FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Design and Implementation of LinqSpace

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Wolfgang Gelbmann
Matrikelnummer 0525873

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuerin: Ao.Univ.Prof. Dipl.-Ing. Dr. eva Kühn
Mitwirkung: Proj. Ass. Dipl.-Ing. Thomas Scheller

Wien, 18.08.2011

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Eidesstattliche Erklärung

Wolfgang Gelbmann

Hasengasse 18 / 2 / 16

1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18.08.2011 _____

Abstract

In the .NET environment LINQ (Language Integrated Query) has established as universal component for data inquiry. The creation of the SQL-like query is strictly separated from the execution and interpretation and therefore represents an appropriate foundation for extensions.

The “Space-based computing” or “Shared Data Spaces” forms the basis for the XVSM (eXtensible Virtual Shared Memory) middleware solution that consistently uses coordinators to shift query logic from the application code into the abstraction framework. The initial intention of this work was to extend XVSM with LINQ in order to enrich the API with uniform and versatile query capabilities.

A portion of this thesis presents a LINQ adapter for XcoSpaces, the .NET reference implementation of XVSM. These module converts LINQ queries, so-called expression trees, to the matching LindaCoordinator inquiry which can be interpreted by XcoSpaces. Because this coordinator is based on template matching, the support of LINQ functionality is mainly restricted to equality comparisons.

As consequence of the insights gained, this work focuses on a new XVSM reference implementation (LinqSpace) with LINQ as primary interface for inquiry. For data storage any system which offers LINQ capabilities would be appropriate. In order to evaluate new opportunities an unconventional form of data storage regarding XVSM was chosen: the relational database.

Finally, the new LinqSpace is compared with XcoSpaces which allows illustrating the paradigm shift from “records” which are stored by the space toward direct interaction with a domain model. An example shows how the entities in an Entity-Relationship model and their relationships can be used as a basis for distributed coordination without additional adaptations.

Kurzfassung

Im .NET Umfeld hat sich LINQ (Language Integrated Query) als Komponente zur universalen Abfrage von Datenquellen durchgesetzt. Das Erzeugen der an SQL angelehnten Anfrage ist konsequent von der Ausführung und der Interpretation getrennt, was eine geeignete Basis für Erweiterungen darstellt.

Das „Space Based Computing“ oder „Shared Data Spaces“ Paradigma dient als Grundlage für die XVSM (eXtensible Virtual Shared Memory) Middleware Lösung, welche durchgängig Koordinatoren einsetzt um Abfragelogik vom Anwendungscode in das Framework zu verschieben. Die einleitende Intention dieser Arbeit resultierte in den Bestrebungen XVSM LINQ-tauglich zu machen und folglich die Interaktionsschnittstelle um eine vielseitige und ausdruckstarke Abfragesprache zu erweitern.

Ein Teilbereich dieser Arbeit präsentiert einen LINQ Adapter für die .NET Referenzimplementierung von XVSM (XcoSpaces). Dieser wandelt LINQ Abfragen, sogenannte Expression Trees, in den für XcoSpaces erfassbaren Linda-Koordinator um. Da dieser Koordinator auf Template-Matching basiert, sind die unterstützen LINQ Anweisungen stark eingeschränkt und umfassen im Wesentlichen Abfragen auf Gleichheit.

Als Konsequenz der erlangten Einsichten richtet sich der Fokus dieser Arbeit auf die Neuimplementierung der XVSM Spezifikation (LinqSpace) mit LINQ als primärer Schnittstelle für Abfragen. Zur Datenspeicherung wäre prinzipiell jedes LINQ-fähige System geeignet, jedoch wurde im Sinne neuer Evaluierungsmöglichkeiten auf eine, im XVSM Umfeld eher unkonventionelle Speicherform zurückgegriffen: die relationale Datenbank.

Abschließend wird der neue LinqSpace mit XcoSpaces verglichen und der Paradigmenwechsel von „Einträgen“ im Space zu einem Arbeiten direkt an einem Domain-Model erläutert. Ein Beispiel zeigt wie die Entitäten eines Entity-Relationship-Modells und deren Beziehungen ohne zusätzliche Adaptionen als Grundlage für verteilte Koordination genutzt werden kann.

Acknowledgements

First I want to thank my supervisor *eva Kühn* for this interesting master thesis topic and for the freedom to experiment far off conventional pathways, as well as *Thomas Scheller* and *Markus Karolus* for supporting me and sharing their great insights.

Furthermore, I want to thank *Doris Wilfinger* for her helpful proofreading.

Special thanks to my parents on which I can always count on.

Finally, my greatest thanks go to my little boy *Jamie* and my wife *Evelin* for encouraging and reminding me about the most important thing in life: family.

Content

1 Introduction

With about 58.4% of the European population using the internet [1] it has become a cornerstone of our digital information society and a reliable source for our daily information retrieval. Nowadays it is more common to search for answers online than to use a dictionary. The reasons are obvious: you can find access anywhere, you can find anything on the internet, it is easy to use and the data supplied is mostly up-to-date. We are so used to that omnipresent connectivity that it seems unusual if a computer is not "online" and therefore cannot communicate immediately with another computer located at the other side of the world.

With increasing data throughput the services provided by the internet quickly raised beyond information retrieval. Cultivating friendships on social networks, phoning people all over the world or even watching TV is all water under the bridge. Moreover, people get displeased when they are faced with long waiting times or complications and they cannot use their internet service 24/7.

The network connectivity model between distributed peers can be divided into two main system topologies: centralized and decentralized [2]. At the beginning of the Internet it was designed solely centralized which means that the data was stored and the web pages were generated on a single server. The reason for this was ease of maintenance and software development according to that topology. The insufficiencies are obvious, depending on the quantity and complexity of requests large web-applications have to be hosted on high performance computers like mainframes[1], grids[2] or clusters[3]. What was initially classified as the advantage of easy maintenance may turn as a momentous disadvantage because the upkeep for an extensive centralized software solution requires special knowledge and hardware.

The success of peer-to-peer networks like Gnutella [3] caused immense research and rethinking in the field of distributed systems [4]. The decentralized topology mainly bases on the idea to let the clients, which in this context are called peers, communicate directly with each other. The absence of a central unit mainly solves the problems of availability, extensibility and fault-tolerance on the topology level by spreading responsibility among the peers. But the decentralized version also has some flaws/disadvantages such as the lack of coherence, security and manageability.

Referring to the .NET Framework there currently exists a very popular library for querying data, namely LINQ[4] (see *Chapter 3.1*). The creation of SQL like inquiry requests is consequently separated from the execution and interpretation which makes LINQ an ideal base for extensions.

The "Space-based computing" (SBC) or "Shared Data Spaces" [5,6,7] paradigm is used as the basis for the XVSM (eXtensible Virtual Shared Memory [8], see *Chapter 1.2*) middleware solution which offers message-oriented, distributed communication and coordination.

[1] Powerful highly specialized computers
[2] The term "Grid computing" describes a loosely coupled distributed system architecture, which may be heterogeneous and geographically dispersed.
[3] Computer clusters are a network of computers whose function is to ensure availability and improve performance over that of a single computer.
[4] Language Integrated Query [96,97]

Inquiries are handled by coordinators which allow to abstract coordination policies through the underlying framework. There is currently no interface which can be used for LINQ interactions.

The ultimate goal of this work is to offer a new way for querying the space, based on the LINQ technology. The result manifests in a new space called LinqSpace which is focused on LINQ as the main interface for inquiry.

1.1 Overview

Chapter 2 provides an overview of middleware technologies in the .NET environment and analyzes some existing frameworks with similarities to this work.

Chapter 3 gives an introduction to the various .NET technologies used for implementation. The final behavior and characteristics mainly shall match the formal model of XVSM [8] wherever possible. LinqSpace largely relies on standard components and libraries of the current .NET Framework version 4.0. If a module or technology does not offer a significant XVSM typical behavior it will be covered by a workaround. Otherwise if the standard component provides a wider range of functionality it is maintained and incorporated with the space.

Chapter 3.1 gives a more detailed look in the LINQ technology. Since LINQ was introduced with the .NET Framework 3.5 it has gained more and more popularity. Its syntactic power and versatility is comparable to SQL statements.

Chapter 3.2 presents the ADO.NET[1] EF[2] which offers the opportunity to map object-oriented models into relational database storages. LINQ is designed to operate directly as query language on these models. The data storage should be switchable, which is one of the core aspects of LINQ. But to go one step further and use the capabilities offered, the secondary objective of this thesis is to evaluate the space as domain model storage.

Chapter 4 presents the results of the first consolidation approach for LINQ and XVSM. A LINQ provider was realized for the current implementation of XcoSpaces.

Chapter 5 shows the design and considerations regarding the implementation of LinqSpace. LinqSpace provides a uniform LINQ interface for coordination and information retrieval beyond network boundaries. This opens a very broad range of new capabilities offered the first time by a space which extents and impacts will be evaluated during this work.

In *Chapter 6* XcoSpaces and LinqSpace are compared regarding their usability, lines of code and concurrency performance.

1.2 An Introduction to eXtensible Virtual Shared Memory (XVSM)

XVMS is a reference architecture of the abstract SBC paradigm. The specification of XVSM, which can be found at [8], introduces new concepts [5,8,9] for space interactions and modifications. The following chapters describe the components which make up the essential parts for data storage, data retrieval and interspace communication.

[1] ActiveX Data Object for .NET [100]
[2] ADO.NET Entity Framework [101]

1.2.1 Container

This is the main structural storage component for data items, which in the context of XVSM are called entries. A container has a name, it holds data units and it can host multiple coordinators. A container basically can be referred to as a collection of entries and it can be addressed using the URI scheme *"xvsm://namespace/ContainerName"*.

1.2.2 Coordinator

Basically coordinators represent coordination policies. They can be declared as obligatory or optional which essentially means that either the coordinator must or can be supplied during a space operation. When a new coordinator is introduced, it must be announced during the creation period of the container and it is bound to the container's lifetime. Coordinators may affect write operations by storing additional coordination information. A component called Selector can additionally deliver discriminative information which the coordinator uses to generate a filtered view of the entries. Selectors are invoked during read and take operations and mainly use coordination information previously stored by Coordinators.

A coordinator that is especially important in the XVSM specification is the QueryCoordinator: Unlike the other coordinators the QueryCoordinator does not rely on previously stored coordination information, but works on the data stored by the entry. The formulation language for requests is called XVSMQL[1] and is loosely based on a subset of SQL statements.

Regarding to the referred data, coordinators can be separated into two types:

- Extrinsic Coordinators: these coordinators are based on additional coordination information (FIFO, LIFO, KEY, LABEL, VECTOR). The coordinators save additional information during write operations. According this classification, it is not significant if the information is passed directly through the interface by the user (KEY, LABEL, VECTOR) or is calculated by coordinators (FIFO, LIFO). Due to hash algorithms which are typically involved in this kind of coordination policy the coordinators are extremely fast when it comes to data retrieval. These coordinators either represent queue or stack typical data access with Fifo- or LifoCoordinator or select precise entries with Key-, Label-, or VectorCoordinator.
- Intrinsic Coordinators: these coordinators are based on entry data (LINDA, QUERY). They do not need extra information for write operations because they rely on data which already exists in the entry. When a query is performed with this kind of coordinator it usually comes down to iterate over a collection of entries stored in memory to apply a filter mask. Therefore they are typically not as performant as coordinators based on hash algorithms. In order to increase inquiry performance, intrinsic coordinators may index relevant entry properties which results in additionally external data storage. Nevertheless, the query expressions still rely to the entry properties.

Any- and RandomCoordinator are omitted regarding this classification, because they generally do not need additional coordination information.

It is also possible to combine different types of coordinators with each other to achieve more complex filter criteria. This combination of coordinators can be seen as a concentration of

[1] Extensible Virtual Shared Memory Query Language

filters, sieving out relevant entries by passing these entries from one coordinator to the other in a pipe like fashion.

1.2.3 Aspects

Aspects are triggered functions before or after write, read, take and destroy operations. If they are connected to a container, they are called local aspects. Global aspects do not rely on containers and are invoked whenever the desired operation is called on an arbitrarily container. Due to cross-platform usage, aspects are to be programmed in a scripting language which allows access to request and result parameters as well as to the space.

1.2.4 Profiles

Profiles can be considered as a bundle of aspects, custom coordinators or other extensions forming so called modules to enrich the XVSM core. Profiles are pluggable and generally designed to achieve a specific overall functionality like security, replication or logging. Further, they can announce their own API in order to extend the interaction opportunities.

1.2.5 Remote communication

For the communication between distributed spaces there is a special XML-based schema named XVSMP[1] which represents a language-independent interface between distributed space boundaries. This ensures interoperable representation of data and operation invocations regarding different platform implementations of XVSM like Java or .NET.

1.2.6 XVSM reference implementations

The following table lists various implementations of XVSM.

Name	Platform
MozartSpaces [9,10,11,12]	Java
XcoSpaces [13,14]	.NET
TinySpaces [15]	.NET Micro Framework
Haskell prototype [8]	Haskell

Table 1, XVSM reference implementations

2 Middleware technology

In this section different frameworks will be evaluated which serve as middleware in distributed environments. Transparency is an important property for this type of software which is typically achieved by hiding the complexity of network interactivity, concurrency and other difficulties which arise between components in a parallel communication process. The services provided by a middleware often go beyond interoperability and may include failover or transactional capabilities. The phrase “middleware” in general addresses all kind of technology connecting software components with each other, located on a single machine or on multiple machines [16]. That results in a wide variety of computer software meeting the

[1] Extensible Virtual Shared Memory Protocol [94]

demands to be classified as middleware. Even a database can be addressed as middleware technology. To narrow down the candidates a categorization according to design principles for middleware interaction, like procedure calls and message passing [17], and support for additional functionalities is presented. These grouping provide the appropriate granularity to classify the behavior of the evaluated middleware solutions [18,19].

2.1 Remoting middleware

The main goal of remoting middleware is to map the OOP[1] paradigm into a distributed environment. Basically, the developer should not differentiate between an ordinary local object and an object which resides on a different machine. This behavior is typically achieved by providing the calling process a proxy object, which shares the same signature and captures the parameters involved. This proxy then initiates the network communication and the actual invocation in the remote process.

DCOM[2] can be seen as one of the first remoting middlewares on the Windows platform capable of RPC[3] which existed before .NET Framework was introduced. Java RMI[4] and .NET Remoting are also typical frameworks of remoting middleware. Other giants in this category are CORBA[5], a standard which allows method-call operations written in different development languages, and the WCF which will be covered in *Chapter 2.6.1*.

As remoting middleware progressed, the paradigm shifted slightly away from invocating a method from a distributed object towards the consumption of distributed services and the functionality they provide. Referring SOA[6], these services no longer represent isolated and incompatible silos of software components and can be arranged to service compositions in order to increase the reusability and accessibility of their functionality [20]. A criticism often mentioned in relation to remoting middleware addresses the transparent network boundaries which are, following this paradigm strictly, completely obscured. Since the developer is no longer in charge or even aware of the distributed invocation there is also no possibility for dedicated configuration, performance or error handling [21].

2.2 Messaging middleware or message-oriented middleware (MOM)

MOM bases on delivering messages between the involved components. Usually these messages are objects carrying information in a manner which can be classified as VO[7] or DTO[8] pattern described by Martin Fowler [21]. The main difference compared to remoting middleware is the asynchronous way in which messages are sent and that no processes are blocked. Receiving notifications is no longer done via return values but also by messages. Furthermore, if required by the application, it is the client's responsibility to correlate received answer packages and request messages sent.

---

[1] Object-oriented programming [102,103]
[2] Distributed Component Object Model [104]
[3] Remote Procedure Calls [105]
[4] Remote Method Invocation [106,95]
[5] Common Object Request Broker Architecture [107]
[6] Service-oriented architecture [20]
[7] Value Object [21]
[8] Data Transfer Object [21]

This paradigm shift creates completely new conditions for programming interactivity of loosely coupled components. The first important aspect is the liberation of time constraints according to communication bridges. The interacting processes no longer have to be ready at the same time in order to initialize an interconnection. Every participating component can send messages anytime to anyone. The middleware takes care of delivery when the receiving process is actually able to receive, which is more convenient to the way internet communication is built-on. This unbounded behavior can manifest, related to the middleware technology used, in various forms toward persistent or transient message subscriptions. That means, if the corresponding receiver is not working when the message is sent it is either discarded (transient communication) or saved in a buffer for later delivery (persistent communication).

Since MOM does not behave like standard method calls, which block until a result is returned, it is often mentioned that programmers familiar to the more mainstream development languages find the programming style complicated and unnatural to use. Other languages like Erlang [22] use message passing as their primary coordination technique to cope with high concurrency.

A good metaphor to point out the differences between remoting and message-oriented middleware is that the former can be compared to making a phone call, where the participant has to answer immediately, and the latter has similarities with writing letters.

Technologies which belong to the category of MOM are MSMQ[1], the software architecture of an ESB[2], JBoss Messaging, JMS[3] and many more.

2.3 Component container middleware

Component container middleware technology serves as a hosting environment where the application logic can be plugged in. Typically they provide a rich ecosystem for security, persistence, transactions, logging and other non-functional requirements. A characteristic approach is to embed the functionality as modular packages which are able to request features from the environment in a declarative way. The interceptor pattern, which also is referred to as aspect pattern, is regularly used as a starting point for the application logic.

The EJB[4] Container or Microsofts IIS[5] can be referred as component container middleware.

2.4 Space Based computing middleware and Space Based computing paradigm (SBC)

The SBC middleware according to XVSM [6,7] basically builds on the methodology described in the blackboard architecture pattern [23,24,25]. A blackboard model is based on the idea of an expert group trying to solve a problem through cooperation. Objects can be put onto this board and participating components can actively be informed about changes (active repository). This is a major difference to a pure database that is only activated by a client, but cannot send notifications. The participants never communicate directly with each other

[1] Microsoft Message Queuing [108]
[2] Enterprise Service Bus [51]
[3] Java Message Service [109]
[4] Enterprise Java Beans [110]
[5] Internet Information Services [111]

because they are only interconnected through the blackboard which allows a parallel execution of the participation processes.

A logical central space which can be accessed by data-driven coordination makes the center of this middleware architecture [5]. The SBC-Interface hides the physical location of the data and assures a homogeneous way of access and manipulation.

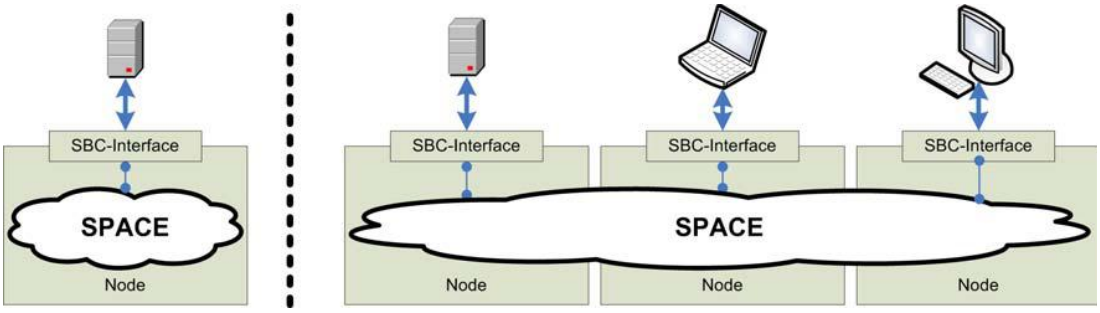

Figure 1: SBC-Interface for client/server (left) and distributed architectures (right) [5]

The main gateways to access data objects are regulated space operations named write, read, take, destroy and notify. Data is stored with additional coordination information according to policies which can be specified when data is written or retrieved. Those policies are exchange- and expandable and change the way data is accessed and viewed. They may vary from simple FIFO or LIFO queues to template matching Linda coordinators [26] or more complex coordination strategies.

In certain cases the SBC paradigm can significantly increase scalability [27] by replicating the data to the participating nodes. This can reduce network traffic and may result in very fast responses because of local operation execution.

2.5 Classification

Since the various middleware solutions which will be presented in *Chapter 2.6* are designed for different problem domains and therefore follow diverse approaches, a set of classification attributes are introduced to share a common ground for comparisons.

- Learnability: addresses the steepness of the learning curve in order to obtain the required knowledge for working with the technology. A short training period is significant for a quick assessment of the framework capabilities in a software project evaluation phase. Although Learnability can be seen as subset of the more general Usability, the two topics will be evaluated separately in order to evaluate the initial difficulties for the individual framework.
- Security: includes authentication, authorization and transfer security. Security is essential to ensure information quality, reliability and privacy in a decentralized system [28,29].
- Discovery: the opportunity to locate resources or services which are not known in advance [30,31,32].
- Replication: can be described as the process of sharing redundant resources with respect of consistency between them. The main goals of replication are to gain accessibility, fault-tolerance and therefore improve reliability [33,34]. The evaluation targets primary data replication not service replication.
- Message exchange pattern fitness: reports the capability of the respective technology to implement typical patterns of distributed communication architectures [35]. In

combination with usability this classification attribute characterizes the overall methodology of the evaluated technology.

- Usability: includes the ease of use when interacting with the framework. Important points are the code readability and the design of the primary interfaces or objects [36,37,38,39]. Easy and self-explaining usage not only increases the acceptance of a framework but also supports a clear and descriptive code footprint. Because usability depends on the "context of use" the evaluation addresses not the technology as a whole, but each message exchange pattern (see *Chapter 2.5.1*) individually.

The reasons for taking discovery and replication into account is because of the highly significance according to the scalability of P2P networks [40,41,42,43].

2.5.1 Exchange patterns used for comparison

Three patterns are used as a basis for comparison, which represent typical problems of communication middleware [44].

2.5.1.1 Extended Producer/Consumer/Observer

The first pattern extends the classic producer/consumer/observer scenario [45] by an additional reply channel, which is used to pass error information back from the consumer to the producer. This set-up can be seen as a combination of the producer/consumer/observer and a reversed request/reply pattern (see *Figure 2*).

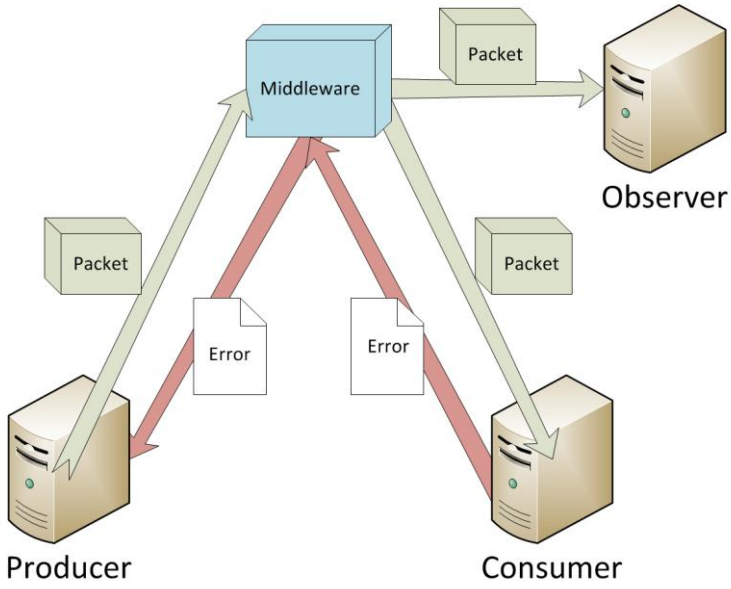

Figure 2: Extended Producer/Consumer/Observer pattern

Packages have a "weight" attribute, which is randomly chosen from the range 0-9. If the weight exceeds the value 5 the producer will be informed about the error through the reply channel. An additional focus of this pattern is whether the middleware offers the opportunity to abstract discriminative filters. This can be used if, for example, consumers are only interested in particular packets depending on attribute values. In the extended Producer/Consumer/Observer pattern it would be a significant breakdown when the consumer has to receive all packets offered by the middleware just to pick the interesting ones out. To achieve loosely coupled components which are required by this type of exchange pattern the technology should be capable of a Publish/Subscribe mechanism.

2.5.1.2 Request/Response

The classic Request/Response pattern matches the basic behavior of RMI which is offered by nearly every software development platform (see *Figure 3*).

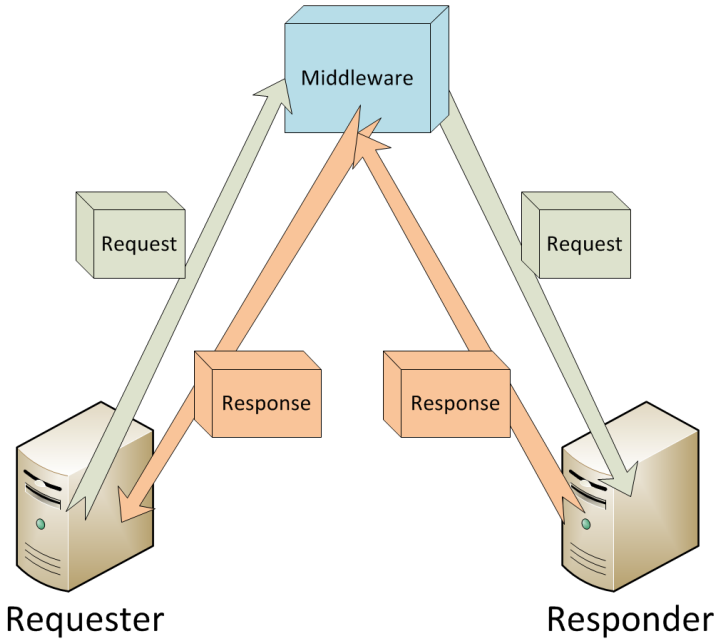

Figure 3: Request/Response pattern

The intentional usage of this pattern in RMI was purely synchronous, meaning after the request is sent the corresponding thread blocks until either it gets the desired response or an error due to possible communication problems. In addition, this pattern will be analyzed whether the middleware is capable of transmitting the request and awaiting the response in an asynchronous fashion, without blocking. Further it will be examined if the technology is capable of linking the requests with its relating response.

2.5.1.3 Single-Request/Multiple-Response

The Single-Request/Multiple-Response pattern extends the classic Request/Response pattern by allowing the responder to reply multiple packages. The test case used for evaluation also varies the type of the responses by sending a special summary object as last package (see *Figure 4*).

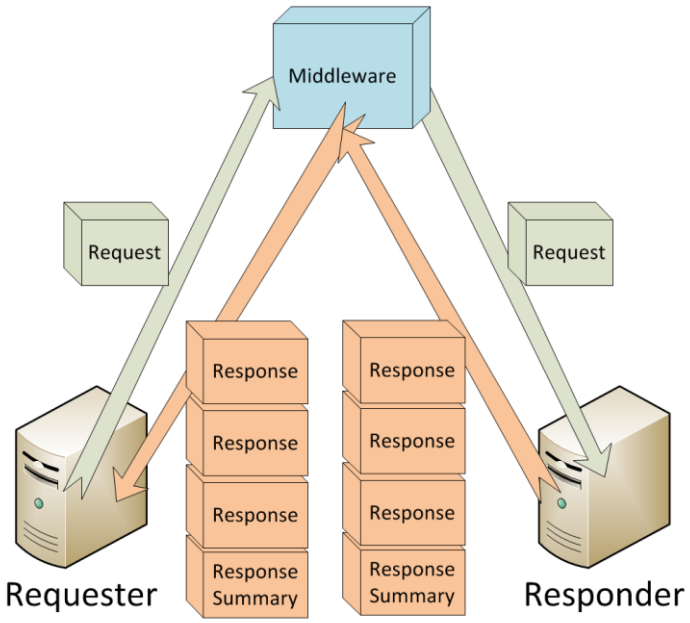

Figure 4: Single-Request/Multiple-Response

2.6 Middleware in the .NET environment

2.6.1 Windows Communication Foundation (WCF)

As the name suggests the WCF is no middleware framework or product, but a foundation. It provides a toolbox of instruments which can be assembled to fit a broad variety of requirements. Microsoft tried to put everyone's need under the hood of the WCF which assembles all major techniques for inter-process communication within a single machine or over the network. .NET Remoting, MSMQ, Named Pipes as well as various WS-* specifications [46] are accessible through a uniform interface and can be interchanged just by adjusting the configuration. Therefore the WCF can be classified as remoting and message-oriented middleware depending on the specific arrangement.

So far the theory, in practice it is not that easy because different middleware or communication methodologies rely on specific architectural designs or explicit coordinated interactions. Since version 3.5 the WCF provides besides the standard SOAP[1] based protocols for service operations additional interfaces toward resource orientated interactions which commonly can be classified as REST[2] architecture. The WCF can propagate transactions across the service boundary allowing multiple clients to participate in an atomic operation. Further the concurrency and instance management can be configured, allowing the manipulation of the way client calls are dispatched on the service-side.

The WCF provides clear and readable interfaces which can be referenced as one of the SOA tenets "Services Share Schema and Contract, Not Class". That means that the service definition is specified as interface and can easily be accessed and propagated via the platform independent description language WSDL[3]. When there is no access to the coded version of the interface there are several tools available which are able to analyze the WSDL description

[1] Simple Object Access Protocol [112]
[2] Representational State Transfer [113]
[3] Web Services Description Language [114]

of a foreign service and generate the corresponding coded interface. This technique goes beyond the interface definitions and also offers the possibility to generate the data types associated with the service. All configurations concerning the hosting service or the client can be made either directly in code or with the .NET typical XML application configuration file.

The WCF is fully supported in the .NET Framework and in addition offers a downgraded version for the .NET Compact Framework.

Learnability

Due to the versatile and flexible nature of the WCF the learning curve is steep at first. Once the concept of contracts and configuration methodology becomes clear the foundation offers many possibilities following the same concept, so the initial difficulties turn to account. The WCF is the main technology for .NET remote communication and therefore is mostly known in the community.

Security

The WCF offers various authentication mechanisms including windows authentication, username and password, certificate authentication with X.509[1] and an adapter which allows developers to implement custom strategies. According authorization, Windows accounts and groups or the ASP.NET[2] membership provider can be used. The latter one provides standard implementations to persist users and roles in a database and an open interface for custom credential storage solutions. The WCF supports three types of transfer security modes:

- Transport transfer security uses a secure communication channel which encrypts the data hardware accelerated on the network card.
- Message transfer security encrypts the message itself and allows communicating securely over non-secure channels like HTTP.
- Mixed transfer security uses both, transport and message transfer security.

Discovery

The WCF offers techniques for passive address discovery, which can be detected by a client via a UDP[3] broadcasts. Further a service can actively announce its endpoint to notify interested peers about its existence. Another possibility would be to configure the WCF to use the Windows Azure AppFabric Service Bus [47] to publish retrievable services.

Replication

Since the WCF is a toolset operates on services, no data can be replicated and this classification attribute cannot be applied.

Extended Producer/Consumer/Oberserver

The ordinary way to achieve a Publish/Subscribe typical behavior with the WCF would be to introduce an intermediate service which handles the infrastructure management. The design of such a subscription services is fairly simple and there are a lot of code examples showing best practices for the implementation. Despite this contingency the WCF does not offer a build-in behavior for the Publish/Subscribe pattern.

[1] Standard for a public key infrastructure [115]
[2] Active Server Pages .NET [116]
[3] User Datagram Protocol, a stateless network protocol [117]

As with the previous Publish/Subscribe pattern an intercessional peer would be able to filter out irrelevant requests before relaying it to the final endpoint. The reason for the absent of such components is the foundational principle of the WCF. It provides a rich toolset for building network infrastructure for broad variety of desired behaviors but there are no complete modules for high-level messaging patterns out of the box available.

Request/Response
The Request/Response pattern represents the classic messaging pattern in the domain of remoting middleware. The WCF is capable of this pattern by providing an interface which gathers the required network communication under the hood of an ordinary method call. Request information is passed over the method parameters and responses can be retrieved over return values. This technique connects the request and the response by the method call and therefore avoids an additional architectural infrastructure to make the link. The hosted functionality can be accessed according the asynchronous method pattern, which does not only target parallel execution. Furthermore it addresses the problematic to allow more concurrent activities than there are threads available [48].

Single-Request/Multiple-Response
In order to achieve this extended behavior of the classic Request/Response pattern the WCF introduces additional callback contracts which can be used by the called peer as retour channel. These interfaces can be invoked multiple times as required by the Single-Request/Multiple-Response pattern.

Usability
The WCF offers a pool of features which can be used to assemble the desired functionality. Patterns like Request/Response and Single-Request/Multiple-Response can be implemented out-of-the-box and are therefore easily realized. The Publish/Subscribe behavior requires an additional intermediate component and subsequently further technology knowledge. This is accompanied by an increase in complexity, and consequently has a negative influence in usability.

2.6.2 XcoSpaces, a reference implementation of XVSM

XcoSpaces clearly resides in the group of SBC middleware (see *Chapter 2.4*) and was created as a reference implementation of the formal XVSM specification. The kernel offers all basic functionality and was built with the intention of fast operation execution and extensible interfaces. XcoSpaces is entirely implemented on the basis of .NET technology [13,14] and uses the WCF for remote communication which allows configuring a wide variety of transport protocols. Although the development of XcoSpaces is still in progress the current version available can be considered as stable and is used for this evaluation.

In addition to the original XcoSpaces Kernel API the so called “XcoSpaces highlevel API” can be used to interact with the space. Latter provides classes and methods for more convenient access and additional functionality like distributed transactions and a container discovery service.

Learnability
If the principles of the SBC paradigm are known in advance the interface provided by XcoSpaces is straightforward. There are good tutorials available for the core and highlevel API which make it easy to locate the desired functionality.

Security

As with the XVSM specification XcoSpaces has no built-in support for authentication and authorization. Such security mechanisms can be injected as extensible features through aspects (see *Chapter 1.2.3*). Since the remote communication bases on the WCF the same transfer security modes can be used for safe message delivery. The WCF authentication and authorization cannot be used because they would require modifying the operation interface contracts which are not accessible by XcoSpaces.

Discovery

The highlevel API allows container discovery built upon the WCF discovery service called PeerResolver and can be configured via the standard application configuration files.

Replication

The XVSM specification considers replication techniques but there is no current implementation.

Extended Producer/Consumer/Observer

According the SBC paradigm the convenient way used to create a Publish/Subscribe typical behavior is via notifications. A container is used where consumers and observers can register their interest in write operations and which are triggered by producers. Further a second container serves as reply channel for error passed from the consumer back to the producer.

Through coordinators XcoSpaces offers the opportunity to sieve information collected by the container so the retrieving components are able to filter relevant entries.

Request/Response

The Request/Response pattern can be achieved by providing one container for requests and typically a distinct response container for each client, subsequently allowing a requesting peer to propagate its response container address along with the request information. The involved peers register notifications in advance and will be informed when desired activity is indicated. The communication offered by the XcoSpaces API is purely asynchronous but it is possible to block the current thread by waiting for a response immediately after a request is put into a container. If desired, it is duty of the surrounding infrastructure to make the association between a request and its correlating response, which represents a typical behavior of MOM.

The setting described is only one opportunity to accomplish this pattern. According to the requirements the organization of containers and coordinators can be fine-tuned to fit a specific behavior.

Single-Request/Multiple-Response

The exactly same setting as described for the Request/Response pattern can be used for implementation. When there are multiple requesters involved, coordinators or distinct response containers per client have to ensure that all correlated response packages are retrieved.

Usability

As with the learnability the usability of XcoSpaces relies on the principles of the SBC paradigm. In general the interfaces are very easy to use which results in clear and understandable code.

When a data class or structure is to be delivered through the space it has to be decorated with the Entry class, or implement IEntry interface. Furthermore there are two possible techniques to prepare the data structure for insertion into XcoSpaces.

- The first option is to hand over the data directly to the Entry class as object or in a generic way (see *Code 1*). A participating component has to be aware of the class in advance to retrieve the information regarding the correct object type.
- The second possibility is to extract the information from the data class and inject them as Tuple with corresponding TupleValues into the space (see *Code 2*). This way participating components do not need object type information but have be aware of the right data types and correct sequence in order to interpret the information.

Code

```
kernel.Write(cref, null, 0, new Entry<TestPerson>(new TestPerson()
                {
                    Age = 20,
                    Firstname = "FirstA",
                    Lastname = "LastA",
                    PersonID = 1
                }));
```

Code 1: XcoSpace data insert with generic entry

Code

```
kernel.Write(cref, null, 0, new Entry(new Tuple(new TupleValue<int>(20), // Age
                                                new TupleValue<string>("Firstname"),
                                                new TupleValue<string>("Lastname"),
                                                new TupleValue<int>(1)))); // PersonID
```

Code 2: XcoSpace data insert with tuples

The Publish/Subscribe functionality required by the extended Producer/Consumer/Observer is fully supported by the XcoSpaces API and is therefore easy to use. Request/Response oriented patterns rely on additional considerations in order to correlate the request invocation with the desired result information. Possible solutions include distinct response containers (see *Chapter 1.2.1*) in order to await the desired result or to mark the requests with identifiers which can subsequently be used to link the response messages to the requests.

2.6.3 XcoAppSpace

XcoAppSpace [49] builds on the asynchronous programming library CCR[1] distributed with Microsoft's RDS[2]. The technology shows more characteristics of MOM rather than SBC middleware. The CCR runtime offers a thread pool dispatcher class to instantiate and coordinate simultaneously executing tasks. A generic port serves as connection between work item objects and delegates which are about to be executed in parallel. Moreover CCR offers various ways to handle and coordinate concurrency by chaining Ports over a uniform architecture.

XcoAppSpace basically distributes the CCR features by making ports remotely available. The remote communication can be selected from a broad variety of supported protocols like TCP

[1] Concurrency and Coordination Runtime [118]
[2] Microsoft Robotics Developer Studio [119]

sockets, MSMQ, Azure AppFabric, Jabber transport service (also known as XMPP [50]) and the communication protocols offered by the WCF.

Learnability

At first it takes some time to understand the methodology of the XcoAppSpace with ports, workers and coordination techniques. Documentation is available on the website and extensive examples facilitate the initial difficulties. When the first complexities are resolved the usage of XcoAppSpace is quite easy and straightforward. In return, the framework offers rich opportunities of concurrency management in a uniform fashion.

Security

XcoAppSpace offers a rudimentary security mechanism with a service called XcoBasicSecurityService. This class manages a basic role-based authentication and authorization strategy which can be achieved by declarative attributes, describing required roles directly within the worker methods signatures. The mapping for username and passwords credentials coupled with the correlated roles can be passed during instantiation of the main space object.

Transfer security mechanisms offered by the WCF or Jabber can be used.

Discovery

The XcoAppSpaces.Discovery feature is capable of hosting a discovery server which can be used to locate distributed workers by name when their network address is not known in advance. Other spaces are able to announce their workers over the discovery service space as well as the discovery service space itself can host workers.

Replication

Since the primary focus of XcoAppSpace is message delivery and not data storage, there are no replication mechanisms required.

Extended Producer/Consumer/Observer

XcoAppSpace offers special classes and methods in form of worker extensions for the Publish/Subscribe pattern, which allow a readable and highly concurrent implementation of the functionality. The postings are forwarded to distributed ports where workers await the arrival of new items. After a published item has been forwarded to all subscribed peers it is removed from the port, so there is no differentiation between a consumer and an observer. During the subscription process a discriminative filter can be announced via a delegate function.

Request/Response

A bidirectional communication according the Request/Response pattern can be achieved over two public ports, each of them representing a one-way channel. This way the middleware itself is not capable of linking the request with the corresponding response object. Another remarkable opportunity offered by XcoAppSpace is to send the reply port along with the actual request as part of the transmitted object. This opens a broad range of usage scenarios and essentially allows associating the request directly with the response.

Although the ordinary communication offered by ports and workers is asynchronous it is also possible to implement synchronous behaviors where, after a request is placed, the current thread locks and waits for a response.

Single-Request/Multiple-Response
With XcoAppSpace this pattern can be implemented like the ordinary Request/Response pattern. Essentially, it does not matter whether a request places one or several responses to the return port. Since each Request can have its own response port there is no interference of concurrent requesters.

Usability
XcoAppSpace provides a rich ecosystem for distributed interactivity. This results in a very short and readable code footprint. Configurations can be made either via a configuration string or via fluent API, both passed at creation of the main XcoAppSpace class.

The port interface is designed generic so every serializable structure or object can be used to interact with XcoSpaces. But this presupposes that the distributed peers must have a coded representation of these items to interact with the port.

The XcoAppSpace offers mechanisms to implement the Publish/Subscribe and subsequently the extended Producer/Consumer/Observer pattern in an easy fashion. The Request/Response and the Single-Request/Multiple-Response pattern can consequently be achieved by response ports transmitted as part of the request objects.

2.6.4 Decentralized Software Services (DSS)

The DSS runtime is capable of exposing services as resources which can be accessed over a REST interface. Services for composition, structured state manipulations and notifications are offered for distributed interactions. DSS relies on the CCR toolkit and is delivered as part of the Microsoft RDS. DSS communicates via DSSP[1], which is a SOAP-based protocol used to define a set of state-oriented message operations for inter-service communication which essentially can be seen as alternate approach to the variety of WS-* specifications. One of the design goals of DSS was to couple performance and robustness.

DSS can be described as component container middleware (see *Chapter 2.3*) because the developed functionality will be hosted either by a dedicated DSS hosting application or self-hosted within another program.

Learnability
The DSS toolkit is an extensive framework and therefore initially more complicated. But there are scripts available to create template projects which demonstrate how to implement ordinary messaging patterns with DSS. However, it is a very comprehensive toolkit requiring a training period in order to exploit all the capabilities.

Security
DSS uses the CLR[2] infrastructure NegotiateStream to encrypt and authenticate TCP connections and offers APIs for HTTP security policies. Further operations, service contract and URI paths can be combined into roles for further usage in DSS.

Discovery
DSSP supports the WS-Addressing basic profile which mainly addresses the message

[1] Decentralized Software Services Protocol [118]
[2] Common Language Runtime [124]

transmission through networks, including firewalls and gateways. Further there is a discovery service using the UPnP[1] protocol to locate distributed nodes.

Replication
DSS is about services, intercommunication and message delivers and not about data storage, so there is no need for data replication mechanisms.

Extended Producer/Consumer/Observer
There are several classes offered to accomplish the Publish/Subscribe behavior. Basically a node can host a subscribe-able service which can be used to broadcast messages to various distributed peers. Since there is no data stored it essentially cannot be distinguished between a consumer and observer because both types are just notified when a new item is published.

There is no opportunity offered to place discriminative filters for published items.

Request/Response
A request communication object usually contains a response port which can be used to place a return item. The result will then be transmitted to the main dispatcher function of the node, which will be invoked for all response objects of the same type. When it is desired to actually match one request to its corresponding response, some kind of identifier is necessary.

The architecture is designed to handle multiple concurrent requests so the main purpose can be identified as asynchronous communication. It would be possible to lock the client and wait for an answer when synchronous interaction is desired.

Single-Request/Multiple-Response
A response port can be used to transmit multiple response items so the implementation architecture mainly matches the one described in the Request/Response pattern.

Usability
Build on top of the CCR the DSS allows to separate concerns into compact modules. The resulting components have a clear code footprint and can flexibly be adjusted for a desired concurrency mode. Once a service is hosted its state can be observed and functions invoked through a HTML interface.

DSS supports functionality to implement all evaluated exchange patterns without additional components or workarounds.

2.6.5 NServiceBus

NServiceBus is an open source, lightweight ESB implementation building on the MSMQ service. This technology can be classified as MOM and component container middleware because a rich environment is available in order to host the user services.

NServiceBus sends and receives messages over the MSMQ protocol which offers since version 3.0 the opportunity to use HTTP and SOAP for communication. Further, NServiceBus allows introducing WebService and WCF endpoints to access the ESB functionality over these technologies.

Learnability
There is a good documentation and various samples available on the NServiceBus homepage

[1] Universal Plug and Play [120]

which facilitate the initial learning phase. In addition, online trainings are offered and public courses are held in order to gain more experience from product experts.

Security
For Authentication and Authorization NServiceBus uses the MSMQ permissions which basically rely on windows authentication. Additional interfaces are available to implement user-defined techniques. Message security can be achieved with the injection of custom or predefined serializers in order to encrypt the message objects.

Discovery
There is no discovery service available according to NServiceBus but a WCF endpoint can be used to implement that functionality.

Replication
An ESB is mainly a massage passing system and has no need for data replication mechanisms.

Extended Producer/Consumer/Observer
NServiceBus offers functionality to implement the Publish/Subscribe behavior. Information about the subscriptions can be stored in memory, in the MSMQ or in a database. As with other MOM frameworks, messages are always consumed by receiving peers resulting in equal treatment of consumers and observers.

NServiceBus allows propagating packet filter criteria during the subscription process.

Request/Response
The framework allows coupling a request message with a response delegate which is invoked when responses are encountered. This technique can be used to establish a full duplex request/response communication between peers. Because of the message-oriented nature of NServiceBus, communication is completely asynchronous.

Single-Request/Multiple-Response
As with the Request/Response pattern, the communication channel can be used to reply to multiple response messages.

Usability
NServiceBus offers extensive possibilities for remote communication and service orchestration and therefore requires initially more time to understand and implement the desired patterns. Nevertheless, the resulting code is easy to understand and components remain loosely coupled. Each evaluated exchange pattern can be achieved with particular functionality offered by the NServiceBus API and without the need for user-defined modules.

2.6.6 Conclusion

The evaluated technologies represent an extract of available middlewares for the Microsoft Windows platform, examining the spectrum of existing frameworks. The following products should not be left unmentioned:

- MSMQ: The message queue service allows a decoupled communication between components. As shown in the individual evaluation chapters, various technologies depend on the MSMQ. The message queue service can be used to extend the WCF with buffers for queued calls in order to decompose a workflow and separate the disjoint operations in

time. Therefore, MSMQ can be considered as a disconnected, message oriented extension of the WCF.

- Other ESBs: An enterprise service bus is often used as catch-all term for a messaging abstraction layer [51]. Although, it is discussed whether an ESB can be considered as an architectural style, this evaluation will reference an ESB as a tangible product. Current implementations include Neuron ESB, BizTalk ESB, Agatha, AppFabric Service Bus, to mention a few of them.
- Distributed Cache: Products providing distributed cached capabilities are ignored in this evaluation because they are typically not capable of event-driven programming [52]. SharedCache, NCache, Java Caching System, Swarmcache are just a small extract of available solutions.

The results of this chapter are shown in a table for easy comparison of the features between the different middleware's.

	Learnability	Usability	Security	Discovery	Replication	Patterns		
						P/C/O	R/R	SR/MR
WCF	+	+	+	+	N/E	o	+	+
XcoSpaces	+	+	-	+	-	+	o	o
XcoAppSpace	o	+	o	+	N/E	o	+	+
DSS	-	+	o	+	N/E	o	o	o
NServiceBus	+	+	o	o	N/E	o	o	o

Table 2: Overview of evaluated middleware and their features

+/o/-	*feature availability: extended/medium/poor or not available* *pattern fitness: excellent/possible with some additions/impossible*
N/E	*Not evaluable for that middleware*
P/C/O	*Extended Producer/Consumer/Observer*
R/R	*Request/Response*
SR/MR	*Single-Request/Multiple-Response*

The comparison highlights a significant difference in paradigms between these middlewares: because of the event-driven and message oriented architecture, most technologies rely on disjoint message passing. Therefore the classification according replication mechanisms mainly cannot be evaluated. In contrast, the SBC paradigm which is represented by XcoSpaces follows other communication strategies. The framework allows simple implementations of the exchange pattern in a decoupled fashion, combining distributed cache aspects with event-driven notification capabilities.

For Request/Response scenarios, WCF provides an easy-to-use but extensible and flexible platform for remote communication. The foundation can be configured to fit the individual needs and can be included into applications without additional hosting environments. The combination and configuration capabilities of the WCF do not allow any narrowing of the usage scenarios.

XcoAppSpace represents a step toward a more decupled communication, allowing an appropriate implementation of the Publish/Subscribe and all evaluated exchange patterns. Because of the initial complexity of the CCR runtime this technology pays off especially when high concurrency is expected beforehand.

DSS has similarities to XcoAppSpace and is suitable for scenarios where concurrency and resource-oriented interfaces are important.

ESB technologies can be used in various settings, depending on the respective product. Frameworks such as the BizTalk Server are large and heavy-weight installations, mediating a broad variety of information and protocols. Other technologies like the evaluated NServiceBus are designed for quick and easy access and can be hosted directly within other applications. ESB products typically provide a wide range of additional services and functionalities like logging, exception handling and security features. ESB-oriented architecture is often mentioned in conjunction with SOA in order to mediate service compositions and to facilitate additional intermediate functionalities.

XcoSpaces and the SBC paradigm use a shared data space approach to achieve remote interaction. This technique can be considered as an event-driven distributed cache, promising significant increase in scalability in some scenarios. Identifying possible deployment scenarios is difficult since the final purpose of the objects transmitted and maintained by the space is still in research. An example application which addresses a setting for usability evaluation will be presented in *Chapter 6.1.1*.

LinqSpace will rely on the XVSM specification. In contrast to the XcoSpaces implementation the entries used to interact with the space will distinctly be specified as entities of a domain model. It should be possible to maintain a predefined ER-Modell with constraints and relationships and use it for remote collaboration through a LINQ interface.

3 .NET technologies used by LinqSpace

This chapter introduces the core .NET technologies used to implement LinqSpace and describes the resulting advantages and disadvantages.

3.1 Language Integrated Query (LINQ)

Many software development projects depend on efficient data manipulation. Conditions like access and query speed, adaptable and versatile code, persistence, reliability and readability are only an excerpt of important indicators which are taken into consideration when evaluating project requirements. When it comes to persistence there is a gap between the object-orientated fashion of programming language and the relational behavior of databases. There are also object-oriented databases which do not face this problem, but they reside in niche markets and are not evaluated in this work [53,54]. The technique to convert the data between these incompatible type systems is mainly addressed as object-relational mapping [55] and was one of the initial motivations behind LINQ [56]. SQL statements are a good example to address this type mismatch. On one side there are rich development runtimes and programming languages (C++, C#, Cyclone, Standard ML) which mostly rely on type safe environments designed to mainly discourage or prevent discrepancy between differing data types. On the other side there is a wide range of storage systems, many with special query

languages or APIs (SQL, NoSQL API, SPARQL, DMX) which offer extensive capabilities for data interrogation. But between these two environments, queries are typically transmitted as simple strings like SQL statements. The request and response communication channels used are neither type safe nor can a static check at build time detect the correctness of SQL statements or their result types.

LINQ is designed to fill this gap between the two environments by offering a type safe, powerful and unified interface for querying a wide-ranged variety of data sources. Query statements are treated as first-class citizens in .NET languages and can be stored and iterated just like ordinary collections or enumerations. LINQ syntax is similar to SQL statements and therefore keeps the learning curve low for people that already have experience with SQL. It is a step toward a more declarative and functional way of expressing requests. LINQ comes with a built-in support for accessing data from in-memory objects (LINQ to Objects), XML (LINQ to XML) and two different providers for databases (LINQ to SQL, LINQ to Entities). The data sources for inquiry are also referred to as LINQ flavors. Further, LINQ allows mixing and interacting with data coming from various sources within a single query.

There is an increasing rate of LINQ-Provider implementations offering query capability of various sources. LINQ to Amazon, LINQ to JSON[1] and LINQ to Google are only a small extract of currently available providers [57].

The .NET Framework 4.0 was released with the IDE Visual Studio 2010 on April 12, 2010 [58]. New features of that framework target an easier way to maintain parallel tasks (TPL[2]) therefore LINQ got a parallel extension called PLINQ[3] allowing concurrent execution of time consuming queries or filters in a unified and easy to read manner.

3.1.1 Extension methods

LINQ heavily relies on extension methods, which were introduced with .NET Framework version 3.0. Basically these kinds of methods are simply static methods which can be called as if they were part of an instance with the associated object as first parameter. Static methods are called in prefix notation in contrast to extension methods which are called in infix notation. The latter produces more readable code, especially when the result is immediately used for another operation as used in fluent interfaces [59] *(*see *Code 3)*. Essentially the whole LINQ library is implemented as extension methods of the two key interface types IEnumerable<T> and IQueryable<T>. Both interfaces can be interpreted as iterators. This makes LINQ very flexible because every type or collection which supports iteration by implementing one of these interfaces can be considered as a LINQ data source. Of course all collections which are part of the .NET Framework support those interfaces and can be queried.

[1] JavaScript Object Notation [121]
[2] Task Parallel Library [122]
[3] Parallel LINQ [123]

Code

```
public static class PersonHelper
{
    // extension method
    public static string GetFullName(this TestPerson person)
    {
        return string.Format("{0} {1}", person.Firstname, person.Lastname);
    }
}

// call extension method as if it were an instance method from testPerson (infix notation)

testPerson.GetFullName();

// the previous call is rewritten during compilation as follows (prefix notation)

PersonHelper.GetFullName(testPerson);
```

Code 3: Extension method example

3.1.2 The IEnumerable interface

IEnumerable<T> is a key interface of LINQ and serves as decorator [60] mainly for collections and in-memory querying. The interface is designed generic which guarantees type safety and avoids type boxing [61]. LINQ allows filtering and modifying the retrieved objects in a more intuitive way than using, for example, if-statements as filter. The example *(Code 4)* shows the stages and different syntax styles of LINQ. The three resulting IEnumerable's are equivalent and differ only in their readability. The query demonstrates a simple filter over an entity property called "Age" combined with a projection over the property "Firstname". The first example shows how to decorate an IEnumerable interface (in this particular case a generic list) with LINQ standard query operators by ordinary extension method calls (also known as method syntax of LINQ). This is how all LINQ queries end up and can be interpreted by the CLR. A delegate is used for the filter and projection expression to customize the outcome when iterated (see *Signature 1*). A more common way to use LINQ extension methods are lambda expressions [62], which are used by the compiler to create the same delegates as in the first example. The last sample adds syntactic sugar by using the LINQ query syntax which is a very declarative way to formulize queries. The CLR itself has no knowledge of this query syntax and it is up to the compiler to create the corresponding method syntax.

Signature

```
public static class Enumerable
{
...
    public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
                                                       Func<TSource, bool> predicate);
...
}
```

Signature 1: Enumerable.Where

Code

```
 // the data source is a list which implements IEnumerable
IList<TestPerson> testPersonList = new List<TestPerson>();

// create an IEnumerable with method syntax and delegate
IEnumerable<string> enumFirstnameDelegate = testPersonList
                .Where(delegate(TestPerson testPerson) { return testPerson.Age > 30; })
                .Select(delegate(TestPerson testPerson) { return testPerson.Firstname; });

// create an IEnumerable with method syntax and lambda expressions
IEnumerable<string> enumFirstnameLambda = testPersonList.Where(testPerson => testPerson.Age > 30)
                                                        .Select(testPerson => testPerson.Firstname);

// create an IEnumerable with query sytnax
IEnumerable<string> enumFirstnameExpressionSyntax = from testPerson in testPersonList
                                                    where testPerson.Age > 30
                                                    select testPerson.Firstname;
```

Code 4: IEnumerable creation examples

It is important to mention that the decoration of IEnumerables with LINQ queries does not touch any data (see deferred execution at *Chapter 3.1.5*). When it comes to iteration of the enumerable, the actual process starts: data is fetched item per item and the result is modified according to the decorated query. The original data source itself is not altered, which reflects similarities to the functional programming paradigm [63]. *Figure 5* illustrates this procedure.

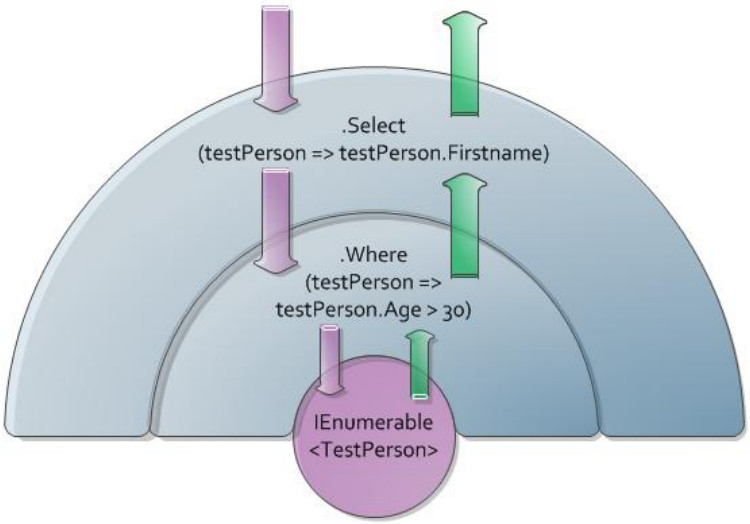

Figure 5: IEnumerable decorator

3.1.3 The IQueryable interface

There are requirements where decorating enumerable interfaces does not solve querying problems. This scenario is encountered if the inquiring process is not executed in the .NET environment. Querying a database, third-party components or even a data source provided by internet services quickly rises beyond IEnumerable capabilities. Furthermore, it is not an option to store the entire library in memory just to pick out the data needed by using the IEnumerable interfaces.

LINQ provides a second possibility to create queries which takes these requirements into account. In contrast to the IEnumerable interface which focuses on decorating the data source with the desired query, the IQueryable interface centers on the query itself. A second set of

standard query operations, also implemented as extension methods, is designed to target the IQueryable interface. These methods are similar to the ones used with the IEnumerable interface according the usability of query creation. The difference lies in the way selectors and filters are passed to extension methods. According to the IEnumerable interface those can be an arbitrary delegate where IQueryable restricts this parameter to be lambda expressions, which can be transformed into expression trees *(see Signature 2)*. Expression trees are topic of *Chapter 3.1.4*.

Signature

```
public static class Queryable
{
...
    public static IQueryable<TSource> Where<TSource>(this IQueryable<TSource> source,
                                                      Expression<Func<TSource, bool>> predicate);
...
}
```

Signature 2: Queryable.Where

LINQ queries created with the IQueryable interface are uncoupled from the underlying data source and represent a composition of standard query operators. Generally speaking, IQueryable represents an immutable query stored as expression tree in the "Expression" property of the interface. In addition, the QueryProvider in the "Provider" property *(see Signature 3)* can be accessed to expand the current query. This usually happens by replacing the IQueryable object by a new one which represents the current query. The QueryProvider stays the same and the generic parameter of the IQueryable interface reflects the actual result type. This technique allows the generic return type to be in sync with the query and therefore a type-safe result enumeration.

When it comes to iteration the QueryProvider is asked to execute the current expression tree and deliver the desired result. The interaction between the IQueryable and IQueryProvider interface is visualized in *Figure 6*.

Signature

```
 public interface IQueryable : IEnumerable
{
    Type ElementType { get; }

    Expression Expression { get; }

    IQueryProvider Provider { get; }
}
```

Signature 3: IQueryable

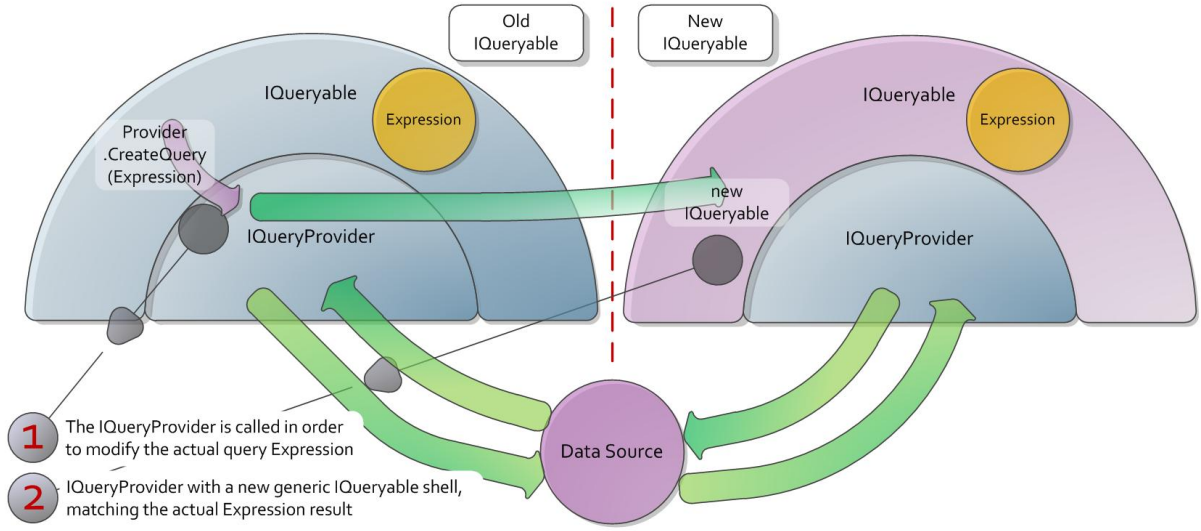

Figure 6: IQueryable and IQueryProvider interaction

It lies in the responsibility of the QueryProvider to take and execute the expression tree provided by the IQueryable interface. This is a great opportunity for interdisciplinary querying but includes that the capabilities of the underlying QueryProvider must be known in advance. Otherwise, if the provider is not capable of the query expressions it will throw an exception at execution time.

3.1.4 Expression trees

Expression trees are a way to express executable code in a tree-like immutable data structure [64]. There is a set of nodes which can be used to build up an expression tree which includes various types of method calls and special nodes for representing formulas, equations and comparisons. The compiler can transform every lambda expression into an expression tree just by assignation.

Expression trees are able to represent all types of standard query operations which basically are static method calls. The query from example *Code 4* would result in the expression tree shown in *Figure 7*.

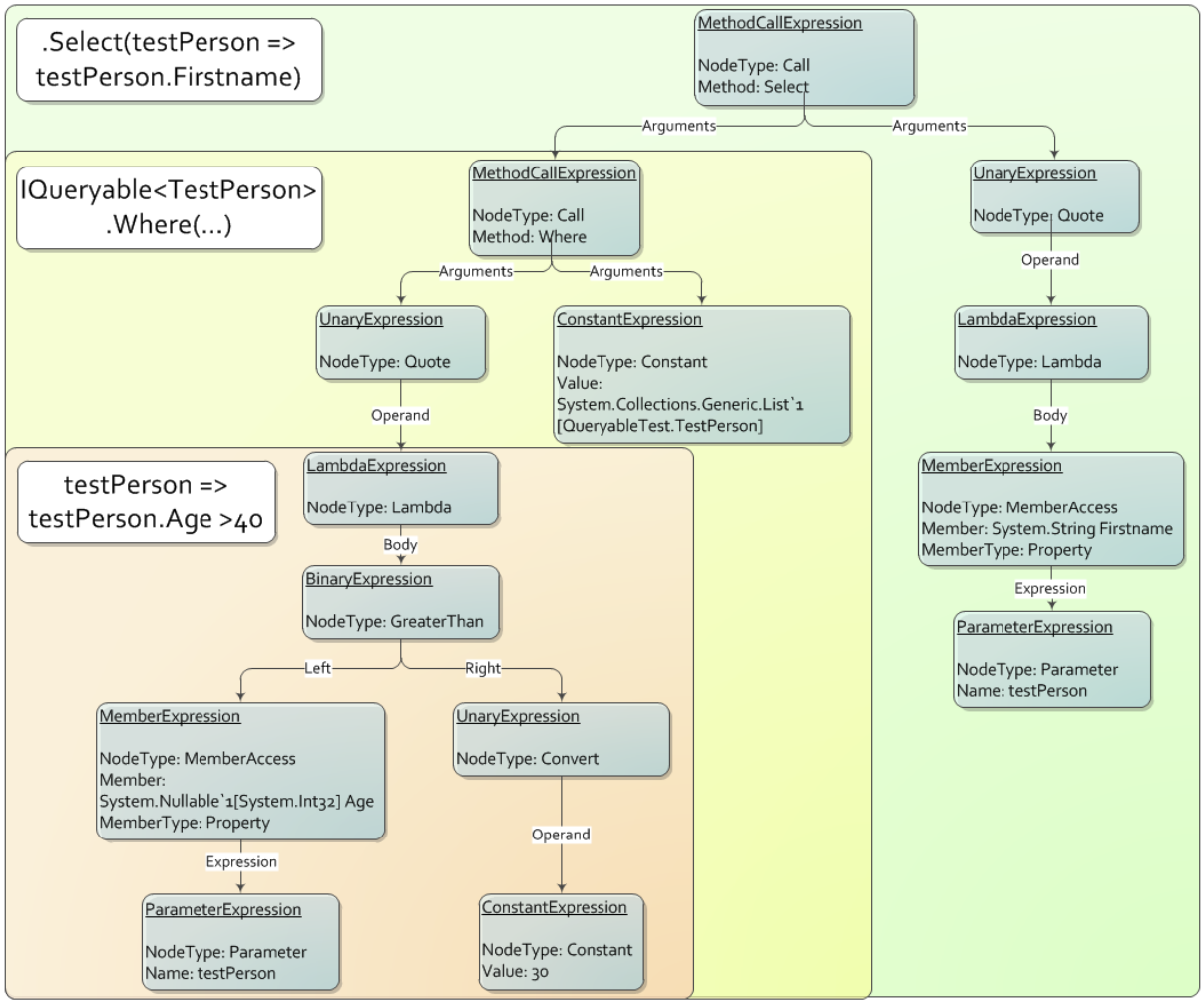

Figure 7: Expression tree example

A query represented as an expression tree is quite valuable if it comes to traversing and transforming the query for other platforms. As shown in *Signature 2* the IQueryable interface relies on query expressions represented as a tree, which is passed as “predicate” parameter of the extension function.

Expression trees are important cornerstones of LINQ’s flexibility. The underlying provider can rely on a well-defined, traversable structure. For example, the LINQ to SQL provider maps the nodes of the tree to their corresponding SQL representations.

3.1.5 Deferred execution

Data is fetched on demand and only when the actual request is placed. That is the case when either an iteration of the interface is started or a LINQ function requires immediate execution, for example aggregation functions like Sum or Count. This technique utilized by LINQ is called deferred query execution or deferred query evaluation and represents the default behavior (see *Code 5*).

Code

```
IQueryable<TestPerson> testPersonQuery = from person in dataSource
                                         where person.Age > 30
                                         select person;

// Query is saved in testPersonQuery but no data is fetched

foreach (TestPerson testPerson in testPersonQuery)
// execution of query which results in an IEnumerable interface
{
    // TestPerson objects are lazily fetched, one by one through IEnumerable interface
    Console.WriteLine("{0} {1} is {2} years old",
        testPerson.Firstname,
        testPerson.Lastname,
        testPerson.Age);
}
```

Code 5: Deferred execution code example

3.2 Entity Framework (EF)

A major challenge of software development is the storage and inquiry of data in an effective way. Referring to the domain model pattern [65,66], data objects are denoted as entities and are mainly modeled from their real world representations. A significant point which had great influence in the evolution of data models is the way relationships between entities are treated. Various data models have been proposed which offer different views on the logical data schema [67].

- Network model: [68,69,70]
  The network model allows a natural representation by separating entities and relationships between entities. The schema can be viewed as graph structure where types are represented as nodes and their relationship as arcs.
- Relational model: [71]
  The network model was mainly displaced by the relational model due to the more declarative and higher-level interface [72]. Based on the first-order predicate logic, the content of the database is defined as a collection of predicates over a finite set of predicate variables which achieves a high degree of data independence. It allows specifying additional constraints which have to be met at any given time and therefore ensure consistency. This environment permits the database designer to create a dependable representation of information.
- Entity-Relationship model (ERM): [67]
  The Entity-Relationship model is supposed to be the most suitable data model because it captures the majority of important facets and semantics of the real world [73,74,75] and expresses them in a natural and easy understandable way [76]. The Entity-Relationship model suggests an abstract and conceptual representation of data which describe the ontology for a certain information environment. The design of the model mainly relies on linguistic aspects to provide natural language constructs, where entities can be thought of as nouns and relationships as verbs [77].

The ADO.NET EF eliminates the object-relational impedance mismatch by providing a bridge between the relational schema which is commonly used by databases and its conceptual schema of the entity-relationship model used by the application [78]. The EF provides a loosely coupled three layer architecture. The first one is a conceptual model which

is the actual EDM[1]. The second one represents the database schema or storage model and the third one describes the mapping between the first and the second one [79].

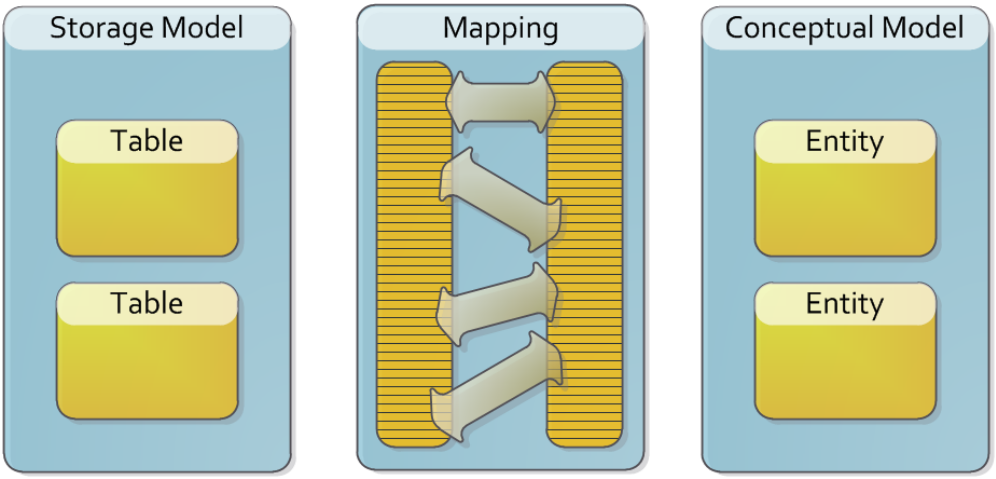

Figure 8: Entity Framework layer architecture

The conceptual data reflects business objects so the application does not have to worry about the structure of the database or storage layer. Data access and storage is done against the conceptual model. Further, schema changes in the storage layer can be compensated through the mapping and may not be reflected in the business objects. Relationships between entities are can be accessed over so called navigation properties which eliminate the usage of SQL-JOINS in the application which normally would be necessary in interaction with relational schemas. The ADO .NET EF communicates with the underlying database on the basis of common SQL query syntax and does not rely on a specific database.

The two different models and the mapping information are stored in an XML metadata representation and can be modified either directly or with a visual design tool included in Visual Studio. The EF version 4 supports the following approaches to build up the models:

- database-first design:
  The database-first design was the first technique available in the EF. It relies on an already existing database which is used to subsequently create the matching conceptual model.
- model-first design:
  In order to allow model driven software development the model-first design is able to create a database reflected from a conceptual model representation. At any given time the model can be used to generate SQL commands which update a current database or create it from scratch.
- code-first design:
  This technique is currently part of the EF Feature CTP[2] (further referred to as EF Feature) and therefore still in development [80]. Following a persistence ignorant strategy [81] the entities are mainly POCO[3] with attributes marking key properties and introducing validation and data annotation information. This approach allows working solely with classes, metadata information for the conceptual and storage model are generated at runtime. In turn, the model can be used for SQL script and database creation.

[1] Entity Data Model [79]
[2] Community Technical Preview
[3] Plain Old Common Language Runtime Objects [21]

Although the code-first approach is still in development it is the best choice for LinqSpace. By omitting the creation of a model and a database beforehand, the behavior of the XVSM specification is mainly approximated. According to that specification the entry class type itself has no significance (schema-free) and is mainly treated as black box.

The following sections (*Chapter 3.2.1 – Chapter 3.2.4*) explain details about the EF and its functionality which will be referenced in the upcoming implementation *Chapter 5*.

3.2.1 LINQ for Database inquiry

Currently two LINQ providers are delivered as part of the .NET platform which offer the ability to query against a database [82,83]. LINQ to SQL evolved from the LINQ project and maps the query directly into its corresponding SQL statement optimized for Microsoft's SQL Server. Linq to SQL supports no other database which makes the provider no first choice as storage backend for LinqSpace. The second provider is called LINQ to Entities and targets the ADO.NET EF. Microsoft announced that the EF team adopted the LINQ to SQL provider and that they would focus on LINQ to Entities in future developments.

3.2.2 Entity change tracking

The ObjectContext is the base class when working with the EF. In practice this class is inherited and extended to fit the application needs for creating and executing queries in a type safe fashion. By default, each object returned as result of a query is still attached to its ObjectContext which keeps track of changes made regarding the objects properties. When the ObjectContext is asked to save the recorded changes only the modified values will be used to update the database. This awareness of modifications can be accomplished in several different ways, but since this work focuses on the use of POCO and persistence ignorance there are two strategies offered by the EF.

- The EF can make a snapshot of the object's properties when it is materialized from the ORM[1]. When the storage is about to be updated the modified object properties are compared with the original ones saved in the snapshot.
- The second possibility EF offers is to deliver a proxy class instead of the original entity. An essential prerequisite of this approach is that every property in the entity class must be marked as virtual. The EF uses reflection to discover the properties and creates a runtime proxy class which inherits from the original entity class. The properties are overridden to send notifications to the ObjectContext when changes are made during process. Using proxies omits the time spent for comparing the entity properties to the snapshot and therefore results in faster database updates. On the contrary it can only be used when the application does not rely on the original entity type. Proxies also enable lazy loading (see *Chapter 3.2.4*) and relationship fix-up[2].

3.2.3 Instance creation

The ObjectContext class is designed as light-weight instance. Typically it can be seen as unit of work pattern [81,84] and should be created for every request independently. Moreover it guides to a very clear convention when the scope of the ObjectContext class is limited by a C# using statement block. Every entity object which lifespan goes beyond the using block

[1] Object-Relational Mapping [79]
[2] method for synchronizing two-way relationships between entities [79]

should be detached from its ObjectContext. Using the ObjectContext as singleton would cause memory problems because the EF will keep tracking objects from query results. ObjectContext is intentionally not thread safe and therefore does not suffer from blocking delays which would be necessary to allow concurrency. This design allows creating distinct instances for parallel tasks and therefore ensures multithreaded access to the DB without interference between instances.

3.2.4 Deferred loading

Deferred loading addresses the supply of linked entity data on demand. The relation toward another entity is expressed as navigation property and is usually retrieved from the underlying database when a specific request is encountered. This can happen explicitly when the ObjectContext is asked to fetch the related entities or implicitly by the decorating proxy class, which is also referred to as lazy loading.

4 XcoSpacesQueryable

The first approach to enrich XVSM with LINQ query capabilities is to implement a prototypic extension for the .NET implementation XcoSpaces. The important aspect of this experiment is to evaluate the query capabilities of a XVSM reference implementation. Another question is the extent to which the API can take advantage of LINQ technology.

A possible solution bases on an additional coordinator, capable of LINQ inquiry. Since LINQ expression cannot be serialized from .NET by default, special serialization functionality is required in order to transmit the query to remote peers. To preserve the enumerable behavior of LINQ (see *Chapter 3.1.2*), the surrounding XcoSpaces infrastructure would have to process and transmit results per entry, which are currently handled at once in a list.

To avoid extending the very foundation of XcoSpaces the idea is to use extension methods to mimic LINQ behavior, parse important information from the expression tree and finally use the existing interface to place the query. Since LINQ queries usually target entity properties, an intrinsic coordinator would be an appropriate candidate for the extension. Obviously the QueryCoordinator defined in the XVSM specification (see *Chapter 1.2.2*) is the best matching interface. Since XcoSpaces has no corresponding implementation for that coordinator the LindaCoordinator is the only intrinsic coordinator left.

4.1 LINQ extension

In order to create a LINQ extension for the LindaCoordinator a special implementation of IQueryable and IQueryProvider is needed to redirect the query execution. An effective way regarding reusable software components is to inject a delegate at the construction of the class implementing IQueryProvider (see *Code 6*) which will be called when the interface is asked to execute a query. The relevant information is passed along with the invocation of the delegate. This technique will disburden the QueryProvider from the exclusive execution logic which will reside in the more suitable extension method.

Code

```
public class XcoSpacesDelegateQueryProvider : IQueryProvider
{
    private Func<Expression, Type, object> _executeDelegate;

    public XcoSpacesDelegateQueryProvider(Func<Expression, Type, object> executeDelegate)
    {
        this._executeDelegate = executeDelegate;
    }

...

    public TResult Execute<TResult>(Expression expression)
    {
        return (TResult)this._executeDelegate(expression, typeof(TResult));
    }

    public object Execute(Expression expression)
    {
        return this._executeDelegate(expression, expression.Type);
    }
}
```

Code 6: Execute delegation of IQueryProvider implementation

The IQueryable implementing class simply acts as storage of the current query.

To offer an intuitive way to start query creation an extension method is provided which targets the XcoKernel class, the main class for XcoSpaces interactions. The parameters of that extension method match to the Read method signature of the XcoKernel class. When the query is executed the anonymous method is called and in turn traverses the achieved expression tree to extract equality expressions and subsequently build a template class for the LindaCoordinator (see *Code 7*).

Code

```
public static IQueryable<T> QueryRead<T>(this XcoKernel kernel, ContainerReference cref,
                                         TransactionReference tref, int timeout, int count)
                                         where T : ILindaMatchable
{
    return new XcoSpacesDelegateQueryProvider((Expression expression, Type type) =>
    {
        T queryObj = Activator.CreateInstance<T>();

        new EqualityExpressionExtractor<T>(queryObj).Visit(expression);

        return kernel.Read(cref, tref, timeout, new LindaSelector(count, queryObj))
                     .OfType<Entry<T>>().Select(entry => entry.Value);

    }).CreateQuery<T>(Expression.Constant(new XcoSpacesQueryable<T>()));
}
```

Code 7: XcoSpaces queryable read extension method

4.2 LINQ API usage

To demonstrate the usage of the new LINQ interface an example is shown which points out the difference in contrast to the traditional interface provided by XcoSpaces. An ordinary query to retrieve entries with matching properties of a “TestPerson” class would look like *Code 8*.

Code

```
List<IEntry> result = kernel.Read(cref, null, 0, new LindaSelector(Selector.COUNT_ALL,
    new TestPerson() { Age = 30, Firstname = "FirstC" }));
```

Code 8: Linda query with ordinary XcoSpaces API

The result is a list containing objects implementing the IEntry interfaces which need to be downcasted before their value can be accessed (see *Signature 4*).

Signature

```
public interface IEntry
{
    bool KeepSerialized { get; }
    List<Selector> Selectors { get; }
    Type Type { get; }
    object Value { get; }
    void Serialize();
}
```

Signature 4: XcoSpaces IEntry

The equivalent query with the new LINQ extension and query syntax is shown in *Code 9*.

Code

```
List<TestPerson> result =
    (from testPerson in kernel.QueryRead<TestPerson>(cref, null, 0, Selector.COUNT_ALL)
     where testPerson.Age == 30 && testPerson.Firstname == "FirstC"
     select testPerson).ToList();
```

Code 9: Linda query with LINQ query syntax

The ToList method is used to force immediate query execution which results in a list containing the desired entries.

4.3 Conclusion of XcoSpacesQueryable

By looking at the code snippets showing the different API interfaces, the simplicity and clarity of the LINQ query is remarkable. Each developer who has experienced LINQ or even has used SQL statements immediately recognizes the syntax and quickly gets a picture of the inquiry. Investigating further there are many opportunities for potential improvements, for example the amount of desired entries could be propagated through LINQ's Count method instead of passing Selector.COUNT_ALL to the extension method.

Despite of the readability of the LINQ query shown in *Code 9* the use of LINQ as a gateway for the LindaCoordinator must be questioned. A developer who is experienced with LINQ would be lured to misguide the query capabilities and try to use expressions which are not supported. In fact the current implementation only supports equality comparisons which can be concatenated by conditional-AND operators (&&). Although the concatenation issue could be solved by simply detecting conditional-OR operators (||) and executing multiple queries against XcoSpaces, the equality comparisons are the only operations supported by template matching strategy offered by the LindaCoordinator. Therefore the spectrum of LINQ operations is very limited and access to unsupported functionality would result in an exception at run time.

5 LinqSpace Implementation

The architecture used for the implementation of LinqSpace relies on the specification prescribed by the formal model of XVSM [8]. *Figure 9* shows the layer diagram with the associated dependencies between the components.

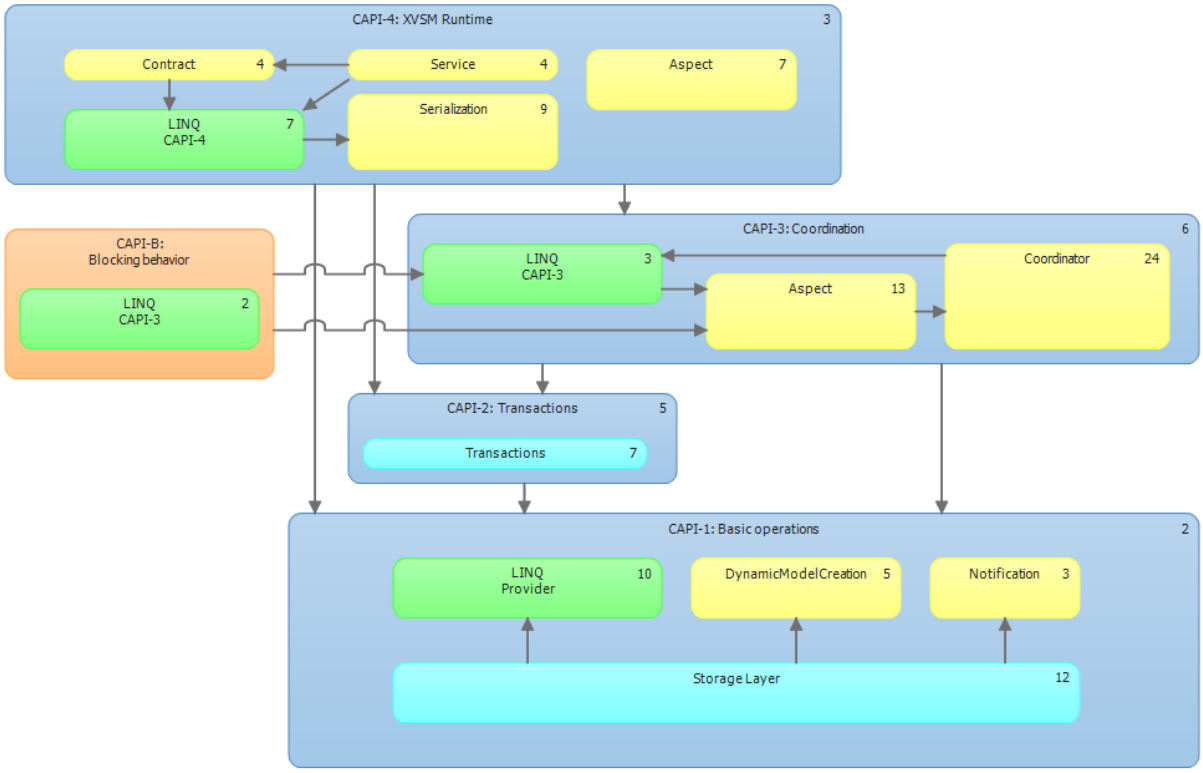

Figure 9: LinqSpace layer diagram

Since LinqSpace targets the domain model, the term "entity" will be used because it is commonly shared in that context. Other space implementations generally refer to data as "entry" because they do not predefine user data as domain model.

The CAPI interfaces inheritance hierarchy should follow the LSP[1] [85], meaning every interface is able to enrich the comportment without altering the base behavior. Semantically the interfaces fulfill the requirements by providing CRUD[2] operations at the CAPI-1 level which are extended throughout the subsequent layers. But since CAPI-3 and CAPI-4 alter the method signature with additional parameters like coordinators used to write entities or space identifiers for distributed execution, the interfaces do not directly inherit from each other and therefore break the LSP syntactically. To provide adequate LINQ access (see coordinator implementations in *Chapter 5.3*) this violation is necessary under these circumstances and therefore this architecture was chosen.

The different CAPI layers provide a fully functional implementation which can be used to operate at the current level of functionality. Each tier in the CAPI architecture can be seen as the final API interface which ensures loosely coupled components but adds some special treatment when it comes to LINQ's deferred execution. But the overhead is not overwhelming and can be outweighed by the loosely coupled and flexible interfaces.

[1] Liskov Substitution Principle
[2] Create Read Update Delete

5.1 CAPI-1: Basic Operations

The purpose of this layer is to provide a uniform way for data storage and to offer the basic operations of the space, namely read, write and take. The natural usage of LINQ queries foresees to return a collection of entities which does not fit with the behavior of the delete operation. It would be possible to offer a workaround by executing the DELETE operation directly against the database and avoid fetching the entities in memory just to delete them afterwards. But for the sake of simplicity this operation is not provided by the current implementation of LinqSpace.

To address the needs of the SBC paradigm, some considerations have to be taken into account. There should be no need to create a domain representation in advance. Due to the capability, flexibility and convenience in the .NET development environment, the ADO.NET EF has been chosen as the primary storage component for LinqSpace. Nevertheless, this basic layer is designed to be interchangeable so other frameworks capable of comparable functionality like LINQ to SQL or NHibernate can be plugged in.

5.1.1 Coordinators in the Entity Framework

The EF is designed to map relational database storage to a conceptual model which refers to the domain model in DDD. The domain entities are known at compile time and, regarding performance concerns and type safety, the interfaces for LinqSpace interactions are provided in a generic way. Queries performed by LINQ typically target entity properties which meets the basic principle of intrinsic coordinators. The question arises where and in which fashion the additional coordination information used by extrinsic coordinators should be stored.

An opportunity would be to separate the concerns of the database, which should only store the relational representation of the domain model, and the space by keeping the coordination information in a in-memory storage. This would result in a highly optimized retrieval of entity identifiers because of hash usage optimized for the needs of each individual coordinator. But the entity object itself must be retrieved from the database in a second step which would ruin the performance benefit encountered by the hash. Moreover the coordination information would not be persistent which, in case of a crash, would result in abandoned entities in the database which no longer can be addressed using extrinsic coordinators.

Another possibility is to store the coordination information along with the domain data in the database. An important decision is how the data is warehoused with respect to the relational, indexed and static behavior of the database storage. A way to provide continual coming and going of coordinators along with their desired information is to store the data directly with the entity data as xml, illustrated in *Figure 10*.

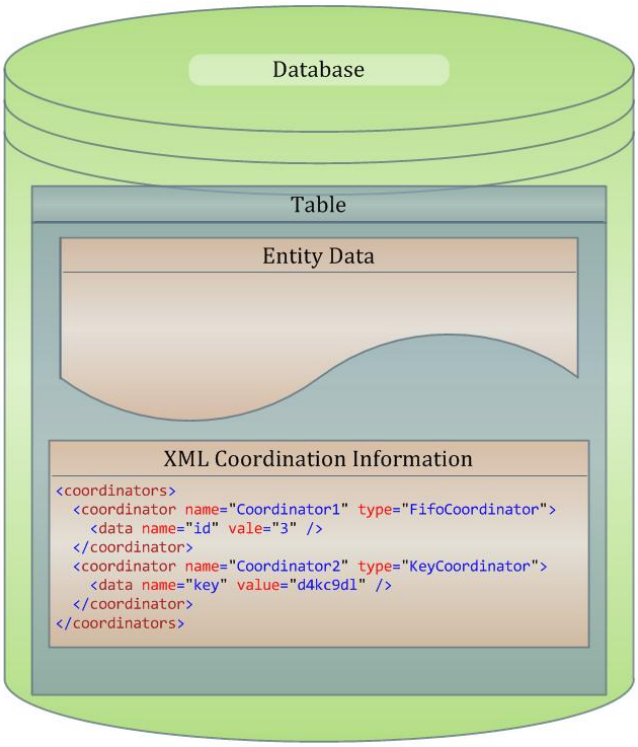

Figure 10: Table with XML coordination information attached

Once the database is created on the basis of the conceptual model, coordination information can be introduced without disturbing the database structure. Intrinsic queries which rely on the entity data can be completely outsourced in the responsibility of the DBMS[1] which should be capable of the execution. On the other hand, extrinsic queries depend on parsing the XML information for each entity. Although DBMS nowadays are capable of XML data types it is still a very expensive task and therefore not an optimal solution for LinqSpace.

The way chosen for the storage of coordination information was to take advantage of the entity relationship model and link the data to the entities. Each coordinator can store the desired data in an explicit database table which is related through foreign keys to the primary entity (see *Figure 11*). This gives the database insight in the coordination information which subsequently can be indexed by the DBMS for faster query execution.

[1] Database management systems

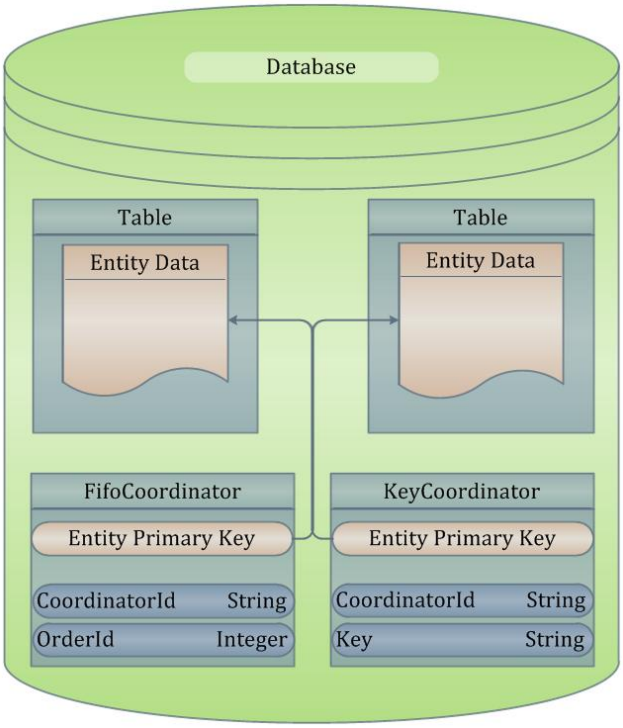

Figure 11: Table with coordination information linked

The entity and coordination data are stored separately which allows attaching coordinators on demand on entity level. Extrinsic queries can now be performed on distinct tables and the association to the entity data can be expressed in a single request. Furthermore, database constraints can be defined to ensure data integrity, for example the key used by a specific coordinator must be unique. Such policies guaranteed from the database are quite useful and transfer complexity away from the space into the DBMS.

The disadvantages manifests in a complex and rigid database structure. Each coordination table must be associated with each entity table, resulting in foreign key columns in the coordination tables for every entity primary key. The entity Primary keys are predefined by the user and the conceptual model and may contain a single or a composition of values. Nevertheless this strategy leaves the original data tables unaffected and gains performance through indexing by the DBMS which justifies the significant structural overhead. The performance influences associated by the cross-table JOIN operations will be presented in *Chapter 6.4*.

5.1.2 A Container-Name extension for the Entity Framework

Referring to DDD there is no reason to split up an entity type across a variety of containers. Usually this behavior is handled under the cover, for example an inheritance hierarchy in the conceptual model may result in an overlapping table with a discriminating column (TPH[1]). However, in order to create a separation within the same entity type LinqSpace injects a custom property into user entities, resulting in a discrimination column for each data table. This property should not be visible in the conceptual model provided by the user. Since there is no actual container object and the container-name just filters a table of user data, no special functionality is offered to create a container (see *Code 10*). Nevertheless, the ClearContainer

[1] Table Per Hierarchy

function API function allows removing all entities from a specified container (see *Signature 6*). The container name is passed as a string to the CAPI operations and there is no ContainerReference object necessary.

Code

```
TestPerson testPerson = capi1.CreateObject<TestPerson>();
testPerson.Age = 22;
testPerson.Firstname = "John";
testPerson.Lastname = "Doe";
capi1.Write(testPerson, "containerName");
```

Code 10: Example for write operation on container

A second possibility would be to store the container-name in a separate table, as it was done with coordinators described in *Chapter 5.1.1*. Since the container is highly related to the entity and due to the performance cost associated with an additional relation lookup the decision was made to directly include the container-name into the entity.

Another consideration was to store the distinct containers as separate tables. With regard to the execution performance, this would be the most obvious since this approach would spare a filter operation in order to discriminate the table. Unfortunately, such an implementation is, due to the EF Feature tool, very difficult and would require a significant amount of workarounds in order to distinct an object type among multiple tables. When the discrimination resides in the same conceptual object, the container-name extension functionality does not require accessing the whole model in order to retrieve the right table.

5.1.3 Database creation

It lies in the responsibility of LinqSpace to create a database schema which meets all necessities of storing entities and coordination information which may be imposed by the user. The entity types delivered are POCOs and designed to meet the PI[1] concept.

The ordinary way to accomplish these requirements would be to use the Entity Data Model Designer included in the Visual Studio IDE. This tool allows the creation of the conceptual model and the mapping to the storage model in a visual manner. Further it can generate SQL scripts for database generation and POCOs which represent entities in the business model.

LinqSpace should preserve the ordinary usability of the SBC paradigm where data is treated mainly as black box object without prior knowledge of the type or properties. Therefore the creation of a domain model by the user should not be a necessary prerequisite to use LinqSpace. The EF Feature [80] accomplishes another way to work with POCOs based on PI and the code first approach. By the time writing this work the version 5 of this EF Feature was already released but LinqSpace was programmed earlier and includes version 4 [86]. The new version introduces great features for example object validation and new change tracking mechanisms. LinqSpace would definitely profit of the new functionality but for presenting a prototype of a space based on the EF the old version is certainly adequate.

The code first approach relies on POCOs whose properties can be annotated by declarative, predefined attributes. Such attributes include primary keys, foreign keys and validation constraints. This mainly mimics the behavior specified by XVSM where no special model is needed to describe the objects. In the current EF Feature tool version, every entity needs one

[1] Persistence Ignorance [125]

or multiple primary keys which are marked with the “Key” attribute. To allow the framework to create a dynamic proxy class for entity change tracking (see *Chapter 3.2.2*) and deferred loading (see *Chapter 3.2.4*) all properties should be marked virtual. The generic ICollection interface can be used to introduce relations between different entity types. A simple example of a domain model containing addresses and persons which are correlated in a many-to-many relationship would look like *Code 11*.

Code

```
public class TestPerson
{
    [Key()]
    [StoreGenerated(StoreGeneratedPattern.Identity)]
    public virtual int PersonID { get; set; }

    [Required()]
    [StringLength(20)]
    public virtual String Firstname { get; set; }

    [Required()]
    [StringLength(20)]
    public virtual String Lastname { get; set; }

    [Required()]
    public virtual int Age { get; set; }

    public virtual int SexInt { get; set; }

    public virtual ICollection<TestAddress> Address { get; set; }
}

public class TestAddress
{
    [Key()]
    [StoreGenerated(StoreGeneratedPattern.Identity)]
    public virtual int AddressID { get; set; }

    public virtual string Street { get; set; }
    public virtual string City { get; set; }

    public virtual ICollection<TestPerson> Persons { get; set; }
}
```

Code 11: Code first example, TestPerson-TestAddress in many-to-many relation

An essential difference should be pointed out: To build up the database schema, an object called DbContext is provided by the EF Feature which basically extends the functionality of the original ObjectContext known by the EF. Simply put, this new context is able to inspect a list of object-types and generate the associated models needed by the EF. As a result, all the object types involved by space operations must be known in advance, precisely at the creation time of the DbContext. Since the object types used for space interaction should be deliberate anyway, this condition should not be a too extensive disadvantage.

5.1.3.1 Dynamically extended entities

To create the desired conceptual model with the extensions for coordinators (see *Chapter 5.1.1*) and container-names (see *Chapter 5.1.2*) a special modification of the user entities is needed. The idea is to inherit from the provided objects and implement interfaces in a new proxy entity object (see *Figure 12*). The dynamic entities have a one-to-many relationship to their coordination tables in order to access the same entity from several coordinators of the same type. For easy access of the dynamically created properties the CAPI levels can check

for the existence of the interface and work with the extension. These new proxy classes are passed to the DbContext for database creation which results in adequate changes in the database schema. By using this technique the extended properties can profit from lazy loading and are fully integrated in the domain model.

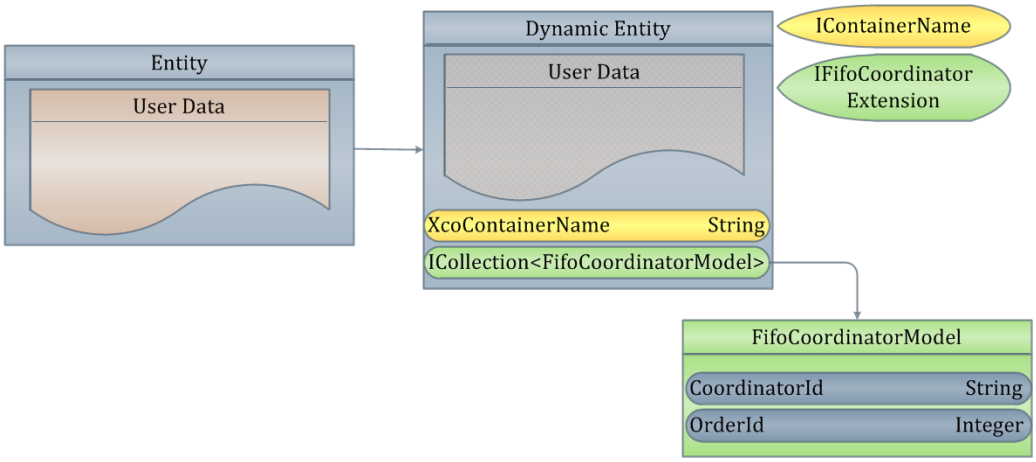

Figure 12: Dynamically created Entity

The IDynamicModelBuilder interface (see *Signature 5*) is responsible for assembling the objects which are about to be used with LinqSpace. The process starts by collecting all entities of the user's domain model. When this step is completed the IDynamicModelBuilder interface will be passed through the various CAPI layers which may include new or announce modifications on the existing entities by specifying additional interfaces. Further, IDynamicModelBuilder also collects SQL Commands which will be executed after the database schema is created. This technique can be considered as workaround or backdoor to update the database schema (the VectorCoordinator uses a statement to add a unique constraint on its table) with ordinary SQL statements. EF Feature also offers a way to alter entities on the level of the conceptual model with a Fluent API. That would be a better and uniform way to specify these modifications, but since the API is not capable of all desired adjustments in version 4 the SQL execution is offered as additional possibility.

Signature

```
public interface IDynamicModelBuilder
{
    void AddEntity(Type entityType);
    bool ModifyEntity(Type baseType, params Type[] newInterfaceTypes);
    Type GetFinalEntity(Type entityType);
    Type GetBaseTypeForType(Type entityType);
    void AddEntityConfiguration(Type entityType, params Type[] configurationTypes);
    void AddCommand(params string[] commands);

    IEnumerable<Type> EntityBaseTypes { get; }
    IEnumerable<string> Commands { get; }
}
```

Signature 5: IDynamicModelBuilder

When it comes to the creation of the database the DynamicModelBuilder class uses dynamic code generation with .NET reflection and the ILGenerator to introduce new entity types which implement the interfaces specified by the various CAPI levels. This technique is used by script engines and compilers, including the EF for dynamic proxy creation. The first step is

to define a new type which inherits from the provided base type and in addition implements all desired interfaces. The dynamic proxy type has to implement the properties desired by the interfaces so they have to be inspected. The properties will be marked virtual which allows the EF to subsequently create its own dynamic proxies. The final result of this process is an in memory type definition which extends the original one. There would be the possibility to create a DLL[1] file including the dynamically generated types which could be used later on, but since the construction process only happens at the creation of the space and the time taken is acceptable it is not implemented by LinqSpace.

After the database has been created the IDynamicModelBuilder interfaces serves as dictionary for type lookups. Since the EF may inherit once more from the entities during proxy creation the actual type of the object cannot serve as identifier for the type created by the DynamicModelBuilder. The methods provided by the CAPI interfaces are designed generic to support type-safe LINQ queries on the API side. Since the generic types provided by the user do not match to the ones used by the EF, a type upcast is inevitable and should happen in a well-designed fashion (see *Chapter 5.1.4*).

5.1.4 Implementation of CAPI1

To ensure independent development and loosely coupled components the bridge design pattern [87] is used to implement the relationship between the CAPI modules and the storage layer. The CAPI side of the bridge is made up of generic methods which provide the LinqSpace operations in an easy to use and type-safe fashion as shown in *Signature 6*.

Signature

```
public interface ICAPI1 : IEntityChangingNotification, IDisposable
{
    string write<T>(T obj, string containerName = null, bool saveChanges = true) where T : class;
    IQueryable<T> read<T>(string containerName = null, bool readPastLock = false) where T : class;
    IQueryable<T> take<T>(string containerName = null, bool readPastLock = false) where T : class;
    int update();

    T CreateObject<T>() where T : class;
    int ClearContainer<T>(string containerName) where T : class;
}
```

Signature 6: ICAPI1

The saveChanges parameter can be used to suppress the database update after the operation and can be used to bundle multiple write operation into a single database update process. This functionality is used by the writeBulk operation in CAPI-3 in order to combine multiple write operations (see *Chapter 5.3*). The readPastLock parameter will be discussed later in this chapter.

Of course the user has no knowledge of the dynamically created proxies and calls the methods with the original entity type as generic parameter. That is fine so far since the entities were instantiated as the extended proxy class and the CAPI modules can rely on casting the entities to the desired interfaces.

This is a crucial prerequisite which should be pointed out. When entities are about to be written the CAPI interface must first ensure that the entities are instantiated as the dynamic proxy type generated by LinqSpace. There are two scenarios according the creation of entity

[1] dynamic linked library

types. The typical way, which is also chosen by the EF, is to provide a factory method to instantiate the objects. LinqSpace uses the same technique by allowing the user to delegate the responsibility of creating a valid entity object to the generic CreateObject method (see *Signature 6*). This is a conventional setup according enterprise-level layered architecture. But to ensure that entities created by the user outside of the space can also be handled correctly, the CAPI write-operations check the type of the entities. If the base type was instantiated the LinqSpace creates the proxy entity type and copies the properties to the newly created object.

When the execution has passed the generic CAPI methods and jumps to the right side of the bridge pattern the generic type parameter which is passed along the method calls is converted into an ordinary "Type" object. From this point on, the implementation follows the decorator design pattern [87] which is split up into two distinct interfaces. The methods are identical besides the fact that the upper half of the decorators (IStorageImplementor, *Signature 7*) uses an object Type to pass the type information and the lower half (IStorageImplementorGeneric, *Signature 8*), including the actual EF Feature DbContext object, uses a generic type.

Signature

```
public interface IStorageImplementor :  IEntityChangingNotification, IDisposable
{
    void Add(Type genericType, object addObj, string containerName = null);
    object CreateObject(Type genericType);
    void Delete(Type genericType, object deleteObj);
    IQueryable Get(Type genericType, string containerName = null, bool readPastLock = false);
    string GetEntityIdentity(Type genericType, object identityObj);
    int SaveChanges();
}
```

Signature 7: IStorageImplementor

Signature

```
public interface IStorageImplementorGeneric : IEntityChangingNotification, IDisposable
{
    void Add<T>(T addObj, string containerName = null) where T : class;
    T CreateObject<T>() where T : class;
    void Delete<T>(T deleteObj) where T : class;
    IQueryable<T> Get<T>(string containerName = null, bool readPastLock = false) where T : class;
    string GetEntityIdentity<T>(T identityObj) where T : class;
    int SaveChanges();
}
```

Signature 8: IStorageImplementorGeneric

The reason for this design decision is due to the easy modification of the Type object compared to the overhead associated when a generic method call must be accomplished with modified generic type parameter through reflections. Decorators which rely on the alteration of the entity type can participate at the non-generic part of the chain where decorators which do not can profit from the generic interface. The duty of mapping the Type object back to a generic method call is done once by an adapter pattern implementation which is positioned between the two interface types (StorageImplementorGenericAdapter).

Before the Type object is transferred back into a generic parameter the LazyDynamicStorageImplementor object ensures that the type is the one anticipated by the EF. This is achieved with a type lookup using the IDynamicModelBuilder interface (see *Signature 5*). Furthermore, the LazyDynamicStorageImplementor class allows the deferred injection of a succeeding chain element. This feature is important because it allows creating

the decorator chain, passing the IDynamicModelBuilder along for model modifications desired by the chain elements and finally creating the database and EF ObjectContext which can be put lazily as the innermost element of the chain.

The bridge design pattern with the significant classes involved is illustrated in *Figure 13*. *Figure 14* shows the call and type mapping sequence during a write operation.

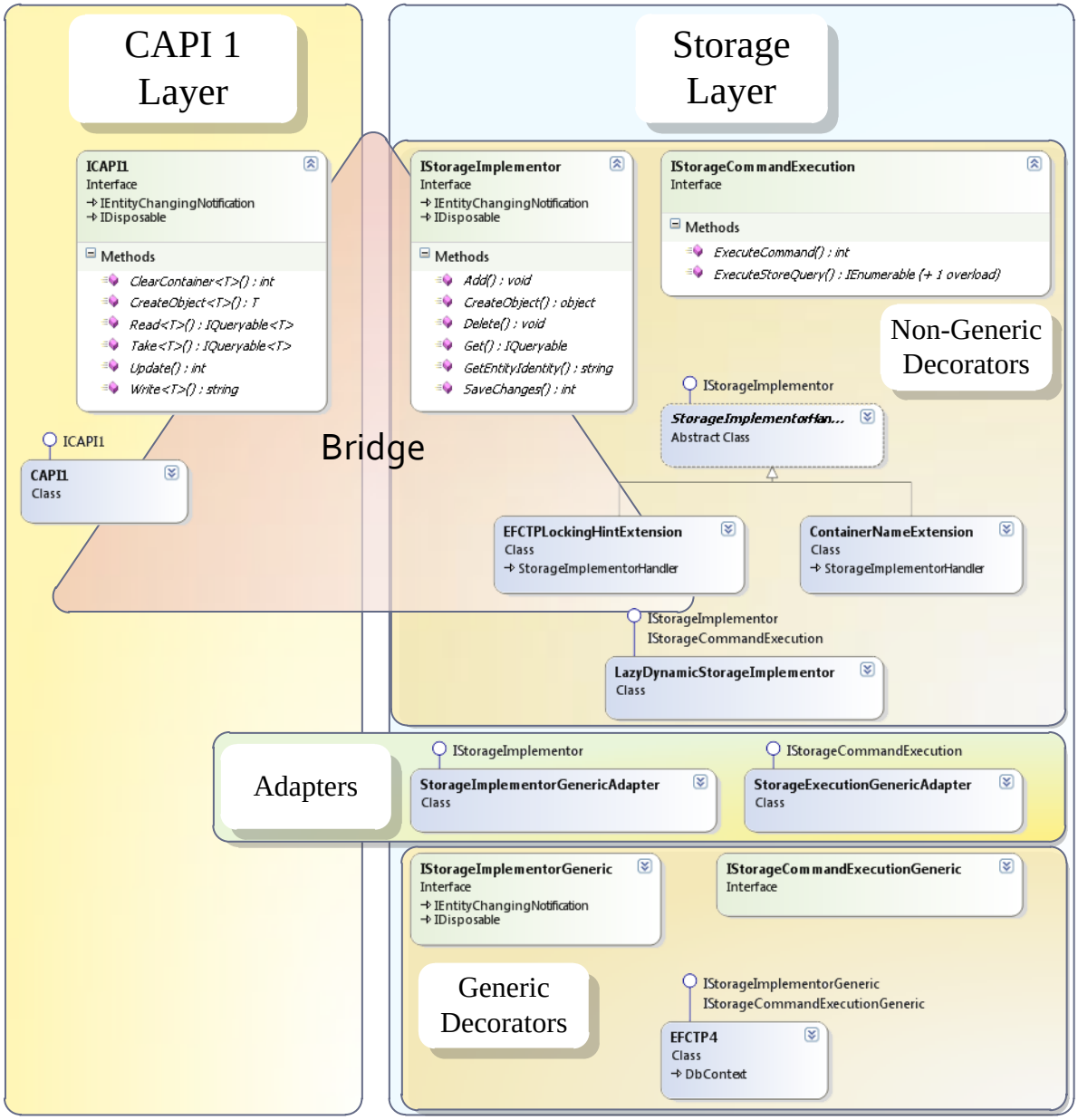

Figure 13: Bridge design pattern connecting CAPI1 and the Storage layer

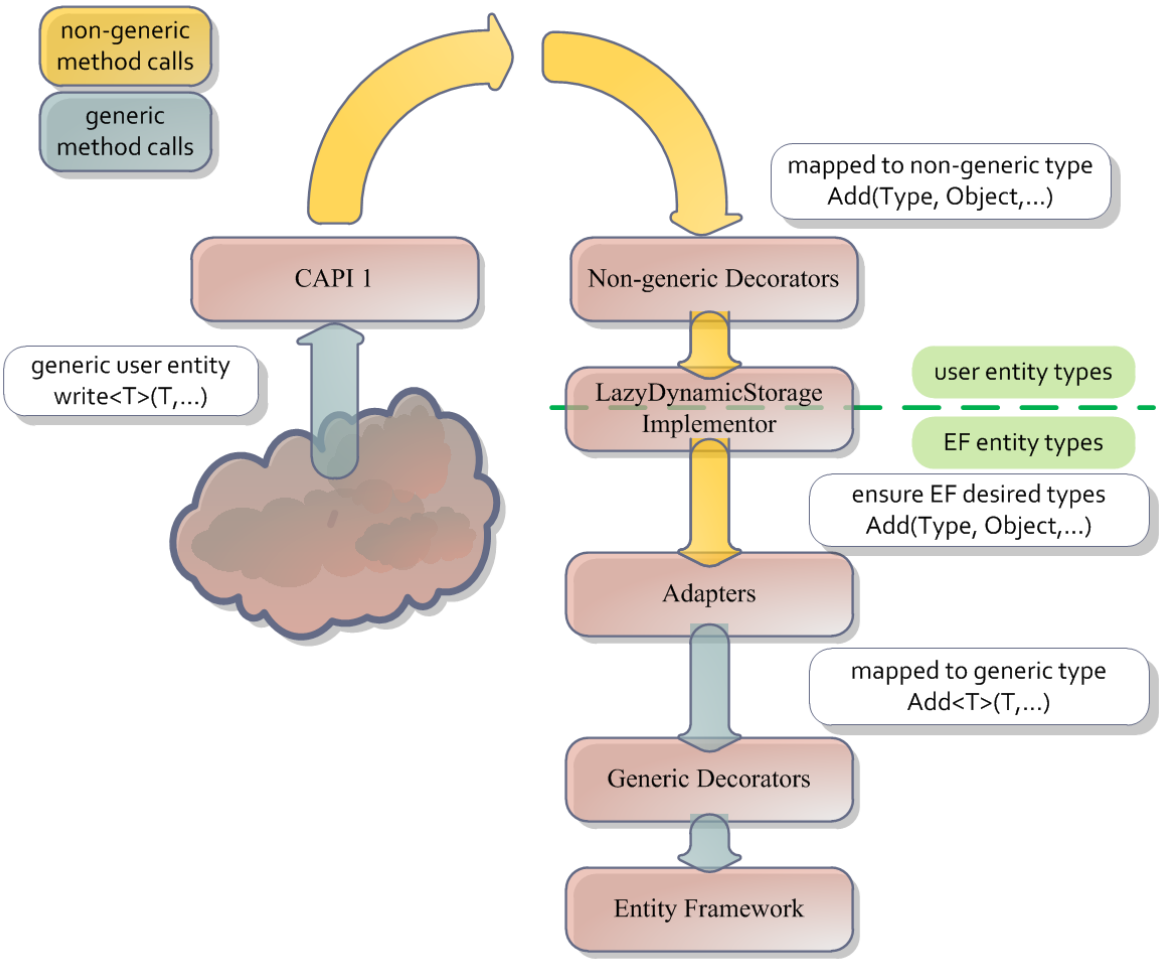

Figure 14: Generic and non-generic method calls and entity type changes illustrated by a write operation

The IStorageImplementor interface basically represents a combination of the unit of work and the repository pattern and is used for storage interactions.

StorageCommandExecution presents a second chain of decorators with equal generic and non-generic interface implementations with the capability of native query execution. This more experimental approach is used by the EFCTPLockingHintExtension decorator class to enrich LINQ to Entities with native SQL statement execution. The decorator processes the readPastLock parameter which can be found throughout the CAPI and storage layer (see *Signature 6*) interfaces. Basically the LINQ expression which is about to be executed is caught and transformed into the corresponding SQL statement. The SQL query is then modified with the “readpast” locking hint, which instructs the DBMS to simply ignore locked rows, ensuring a locking free query execution. The EF provides a query gateway for SQL statements which is used to forward the query. The resulting entities will be treated equivalent to results of a LINQ query execution, with all the proxy creation and lazy loading. This behavior is not specified by XVSM but it definitely is an interesting side effect that any SQL functionality can be achieved by this technique even if not supported by LINQ to Entities directly. An example which uses this locking-ignoring strategy will be presented in *Chapter 6.1* as part of the LinqSpace evaluation.

5.1.4.1 Entity change notifications

The IEntityChangingNotification interface is implemented all the way through the CAPI and storage interfaces. Since the EF keeps track of modified entity properties it is the only instance where changes actually can be triggered. The interface event is triggered just before the updates are persisted in the database which ensures that the notification bubbles up the decorator and CAPI chain.

There is a significant drawback anchored in the EF and the way it updates the database which has great influence on the architecture of LinqSpace. When the SaveChanges method is called to persist the changes, a graph walking algorithm is used by the framework to determine the order of the operations which are about to be executed. Obviously the process does not preserve the original order entities were updated, changed or deleted because of relationships between the objects [88]. All changes are made as atomic operation which means either all updates succeed or the whole task is rolled back. When a VectorCoordinator is about to modify its coordination information because of a deleted entity notification the bag of changed objects contains a DELETE operation to remove the corresponding column from the database and may contain multiple UPDATE statements to close the gap of vector keys to the succeeding elements. In this scenario, the EF executes the UPDATE before the DELETE operation which causes a database unique constraint error due to multiple primary indexes. There is no possibility to influence the ordering of these database operations. But the DELETE statements should be executed before the UPDATE statements. LinqSpace deals with that issue by splitting the entity change notification procedure into two phases. Modified and added entity change notifications are sent as usual before the EF persist the changes. Afterwards a SaveChanges call is placed to save all modifications, including deleted objects, to the database. Subsequently in the second phase, the notifications for the deleted entities are sent, allowing the VectorCoordinator to announce UDPATE operations for the succeeding database rows. A second SaveChanges method call ensures the persistence of these operations, avoiding collisions with the previously executed DELETE operations. The second phase can be omitted if there are no delete operations in the bag of changed entities, but consequently the SaveChanges procedure provided by LinqSpace can no longer be assumed as an atomic operation. To solve this issue the update process is enclosed automatically by a transaction, but in order to follow the layered architecture of the XVSM specification the implementation is postponed to CAPI-2. What should be noticed is that the procedure of updating the database in CAPI-1 is not atomic and may result in an inconsistent state.

5.1.4.2 LinqSpace queries

The deferred fashion of LINQ as described in *Chapter 3.2.4* suggests lazy oriented CAPI operations. Read and take method calls should not instantly invoke the query execution but return the standardized LINQ interface IQueryable (see *Chapter 3.1.3*). When the actual request is encountered, the IQueryProvider interface is invoked to deliver the desired results. To separate the LINQ functionality, the CAPI interfaces do not implement the IQueryProvider interface directly.

LINQ queries are created by the IQueryProvider implementing classes XcoQueryProviderQueryableDelegate and XcoQueryProviderExecuteDelegate, both inheriting from the same base class XcoQueryProviderBase (see *Figure 15*).

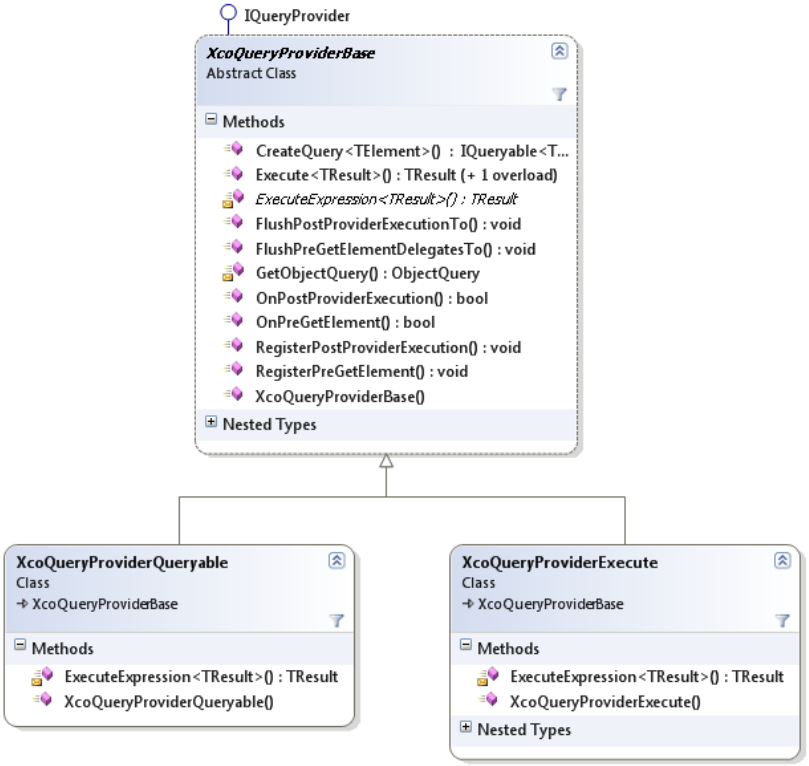

Figure 15: IQueryProvider implementations in CAPI-1

The take operation requires special treatment according to LINQ which is primary designed to retrieve entity objects. Subsequently, the objects returned from the query result have to be removed from the storage. Since the initial query execution is not invoked in the CAPI objects, the base XcoQueryProviderBase class allows to register delegates which are triggered before the retrieved entities are returned (RegisterPreGetElement) or after all elements are processed (RegisterPostProviderExecution). Further the base class allows flushing the registered delegates to another XcoQueryProviderBase class with respect to a stack ordered execution. This is important because the QueryProvider may change during the various CAPI levels but the final object retrieved by the API must be aware of the correct execution sequence.

The children of the XcoQueryProviderBase class only differ in the way the provider execution is handled. XcoQueryProviderQueryableDelegate simply forwards the inquiry to a subsequent class implementing the IQueryable interface which is passed at object creation. The XcoQueryProviderExecuteDelegate class redirects the execution to a delegate which actually offers a way to implement the executing logic directly in the CAPI objects and inject it into the QueryProvider.

The CAPI-1 object simply forwards the read and take operations with the XcoQueryProviderQueryableDelegate to the IQueryable interface, obtained from the Get method call of the succeeding IStorageImplementor interface. In addition, the take operation registers delegates to mark the retrieved entities as deleted and to persist the changes into the database.

5.1.4.3 Take operation in the Entity Framework

A drawback from the use of the DBMS and its locking mechanisms addresses the take operation. A relational database supports no native operation to perform a consuming read

request. Furthermore, when a query is placed the EF keeps the connection to the database open and iterates the results row by row. This behavior comes in handy when there is a large amount of results or because the user may decide to quit the iteration. The ORM only manifests objects which are actually requested by the interface. There are various strategies to map the take operation to an atomic read and delete operation for interaction with the EF.

- The first opportunity is to mark the entity as deleted immediately after the QueryProvider fetched it from the EF. This would result in a fast execution since the ObjectStateManager, which keeps track of the entity states, works in memory. Further the deferred behavior (see *Chapter 3.1.5*) would be preserved because the EF does not have to fetch all results immediately and can deliver them as they come. The drawback results in the unit of work design of the EF (see *Chapter 3.2.3*). Since the EF, and therefore LinqSpace, is not designed to work as singleton, there may be multiple instances even within the same application. The ObjectStateManager marks the deleted entity in memory and does not propagate the change to the database. The consequence is that the locking must in fact be handled by the DBMS to be acknowledged beyond the boarder of a single LinqSpace instance.
- As result of the experience gained from the first approach the logical consequence would be to persist the deletion of the entity. In order to keep the enumerable behavior, the SaveChanges method is invoked immediately after the entity has been fetched. This would result in multiple database updates for each entity retrieved. Since the iteration procedure of the EF is yet in process by the time the persistent call takes place, and therefore the database connection is still open, this operation results in an exception. An EF ObjectContext can only have one active database connection at a time. A workaround for that issue would be to keep two instances of the EF storage implementation in the CAPI-1 object, one for retrieving and the other one for database changing tasks. Furthermore, it would be possible to enlist the distinct database connections into a shared transaction which results in an atomic database operation. Nevertheless, this opportunity was not chosen due to the performance overhead associated with the increased database updates.
- The solution chosen for LinqSpace was to initially retrieve all results corresponding to the query. Therefore, the infrastructure is ignorant according storage concerns like open database connections and can simply propagate entity deletions as needed. The trade-off for database persistence lies in the pre-execution of the RegisterPreGetElement delegates, meaning that the entities are iterated entirely to invoke the registered functions. This mechanism subsequently allows the EF storage implementations to mark all retrieved entities as deleted and to persist the changes in a single step before the result is propagated throughout the API. The resulting disadvantage of this solution is that the enumerator behavior is essentially disabled. The objects returned by the CAPI are actually deleted or, in case of a transaction, locked by the DBMS, regardless of the user iteration. This may become problematic when entities are queried without the intention to iterate them entirely. In those scenarios the suggestion would be to page the results through the query either via coordinators or simply with LINQ's Skip and Count operations.

5.2 CAPI-2: Transactions

The CAPI-2 class can be used to decorate the CAPI-1 interface in order to enrich LinqSpace with transaction capabilities. Actually the transaction management is already provided by the

EF storage class so this layer is mainly used to separate concerns for loosely coupled components. The CAPI-1 interface inherits from the CAPI-2 and extends a single function for transaction creation (see *Signature 9*).

Signature

```
public interface ICAPI2 : ICAPI1
{
    IXcoTransaction CreateTransaction(IsolationLevel isolationLevel = IsolationLevel.RepeatableRead,
                                      TimeSpan? timeout = null);
}
```

Signature 9: ICAPI2

A timeout can be specified to limit the lifetime of a transaction. The supported isolation levels depend on the underlying database and the EF provider.

The CAPI-2 follows the abstract factory pattern to delegate the creation of objects representing a context for atomic LinqSpace interactions. The transaction objects implement the IXcoTransaction interface (see *Signature 10*) which simply offers the functionality to commit the current changes and to be disposed at the end of the transaction scope. Implementing the IDisposable interface allows to encapsulate the interface in a C# using block which basically wraps the code in a try-finally block to ensure disposal even if exceptions are thrown.

Signature

```
public interface IXcoTransaction : IDisposable
{
    void Commit();
}
```

Signature 10: IXcoTransaction

The .NET TransactionScope class is used for LinqSpace's EF transactions. Basically this mechanism allows nested transactions, but when a subtransaction fails the overall transaction will also become invalid. Subsequently no partial rollback of nested transactions is provided.

5.2.1 Atomic take operation

To work around the issue described in *Chapter 5.1.4.3* the CAPI-2 take operation registers delegates to wrap the inquiry and deletion process in a transaction. The procedure is illustrated as sequence diagram in *Figure 16*.

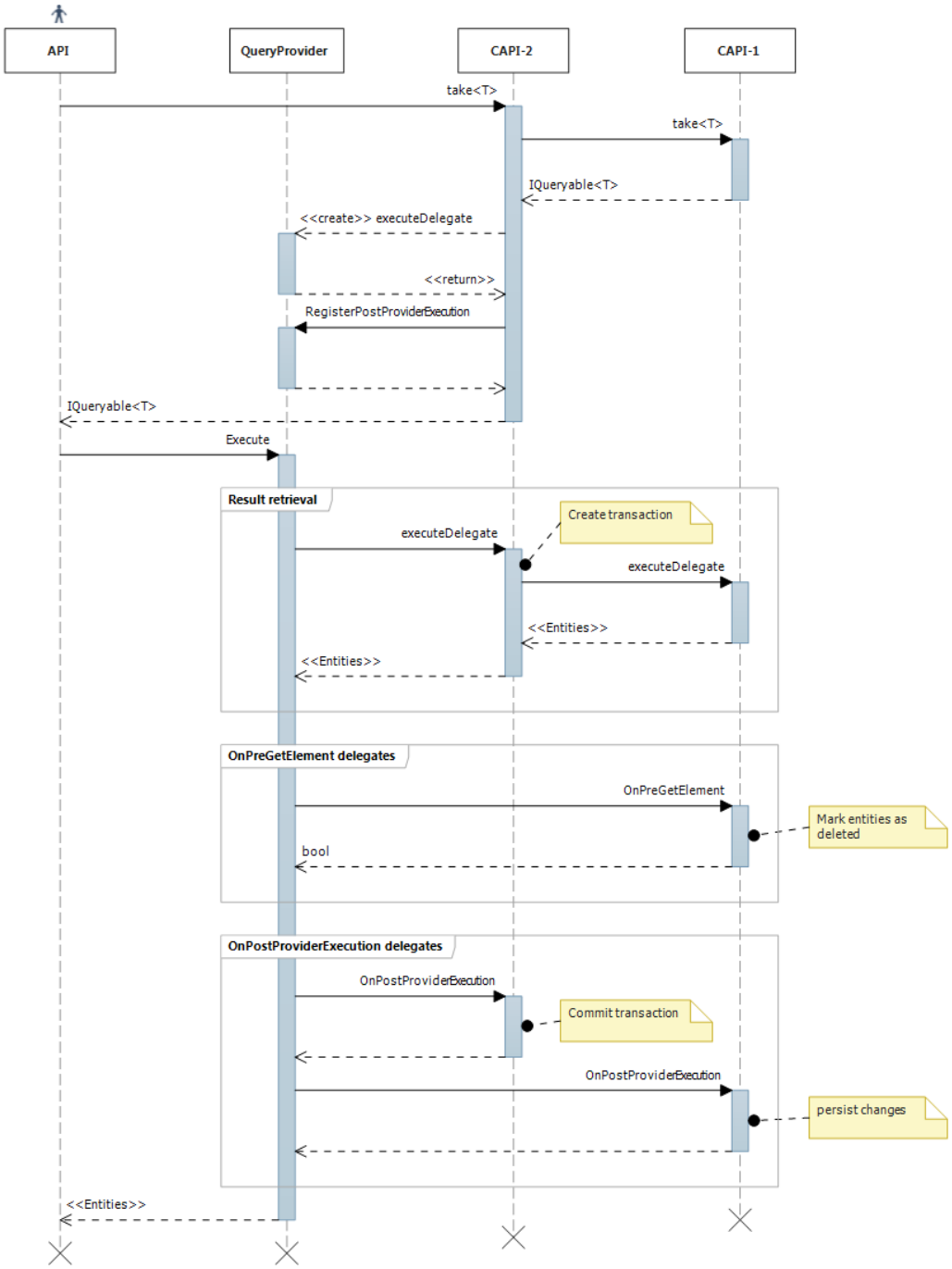

Figure 16: Take operation wrapped into a transaction

The CAPI-2 object uses the IQueryable interface retrieved from the succeeding CAPI-1 layer within its own QueryProvider implementation. Therefore the execute call can be intercepted within the CAPI-2 class which subsequently creates the transaction before the query request is delegated. The execution of the OnPostProviderExecution delegates once more accentuates the importance of the stacked invocation sequence described in *Chapter 5.1.4.2*. Accordingly it is ensured that the transaction is committed before the SaveChanges call is placed.

5.3 CAPI-3: Coordination

The fundamental challenge for the design of the CAPI-3 layer is to unite the coordination mechanism of the XVSM specification with the fluent API of LINQ. The appropriate

technique is obviously the integration through extension methods since the LINQ library is designed that way. An easy solution would be to extend the IQueryable interface because it is the return value of the underlying CAPI interfaces. The problem with this approach is that the extension method would be available on every IQueryable interface whether from LinqSpace or not. Of course it would be possible to verify the appropriateness of the interface in the extension method and otherwise return the IQueryable interface untouched, but the decision was made toward a more exclusive interaction.

The CAPI-3 layer introduces a new interface which meets the requirements for coordinated space interactions (see *Signature 11*). Accordingly, it does not inherit from the previous CAPI-2 interface because of changes in the return type of the read and take operation and an additional parameter to specify associated coordinators along with the write method invocation.

Signature

```
public interface ICAPI3 : IEntityChangingNotification, IDisposable
{
    IList<Type> SupportedCoordinators { get; }
    IList<Type> SupportedSelectors { get; }

    ICAPI3Aspect Aspect { get; }

    IXcoTransaction CreateTransaction(IsolationLevel isolationLevel = IsolationLevel.RepeatableRead,
                                      TimeSpan? timeout = null);
    int Update();

    T CreateObject<T>() where T : class;
    int ClearContainer<T>(string containerName) where T : class;

    TCoordinator CreateCoordinator<TCoordinator>(string containerName = null,
                                                 string coordinatorId = null,
                                                 bool isOptional = true)
                                                 where TCoordinator : AbstractCoordinator;

    IXcoCapi3Queryable<T> Read<T>(string containerName = null, bool readPastLock = false)
                                  where T : class;

    IXcoCapi3Queryable<T> Take<T>(string containerName = null, bool readPastLock = false)
                                  where T : class;

    string Write<T>(T obj, string containerName = null, bool saveChanges = true,
                    params AbstractCoordinator[] coordinators) where T : class;

    IList<string> WriteBulk<T>(IEnumerable<T> entities,
                               string containerName = null,
                               bool saveChanges = true,
                               params AbstractCoordinator[] coordinators) where T : class;
}
```

Signature 11: ICAPI3

The ICAPI3Aspect interface propagates events which can be used for CAPI operation intervention either at the beginning or ending of the procedure. ICAPI3Aspect is the implementation according to the XVSM specification of aspects which actually allows modifications of LINQ Expressions and returned entities. There is also a light-weight version of the interfaces provided (ICAPI3TinyAspect). The essential difference is that the notification information has been constricted to the operation type (read/take/write) and the name of the involved container. This makes it easier to transport notifications across network boundaries since serialization of expression trees and entities are CPU expensive operations

which may also reflect in a significant increase of network traffic. The ICAPI3Aspect represents the notification interface of CAPI operations. The activating parts are called ICAPI3AspectManager and ICAPI3TinyAspectManager which extend the event interfaces with activation methods used by the CAPI-3 object. The ICAPI3 interface with its involved aspect notification mechanism is illustrated in *Figure 17* and coded usage examples are shown in *Code 12*.

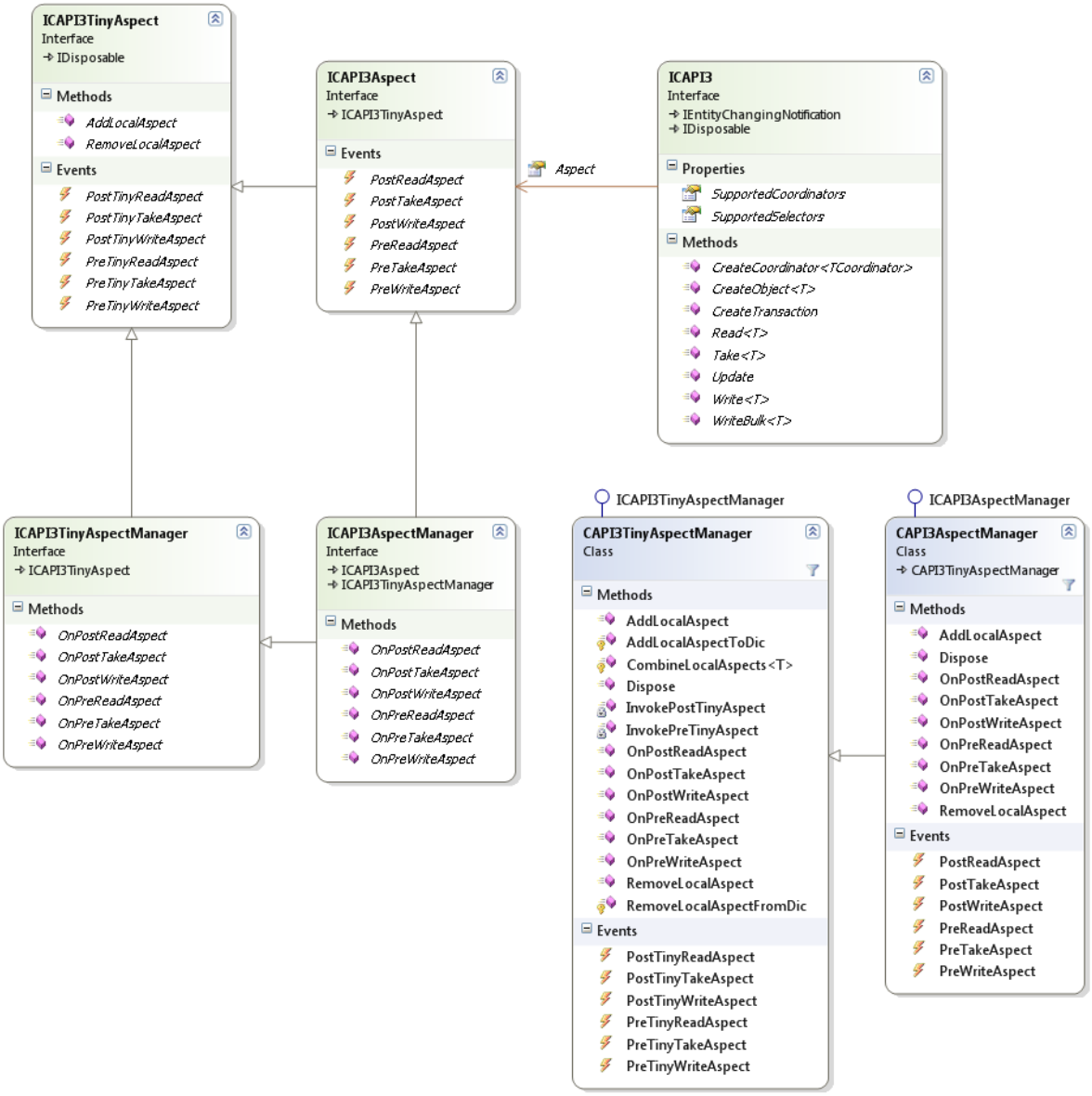

Figure 17: CAPI-3 Interface and Aspects

Code

```
// global aspect
capi3.Aspect.PostTakeAspect += new PostGetAspect((Type objType, // entity type
                                                  string containerName, // container name
                                                  Expression expression, // LINQ expression
                                                  ref object result) => // result
    {
        // global aspect, invoked after take operation
        // ability to modify result parameter
    });


// local aspect
capi3.Aspect.AddLocalAspect(CAPI3Operation.write, // capi operation
                            "testContainer", // container-name
                            new PostTinyAspect((CAPI3Operation operation, Type objType,
                                                string containerName) => // post delegate
    {
        // local aspect on 'testContainer' container, invoked after write operation
    }));
```

Code 12: Global and local aspect example

The CAPI-3 layer extends the entity model with a new type (CoordinatorInfo), which will be used to persist information about coordinators and their relation to containers in the database. The entity maintains data about the coordinator type and a unique id for identification. This data is mainly used to verify that all coordinators marked as obligatory are committed during the write operation. A memory-cache ensures fast retrieval of that information by omitting additional database lookups.

The static database schema must consider all combinations of entities to the coordination tables. Therefore, each coordinator which is about to be used for LinqSpace interaction must be announced at the CAPI-3 object instantiation in order to modify the conceptual schema and to construct the database (see *Code 13*).

Code

```
CAPI3 capi3 = new CAPI3(
// ICAPI2 capi2: CAPI2
    capi2,
// IDynamicModelBuilder modelBuilder: IDynamicModelBuilder, to register and modify conceptual model
    builder,
// bool updateModel: true to modify model, false to keep the reference for lookup
    true,
// ICoordinatorInfoCache coordinatorCache: cache strategy
    new CoordinatorInfoCache(capi2),
//ICAPI3AspectManager aspectManager: null for new aspectManager or instance for shared/reuse of aspectM
anager
    null,
//IEnumerable<Type> coordinatorTypes: register Coordinators
    new List<Type>{typeof(FifoCoordinator), typeof(KeyCoordinator)});
```

Code 13: CAPI-3 instantiation example

The initialization is done in the following sequence:

1. Registration of user entities
2. Registration of coordinators
3. Modification of the user entities according to the used coordinators

The current implementation links the user entities to all registered coordinators.

The CreateCoordinator method allows LinqSpace to keep track of the involved coordinators by persisting them with the previously mentioned CoordinatorInfo entity. Further, the method is designed generic which makes it easy to introduce new coordinator types. Each coordinator can announce:

- new interfaces which the entity objects are about to implement dynamically
- completely new entities to be added into the domain model
- configuration files for the EF Feature
- SQL commands for direct DBMS interactions

The base class for the coordinators is called AbstractCoordinator and consists of the coordinator id, a flag indication if the coordinator is optional or not and the IDynamicModelBuilder interface which will be needed by the coordinator extension method to identify the correct entity type. The coordinator classes have to be serializable in order to transmit their state across network boundaries.

In order to initiate activities, a coordinator can override the OnInsert and OnRemove methods to be notified from the CAPI-3 object when entities are added or removed. Since the coordinators are passed along the write method call, the OnInsert function of the coordinator instance can be invoked. According the OnRemove method, a take operation can be invoked without an optional coordinator. Therefore, the OnRemove method can be implemented statically within the coordination class in order to be called without an actual object instance. The static method is omnipresent and can be detected with .NET reflections.

The new generic interface IXcoCapi3Queryable is used to associate extension methods for the interaction with coordinators. *Code 14* shows the convention used for the extension method. The returned query is modified to fit the requirements of the coordinator specified by the selector parameter.

Code

```
public static IXcoCapi3Queryable<TSource> WithCoordinator<TSource>(
this IXcoCapi3Queryable<TSource> query, [SelectorType] selector)
```

Code 14: Convention for IXcoCapi3Queryable extension methods

The IXcoCapi3Queryable interface inherits from ITinyAspectQueryable which allows static extension methods to access ICAPI3TinyAspect for additional notification and blocking behaviors (see *Signature 12*).

Signature

```
public interface ITinyAspectQueryable : IQueryable
{
    ICAPI3TinyAspect TinyAspect { get; }
    Type BaseElementType { get; }
    string ContainerName { get; }
}
```

Signature 12: ITinyAspectQueryable

The extension methods used to include coordinators are called from the API and therefore reflect the user entity types and not the ones used by the EF to create the initial IQueryable object. Consequently the generic parameter cannot be used by the extension method to alter the query with standard LINQ methods (see *Chapter 3.1*). A way to achieve query

modification without generic parameters would be to use the static methods provided by the Expression class, since this is the way queries are built internally. But the resulting code would not be very readable which resulted in the decision to transfer the logic in a separate generic method. The method will be invoked with the right generic type through .NET reflection. Although this allows extending the current expression in LINQ query syntax, a special preparation is needed afterwards to remove cast and type conversion operations automatically added by the LINQ framework. The generic query extension methods encapsulate the main functionality of coordinators and will be presented in the subsequent chapters.

5.3.1 FIFO / LIFO Coordinator

The Fifo- and LifoCoordinator share mainly the same logic, so they will be covered both in this chapter, represented by the FifoCoordinator. *Code 15* shows the new entity introduced by the coordinator (FifoCoordinatorModel) and the interface which is used to extend the user entities (IFifoCoordinatorExtension).

Code

```
public class FifoCoordinatorModel
{
    [Key(), DataMemberAttribute(Order = 1)]
    [StringLength(20)]
    public virtual string CoordinatorId { get; set; }

    [Key(), DataMemberAttribute(Order = 2)]
    [StoreGenerated(StoreGeneratedPattern.Identity)]
    public virtual Int64 OrderId { get; set; }
}

public interface IFifoCoordinatorExtension
{
    ICollection<FifoCoordinatorModel> FifoCoordinatorModel { get; set; }
}
```

Code 15: FifoCoordinatorModel and interface for user entities extension

Both properties of the FifoCoordinatorModel class have the Key attribute attached resulting in a composite primary key in the database. This ensures that each coordinator (identified by the CoordinatorId) can have only one occurrence of a specific OrderId. Further the OrderId is marked with the *StoreGenerated(StoreGeneratedPattern.Identity)* attribute, instructing the EF Feature to use the property as identifier by increasing the count on every new entity. The property is used by the coordinator as chronological identifier of the entities. The StringLength attribute is used to limit the maximum length of the properties in order to reduce the storage amount. The data integrity is completely ensured by this conceptual model and therefore keeps LinqSpace free of validation checks. The EF Feature will detect the reference from the user entities toward the FifoCoordinatorModel class and consequently add the primary keys of all user entities as foreign keys. That is carried out by the storage model underneath and is no burden of the conceptual model. An example of the final data structure using the FifoCoordinator is illustrated in *Figure 18*.

TestPersons					
Age	Discriminator	Firstname	Lastname	PersonID	XcoContainerName
22	XcoDynamic_TestPerson	John	Doe	1	NULL
34	XcoDynamic_TestPerson	Richard	Smith	2	NULL

FifoCoordinatorModels		
CoordinatorId	OrderId	PersonID
testCoord	1	1
testCoord	2	2

Figure 18: FifoCoordinator example of database structure and data

The Discriminator column is added by the EF Feature tool because of the entities inheritance structure (see *Chapter 5.1.3.1*) and is of no further use according the conceptual model.

The FifoCoordinator inherits from the AbstractCoordinator and overrides the OnInsert method of the base class in order to add a reference to a coordination entity with the associated CoordinatorId for each inserted entity (see *Code 16*).

Code

```
public class FifoCoordinator : AbstractCoordinator
{
    public override bool OnInsert(IEnumerable<object> objects)
    {
        foreach (object obj in objects)
        {
            IFifoCoordinatorExtension fifoObj = obj as IFifoCoordinatorExtension;

            if (fifoObj == null)
                throw new ArgumentException(
                    string.Format("Cannot cast object of type {0} to IFifoCoordinatorExtension",
                    obj.GetType().Name));

            fifoObj.FifoCoordinatorModel.Add(
                new FifoCoordinatorModel() { CoordinatorId = (this.CoordinatorId ?? string.Empty) });
        }
        return true;
    }

    ...
}
```

Code 16: FifoCoordinator implementation

A FifoSelector object can be used to represent the current state of iteration (see *Code 17*).

Code

```
public class FifoSelector : BaseSelector
{
    internal FifoSelector(FifoCoordinator coordinator)
        : base(coordinator)
    {
        this.CurrentPosition = 0;
    }

    internal Int64 CurrentPosition { get; set; }

    ...
}
```

Code 17: FifoSelctor implementation

Code 18 shows the generic method used to extend the query for a specified FifoSelector. This procedure represents the core query functionality of the coordinator and gets invoked after the WithCoordinator (see *Code 14*) method call.

Code

```
private static IQueryable<TSourceCasted> MakeFifoExtension<TSourceOriginal, TSourceCasted>
(IXcoCapi3Queryable<TSourceOriginal> query, FifoCoordinator.FifoSelector selector)
    where TSourceCasted : IFifoCoordinatorExtension
{
    return from queryObj in query.Provider.CreateQuery<TSourceCasted>(query.Expression)
           let fifoExtension = queryObj.FifoCoordinatorModel
                                       .FirstOrDefault(fifoCoordModel =>
                                  fifoCoordModel.CoordinatorId == selector.Coordinator.CoordinatorId)
           where fifoExtension != null && fifoExtension.OrderId > selector.CurrentOrder
           orderby fifoExtension.OrderId
           select queryObj;
}
```

Code 18: FifoCoordinator query extension

The TSourceCasted generic parameter can be restricted to implement the IFifoCoordinatorExtension interface which is ensured by the CAPI objects (see *Chapter 5.1.4*). This interface is essentially the bridge for linking user entities to the associated coordination entities. The EF deferred loading feature (see *Chapter 3.2.4*) ensures that the related entities are lazily fetched from the database without any special action at this point. The LINQ query basically filters relevant entities regarding the existence of the related coordination entity with the correct CoordinatorId and sorts the result by ascending OrderId. Further the CurrentOrder property of the FifoSelector is used to identify the already read entities which are taken into account with a greater-than expression. The LifoCoordinator extension method is mainly the same except for the descending ordering and the corresponding less-than expression. *Figure 19* illustrates the sequence of operations in order to execute a CAPI-3 take operation with a FifoCoordinator.

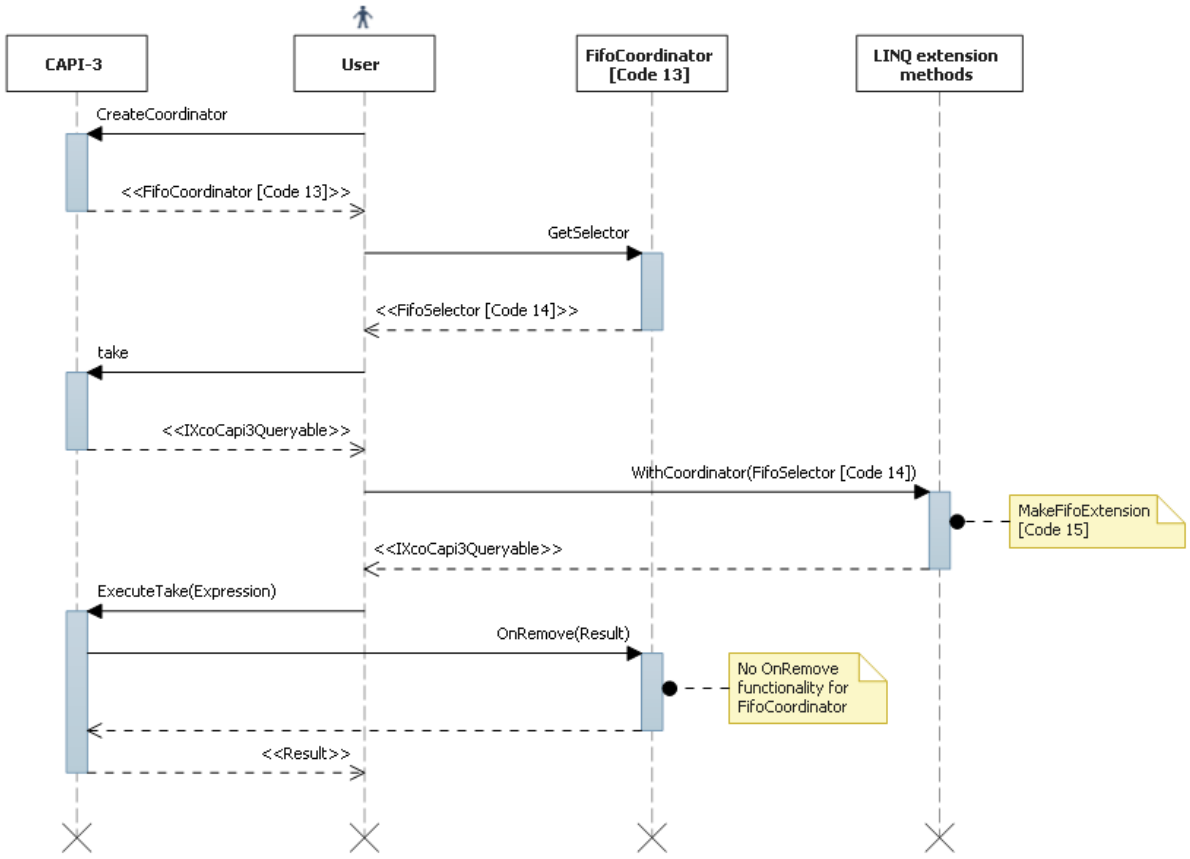

Figure 19: Sequence diagram for FifoCoordinator interaction

Before returning the modified query the extension method registers a PreGetElement event (see *Chapter 5.1.4.2*) to alter the CurrentOrder property of the passed FifoSelector to the last OrderId returned when the query is executed.

The FifoCoordinator propagates a conceptual model configuration which allows deleting the coordination related entities along with the user entities with the on-delete-cascade option offered by databases. Therefore, no additional treatment is required in case of removal.

5.3.2 KEY Coordinator

The model used to implement the KeyCoordinator is shown in *Code 19*.

Code

```
public class KeyCoordinatorModel
{
    [Key(), DataMemberAttribute(Order = 1)]
    [StringLength(20)]
    public virtual string CoordinatorId { get; set; }

    [Key(), DataMemberAttribute(Order = 2)]
    [StringLength(20)]
    public virtual string Key { get; set; }
}

public interface IKeyCoordinatorExtension
{
    ICollection<KeyCoordinatorModel> KeyCoordinatorModel { get; set; }
}
```

Code 19: KeyCoordinatorModel and interface for user entities extension

The OnInsert method is overridden by the KeyCoordinator to add the KeyCoordinatorModel reference for the inserted entities.

The query is mainly extended to take the first appearance of an entity related to a KeyCoordinatorModel with the key specified by the selector (shown in *Code 20*).

Code

```
private static IQueryable<TSourceCasted> MakeKeyExtension<TSourceOriginal, TSourceCasted>
(IXcoCapi3Queryable<TSourceOriginal> query, KeyCoordinator.KeySelector selector)
where TSourceCasted : IKeyCoordinatorExtension
{
    return (from queryObj in query.Provider.CreateQuery<TSourceCasted>(query.Expression)
            where queryObj.KeyCoordinatorModel.FirstOrDefault(keyCoordModel =>
           keyCoordModel.CoordinatorId == selector.Coordinator.CoordinatorId).Key == selector.SearchKey
            select queryObj).Take(1);
}
```

Code 20: KeyCoordinator query extension

The KeyCoordinator also specifies the on-delete-cascade option to clean up coordination information without additional logic.

5.3.3 VECTOR Coordinator

The VectorCoordinator extends the user-defined domain with the model shown in *Code 21*.

Code

```
public class VectorCoordinatorModel
{
    [Key(), DataMemberAttribute(Order = 1)]
    [StringLength(20)]
    public virtual string CoordinatorId { get; set; }

    [Key(), DataMemberAttribute(Order = 2)]
    [StoreGenerated(StoreGeneratedPattern.Identity)]
    public virtual Int64 Id { get; set; }

    public virtual Int64 Position { get; set; }
}

public interface IVectorCoordinatorExtension
{
    ICollection<VectorCoordinatorModel> VectorCoordinatorModel { get; set; }
}
```

Code 21: VectorCoordinatorModel and interface for user entities extension

Since it is problematic to alter a primary key the Position property is not part of the composite key. Therefore an Id property is introduced which mainly serves as placeholder and has no special usage. To guarantee data integrity, the VectorCoordinator propagates an SQL command which alters the database table by adding a unique constraint for the CoordinatorId and Position column.

When entities are inserted, the coordinator has to ensure that the succeeding entities Position properties are shifted. This is done by an additional request to fetch the following VectorCoordinatorModels which are subsequently iterated to increase the Position property.

Code 22 shows the query extension performed to implement the VectorCoordinator behavior.

Code

```
private static IQueryable<TSourceCasted> MakeVectorExtension<TSourceOriginal, TSourceCasted>
(IXcoCapi3Queryable<TSourceOriginal> query, VectorCoordinator.VectorSelector selector)
where TSourceCasted : IVectorCoordinatorExtension
{
    return (from queryObj in query.Provider.CreateQuery<TSourceCasted>(query.Expression)
            let vectorExtension = queryObj.VectorCoordinatorModel.FirstOrDefault(
                fifoCoordModel => fifoCoordModel.CoordinatorId == selector.Coordinator.CoordinatorId)
            where vectorExtension != null && vectorExtension.Position >= selector.SearchPosition
            orderby vectorExtension.Position
            select queryObj).Take(selector.TakeAmount);
}
```

Code 22: VectorCoordinator query extension

The query mainly matches the one used for the Fifo- and LifoCoordinator (see *Code 18*) but adds a Take operation to limit the resulting entities to the amount specified by the delivered selector.

The VectorCoordinator is at the moment the only coordinator which requires an OnRemove method. With an additional request, the succeeding entities VectorCoordinator models are fetched to decrease the Position property. The VectorCoordinator also uses the on-delete-cascade option.

5.4 CAPI-4: Runtime and Remoting

The easiest way to access a remote data storage would be to simply redirect the connection string to the database. Regarding Microsoft's SQL Server the TDS[1] application layer protocol is used for client and database server interaction. The requests are essentially encapsulated into packages which are subsequently transmitted over various configurable standard communication mechanisms like shared memory, named pipes, TCP/IP or HTTP. The latter allows a routable communication channel between the involved peers.

Since the standard SQL connection is not capable of notifications which are needed for the aspect behavior, a second interaction technique would be necessary to actively inform the peers about triggered aspects. This approach was firmly rejected in order to implement a uniform interface.

5.4.1 The WCF Data Services approach

The WCF Data Services [89] is a component of the .NET Framework which allows exposing and consuming data using the semantics of REST. Data is exposed over the OData[2] protocol with JSON or Atom as transfer format. The service can easily propagate the domain model with all CRUD operations just by wrapping the EF ObjectContext in a special WCF service class. The client side class offers mainly the same functionality as the EF QueryProvider, namely LINQ support, entity change tracking and deferred loading.

The characteristics of the WCF Data Service make the framework a fabulous candidate for remote LinqSpace interactions. Since there is no mechanism available in the .NET Framework to serialize and transmit expression trees for remote execution, the WCF Data Services service solves this problem by mapping the query as HTTP REST service request. This results in a standardized interface for LinqSpace interactions which easily can be used by third-party components.

Where deficits of the stateless REST architecture like transactions and aspect notifications can be dissolved by workarounds, there is currently one issue which results in an exclusion criterion for WCF Data Services: Navigation properties are allowed only in the base classes. Since the references to the coordination domain entities are included as extension in the dynamically created inheriting class, the domain model cannot be used. Currently the support of navigation properties on derived types is the most voted feature suggestion for WCF Data Services [90].

Since the WCF Data Services cannot be used for LinqSpace, another technique will be presented in the following chapter.

5.4.2 Remote CAPI-3 access

The CAPI-4 layer exposes the CAPI-3 functionality over a WCF service (see *Signature 13*).

[1] Tabular Data Stream [126]
[2] Open Data [127]

Signature

```
[ServiceContract(SessionMode = SessionMode.Required)]
internal interface ICAPI3RemoteContract
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    CAPI3RemoteResponse Read(XcoCapi4QueryableBase query, string containerName, bool readPastLock);

    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    CAPI3RemoteResponse Take(XcoCapi4QueryableBase query, string containerName, bool readPastLock);

    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    IEnumerable<string> WriteBulk(IEnumerable<object> entities, string containerName, bool saveChanges,
                                  IEnumerable<AbstractCoordinator> coordinators);

    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    AbstractCoordinator CreateCoordinator(string coordinatorType, string containerName,
                                          string coordinatorId, bool isOptional);
}
```

Signature 13: ICAPI3RemoteContract

The TransactionFlow attribute is specified which allows the client to distribute transactions. Further the communication requires a session which is used by the hosting service to dedicate a distinct CAPI-3 instance for each active connection allowing a high degree of concurrency (see *Chapter 3.2.3*).

The serialization of expression trees is a fairly complex task since there is no mechanism provided by the .NET Framework. To keep the queries as simple as possible, the coordinators should not add supplementary complexity by altering the query on the client side. Instead the XcoCapi4Queryable object uses an instance method with the same name as the coordinator extension method (WithCoordinator) to collect involved selectors (see *Signature 14*).

Signature

```
public interface IXcoCapi4Queryable<T> : IQueryable<T>, IOrderedQueryable<T>, ITinyAspectQueryable
{
    XcoCapi4Queryable<T> WithCoordinator(BaseSelector selector);
}
```

Signature 14: IXcoCapi4Queryable

When the QueryProvider is executed, the whole query object (including the collected selectors and the user expression) is serialized and transmitted to the remote peer where the selectors are inspected and the CAPI-3 WithCoordinator extension method is called to involve the coordinators. The remote communication is illustrated by a take operation in *Figure 20*.

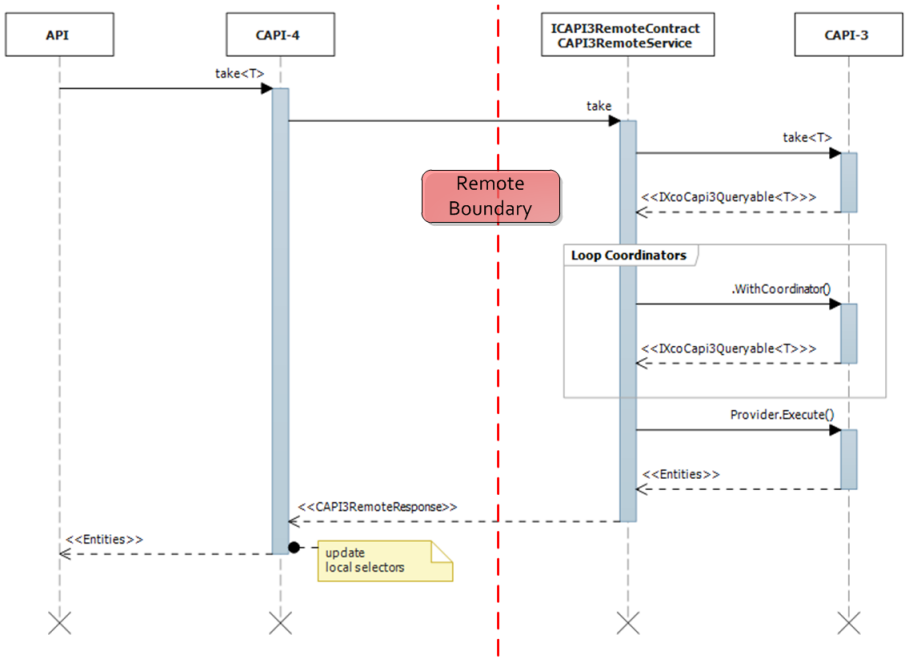

Figure 20: CAPI-4 remote communication7

The coordinator and selector classes are marked with the DataContract and DataMember attributes which allows serialization and consequently the transmission over the WCF interface. The CAPI3RemoteResponse object returned by the read and take operation contains the serialized query result and the modified selectors which subsequently are used to update the local selectors.

The expression tree serialization technique has its origins in the MSDN archive [91] and was modified to fit the needs of LINQ in the .NET Framework 4.0 and LinqSpace. Basically the tree is serialized into a corresponding XML expression.

Since the transmitted entities are decoupled from their EF ObjectContext the following EF features are no longer supported in CAPI-4:

- Entity change tracking and update functionality (see *Chapter 3.2.2*)
- Deferred loading and navigation properties (see *Chapter 3.2.4*)

Both functionalities could be preserved by self-implemented workarounds, but the effort for a prototype would beyond the scope of this work. In order to track modified properties, a client extension is necessary which is included or wrapped over the entities. Approaches to achieved this functionality would be the use of transparent proxies (as with the user entity extension, see *Chapter 5.1.3.1*) or ADO.NET self-tracking entities [92]. When an update operation is requested, only the modified properties are transmitted to the responsible space. To lazily load related properties, the proxy classes can trigger an additional remote request in order to retrieve the required information.

5.4.3 Remote Aspects

In order to publish aspects across network boundaries a second WCF service is hosted by the CAPI-4 object which offers the contract shown in *Signature 15*.

Signature

```
[ServiceContract(SessionMode = SessionMode.Required)]
internal interface ICAPI3RemoteTinyAspectContract
{
    [OperationContract()]
    void AddLocalPostAspect(CAPI3Operation operation, string containerName);

    [OperationContract()]
    void AddLocalPreAspect(CAPI3Operation operation, string containerName);

    [OperationContract()]
    void RemoveLocalAspect(CAPI3Operation operation, string containerName);

    [OperationContract()]
    List<CAPI3RemoteTinyAspectResponse> ReceiveNotifications();
}

[ServiceContract(Name = "ICAPI3RemoteTinyAspectContract", SessionMode = SessionMode.Required)]
internal interface ICAPI3RemoteTinyAspectClientContract
{
    [OperationContract()]
    void AddLocalPostAspect(CAPI3Operation operation, string containerName);

    [OperationContract()]
    void AddLocalPreAspect(CAPI3Operation operation, string containerName);

    [OperationContract()]
    void RemoveLocalPostAspect(CAPI3Operation operation, string containerName);

    [OperationContract(AsyncPattern = true)]
    IAsyncResult BeginReceiveNotifications(AsyncCallback callback, object userState);
    List<CAPI3RemoteTinyAspectResponse> EndReceiveNotifications(IAsyncResult result);
}
```

Signature 15: ICAPI3RemoteTinyAspectContract

To actively notify the client a long polling strategy [93] is used, resulting in a blocking ReceiveNotifications method on the remote peer until aspects are triggered. This ensures that communication channels are created only in one direction which allows passing HTTP proxies and firewalls. The client side interface exposes the method asynchronously which permits blocking-free invocation.

5.5 CAPI-5: Blocking behavior

In order to await the delivery of results the generic BlockingQuery class can be used. This class mainly represents a façade pattern which encapsulates the registration of a write operation aspect on the requested container and, if no satisfying result is encountered by the first query execution, blocks the current thread. When the aspect triggers a notification a subsequent query request is placed. In addition a timeout condition can be set to stop the procedure. An extension method is provided to encapsulate a created query within a BlockingQuery object (see *Signature 16*) using fluent code.

Signature

```
public static class LINQCapiBExtension
{
    public static BlockingQuery<IEnumerable<T>> WithBlocking<T>(this IQueryable<T> query,
                                                                TimeSpan? waitTime = null);
}
```

Signature 16: WithBlocking extension method

Code 23 illustrates an example of the WithBlocking extension method usage.

Code

```
BlockingQuery<IEnumerable<TestPerson>> request =
                        clientCapi4.Read<TestPerson>().WithBlocking(TimeSpan.FromSeconds(10));
// blocking call
List<TestPerson> resultList = request.Value.ToList();
```

Code 23: WithBlocking extension example

6 LinqSpace compared

In order to evaluate the new LinqSpace, it is compared to its .NET counterpart XcoSpaces. The major difference at first examination is the opportunity to work on related entries and therefore directly with the domain model. Other interesting features include functionality provided by the database like grouping, which is not considered by the specification of the SBC paradigm. The following chapter addresses the usability reviewed by an example scenario which points out the major differences in methodology used to implement the desired requirements.

6.1 Example for usability comparison: Kitchen Order Ticket (KOT)

This example is about a restaurant ticket system which extends the scenario presented in [44]. The included actors are costumers which can order meals and drinks, cooks which are responsible for supplying the order items and waiters which are responsible to serve the food in an organized fashion.

One of the fundamental tasks of software development is to identify and describe the problem domain. This ensures that the conceptual formulation is completely understood and provides a common ground of knowledge for every person involved. The entities and relationships are typically centralized in an ER-Modell[1]. Objects in the real world cannot be mapped completely into a model because their properties are nearly endless. Therefore it is important to create the conceptual model with a well-defined assignation which may result from use cases or requirements to be met by the software product. *Figure 21* summarizes the usage of the KOT application.

[1] Entity-Relationship-Modell

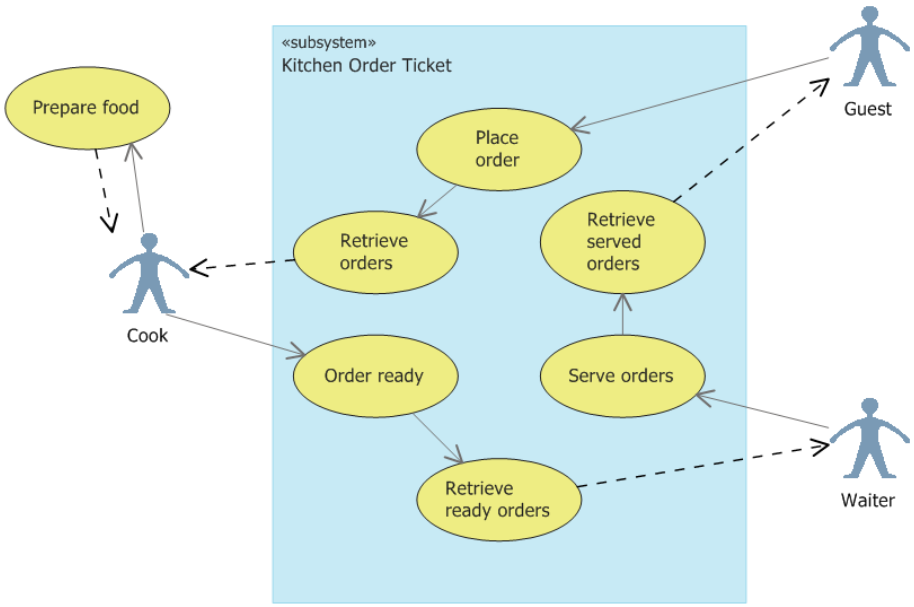

Figure 21: KOT use case diagram

The use case diagram illustrates the desired functionality. In order to identify the appropriate granularity for the domain model the functional requirements must be specified:

- Items which can be ordered are pre-stored in the system. These are classified to different categories like “Starters”, “Main Dishes” or “Dessert”. Further the items have a weight, in order to limit the amount a waiter can carry, and an established production time for cooks.
- A guest can order items for a specified Table.
- A cook retrieves orders which are placed by a guest (in a real restaurant the waiter receives the orders, but this step is omitted by this model) and creates them with respect to the specified item category order (“Starters” before “Main Dishes” before “Dessert”). Depending on the experience of the cook, he can create multiple items simultaneously. Each order has a specific preparation time.
- The waiter is responsible for serving created items to the waiting guests. The delivery takes a specified period of time, depending on the waiter. The following points describe the strategy for item delivery:
    - Orders are started to serve when all orders are finished in a category for a table.
    - A waiter can carry a maximum weight of items.
    - When not all orders can be served at once the remaining items in the category gain a higher priority than category orders where no item has been served. When the main meal is served for a table except for one person this behavior ensures that the next free waiter will give precedence to the outstanding meal so the table category is finished.
- Any actor can occur multiple times and it must be possible to dynamically add or remove actors.

With the requirements determined the ER-Modell can be created (see *Figure 22*).

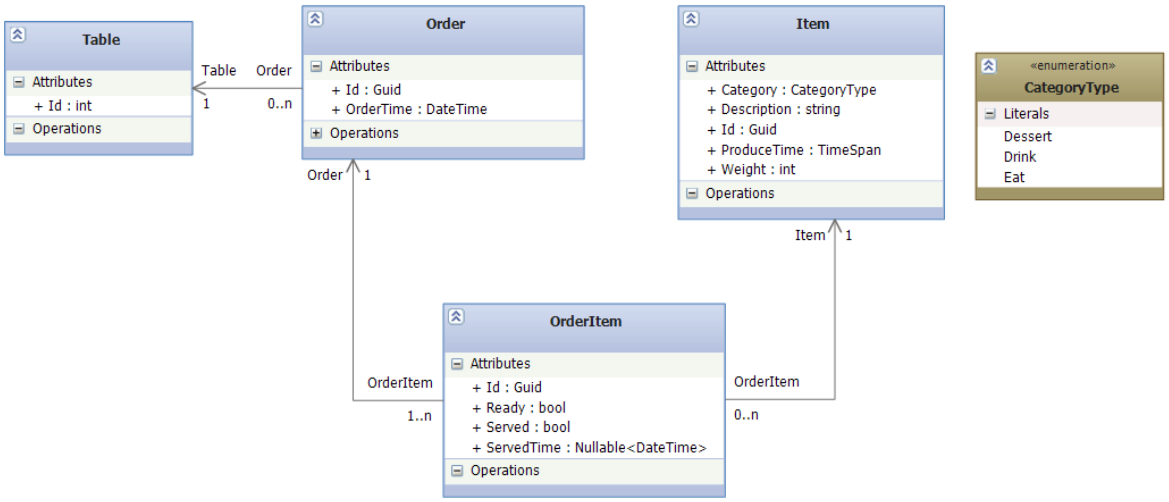

Figure 22: ER-Modell KOT

In a business environment it would be appropriate to extend the Ready property of the OrderItem class with a relation to the cook who produced the item. But for the sake of simplicity and in order to keep the model simple the state of an OrderItem is just flagged with booleans.

6.1.1 Implementation approach: XcoSpaces

Since the SBC paradigm does not support relations between the entries some considerations according the mapping of the ER-Model into a container-based model must be taken into account.

- The Item entity will be mapped into a container with an additional ListCoordinator because the collection will typically be read as list to present the guest all items which can be ordered.
- The table entity can also be put in a container with an additional ListCoordinator since the guest can choose from all available tables.
- Because of the coordination strategy demanded by the waiter, the mapping of the Order/OrderItem requires additional investigations. The first examination addresses the storage of the ordered items. A possibility would be to use one container for the kitchen and a distinct container for each table. The cooks are waiting for incoming orders in the kitchen-container which are posted there by guests. When a cook finished producing an item, it is put into an order-ready container which will be used to notify the waiters. A waiter identifies the destination table by a reference stored in the Order/OrderItem.

6.1.1.1 Cook

When orders are placed into a single container a LabelCoordinator can be used to separate the items category. This would demand the cook to know the accurate sequence of categories in advance and place multiple take-operations on the container, each masked with the category identifier. Further, this approach would entail coordination policy outside the abstraction framework. Alternately, a user-defined coordinator can be introduced which consolidates the logic and omits multiple requests.

Another interesting point is the locking strategy used by the cook. The intention is to wrap the take operation for guest orders, the creation of the item and the write operation for the newly created item into a transaction. This allows rolling back the initial take operation when there are problems encountered during the cooking process. Since the creation of an item may take a longer period of time the order is locked and a standard coordinator like the FifoCoordinator subsequently forbids a second cook to skip the locked entry and take the next available order. As with the suggested strategy for category discrimination, a user-defined coordinator would be appropriate to achieve this behavior. An implementation of this coordinator is illustrated in *Code 24* without exception and additional pre-selection handling.

Code

```
public class CookCoordinator : ICoordinator
{
    // category sorted list
    private SortedList<int, IEntry> _orderItemEntryList = new SortedList<int, IEntry>();

    ...

    // allow locking on entry level
    public bool AllowsEntryLocking
    {
        get { return true; }
    }

    public bool Write(IEntry entry, ITransaction t)
    {
        CookSelector ws = GetSelector(entry);

        _orderItemEntryList.Add(ws.Category, entry); // Category is passed by CookSelector
        t.AddLog(new TransactionLog(() => _orderItemEntryList.Remove(ws.Category)));
        return true;
    }

    // get CookSelector for entry
    private CookSelector GetSelector(IEntry entry)
    {
        foreach (Selector selector in entry.Selectors)
        {
            if (selector is CookSelector)
                return (CookSelector)selector;
        }
        return null;
    }

    public List<IEntry> Read(Selector selector, ITransaction t, List<IEntry> preSelectedEntries)
    {
        return _orderItemEntryList.Take(selector.Count) // take amount specified by selector
                                  .Select(o => o.Value) // select Value Property (IEntry)
                                  .ToList(); // immediate execution
    }

    public int Remove(List<IEntry> entries, ITransaction t)
    {
        int count = 0;

        foreach (KeyValuePair<int, IEntry> entry in _orderItemEntryList.Where(o =>
                                                                              entries.Contains(o.Value)))
        {
            KeyValuePair<int, IEntry> curEntry = entry;
            if (_orderItemEntryList.Remove(curEntry.Key))
            {
                ++count;
                t.AddLog(new TransactionLog(() =>
                                                _orderItemEntryList.Add(curEntry.Key, curEntry.Value)));
            }
        }
        return count;
    }
}
```

Code 24: User-defined coordinator for cook

A SortedList is used to arrange the written OrderItem entries according their category. The coordinator uses the AllowsEntryLocking property in order to lock on entry level and skip locked entries as specified by the requirements for the cook actor.

This example demonstrates how additional coordination information can be passed into the coordinator as used for the Category identifier in the Write operation. Additionally, the ProduceTime information of the corresponding Item entity would be necessary for the cook actor to associate the time for creating the product. The ProduceTime can also be passed along with the CookSelector. Coordinators only return user entries, but the selector could also be used to reply this additional information from the coordinator. Alternately, the domain model can be altered to combine all necessary coordination information in the OrderItem object.

6.1.1.2 Waiter

The strategy for the order delivery cannot be achieved with concatenations of standard coordinators. Therefore a specialized user-defined coordinator is appropriated which meets the following requirements:

- Each order delivered by a waiter or placed by a guest may influence the prioritization of the remaining entries according the specified order delivery requirements (see *Chapter 6.1*).
- Items are not retrieved till the whole category for a table is created by the cook.
- Orders are taken with an overall maximum weight in a single operation. This requirement results in further considerations since the intentional model foresees a separation of Items and OrderItems which would require the coordinator to cross-reference containers in order to retrieve the weight information. An alternative would be to inject the Weight property directly within the user-defined coordinator when OrderItems are written. The duplicated and redundantly stored information would have no significance regarding the KOT example because the OrderItem entities are short-living. What if the items are stored over a longer period and the weight changes? A user-defined coordinator would be necessary to take orders with a specified overall weight. Alternatively, the orders can be separated with multiple take operations till the maximum weight is exceeded. This concludes that the last element must be written back into the container.

The primary functionality for a user-defined coordinator meeting the waiter actor requirements is shown in *Code 25*.

Code

```
public class WaiterCoordinator : ICoordinator
{
    private struct OrderItemEntry
    {
        public IEntry Entry;
        public int TableNumber;
        public int Category;
        public int Weight;
    }

    private List<OrderItemEntry> _orderItemEntryList;

    ...

    public bool Write(IEntry entry, ITransaction t)
    {
        if (!(entry.Value is OrderItem))
            throw new XcoContainerWriteException("WaiterCoordinator can only be used for entries of
                                                  type OrderItem");

        WaiterSelector ws = GetSelector(entry);
        // WaiterSelector is used to pass additional entry information
        OrderItemEntry oie = new OrderItemEntry
        {
            Entry = entry,
            TableNumber = ws.TableNumber,
            Category = ws.Category,
            Weight = ws.Weight
        };

        _orderItemEntryList.Add(oie);

        t.AddLog(new TransactionLog(() => _orderItemEntryList.Remove(oie)));
        return true;
    }

    // get WaiterSelector for entry
    private WaiterSelector GetSelector(IEntry entry)
    {
        foreach (Selector selector in entry.Selectors)
        {
            if (selector is WaiterSelector)
                return (WaiterSelector)selector;
        }
        return null;
    }

    public List<IEntry> Read(Selector selector, ITransaction t, List<IEntry> preSelectedEntries)
    {
        WaiterSelector ws = (WaiterSelector)selector;

        return (from orderItemEntry in _orderItemEntryList

                let takeOderItemEntries = (
                    from TakeorderItemEntry in _orderItemEntryList

                    group TakeorderItemEntry by new // group by TabledId and Category
                    {
                        TakeorderItemEntry.TableNumber,
                        // according the domain model, TableNumber is part of Table entity
                        Category = TakeorderItemEntry.Category
                        // according the domain model, Category is part of the Item entity
                    } into tableClassGroups

                    where !tableClassGroups.Any(oi => (oi.Entry.Value as OrderItem).Ready == false)
                          // all orderitems of category and table ready
                           && tableClassGroups.Any(oi => (oi.Entry.Value as OrderItem).Served == false)
```

```
                        // some not served

                    let sumItemTableClasses = tableClassGroups.Count()
                    // sum of category for table
                    let sumItemTableClassesNotServed = tableClassGroups.Count(oi =>
                                                        (oi.Entry.Value as OrderItem).Served == false)
                    // sum of category for table, which are not already served by a waiter
                    orderby tableClassGroups.Key.Category ascending, sumItemTableClasses -
                                                            sumItemTableClassesNotServed descending
                    // order by category and respect partial served categories
                    // as specified by the delivery requirements
                    select tableClassGroups)
                .SelectMany(s => s) // flattening of groups
                .Where(takeOrderItem => (takeOrderItem.Entry.Value as OrderItem).Served == false)
                // filter out served items

                let sumWeightBevoreOrderItem = (from orderItemBevore in takeOderItemEntries
                                                where (orderItemBevore.Entry.Value as OrderItem).OrderN
umber <= (orderItemEntry.Entry.Value as OrderItem).OrderNumber
                                                select orderItemBevore.Weight
                                        // according the domain model, Weight is part of the Item entity
                                                ).Sum()


                where sumWeightBevoreOrderItem <= ws.MaximumWeight
                     // take maximum weight into accont, retrieved by WaiterSelector
                where takeOderItemEntries.Select(to => (to.Entry.Value as OrderItem).Id)
                                         .Contains((orderItemEntry.Entry.Value as OrderItem).Id)
                select orderItemEntry.Entry).ToList();
    }

    public int Remove(List<IEntry> entries, ITransaction t)
    {
        int count = 0;

        foreach (OrderItemEntry oie in _orderItemEntryList.Where(o => entries.Contains(o.Entry)))
        {
            OrderItemEntry curoie = oie;
            if (_orderItemEntryList.Remove(curoie))
            {
                ++count;
                t.AddLog(new TransactionLog(() => _orderItemEntryList.Add(curoie)));
            }
        }
        return count;
    }
}
```

Code 25: User-defined coordinator for waiter

The coordinator uses a single list of OrderItemEntry objects, gathering important coordination information, and LINQ to Objects (see *Chapter 3.1*) in order to realize the waiter-specific coordination. This functionality can be optimized by hashtables and flags, identifying certain states of served categories and tables for performance improvements. These modifications would have negative influence according readability and are omitted in this example. Further, an adapted version of the LINQ query, used by the Read method, will be implemented and discussed in the LinqSpace example in *Chapter 6.1.2.2*.

Since relationships between entities are not supported, the required properties from foreign entity types (TableNumber from Table entity, Category and Weight from Item entity) are passed to the Write operation with the WaiterSelector. The OrderItemEntry object maintains this information for subsequent inquiry.

Another approach to gather coordination relevant information would be to change the domain model with the intention to group all required properties in the OrderItem entity, as already mentioned in the previous cook implementation (see *Chapter 6.1.2.1*). This would result in redundant data maintenance on the conceptual model layer.

6.1.2 Implementation approach: LinqSpace

LinqSpace allows working directly on the conceptual model without additional mappings. Because of the CAPI-4 limitation according navigation properties (see *Chapter 5.4.2*) the relationship between entities must be established with foreign key identifiers.

The coordination information is part of the domain model so there are no coordinators required to implement the desired functionality. Furthermore, the implementation approach for XcoSpaces (see *Chapter 6.1.1*) has shown that the requirements could not be implemented with standard coordinators anyway. User-defined coordinators would only result in redundant information in the database.

The queries presented in the subsequent chapters are triggered by aspects, after a write operation has been invoked (see *Chapter 5.3*).

6.1.2.1 Cook

The coordination strategy for the cook actor can be represented in a single operation (see *Code 26*).

Code

```
List<OrderItem> orderItemList = from oderItem in
    capi4.take<OrderItem>("OrderItem_cook", true)
    join item in capi4.Read<Item>(typeof(Item).Name, false) on oderItem.ItemId equals item.Id
    where oderItem.Ready == false
    orderby item.Category ascending
    select oderItem)
```

Code 26: Cook LinqSpace coordination

The query simply joins the OrderItems with the corresponding Item entities, filters out the ready ones and orders them according their category.

The LinqSpace approach faces the same locking problem described in the XcoSpaces cook implementation (see *Chapter 6.1.1.1*). The “readpast” locking hint (see *Chapter 5.1*) was initially intended to resolve that issue, but investigations on the KOT example surfaced a significant problem. The extension operates on SELECT statements and consequently only on read operations. A take operation involves a DELETE statement as second step which seems to block as soon as locked rows are implicated. To work around that problem a second container can be used (OrderItem_cook), which allows a cook to take the orders without a surrounding transaction and subsequently without locking. When the item is produced the cook takes the order from the main container and writes the finished order within a transaction. The sequence is illustrated in *Figure 23*.

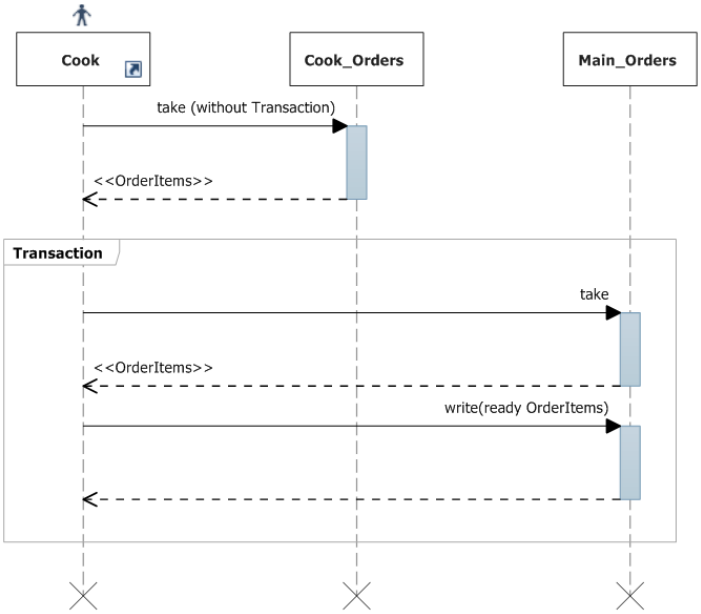

Figure 23: Cook retrieve OrderItem sequence

This strategy requires, that initial orders are written in both containers. Alternatively, an aspect can be used to initiate the write operation to the second container.

When a cook encounters a problem after the first take operation (see *Figure 23*), it cannot be rolled back because it is not part of the subsequent transaction.

6.1.2.2 Waiter

The waiter coordination is summarized in the query shown in *Code 27*.

Code

```
List<OrderItem> serveList = (from orderItem in capi4.Take<OrderItem>("OrderItem")

  let takeoderItems = (
    from orderItemTake in capi4.Read<OrderItem>(typeof(OrderItem).Name, false)
    join item in capi4.Read<Item>(typeof(Item).Name, false) on orderItemTake.ItemId equals item.Id
    join order in capi4.Read<Order>(typeof(Order).Name, false) on orderItemTake.OrderId equals order.Id
    group orderItemTake by new { order.TableId, Category = item.Category } into tableCategoryGroups
// join tables and group by TabledId and Category

    where !tableCategoryGroups.Any(oi => oi.Ready == false)
                                // all orderitems of category and table ready
          && tableCategoryGroups.Any(oi => oi.Served == false)
                                  // some not served

    let sumItemTableCategories = tableCategoryGroups.Count()
        // sum of category for table
    let sumItemTableCategoriesNotServed = tableCategoryGroups.Count(oi => oi.Served == false)
        // sum of category for table, which are not already served by a waiter
    orderby tableCategoryGroups.Key.Category ascending, sumItemTableCategories -
                                              sumItemTableCategoriesNotServed descending
        // order by category and respect partial served categories
        // as specified by the delivery requirements
    select tableCategoryGroups)
  .SelectMany(s => s) // flattening of groups
  .Where(takeOrderItem => takeOrderItem.Served == false) // filter out served items

  let sumWeightBevoreOrderItem = (from orderItemBevore in takeoderItems
                                  join item in capi4.read<Item>(typeof(Item).Name, false)
                                            on orderItem.ItemId equals item.Id
                                  where orderItemBevore.OrderNumber <= orderItem.OrderNumber
                                  select item.Weight).Sum()

  where sumWeightBevoreOrderItem <= maxServeWeight // take maximum weight into accont
  where takeoderItems.Select(to => to.Id).Contains(orderItem.Id)
  select orderItem).ToList();
```

Code 27: Waiter LinqSpace coordination

Despite the advanced complexity of the query, it is still descriptive and covers the whole coordination policy. The first step is to join all required tables and group them regarding their TableId and Category. The subsequent filter ensures that all passing groups contain ready items (which means produced by the cook) with some of them still to be served. The filtered groups are sorted regarding their category and amount of not served items. A subselect is used for the calculation of the maximum weight, which must not be exceeded.

6.1.3 Conclusion of KOT example

6.1.3.1 XcoSpaces example

The SBC paradigm does not expect entries to be related. Therefore an ER-Modell is essentially useless and must be mapped into an appropriate container/coordinator model.

Since the final purpose of the objects transmitted and maintained by the space is still in research, evaluating the usability becomes a very hard task. Initially, there is the obvious question of what objects types can be handled by XcoSpaces. Considering DDD[1] the objects transmitted may be:

[1] Domain-driven design [128]

- Business entities of the domain model: This type of objects would fit the XVSM methodology of strictly separating the data and coordination information. Moreover the general intention of coordinators, except the way Query- and LindaCoordinator handle entries (see *Chapter 1.2.2*), was to treat the information objects as unreadable black boxes. The difficulties arise when it comes to relationships and their management which is currently not part of the SBC paradigm. Although these interconnections between entities make up an essential aspect of an ERM (see *Chapter 3.2*) they perhaps can be converted in a corresponding representation using containers and coordinators. However, this would require a significantly different approach of describing a problem domain which mainly clashes with the practices nowadays. The usage of spaces as a transport layer for the domain model should be determined at an early stage of the software development cycle because once an ERM is designed it may not be easy to adopt it. Of course, the choice of the middleware used has significant impact on the overall architectural structure. To what extent these involvement can or should reach is a very interesting question but beyond the scope of this work.
- Data Transfer Objects (DTOs) / Value Objects (VOs): This way a space would mainly serve as information mediator between service layers. A typical deployment scenario would be the command pattern where each object triggers functionality at the receiving peer.

Standard coordinators cover simple and efficient coordination policies for Hashtables like Key-, Label- and VectorCoordinator and Stack or Queue structures like Fifo-, LifoCoordiators. In business applications the need for advanced coordination capabilities like grouping, ordering or aggregate functions quickly arises which are not supported and may result in many specialized, user-defined coordinator implementations.

Further, the SBC paradigm specifies that coordinators are bound to the lifetime of a container and cannot be added, modified or deleted. The only technical way to remove or introduce a coordinator is to create a new container and shuffle the entries from the old one. This may become impractical when a larger amount of data is stored in the container or the coordination policy relies on complex evaluations.

6.1.3.2 LinqSpace example

The LinqSpace sample implementation of KOT manifests some anti-pattern characteristics regarding the SBC paradigm where coordination complexity is intended to be shifted into the space framework. The basic purpose of this example is to highlight one essential fact: Referring to domain driven design, coordination information is part of the domain model. LINQ mainly can be seen as extended QueryCoordinator which addresses intrinsic information residing in the domain model. No more extrinsic coordination data is necessary, even for a sample which exceeds the functionality of standard coordinators. Moreover, the use of coordinators would result in redundant data and, in the case of LinqSpace and a database storage layer, likely decrease the speed because of an additional table join.

Another aspect illustrated by the example is that the capabilities of standard coordinators can easily be covered by LINQ. This is comprehensible since under the covers LinqSpace uses LINQ to implement the coordinators. In fact, the functionality is only a small part of the expressive potential.

An approach to efficiently shift coordination policy into LinqSpace would be the introduction of a new PrecompiledQuery-Coordinator which allows injecting LINQ queries to the abstraction framework. These queries can be precompiled for faster execution and even can be passed to the DBMS as stored procedure. The PrecompiledQueryCoordinator would replace the existing coordinators and combine the expression capabilities of LINQ with the coordination policies known by the SBC paradigm. Further, it is an interesting facet that coordination complexity can be shifted into a data-centered architecture like a relational database with stored procedures.

6.1.3.3 Application areas and similarities

6.1.3.3.1 XVSM/XcoSpaces

Application areas for XcoSpaces and XVSM in general are scenarios where persistent and relational data storage is not required. Optimal conditions are performance critical and event-based processing of short-lived objects which can be coordinated with information within or attached to the entries. Furthermore XVSM can be seen as lightweight version of an ESB because of implementing some of the fundamental patterns like Asynchronous Queuing, Intermediate Routing, Event-Driven Messaging [20] and the opportunity to register Service Agents via aspects. Service Broker functionalities like Data Format Transformation, Protocol Bridging and Data Model Transformation are not supported.

6.1.3.3.2 Relational Database

Main arguments for the use of a relational database are obviously persistence, reliability, relationships and an extensive query language. Locking and isolation levels are fully supported by the DBMS. Linq to Entities only supports optimistic concurrency, which subsequently requires a distinct read (without lock) and delete process for the take operation (see *Chapter 5.1.4.3* and *Chapter 6.4*). Basically, pessimistic concurrency would be supported by SQL over the "SELECT FOR UPDATE" clause. Event-Driven Messaging and aspect-oriented behavior can be achieved by data manipulation language (DML) triggers, which are typically used to execute a stored procedure within the database. In order to notify an application, some relational databases, like the MS SQL Server, support special "Code Triggers" (CLR Triggers) which in turn are able to send data manipulation notifications. To accomplish coordinator typical behavior, stored procedures can be used to abstract query an data access information into the database infrastructure. Furthermore, stored procedures create and additional layer, allowing to introduce security policy and providing a quick entry point for database specialist which can optimize the inquiry.

Besides the blocking behavior, which would require an additional façade object in order to block the querying thread and register notifications, all XVSM specifications can be achieved with the functionality of a relational database. The added expense of infrastructure is the reason why simple inquiry operations cannot be performed as fast as for example in XcoSpaces. Databases have an advantage when complex ER-Models must be mapped in order to carry out extensive quires.

6.1.3.3.3 LinqSpace

Since LinqSpace builds on a relational database, the usage scenarios mostly overlap. Typical setups include persistent storage of related entities and a rich framework for inquiry. In

addition, the framework supports blocking and notifications and therefore an event-driven architecture without database triggers. Stack and Queue structures for coordination are generally supported but cannot be performed as fast as with the hashed and in-memory execution of XcoSpaces (see *Chapter 6.4*).

LinqSpace combines the advantages and disadvantages of relational databases and the XVSM specification. Complex and type-safe queries are supported and can be used to inquiry entities and relationships, but with the additional overhead of a persistent data storage.

6.2 Usage examples of CAPI-3 Coordinators

The extension method used to interact with coordinators (see *Chapter 5.3*) allows a fluent integration with LINQ.

Code

```
LifoCoordinator.LifoSelector lifoSelector = capi3.CreateCoordinator<LifoCoordinator>().GetSelector();
VectorCoordinator.VectorSelector vectorSelector =
capi3.CreateCoordinator<VectorCoordinator>().GetSelector(1);

 IQueryable<TestPerson> testPersonQuery =
    from person in capi3.Read<TestPerson>().WithCoordinator(lifoSelector)
                                           .WithCoordinator(vectorSelector)
    where person.Age > 20
    select person;
```

Code 28: Usage example CAPI-3 coordinators

Code 28 shows preceding coordinators with a final LINQ query filter. Since the standard LINQ query functions change the returned interface from IXcoCapi3Queryable to IQueryable, a type cast is needed if the coordinators should be placed after the LINQ expression (see *Code 29*).

Code

```
IQueryable<TestPerson> testPersonQuery =
((IXcoCapi3Queryable<TestPerson>)from person in capi3.Read<TestPerson>()
                                 where person.Age > 20
                                 select person).WithCoordinator(lifoSelector)
                                               .WithCoordinator(vectorSelector);
```

Code 29: Usage example CAPI-3 posteriori coordinators

This type cast could be avoided with an adapter for LINQ query functions matching the IXcoCapi3Queryable interface, but is currently not offered by LinqSpace.

In the shown example the sequence of the LINQ expression and coordinators has no influence on the result. Nevertheless, the execution speed of the resulting SQL statements may depend on it, especially when JOIN and aggregate functions are involved.

6.3 Lines of code

The following tables show the lines of code in the core assemblies of XcoSpaces and LinqSpace.

Assembly	Lines of Code
XcoSpaces.Kernel	**3.122**

XcoSpaces.Kernel.Communication.TCP	**196**
XcoSpaces.Kernel.Communication.WCF	**237**
XcoSpaces.Kernel.Communication.XML	**2.100**
XcoSpaces.Kernel.Selectors	**603**
XcoSpaces.Kernel.Microkernel	**203**
Total	**6.461**

Table 3: Lines of Code XcoSpaces

Assembly	Lines of Code
CAPI-1	**623**
CAPI-2	**134**
CAPI-3	**812**
CAPI-4	**983**
CAPI-B	**35**
Total	**2.587**

Table 4: Lines of Code LinqSpace

The design of LinqSpace bases on standard .NET technologies and frameworks which has a significant influence on the code extent. It must be considered that XcoSpaces supports more coordinators and communication protocols.

6.4 Stress test

The purpose of this test is to give evaluate the performance and characteristics of XcoSpaces and LinqSpace CAPI operations regarding concurrent execution. Standard coordinators have been used to share a common ground for result comparison. The first step is to invoke multiple write operations to fill the framework with a fixed amount of 1000 entries. Subsequently in the read phase, parallel read operations are executed and in the last step the take process is tested. Each concurrent read or write operation is limited to fetch a single entry. The following diagrams represent an average time taken to finish write, read and take operations over 5 testing iterations. If an error occurs, the faulty operation is repeated until it succeeds. The average amount of errors for each concurrency level is recorded. To omit network latency which would distort the test results the examination is performed locally under the following conditions:

- Intel® Core™2 Duo CPU
- 4,00 GB RAM
- Windows 7, 64 Bit
- SSD Drive
- Microsoft SQL Server 2008 Express

The tests for XcoSpaces are taken on a single XcoKernel instance (see *Figure 24*).

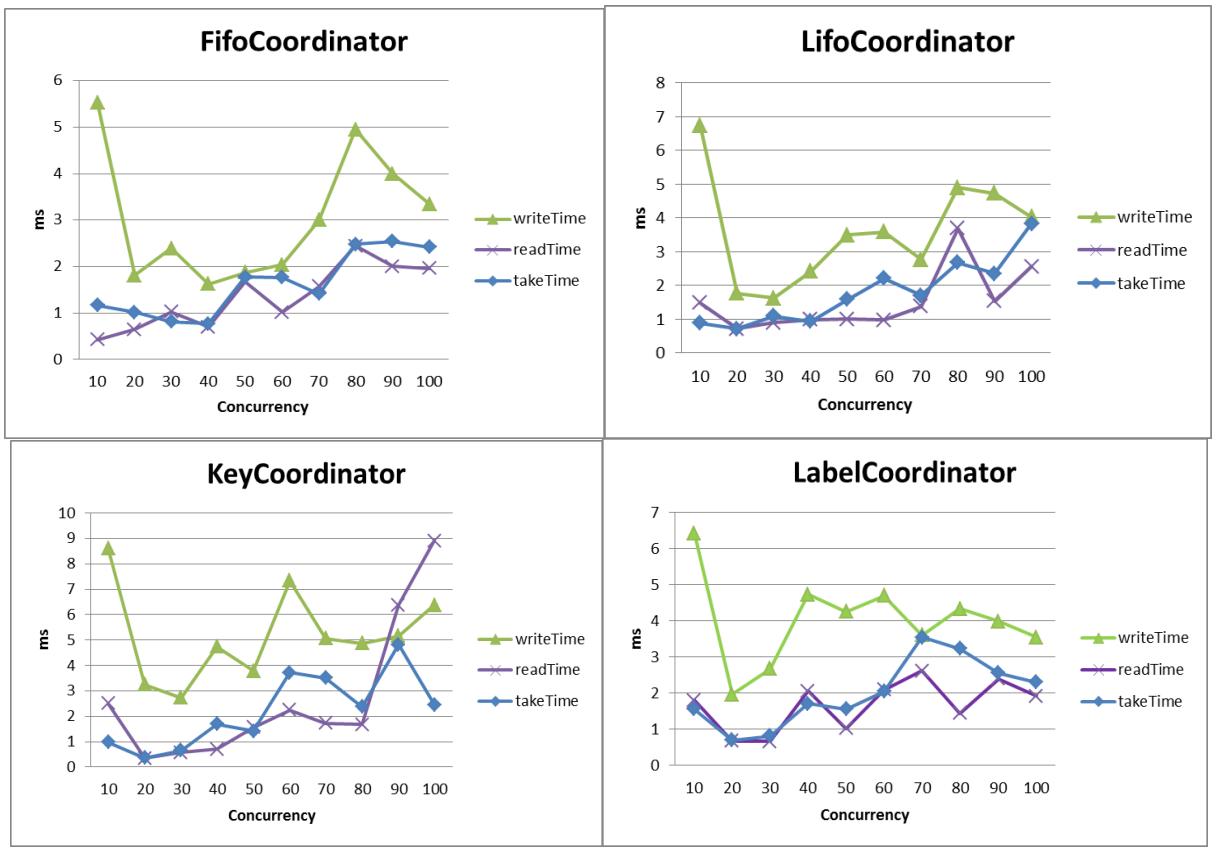

Figure 24: Stress test XcoSpaces

All operations are executed extremely fast and without errors. There is even no trend of increasing computation time recognizable.

Figure 25 shows the test results for LinqSpace. Because of the light-weight instance creation offered by the EF (see *Chapter 3.2.3*), the concurrent operations create their own LinqSpace object, all sharing the same underlying database.

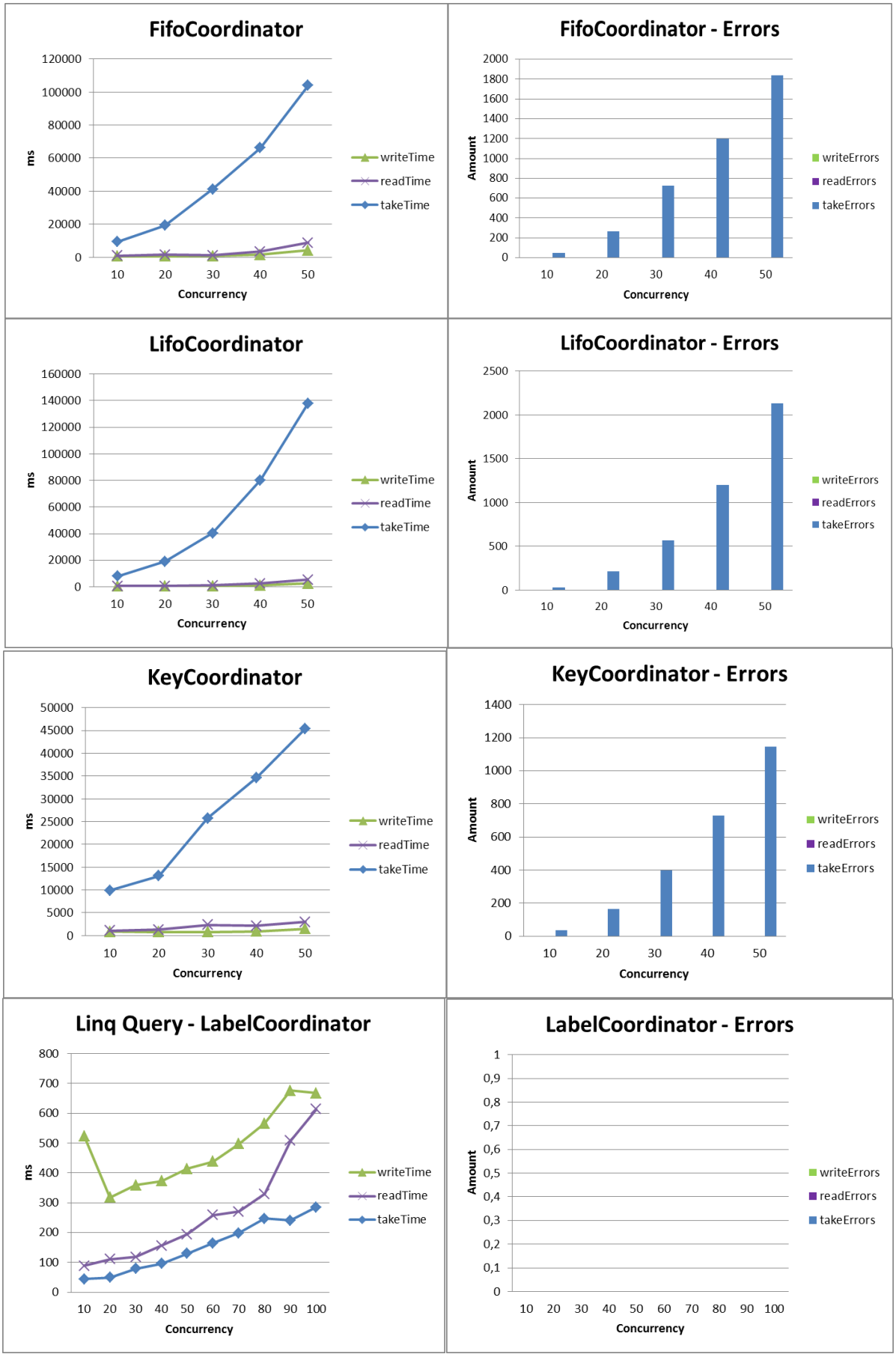

Figure 25: Stress test LinqSpace

The results for Fifo-, Lifo- and KeyCoordinator share the same horrible picture. The test was stopped at a concurrency amount of 50 because of the unpromising result and extensive test iterations. The reason can be found in the two-phased nature of the take operation. In the first step, all concurrent operations read the same entry which in the second step can only be removed by one task. The remaining operations have to be repeated because of optimistic concurrency exceptions, which can be seen in the error diagrams. This explanation holds for the Fifo-, and LifoCoordinator but the functionality used to test the KeyCoordinator chooses the entries for the take operation randomly. The reason for the deadlock exceptions thrown by the usage of this coordinator seems to reside in the DBMS and in the way EF navigation properties are locked.

Further, the EF Feature tool is not capable of automated database index creation on the foreign key columns. Indexed foreign keys would decrease table JOIN operations and therefore Fifo-, Lifo- and KeyCoordinator execution times. This index could also be created by a custom SQL statement which is directly invoked during database creation, but was omitted by the LinqSpace prototype because of additional implementation expenses.

An interesting result is obtained by the LabelCoordintor, or rather its absence. In this test the inquiry is formulated using LINQ expressions directly over the testing entity. Therefore the cross table JOIN is omitted which consequences in faster an errorless execution. The read operations are actually faster than the subsequent take operations which may be explained by caching functionality carried out in the DBMS. In summary, it can be said that coordinators which use cross-table connections to link their coordination information result in unacceptable performance.

The following tables record present the average execution time broken down by coordinator.

LinqSpace			
	Write (ms)	Read (ms)	Take (ms)
FifoCoordinator	1478,850956	3084,60064	47957,93596
LifoCoordinator	1001,815576	2023,407192	57019,53129
KeyCoordinator	950,77348	1936,106492	25746,89733
LabelCoordinator	483,395128	264,713624	153,48704

Table 5: Average execution time LinqSpace

XcoSpaces			
	Write (ms)	Read (ms)	Take (ms)
FifoCoordinator	2,647132	0,891856	1,106828
LifoCoordinator	3,204576	1,01966	1,043704
KeyCoordinator	4,619732	1,133512	1,018176
LabelCoordinator	4,023404	1,667894	2,001272

Table 6: Average execution time XcoSpaces

The evaluation shows a significant speed difference. The persistent storage of the database, the missing foreign key index, the ORM layer and the two-phase take operation can be identified as reason for this performance diversity.

7 Future Work

LinqSpace has the status of a technical prototype. The following points give an overview of further investigations:

- The speed of LinqSpace currently depends on the database and the DBMS. A cache in the decorator chain would decrease access time by holding frequently requested entities in memory. Further, there is the possibility to precompile LINQ queries and omit the expensive task of expression tree creation and serialization.
- A CAPI-4 implementation on WCF DataServices would allow LinqSpace to offer a space interface over a REST service. Probably it is only a matter of time until navigation properties are supported on derived classes (see *Chapter 5.4.1*). In addition, the complex task of expression tree serialization and desterilizing would no longer be necessary.
- Support of entity change tracking and subsequently allowing update operations across remote boundaries. Currently this capability is not supported in the CAPI-4 layer. The WCF DataServices mentioned in the previous point would already include this functionality.
- Evaluating the domain model pattern as primary storage strategy of the SBC paradigm. The handling of relationships between entities is a complex task with significant influence on locking strategies and query execution performance.
- Currently the Microsoft SQL Server and Microsoft SQL Server Compact Edition 4.0 have been tested as LinqSpace storage layer. Other databases which support the EF are about to be evaluated.
- Usage of replication technologies available for databases which can be integrated into LinqSpace. One possible candidate would be the Microsoft Sync Framework which allows synchronization of various sources.
- The evaluation of other frameworks for the storage layer which are capable of LINQ query execution like NHibernate or LINQ to SQL.

8 Conclusion

Initial attempts to use LINQ as an interface for XcoSpaces failed, since the query capabilities of the LindaCoordinator are too limited. For an appropriate porting of the functionality the QueryCoordinator is considered which is currently not implemented as part of a XVSM reference implementation in the .NET environment.

The combination of LINQ, databases and the EF results in an interesting mixture for a new XVSM reference implementation. This document presents the architecture, core functionality and considerations which led to the LinqSpace prototype. LinqSpace also can be considered as bridge between the specific requirements of the SBC paradigm and standard technologies. This approach offers completely new opportunities whose extents still have to be evaluated:

- persistent data storage
- database replication
- large amount of entries
- powerful and versatile query language
- querying a domain model

Despite the new prospects there are also additional consequences resulting from the usage of databases as primary storage, especially when it comes to inquiry and locking. Coordinators which are initially designed to increase query performance by hashing extrinsic information are inoperative in context of relational database storage. The technique used by LinqSpace involves additional coordination tables which require a computationally intensive JOIN operation and are subsequently contra productive. The stress tests provided in this document reveal the final impact of this design decision, being significant slower than the compared XcoSpaces middleware.

LinqSpace mainly implements the directives given by the XVSM specification. The standard coordinators can be mapped entirely to LINQ expressions. Therefore, it is proposed to replace the known coordinators by a single LINQ-based coordinator which combines aspect-oriented and coordination principles with flexible query capabilities.

9 List of Figures

10 List of Tables

11 List of Code samples

12 List of Abbreviations

ADO.NET	ActiveX Data Object for .NET
ASP.NET	Active Server Pages .NET
CCR	Concurrency and Coordination Runtime
CLR	Common Language Runtime
CORBA	Common Object Request Broker Architecture
CRUD	Create Read Update Delete
CTP	Community Technical Prieview
DBMS	Database management systems
DCOM	Distributed Component Object Model
DDD	Domain-driven design
DLL	dynamic linked library
DSSP	Decentralized Software Services Protocol
DTO	Data Transfer Object
EDM	Entity Data Model
EF	Entity Framework
EJB	Enterprise Java Beans
ER-Modell	Entity-Relationship-Modell
ESB	Enterprise Service Bus
IIS	Internet Information Services
JMS	Java Message Service
JSON	JavaScript Object Notation
LINQ	Language Integrated Query
LSP	Liskov Substitution Principle
MOM	Message-oriented middleware
MSMQ	Microsoft Message Queuing
OData	Open Data
OOP	Object-oriented programming
ORM	Object-Relational Mapping
PI	Persistence Ignorance
PLINQ	Parallel LINQ
POCO	Plain Old Common Language Runtime Objects
RDS	Robotics Developer Studio
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Calls
SOA	Service-oriented architecture
SOAP	Simple Object Access Protocol
TDS	Tabular Data Stream
TPH	Table Per Hierarchy
TPL	Task Parallel Library
UDP	User Datagram Protocol
UPnP	Universal Plug and Play
VO	Value Object
WSDL	Web Services Description Language
XVSMP	Extensible Virtual Shared Memory Protocol
XVSMQL	Extensible Virtual Shared Memory Query Language

13 References

4. **XIA, Yang, Qi LI and Lei WANG**. *Research on Decentralized E-commerce Architecture in P2P Environment*. Wuhan: International Conference on Electrical and Control Engineering, 2010.

5. **MORDINYI, Richard**. *Managing Complex and Dynamic Software Systems with Space-Based Computing*. Vienna: Vienna University of Technology, Institute of Computer Languages, 2010.

6. **KÜHN, eva, Richard MORDINYI and Christian SCHREIBER**. *An Extensible Space-based Coordination Approach for Modeling Complex Patterns in Large Systems*. Vienna: Vienna University of Technology, Institute of Computer Languages, 2008.

7. **MORDINYI, Richard, eva KÜHN and Alexander SCHATTEN**. *Space-based Architectures as Abstraction Layer for Distributed Business Applications*. Vienna: International Conference on Complex, Intelligent and Software Intensive Systems, 2010.

8. **CRAß, Stefan**. *A Formal Model of the Extensible Virtual Shared Memory (XVSM) and its Implementation in Haskell: Design and Specification*. Vienna: Vienna University of Technology, Institute of Computer Languages, 2010.

9. **BARISITS, Martin-Stefan**. *Design and Implementation of the next Generation XVSM Framework: Operations, Coordination and Transactions*. Vienna: Vienna University of Technology, Institute of Computer Languages, 2010.

10. **SCHREIBER, Christian**. *Design and Implementation of MozartSpaces, the Java Reference Implementation of XVSM: Custom Coordinators, Transactions and XML protocol*. Vienna: Vienna University of Technology, Institute of Computer Languages, 2008.

11. **PRÖSTLER, Michael**. *Design and Implementation of MozartSpaces, the Java Reference Implemention of XVSM: Timeout Handling, Notifications and Aspects*. Vienna: Vienna University of Technology, Institute of Computer Languages, 2008.

12. **DÖNZ, Tobias**. *Design and Implementation of the next Generation XVSM Framework: Runtime, Protocol and API*. Vienna: Vienna University of Technology, Institute of Computer Languages, 2011.

13. **SCHELLER, Thomas**. *Design and Implementation of XcoSpaces, the.Net Reference Implementation of XVSM: Core Architecture and Aspects*. Vienna: Vienna University of Technology, Institute of Computer Languages, 2008.

14. **KAROLUS, Markus**. *Design and Implementation of XcoSpaces, the.Net Reference Implementation of XVSM: Coordination, Transactions and Communication*. Vienna: Vienna University of Technology, Institute of Computer Languages, 2009.

15. **MAREK, Alexander**. *Design and implementation of TinySpaces, the.NET Micro Framework based implementation of XVSM for embedded systems*. Vienna: TU Vienna, Institute of Computer Languages, 2010.

20. **ERL, Thomas**. *SOA Design Patterns*.: Prentice Hall/PearsonPTR, 2009.

21. **FOWLER, Martin, David RICE, Matthew FOEMMEL et al.** *Patterns of Enterprise Application Architecture*.: Addison Wesley, 2002.

22. **ARMSTRONG, Joe**. *Concurrency Oriented Programming in Erlang*.: Distributed Systems Laboratory, Swedish Institute of Computer Science, 2003.

23. **NII, H. Penny**. *The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures*. California: Al Magazine Volume 7 Number 2, 1986.

24. **HAYES-ROTH, Barbara**. *The blackboard architecture: A general framework for problem solving?* Stanford University: Tech. Rep. Knowledge Systems Laboratory, Computer Science Department, 1983.

25. **HAYES-ROTH, Barbara**. *Blackboard architecture for control*.: Journal of Artificial Intelligence, 1985.

26. **GELERNTER, David**. *Generative communication in linda*, SYST, Program. Lang.: ACM Trans, 1985.

27. **HILL, Mark Donald**. *What is scalability?* New York: ACM SIGARCH, 1990.

28. **SCHWEITZER, James A.** *How security fits in -- a management view : Security is an essential for quality information*.: Computers & Security, Volume 6, Issue 2, 1987.

29. **RADACK, M Shirley**. *Security in open systems networks*.: Computer Standards & Interfaces, Volume 10, Issue 3, 1990.

30. **GADALLAH, Yasser, Mohamed Adel SERHANI and Nader MOHAMED**. *Middleware support for service discovery in special operations mobile ad hoc networks*.: Journal of Network and Computer Applications, Volume 33, Issue 5, 2010.

31. **MESHKOVA, Elena, Janne RIIHIJARVI, Marina PETROVA and Petri MAHONEN**. *A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks*. Aachen: Computer Networks, Volume 52, Issue 11, 2008.

32. **MESHKOVA, Elena, Janne RIIHIJARVI, Marina PETROVA and Petri MAHONEN**. *A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks*. Aachen: Computer Networks, Volume 52, Issue 11, 2008.

33. **KHAN, Samee Ullah and Ishfaq AHMAD**. *Comparison and analysis of ten static heuristics-based Internet data replication techniques*. Arlington: Journal of Parallel and Distributed Computing, Volume 68, Issue 2, 2008.

34. **GAO, Guoqiang, Ruixuan LI, Kunmei WEN and Xiwu GU**. *Proactive replication for rare objects in unstructured peer-to-peer networks*. Wuhan: Journal of Network and Computer Applications, 2011.

35. **ARROYO, Sinuhe, Miguel-Angel SICILIA and Jose-Manuel LOPEZ-COBO**. *Patterns of message interchange in decoupled hypermedia systems*.: Journal of Network and Computer Applications, Volume 31, Issue 2, 2008.

36. **BOSCH, Jan and Eelke FOLMER**. *Architecting for usability: a survey*.: Journal of Systems and Software, 2004.

37. **SEFFAH, Ahmed, Mohammad DONYAEE, Rex Bryan KLINE and Harkirat Kaur PADDA**. *Usability measurement and metrics: A consolidated model.* Hingham: Kluwer Academic Publishers, 2006.

38. **JURISTO, Natalia, Ana M. MORENO and Maria-Isabel SANCHEZ-SEGURA**. *Analysing the impact of usability on software design*.: Journal of Systems and Software, Volume 80, 2006.

39. **SCHELLER, Thomas and eva KÜHN**. *Measurable Concepts for the Usability of Software Components*.: Submitted for publication.

40. **LV, Qin, Pei CAO, Edith COHEN, Kai LI and Scott SHENKER**. *Search and replication in unstructured peer-to-peer networks*. New York: ICS '02 Proceedings of the 16th international conference on Supercomputing , 2002.

41. **ALTıNBÜKEN, Deniz and Öznur ÖZKASAP**. *SCALAR: Scalable Data Lookup and Replication Framework for Mobile Ad-hoc Networks*. Karabuk: 5th International Advanced Technologies Symposium, 2009.

43. **ZHAO, Weibin and Henning SCHULZRINNE**. *Enhancing Service Location Protocol for efficiency, scalability and advanced discovery*. Amsterdam: Journal of Systems and Software, Volume 75, Issues 1-2, 2005.

44. **KÜHN, eva**. *Verteiltes Programmieren mit Space Based Computing Middleware (185.226)*. Vienna, TU Wien: Institut für Computersprachen.

47. **REDKAR, T**. *Windows Azure Platform*.: Springer, 2009.

51. **CHAPPELL, David A.** *Enterprise Service Bus*, HENDRICKSON, Mike.: O'Reilly Media, Inc., 2004.

52. **TEWARI, R., M. DAHLIN, H.M. VIN and J.S. KAY**. *Design considerations for distributed caching on the Internet*.: IEEE Computer Society, 1999.

54. **ZAQAIBEH, Belal and Essam Al DAOUD**. *The Constraints of Object-Oriented Databases*. Jordan: Int. J. Open Problems Compt. Math., Vol. 1, No. 1, 2008.

56. **MEIJER, Erik, Brian BECKMAN and Gavin BIERMAN**. *LINQ: Reconciling Object, Relations and XML in the.NET Framework*. Chicago, Illinois: ACM Press, 2006.

65. **REINHARTZ-BERGER, Iris and Arnon STURM**. *Utilizing domain models for application design and validation*. Haifa: Information and Software Technology, Volume 51, Issue 8, 2009.

67. **CHEN, Peter Pin-Shan**. *The entity-relationship model—toward a unified view of data*. Massachusetts: Massachusetts Institute of Technology, 1976.

68. **BACHMAN, Charles W.** *Software for random access processing*.: ACM SIGMIS, 1965.

69. **BACHMAN, Charles W.** *Data Structure Diagrams*. New York: ACM, 1969.

70. **CODASYL**. *Data base task group report*. New York: ACM, 1971.

71. **CODD, Edgar F.** *A Relational Model of Data for Large Shared Data Banks*, BAXENDALE, P. San Jose: IBM Research Laboratory, 1970.

73. **DEHENEFFE, C., H. HENNEBERT and W. PAULUS**. *Relational model for data base*. Amsterdam: Norh-Holland Pub. Co., 1974.

74. **HAINAUT, J.L. and B. LECHARLIER**. *An extensible semantic model of data base and its data language*. Amsterdam: North-Holland Pub., 1974.

75. **SCHMID, H.A. and J.R. SWENSON**. *On the samantics of the relational model*. San Jose: ACM-SIGMOD, 1975.

76. **GOGOLLA, Martin**. *An Extended Entity-Relationship Model*. Berlin Heidelberg: Springer-Verlag, 1994.

79. **LERMAN, Julia**. *Programming Entity Framework*. Sebastopol: O'Reilly Media, Inc., 2010.

87. **GAMMA, Erich, Richard HELM, Ralph JOHNSON and John M. VLISSIDES**. *Design Patterns: Elements of Reusable Object-Oriented Software*.: Addison-Wesley Professional, 1994.

94. **ECKER, Severin**. *Communication protocols in xvsm-design and implementation*. Vienna: TU-Vienna, Insititute of Computer Languages, 2007.

95. **WOLLRATH, Ann, Roger RIGGS and Jim WALDO**. *A Distributed Object Model for the Java System*. Toronto: USENIX 1996 Conference on Object-Oriented Technologies, 1996.

102. **RENTSCH, Tim**. *Object oriented programming*. New York: ACM SIGPLAN Notices, 1982.

103. **COX, B.J.** *Object oriented programming*.: Addison-Wesley, 1985.

104. **GRIMES, Richard**. *Professional Dcom Programming*. Birmingham: Wrox Press, 1997.

107. **ORFALI, Robert, Dan HARKEY and Jeri EDWARDS**. *Instant CORBA*. New York: John Wiley & Sons, Inc., 1997.

112. **BOX, D, D EHNEBUSKE, G KAKIVAYA et al.** *Simple object access protocol (SOAP) 1.1*.: May, 2000.

113. **JAKL, Michael**. *REST Representational State Transfer*.: Citeseer, 2008.

114. **CHRISTENSEN, E, F CURBERA, G MEREDITH and S WEERAWARANA**. *Web services description language (WSDL) 1.1*., 2001.

115. **HOUSLEY, R., W. FORD, W. POLK and D. SOLO**. *Internet X. 509 public key infrastructure certificate and CRL profile*.: RFC Editor, 1999.

116. **EVJEN, B. and F. MUHA**. *Professional ASP. Net 2.0*.: Wiley-India, 2008.

117. **POSTEL, J.** *User datagram protocol*.: ISI, 1980.

118. **QIU, X., G. FOX and A. HO**. *Analysis of Concurrency and Coordination Runtime CCR and DSS*.: Citeseer, 2007.

119. **JACKSON, J.** *Microsoft robotics studio: A technical introduction*.: Robotics & Automation Magazine, IEEE, 2007.

14 Links

1. **STATS, Internet World**, , *Internet Usage in Europe*. 30 June 2010, Accessed 4 February 2011. <http://www.internetworldstats.com/stats4.htm>

2. **MINAR, Nelson**, *O'Relly P2P, Distributed Systems Topologies*. 14 December 2001, Accessed 07 February 2011. <http://openp2p.com/pub/a/p2p/2001/12/14/topologies_one.html>

3. **WIKIPEDIA**, , *Gnutella*, Accessed 07 Feburary 2011. <http://en.wikipedia.org/wiki/Gnutella>

16. **WIKIPEDIA**, , *Middleware*, Accessed 18 May 2011. <http://en.wikipedia.org/wiki/Middleware>

17. **WIKIPEDIA**, , *Message passing*, Accessed 23 July 2011. <http://en.wikipedia.org/wiki/Message_passing>

18. **DEFINING TECHNOLOGY, Inc.**, *Middlware Resource Center*, Accessed 23 July 2011. <http://www.middleware.org>

19. **ITWISSEN**, , *Middleware*, Accessed 23 July 2011. <http://www.itwissen.info/definition/lexikon/Middleware-middleware.html>

42. **WIKIPEDIA**, , *Peer-to-peer*, Accessed 15 February 2011. <http://en.wikipedia.org/wiki/Peer-to-peer>

45. **WIKIPEDIA**, , *Producer-consumer problem*, Accessed 18 March 2011. <http://en.wikipedia.org/wiki/Producer-consumer_problem>

46. *SOASpecs.com*, Accessed 19 March 2011. <http://www.soaspecs.com/ws.php>

48. **ALBAHARI, Joseph**. *Threading in C#*.: albahari.com, 2010.

49. **CODEPLEX**, , *Xcoordination Application Space*, Accessed 24 March 2011. <http://xcoappspace.codeplex.com/>

50. **WIKIPEDIA**, , *Extensible Messaging and Presence Protocol*, Accessed 24 March 2011. <http://en.wikipedia.org/wiki/Extensible_Messaging_and_Presence_Protocol>

53. *Leavitt Communications, Whatever Happened to Obect-Oriented Databases?*, Accessed 20 May 2011. <http://www.leavcom.com/db_08_00.htm>

55. **WIKIPEDIA**, , *Object-relational mapping*, Accessed 20 February 2011. <http://en.wikipedia.org/wiki/Object-relational_mapping>

57. **SHELTON, Robert**, *Robert Shelton's Blog, List of LINQ Providers*, Accessed 21 February 2011. <http://www.sheltonblog.com/archive/2008/07/11/list-of-linq-providers.aspx>

58. **WIKIPEDIA**, ,*.NET Framework*, Accessed 29 April 2011. <http://en.wikipedia.org/wiki/.NET_Framework#Versions>

59. **FOWLER, Martin**, , *FluentInterface*. 20 December 2005, Accessed 22 February 2011. <http://www.martinfowler.com/bliki/FluentInterface.html>

60. **WIKIPEDIA**, , *Decorator pattern*, Accessed 22 February 2011. <http://en.wikipedia.org/wiki/Decorator_pattern>

61. *MSDN, Visual C# Developer Center, Boxing and Unboxing (C# Programming Guide)*. July 2010, Accessed 29 April 2011. <http://msdn.microsoft.com/en-us/library/yz2be5wk.aspx>

62. **MSDN**, *Visual C# Development Center, Lambda Expressions (C# Programming Guide)*, Accessed 22 February 2011. <http://msdn.microsoft.com/en-us/library/bb397687.aspx>

63. **WIKIPEDIA**, , *Functional programming*, Accessed 29 April 2011. <http://en.wikipedia.org/wiki/Functional_programming>

64. **MICROSOFT**, *MSDN Blogs*, *Dealing with Linq's Immutable Expression Trees*. 23 May 2007, Accessed 3 February 2011. <http://blogs.msdn.com/b/jomo_fisher/archive/2007/05/23/dealing-with-linq-s-immutable-expression-trees.aspx>

66. **DAHAN, Udi**, *Udi Dahan - The Software Simplist: Enterprise Development Expert & SOA Specialist*, *Domain Model Pattern*. 21 April 2007, Accessed 20 May 2011. <http://www.udidahan.com/2007/04/21/domain-model-pattern/>

72. **WIKIPEDIA**, , *Network model (database)*, Accessed 1 March 2011. <http://en.wikipedia.org/wiki/Network_model_(database)>

77. **WIKIPEDIA**, , *Entity-relationship model*, Accessed 1 March 2011. <http://en.wikipedia.org/wiki/Entity-relationship_model>

78. **WIKIPEDIA**, , *ADO.NET Entity Framework*, Accessed 1 March 2011. <http://en.wikipedia.org/wiki/ADO.NET_Entity_Framework>

80. *ADO.NET team blog*, *EF Feature CTP5 Released!* 6 December 2010, Accessed 1 March 2011. <http://blogs.msdn.com/b/adonet/archive/2010/12/06/ef-feature-ctp5-released.aspx>

81. **MILLER, Jeremy**, *MSDN Magazine*, *Patterns in Practice: The Unit Of Work Pattern And Persistence Ignorance*, Accessed 1 March 2011. <http://msdn.microsoft.com/en-us/magazine/dd882510.aspx#id0420053>

82. **FLASKO, Elisa**, *MSDN*, *Introducing LINQ to Relational Data*. January 2008, Accessed 15 February 2011. <http://msdn.microsoft.com/en-us/library/cc161164.aspx>

83. **SCHWICHTENBERG, Holger**, *heise Developer*, *Verwirrung um objekt-relationale Mapper: LINQ-to-SQL oder ADO.NET Entity Framework?* 31 July 2009, Accessed 15 February 2011. <http://www.heise.de/developer/artikel/Verwirrung-um-objekt-relationale-Mapper-LINQ-to-SQL-oder-ADO-NET-Entity-Framework-227256.html>

84. **STACKOVERFLOW**, , *Entity Framework ObjectContext re-usage*, Accessed 1 March 2011. <http://stackoverflow.com/questions/2724176/entity-framework-objectcontext-re-usage>

85. **WIKIPEDIA**, , *Liskov substitution principle*, Accessed 7 March 2011. <http://en.wikipedia.org/wiki/Liskov_substitution_principle>

86. **MICROSOFT**, *MSDN*, *EF Feature CTP4*. 14 July 2010, Accessed 2 February 2011. <http://blogs.msdn.com/b/adonet/archive/2010/07/14/ctp4announcement.aspx>

88. *Data Developer Center*, *Delete before Insert during SaveChanges?*, Accessed 25 March 2011. <http://social.msdn.microsoft.com/Forums/en/adodotnetentityframework/thread/d8a01422-dbcb-49df-a42d-02484e2c9aab>

89. **MSDN**, , *WCF Data Services*, Accessed 6 April 2011. <http://msdn.microsoft.com/en-us/library/cc668792.aspx>

90. **MICROSOFT**, *Data Developer*, *WCF Data Service Feature Suggestions*, Accessed 6 April 2011. <http://data.uservoice.com/forums/72027-wcf-data-services-feature-suggestions/suggestions/1012603-support-navigation-properties-on-derived-types?ref=title>

91. **MSDN**, , *Expression Tree Serialization fixed for VS2010 beta 2*, Accessed 6 April 2011. <http://archive.msdn.microsoft.com/ExpressionSerializer/Thread/List.aspx>

92. *MSDN*, *ADO.NET Self-Tracking Entity Generator Template*, Accessed 25 May 2011. <http://msdn.microsoft.com/en-us/library/ff477604.aspx>

93. **WIKIPEDIA**, , *Push technology*, Accessed 6 April 2011. <http://en.wikipedia.org/wiki/Push_technology#Long_polling>

96. **MICROSOFT**,.*NET Framework Developer Center*, *LINQ*, Accessed 15 February 2011. <http://msdn.microsoft.com/en-us/netframework/aa904594.aspx>

97. **WIKIPEDIA**, , *LINQ*, Accessed 15 February 2011. <http://en.wikipedia.org/wiki/Language_Integrated_Query>

98. **WORTHINGTON, David**, *SDTimes*, *Does.NET With LINQ Beat Java?* 29 January 2008, Accessed 21 February 2011. <http://www.sdtimes.com/content/article.aspx?ArticleID=31643&page=1>

99. **WIKIPEDIA**, , *Microsoft Visual Studio*, Accessed 16 February 2011. <http://en.wikipedia.org/wiki/Microsoft_Visual_Studio>

100. **WIKIPEDIA**, , *ADO.NET*, Accessed 22 July 2011. <http://en.wikipedia.org/wiki/ADO.NET>

101. **WIKIPEDIA**, , *ADO.NET Entity Framework*, Accessed 22 July 2011. <http://en.wikipedia.org/wiki/ADO.NET_Entity_Framework>

105. **WIKIPEDIA**, , *Remote procedure call*, Accessed 23 July 2011. <http://en.wikipedia.org/wiki/Remote_procedure_call>

106. **WIKIPEDIA**, , *Java remote method invocation*, Accessed 23 July 2011. <http://en.wikipedia.org/wiki/Java_remote_method_invocation>

108. **WIKIPEDIA**, , *Microsoft Message Queuing*, Accessed 23 July 2011. <http://en.wikipedia.org/wiki/Microsoft_Message_Queuing>

109. **WIKIPEDIA**, , *Java Message Service*, Accessed 23 July 2011. <http://de.wikipedia.org/wiki/Java_Message_Service>

110. **ORACLE**, , *Enterprise JavaBeans Technology*, Accessed 23 July 2011. <http://www.oracle.com/technetwork/java/javaee/ejb/index.html>

111. **MICROSOFT**, , *IIS*, Accessed 23 July 2011. <http://www.iis.net>

120. **WIKIPEDIA**, , *Universal Plug and Play*, Accessed 23 July 2011. <http://en.wikipedia.org/wiki/Universal_Plug_and_Play>

121. , *Introducing JSON*, Accessed 23 July 2011. <http://www.json.org/>

122. **MSDN**, , *Task Parallel Library*, Accessed 23 July 2011. <http://msdn.microsoft.com/en-us/library/dd460717.aspx>

123. **MSDN**, , *Parallel LINQ (PLINQ)*, Accessed 23 July 2011. <http://msdn.microsoft.com/de-de/library/dd460688.aspx>

124. **WIKIPEDIA**, , *Common Language Runtime*, Accessed 23 July 2011. <http://en.wikipedia.org/wiki/Common_Language_Runtime>

125. **MILLER, Jeremy**, *MSDN Magazine*, *The Unit Of Work Pattern And Persistence Ignorance*, Accessed 18 August 2011. <http://msdn.microsoft.com/en-us/magazine/dd882510.aspx>

126. **WIKIPEDIA**, , *Tabular Data Stream*, Accessed 18 August 2011. <http://en.wikipedia.org/wiki/Tabular_Data_Stream>

127. *Open Data Protocol*, Accessed 18 August 2011. <http://www.odata.org/>

128. *Domain-Driven Design Community*, Accessed 18 August 2011. <http://domaindrivendesign.org/>