is limited, they are well-documented, and their benefit outweighs the increase in complexity.

Often, a layer can be further refined by horizontally subdividing it into independent subcategories. These subcategories can again be treated as layers and may even form subsystems with an individually layered structure. An example is shown in Figure 3.

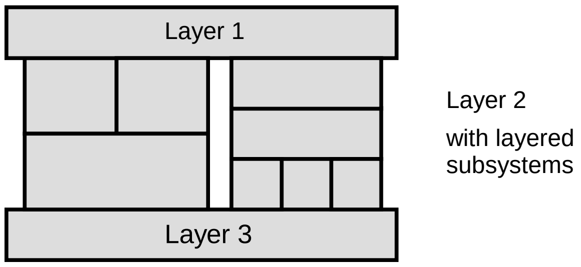

Figure 3. Example of horizontally subdivided layers.

3. TIERS

Having defined layers and multi-layer structures we will have a closer look at tiers, multi-tier structures, and n-tier structures in this section.

A *tier* is a layer that corresponds to a process or a collection of processes. A tier contains all artifacts of a software system that can be associated with the tier's process(es). Consequently, we can define a *multi-tier structure* as a multi-layer structure with a layering criterion that is dominated by a process layering style. Other layering styles can only serve as a further refinement of the structure defined by the process layering style.

Multi-tier structures are especially suited for distributed systems as machine boundaries always denote process (address space) boundaries. Often, each machine will run only exactly one (application) process that is explicitly incorporated into the multi-tier structure, although, in principle, any number of processes may run on a single machine. The assignment of processes to machines in a system is independent of its multi-tier structure.

Figure 4 shows an example of a multi-tier system with its processes (squares), communication relationships (edges), and tiers (gray rectangles).

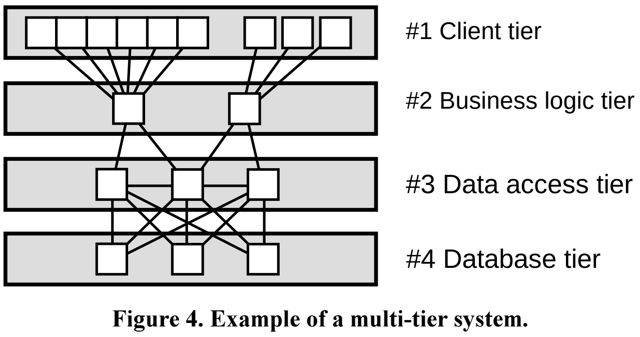

Figure 4. Example of a multi-tier system.

The condition that layers should communicate with neighboring layers only applies to multi-tier structures, too, and is much more important. Relaxing this condition does not only result in more complexity but can have a negative impact on scalability, security, and configuration.

In the past years the term *n-tier* has been used with varying semantics and not always consistently. The most common meanings found are:

(a) A synonym for “multi-tier” (which simply means “more than one tier”).

(b) A fixed number of tiers that is greater than three (in contrast to two-tier and three-tier structures).

(c) An arbitrary, variable number of tiers. The number of tiers can be freely chosen and, if necessary, easily be adapted.

Option (a) is unsatisfactory because giving the same entity two names does not help to refine our terminology and increases the risk of misunderstanding. (b) has the disadvantage that it defines a structure simply by defining what it is not. Furthermore, it only makes sense in the domain of enterprise applications where people have a common understanding of two- and three-tier structures (see Section 4). Even as a domain-specific term it is rarely useful because for describing concrete structures it is much more informative to state the actual number of tiers (e.g. “a five-tier application”).

For our work we adopted meaning (c). Multi-tier structures can either be n-tier structures *or* have a fixed number of tiers. A concrete instance of an n-tiered software system usually has a specific number of tiers, but its architecture and design are flexible and allow for an easy adaption of the number of tiers.

4. ENTERPRISE APPLICATIONS

We define enterprise applications as distributed, transactional, multi-user applications that are employed by organizations to control, support, and execute business processes. Transactional behavior is required to guarantee consistency of business data being processed. Historically, the need for distribution originated from the fact that multiple users had to work with an application, each one requiring an own front-end (terminal). Today’s enterprise applications often form much more complex distributed systems, for instance, because of scalability or fault tolerance requirements, or because of the integration of several existing, previously isolated systems.

Most enterprise applications have been constructed based on the concept of multi-tier structures. Typically, the first tier is responsible for presenting the user interface, while the last tier persistently stores business data. In that context the terms “two-” and “three-tier application” are being used to designate two very specific structures:

Two tiers are associated with a client tier that contains presentation and a database tier that stores business data and is directly accessed by the client tier. Business logic is implemented either entirely within the client tier or within the database tier or distributed among both tiers.

Three-tier applications aim at separating business logic from presentation and database concerns by introducing an additional middle tier. The middle tier mediates between client tier and database tier and contains most of the business logic - although some parts may still reside in the other tiers.

Depending on the amount of business logic and application-specific data access logic they contain, clients are often referred to as “fat”/“rich” (much), “thin” (little), or “ultra-thin” (none). Examples for ultra-thin clients are web browsers or X terminals.

4.1 N-Tiered Enterprise Applications

Enterprise applications are not limited to two or three tiers, in fact, there are many cases where additional tiers can be of significant advantage, for example:

- A *concentration tier* – Instead of connecting a large number of clients (e.g., >1k) directly to a server machine, a group of replicated “concentration servers” mediates between clients and server machines. Each concentration server handles requests of a subset of the clients, optionally performs some pre-processing, and forwards the requests to a server machine. Thereby the burden of handling a large number of connections (I/O, CPU cycles, memory) is shifted from the server machines to the concentration tier which improves scalability and throughput.
- A *workflow tier* that is placed between presentation and business logic. The tier manages the execution of workflows, routes data, and assigns work items to users.
- An *access control tier* that provides fine-grained, application-specific access control. Because the access control code is strictly separated from other tiers it can be kept simple, isolated, and under the physical control of a security specialist.
- When functions are to be replicated, for instance, because of fault tolerance or (in conjunction with load balancing) to improve a system’s throughput, it is often reasonable to place them in a separate new tier.
- An *application integration tier* – When two or more applications are to be integrated (e.g., an application has to access data managed by a legacy system) it is a well-tried approach to introduce a new tier that mediates between the applications.

Because typical enterprise applications are constantly subject to change, it is reasonable to design them as n-tiered applications. Ideally, it should be possible to add or remove tiers without significantly affecting code and design of other tiers. This allows for a clear, intuitive, and flexible separation of many application concerns.

The main architectural building-blocks of such an n-tiered application are distributed, loosely coupled processes with only few dependencies. Replication fits well into the model (and is encouraged), and evolution and integration with other n-tiered applications becomes much easier.

4.2 Performance Aspects

Performance is an aspect that is usually paid close attention to during the development of a distributed system. A prevalent argument against using more tiers for enterprise applications than absolutely necessary is the potential overhead. Each additional tier can increase latency, and thus reduce response time because in many situations control flow and data have to pass through all tiers of a system.

While this is true, we believe that the effect is overrated. Whether performance is acceptable, or not, is usually determined by other factors, for example, caching strategies, adequate hardware for a given load, available bandwidth, or granularity of inter-process communication. For a reasonable number of tiers the resulting latency seems acceptable as, in the worst case, it grows only linearly with the number of tiers. We think that even up to ten tiers would be acceptable from a performance point of view – although it is very unlikely that an application will need that many tiers.

Moreover, the positive effects of n-tier structures on performance must also be taken into account: Applying replication to specific components becomes much easier, which in turn helps to improve scalability and throughput. In addition, shifting work to higher or lower tiers can help to relieve machines of a tier that would otherwise become a bottleneck. In that case the extra processing power of machines in an additional tier can very well offset the increased communication costs.

Finally, we believe that the reduction of complexity in n-tiered enterprise applications (through improved modularity and better separation of concerns) by far outweighs potential performance penalties. With less complexity development and maintenance costs can be significantly reduced, as well as the risk of failure. Compared to the costs of higher complexity the costs of extra machines (for additional tiers) and network bandwidth are relatively low.

5. PROBLEMS WITH OBJECT-ORIENTED MIDDLEWARE AND COMPONENT MODELS

Enterprise applications are rarely built from scratch. Instead they are commonly built on top of a software infrastructure that provides basic functionality like persistence, transactional access, and remote communication. In theory, this allows enterprise application developers to focus on application-specific aspects while leaving low-level, infrastructure aspects to infrastructure component vendors. Today, object-oriented middleware standards (e.g., CORBA or Java/RMI) and component models (like Enterprise JavaBeans and the CORBA Component Model) play a key role in many state-of-the-art software infrastructures.

Unfortunately, developers encounter significant architectural and performance problems when attempting to build n-tiered enterprise applications on top of object-oriented middleware and component models. These problems closely relate to how *business objects* are represented and incorporated into the infrastructure.

Business objects are those objects (or components) in an enterprise application's object model that cover domain-specific aspects. They represent real or abstract entities of the business domain, for example, `Customer`, `Order`, or `Payment`. We distinguish two types of business objects: *Business data objects* are data-centric and primarily represent business data that are stored in persistent storage and accessed within transactions. *Business process objects* are process-centric and primarily contain business logic that accesses business data objects. In the following discussion we will focus on business data objects only.

Current object-oriented middleware and component models are centered around the concept of remote objects. The standard way of incorporating business data objects into a system is to represent them as remote objects that are located in the second last tier of the enterprise application. Their persistent state is stored in the last tier (typically in a relational database) with the help of services provided by, for example, JDBC, CORBA's Persistent State Service, or various object-to-relational mapping tools.

This approach causes a number of severe problems for n-tier applications (and for many multi-tier applications as well) that are outlined in the following subsections.

5.1 Corrupted N-Tier Structure

In an n-tiered enterprise application it is very likely that most tiers contain business logic that needs to access business data objects. But with business data objects being represented as remote objects in tier *n*-1, all tiers are forced to bypass intermediate tiers and directly access tier *n*-1 via remote calls, as depicted in Figure 5.

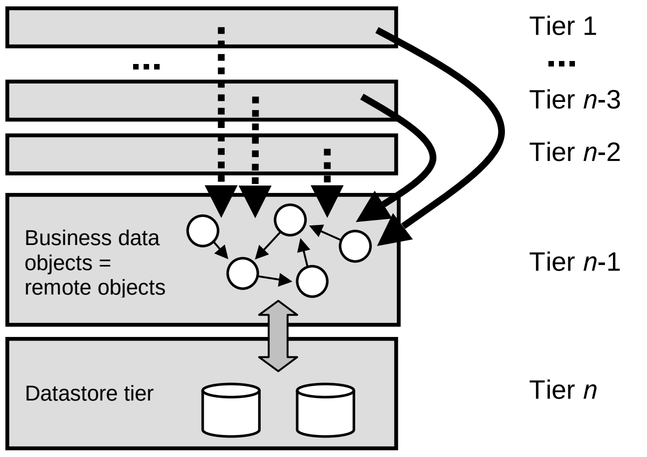

Figure 5. Tiers are forced to directly access tier *n*-1, bypassing other tiers instead of adhering to the tiered structure (dashed arrows).

This violates the condition that tiers should communicate with neighboring tiers only (see Sections 2 and 3) and thus significantly complicates the construction of well-structured n-tier applications. Tiers between the calling tier and tier *n*-1 are given no chance to interfere and provide their services which renders them useless. For example, a security tier, a concentration tier, or a tier that caches business data require to be involved and should not be skipped.

The problem occurs in all n-tier and multi-tier applications where business logic and business data objects are distributed over three or more tiers, excluding the database tier. To avoid the problem the total number of tiers must not exceed four with ultra-thin clients and three in all other cases. This is one of the main reasons why architects stick to these structures, as additional tiers would corrupt the design.

In principle, this architectural problem could be solved by adding artifacts to each tier that act as proxies to artifacts of the underlying tier that, in turn, can be proxies. This is straightforward for procedural systems where each tier can provide a facade with functions that are either implemented locally or simply call their equivalent in the underlying tier. However, for objects, especially business data objects, there is no such simple solution. Object identity, references between business data objects, references in parameters or return values of remote operations, garbage collection, and transactional access are just a small selection of issues. Also, introducing optimization techniques like caching or bulk transfer of data would lead to additional challenges. In any case, such an approach would require major extensions to the infrastructure. It is not supported by existing object-oriented middleware and component models.

5.2 Fine-Grained Inter-Process Communication

The granularity of inter-process communication has a significant impact on performance. A fine-grained communication style often drastically increases network traffic, workload, and latency, and thus can be a severe threat to performance and scalability [10]. Unfortunately, with existing object-oriented middleware and component models only fine-grained communication is well supported. Access to business data objects is provided on a per-attribute basis.

For most applications it is not acceptable to perform a remote operation call for each business data object attribute to be accessed. For example, a client with a complex, powerful GUI that displays several tables, trees, and combo boxes containing business data, could easily issue hundreds of remote invocations just for displaying a single view. But more efficient access techniques are not directly supported and are completely left to application developers (see 5.3 and 5.5).

5.3 Application Design Dominated by Performance Aspects

With the infrastructure lacking support for more efficient access, application developers have to design all parts of an enterprise application with respect to performance. This is not desirable because architecture, object model, and business logic become heavily dependent on performance optimization aspects.

The problem is likely to be encountered in all applications where business logic (or presentation) on other tiers needs direct access to business data objects. Only three-tier applications with ultra-thin clients do not suffer from it as the middle tier contains both business logic and business data objects and the presentation is pre-calculated there, too. This is one of the main reasons why ultra-thin clients are so popular and their use is advocated, although they are often unsuitable for complex, powerful, and user-friendly user interfaces.

Typical optimization strategies used for reducing the number of remote operation calls are:

(a) New data structures and access methods are defined to transfer multiple attribute values of a business data object with a single remote call. Alternatively, the complete internal state of business data objects can be extracted and transferred with the help of container classes that are either generic (e.g., Java `Hashtable`) or business data object specific (e.g. `AccountState` for a business data object `Account`).

(b) Direct remote access to business data objects is considered strictly prohibited in order to prevent fine-grained access. They can be accessed by other tiers only through a procedural facade (e.g. EJB session beans) that resides local to the business data objects. The facade contains operations for efficient bulk transfer of object state. For example, the states of a set of business data objects are transmitted as a result of a query operation. The EJB 2.0 specification [9] introduces the concept of *dependent value classes* that helps application developers to implement a relaxed variant of this optimization strategy: Business data objects are classified into two types, coarse-grained entity beans that are remotely accessible and fine-grained dependent objects that are passed by value. Entity beans provide a procedural facade for remotely accessing data of their private, dependent objects.

(c) State information of business data objects is cached in other tiers so that subsequent accesses to the same objects can be performed locally. Caching can be transparent, for example, by using smart proxies (which leads to proxy problems outlined in 5.1), or explicit by using a cache structure with an entry for each cached business data object state. Cache entries can be generic or business data object specific (see a).

5.4 Corrupted Object Model

While the strategies listed in the subsection above help to improve performance they often damage the quality and usefulness of an n-tiered enterprise application's object model.

It is paradoxical that, on the one hand, the object model is considered to be one of the main foundations of an enterprise application and is carefully populated with objects, but, on the other hand, many parts of the business logic are denied direct access to business data objects. Working with data structures that contain state information instead of working with the original business data objects means to abandon object-oriented principles and features in many cases.

Upgrading the data structures to objects again may preserve the object-oriented view, but results in a duplication of all business data object classes in the object model because different versions are needed for the second last tier on the one hand and all tiers above on the other hand.

5.5 Proprietary and Costly Extensions to the Infrastructure

In order to build large-scale n-tiered enterprise applications with acceptable performance application developers have to extend the existing infrastructure with features like caching and bulk transfer of object state. Building these extensions without flawing the design is highly complex and may involve some or all of the following aspects (selection):

- Synchronization of cached values,
- management of identity and references of cached business data objects (mechanisms provided by Java serialization or CORBA value types are not sufficient for this purpose),
- garbage collection and cache replacement strategies,
- transactional access to cached data,
- locking strategies,
- session management,
- development of new tools, e.g. code generators,
- and integration with existing infrastructure, tools, and applications.

Obviously, this goes far beyond the scope of application development and the skills of many developers who expect their existing infrastructure to cover these features. Please note that this problem is not limited to n-tier structures, but can occur in many demanding, conventional three-tier applications, too.

Often the result is that application developers spend most of their time with complex infrastructure aspects instead of concentrating on the application's business logic. They extend the existing infrastructure with proprietary, highly application-specific, and limited features which leads to systems that, although built on standardized object-oriented middleware and component models, are difficult to understand and hard to maintain.

6. A FRAMEWORK FOR N-TIERED ENTERPRISE APPLICATIONS

To overcome the problems discussed in the previous section a novel approach for enterprise application frameworks is needed. Currently, we are designing and implementing a prototype framework that provides explicit and efficient support for well-structured, object-oriented, n-tiered enterprise applications. In this section we describe basic ideas and concepts our framework is based on. Please note that this is work in progress. As many technical details are likely to be subject to changes at this early stage we focus on high-level, architectural aspects in this section. The goal is to give the reader an impression of our particular solution.

6.1 Copies of Business Data Objects

Instead of accessing business data objects remotely they are copied across process and machine boundaries so that application code always works on local *copies*. More precisely, a business data object is an abstraction that consists of a persistent state stored in a transactional data store, zero or more copies, and interfaces through which application code can access a copy.

6.2 Object Managers for Managing Copies

Copies live within and are managed by *object managers*. An object manager manages object identity, life cycle, and relationships between business data objects. Furthermore, an object manager lets application threads perform queries, set transaction boundaries, and it provides application threads with a transactional view on business data objects. Object managers maintain a cache for copies and are responsible for passing and synchronizing copies across process boundaries.

6.3 DAG of Generic Object Managers

Each application process of the n-tiered enterprise application has an object manager that manages local copies for that process. An object manager loads copies from, stores them to, and synchronizes them with object managers of the underlying tier. Each communication relationship between two application processes implies a communication relationship of their object managers, too. Like their application processes object managers can be viewed as nodes of a directed acyclic graph (DAG) with edges that represent client-server communication relationships.

An object manager is a generic component that can run on any node of the DAG, it is not specific for a particular tier or application. Object managers can use any combination of other object managers and transactional datastores (e.g., relational databases) in the underlying tier as data sources. Typically, transactional datastores will reside in the last tier and serve as data sources for object managers in the second last tier, although this is not a requirement.

Figure 6 illustrates a simple n-tier structure with generic object managers.

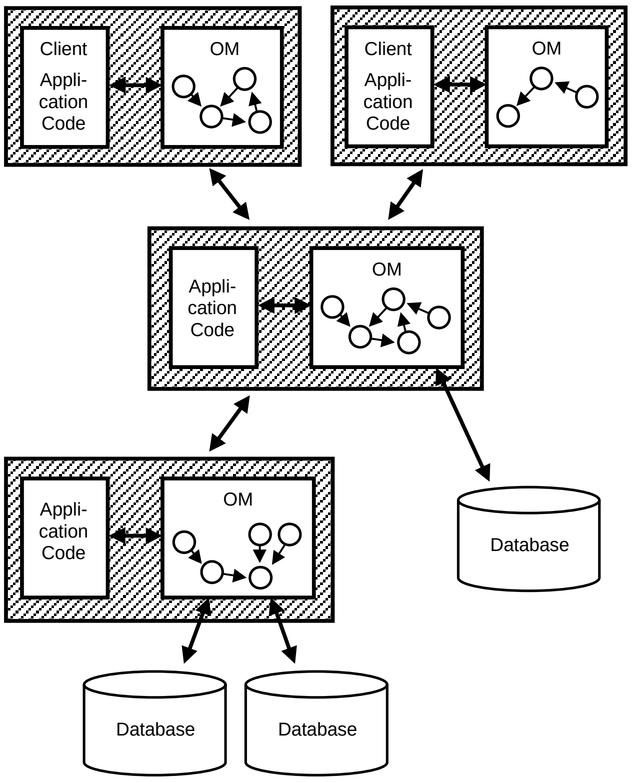

Figure 6. Example of a n-tier structure with generic object managers. Each application process contains application code and an object manager (OM) that manages local copies.

6.4 Transactions

In simple cases a transaction involves all nodes on a path from a client to a transactional data store. For example, a thread in a client process starts a new transaction and queries data. The local object manager forwards the query to an underlying object manager (unless the data requested is already available locally) which forwards it again, and so on, until, finally, an object manager can execute the query on an underlying database. The result is transformed into a set of copies and sent back to the client, traversing all object managers on the path the query took in reverse order. The client thread performs read and write operations on business data objects through local copies and finally requests to commit the transaction. The object managers on the path to the database propagate all changes down until, finally, the last object manager on the path writes new values to the database. A possible client-side implementation of that scenario (Java) might look as follows:

```
myObjectManager.beginTx();
List orderList = myObjectManager.query( query );
Order order = (Order) orderList.get( 0 );
Customer customer = order.getCustomer();
if( customer.getName().equals( "Mr. X" ) ) {
    order.setDiscount( 0.06 );
}
myObjectManager.commitTx();
```

Transactions that access business data objects from more than one data store are more complex. They involve nodes of a rooted DAG instead of a simple path. Also, a two-phase commit is required in that case.

For an effective and non-blocking concurrency control that allows for caching of transactional data optimistic locking and versions can be utilized.

6.5 Inter-Tier Application Control Flow

Many applications will leave inter-tier communication completely to their object managers. Processes between client object managers and data stores are used as caches, for database access, for concentration, or to integrate multiple data sources. Application code in that processes is either not required or is implicitly invoked as rules or triggers (e.g., for checking the validity of a request).

But more demanding applications may require application-specific remote invocations from one tier to a neighboring tier, especially for invoking remote business process objects. To guarantee a consistent, transactional view on business data objects for threads in all tiers a transaction context has to be propagated with such invocations. Contexts have to be sent in both ways, one with the request message and another with the response message. For instance, for CORBA remote operation calls interceptors/service contexts can be used. A transaction context consists of a transaction id and synchronization data. After a request message arrived but before the remote operation is executed the local object manager uses the synchronization data to update and (optimistically) lock copies according to the caller's current transaction state. Then the remote operation is executed within the given transaction, a response message with a transaction context is sent back to the calling thread, synchronization takes place again, and, finally, the calling thread is allowed to proceed.

An alternative is to propagate transaction ids only and allow object managers to exchange synchronization data on demand with separate messages.

6.6 Discussion

There are some parallels between our object managers/copies approach and EJB containers/entity beans as well as CCM containers/entity components. But there are also important differences – namely that we use copies instead of remote objects and that our architecture is based on a DAG of generic object managers.

With our approach all of the problems discussed in Section 5 can be solved or at least be handled much more convenient:

- Corrupted n-tier structures (5.1) are avoided as tiers communicate with neighboring tiers only. There is no need to bypass tiers anymore because business data objects are accessed through local copies obtained from object managers of the underlying tier.
- Fine-grained inter-process communication (5.2) is reduced significantly because copies are cached local to the accessing code and states of multiple objects are transmitted within a single message.
- Application design (5.3) is relieved of performance aspects. Application developers can return to clear, simple, intuitive, object-oriented designs as the framework transparently handles synchronization and performance optimizations. Known optimization techniques for object-oriented databases and object-relational mapping tools [2], like caching, bulk transfer of object state, prefetching, and lazy loading of collections and attribute values, can be extended to n tiers and be utilized as part of the framework. This does not guarantee best performance in all cases, but we believe that in most cases sufficient performance can be achieved. Application developers can focus on a clear design and, if necessary, on a small number of performance "hot spots".
- There is no need any more to duplicate parts of the object model or deny business logic direct access to it (5.4). All tiers have direct and unlimited access to the object model in a uniform way.
- Performance optimizations are implemented as part of the framework and are not application-specific. Thus, application developers can concentrate on their application's business logic instead of creating complex, proprietary, and costly extensions to the infrastructure (5.5).

Our framework does not replace object-oriented middleware which is still needed for application-specific inter-tier communication and possibly communication between object managers.

Direct and uniform access to the object model from all tiers is a powerful feature as code that accesses business data objects is independent of a particular tier. For example, a copy may implement a method `calcTotalPrice` of a business data object `Order`. The total price of an order is calculated by adding the current prices of its associated business data objects of type `OrderItem`. The implementation can be invoked and executed in any tier. It is up to the application developer to decide where it is invoked: When all order items are likely to be in the client cache it makes sense to execute `calcTotalPrice` on the client. But when an order typically has a large number of order items that are not likely to be in a cache it is much more efficient to execute the method in a process near to the database that stores order items. This example shows that application developers still have to deal with performance aspects in some cases, but at a much higher and more appropriate level of abstraction.

7. SUMMARY

We define *n-tier structures* as multi-layer structures with a layering criterion that is dominated by a process layering style. The number of tiers can be freely chosen and, if necessary, easily be adapted. We motivated that n-tier structures are well suited for large-scale enterprise applications because they provide a scalable, modular, and flexible design with a good separation of concerns.

Unfortunately, state-of-the-art object-oriented middleware and component models do not support enterprise applications with an arbitrary number of tiers. The two basic problems are that business data objects are represented as remote objects in the second last tier and that the infrastructure does not support efficient access to them from other tiers. This leads to severe architectural and performance problems that force developers to strictly limit the number of tiers and build costly, complex, and proprietary extensions to the existing infrastructure. The problems we identified effectively prevent the construction of well-structured n-tiered enterprise applications on top of object-oriented middleware and component models.

We are designing and implementing a prototype framework that provides explicit and efficient support for well-structured, object-oriented, n-tiered enterprise applications. The main differences between our framework and existing object-oriented middleware and component models are that we use copies instead of remote objects for business data objects and that our architecture is based on a DAG of generic object managers.

Future work includes refinement and enhancement of our framework, performance optimization, efficient synchronization, case studies with n-tiered applications, and a performance analysis.

8. REFERENCES

[1] Bass, L., Clements, P., and Kazman, K. Software Architecture in Practice. Addison-Wesley, 1998.

[2] Bernstein, P.A., Pal, S., and Shutt, D. Context-based prefetch - an optimization for implementing objects on relations. VLDB Journal 9 (3), 177-189, 2000.

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

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

[5] Object Management Group. The Common Object Request Broker: Architecture and Specification. Rev. 2.5, Sep. 2001. http://www.omg.org

[6] Object Management Group. CORBA Components. Joint Revised Submission. OMG TC document orbos/99-07-01, Aug. 1999. http://www.omg.org

[7] Shaw, M., and Garlan, D. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

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

[9] Sun Microsystems. Enterprise JavaBeans Specification, Version 2.0. Final Release, Aug. 2001. http://java.sun.com/products/ejb/2.0.html

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