Karlsruhe Institute of Technology

Evaluating the Xydra Framework for Social Web Applications

Diploma Thesis of

Thomas Lichtenstein

At the Faculty of Economics and Business Engineering
Institute of Applied Informatics
and Formal Description Methods (AIFB)

Reviewer:	Prof. Dr. Rudi Studer
Advisor:	Dr. Max Völkel

Duration: January 31, 2012 – July 31, 2012

Submission date: July 31, 2012

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, July 31, 2012

. . . .

(**Thomas Lichtenstein**)

Contents

1 Introduction

Problem statement. General availability of computing devices, the Internet and the Web has enabled users to communicate by email, instant messaging, and other applications. In recent years, in particular social web applications have become very popular, which - as the name suggests - focus on social features. Social web applications typically allow the formation of communities in which users can befriend and message each other as well as collaboratively edit content.

A single application can have thousands or even millions of simultaneous users. The huge amount of users and the high level of interaction raise challenges in building social web applications in a cost-effective way. For instance, scalability and performance of the applications must be taken into account. Furthermore, in order to provide a seamless user experience, responsiveness of the application is an important factor. Due to increasing mobile Web usage, applications need to be able to deal with intermittent loss of connectivity of mobile clients. Ideally, applications can be used even when the user is offline.

These and other problems must be faced when building social web applications. Traditional web application approaches often fail to meet the new requirements of social web applications.

Idea and goal. In this work, a new approach in building social web applications in a cost-effective way is designed and evaluated. Subject to the new approach is Xydra, an open-source platform-independent data management framework. Xydra's features are already in compliance with several of the requirements for social web applications. For instance, scalability and performance are claimed to be an integral part of the Xydra framework. Overall, it presents itself as a promising framework for the development of social web applications. Until now, however, it has been unclear whether Xydra is suitable for such applications. Therefore, the goal of this work is to evaluate Xydra in regards to its suitability for building social web applications in two steps. First, a blueprint design of building social web applications with Xydra in a cost-effective way is developed. Second, this solution is evaluated in respect to the requirements of social web applications and compared to alternative approaches.

Outline. Chapter 2 presents a brief definition of social web applications as well as a comprehensive introduction to the Xydra framework. The chapter closes with an overview of core technologies used within this work.

Chapter 3 discusses the requirements of social web applications. Subject to the discussion are issues such as scalability, responsiveness and cost-effectiveness among other requirements.

Chapter 4 analyzes traditional web applications and their short-comings when building social web applications. As the costs of an application are an important aspect of the evaluation in this work, the analysis contains an overview and comparison of pricing models of cloud service providers.

A central part of this work is Chapter 5 - the Design Chapter. Therein a blueprint of building social web applications with Xydra in a cost-effective way is presented. The architecture and application structure are discussed in detail.

Chapter 6 presents an experimental application that adheres to the formerly described design.

Chapter 7 discusses the extent to which the design meets the formerly discussed requirements. A closer look is given on the aspects of responsiveness and cost-effectiveness of the provided solution. The chapter closes with a comparison of the solution presented in this work and other third-party solutions.

Chapter 8 gives a resume of the solution and findings presented in this work and a future outlook.

2 Background

This chapter introduces social web applications and an overview of the Xydra framework, that is used in the remainder of this work, is given. Furthermore, the replication capability of Xydra is characterized. The chapter closes with a brief description of other technologies used.

2.1 Social web applications

The term social Web application describes a rapidly evolving concept of web sites that focus on social features. The feature set usually encompasses the formation of communities: users can publish a profile, associate themselves with other users and have the ability to discover new users [BE07].

Usually, the communities serve a certain purpose that can be as broadly as mapping the real-life social relationships between people onto the web or be as specific as forming communities around a certain topic. The concrete feature set of a social web applications is diverse and usually bound to the aspect it focuses on: for instance, a social web application for sharing of photos will likely offer ways to annotate photos or comment on them. A social web application for broadcasting short messages may focus on the near instant delivery of messages. In this work, a social web application for collaboratively managing and delegating tasks is presented.

A common concept among social web applications is the ability for users to publish user-created content (UCC) [VWV07]. The UCC may be photos, audio or video content, books or to-do items. Users of a social web application may be able to collaboratively edit the UCC. As a single social web application can have a user base of millions of users with a high level of interaction, architecture challenges such as performance, scalability and availability of the application are raised [KJL10]. Other important aspects of social web applications are the responsiveness of the application, as a seamless user experience is a significant factor for success of an application. With the rise in mobile Web usage, unstable connectivity of users must increasingly be taken into account and raises questions about the consistency of the application.

The requirements of social web applications are discussed in more detail in Chapter 3.

2.2 Xydra

In this section Xydra[1], an open-source platform-independent application model and data management framework, is described.

Data model

Xydra uses a hierarchical, tree-structured data model that consists of five types of singular entities, namely *XRepository*, *XModels*, *XObjects*, *XFields* and *XValues*.

Entities

At the root level, a *XRepository* is a container of a set of *XModels*. A repository provides methods for creating, retrieving, and removing *XModels*. A *XRepository* is only used to store and organize *XModels*, but is generally not used to model data by itself. *XModels*, however, are used to model the application data at a high level. Similar to the methods provided by a *XRepository*, *XModels* support methods for creating, retrieving and removing *XObjects* entities. *XObjects* contain a set of *XFields* which represent attributes of the *XObject*. *XObjects* also support the respective methods for creating, retrieving and removing *XFields*. *XFields*, in contrast to the former entities are a container for at most one *XValue* entity. Besides the methods for creating, retrieving and removing *XValues*, *XFields* also support a modify method to change the value of a *XField*. *XValue* entities, being at the leaf level of the tree structured data model, represent primitive data structures to store single values of the type boolean, integer, long, double or string. In addition to single value types, Xydra also supports *XValue* entities that store abstract data structures like lists or sets of primitive values. List *XValue* entities are used to represent structured or ordered data while sets represent a duplicate-free, unordered collection of primitive values.

How these entities relate to entities in other domains is determined in Table 2.1. Figure 2.1 illustrates the hierarchical, tree-structured entities.

Xydra	**Database**	**Object Oriented Programming**	**Example**
XRepository	Database	-	-
XModel	Table	Packgage	Todolist
XObject	Row	Object	TodoItem
XField	Column	Field	TodoDescription

Table 2.1: Xydra entities and corresponding entities in the domain of databases, object oriented programming, and an example use case.

1 Xydra - The application model for seamless usage on GAE, GWT and pure Java - Google Project Hosting, http://code.google.com/p/xydra/ accessed July 12, 2012

Ids and Addresses

Each entity type with the exception of the *XValue* entity type has an *XId* that is used to address the entity with respect to its parent entity. The *XId* is a unique identifier among its siblings but may not be globally unique. An *XId* can be thought of as the name of an entity much like a filename in a folder. In order to be able to globally address entities, each entity also has an *XAddress* that is a concatenation of its *XId* and the *XAddress* of its parent. An *XAddress* is therefore similar to a filename including the absolute path. As Figure 2.1 shows, the addressing scheme follows the pattern of addressing in traditional directory trees. Both *XId* and *XAddress* are specific *XValue* entities in addition to the primitive and abstract *XValue* entities mentioned above.

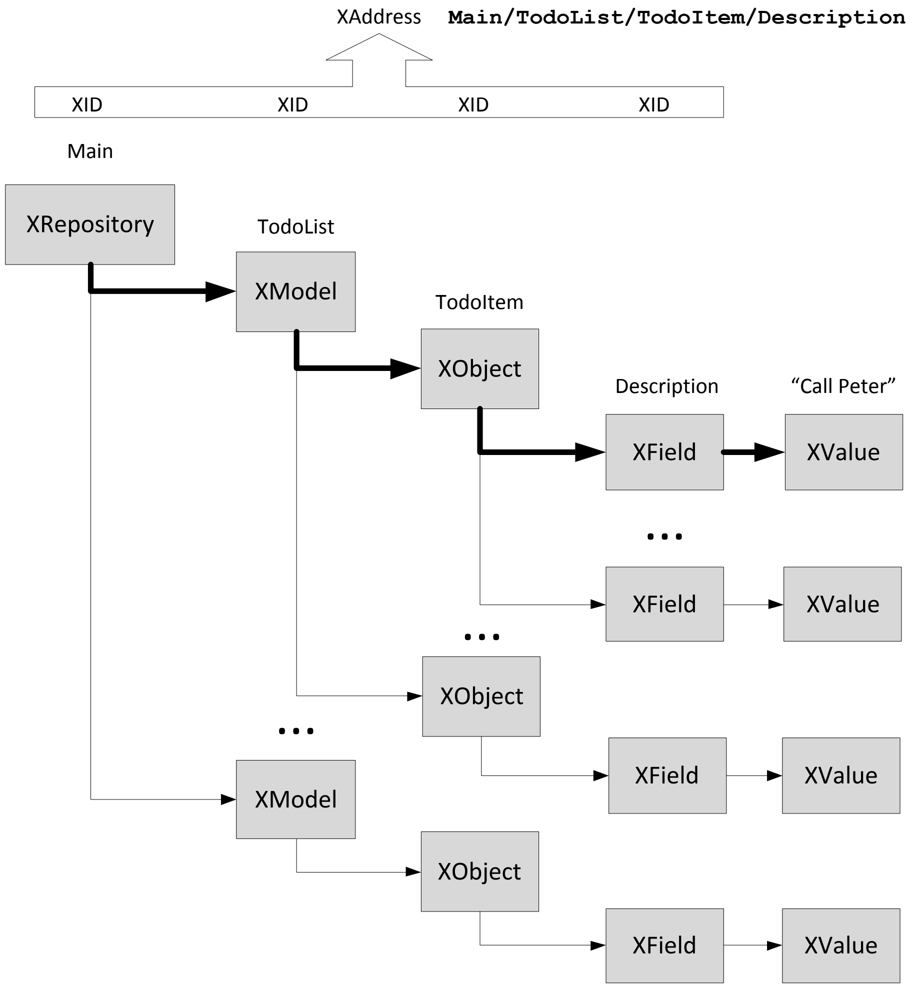

Figure 2.1: Hierarchical model of Xydra entities[1]

Commands

Xydra supports operations for creating and removing entities on each level with the additional method of changing the value of a *XField* entity. The operations are named *XCommands*, that encapsulate single or multiple change operations on any type of entity.

Xydra distinguishes between two types of *XCommands*, safe and forced *XCommands*. Safe *XCommands* contain operations that require a certain precondition to be fulfilled in order to be executed. Suppose an operation of the type *Add*, e.g. a new *XModel*, with a certain *XId*, should be added to a *XRepository*. Using a safe *XCommand*, this operation will only succeed if the *XModel*'s *XId* doesn't already exist within the *XRepository*. Analogously, a safe *XCommand* with an operation of the type *Remove* and a given revision number has an implicit precondition that the entity to be removed must have a matching revision number and it must exist before it can be removed. In addition to that, safe *XCommands* with an operation of the type *Change* only succeed if the *XField* to be changed has the same revision number as specified in the *XCommand* and will fail otherwise.

On the other hand, forced *XCommands* succeed but may have no effect. Again, suppose an *Add* operation of an *XModel* to an *XRepository* and the *XModel* already exists within the *XRepository*. The forced *XCommand* will succeed but have no effect as its intended result is already present. Correspondingly, an *XCommand* with a *Remove* operation always succeeds irrespective of the existence of the entity to be removed. As stated above, a safe *XCommand*, however, would fail in both scenarios. Finally, a forced *XCommand* with a *Change* operation is executed on an *XField* without requiring a certain revision number. As a summary, forced *XCommands* care about a postcondition whereas safe *XCommands* require a precondition to be fulfilled.

Revisions

Given the provided *XCommands*, Xydra increases a revision number on each entity affected by an *XCommand* issued. When a child entity is added or removed as well as when an *XValue* is changed, the revision number of all its parents is increased.

Actors

Each *XCommand* is executed on behalf of a so called actor. The presence of actors allows Xydra the support of access rights management and authorization on the application data.

Events

Xydra generally supports concurrent access and modification of application data by multiple actors. The application data is modified by applying *XCommands* on any type of entity

1 A similar illustration originally appeared in the Xydra Wiki, http://code.google.com/p/xydra/wiki/XydraBasics, accessed Jul 12, 2012

on behalf of an actor. To alleviate application development in respect to changes made to the application data by different actors, Xydra sends out and allows observing change events, namely *XEvents*, as a result of executed *XCommands*. For one, *XEvents* include information on the actor responsible for the change and the type of change, i.e. *Add*, *Remove*, *Change* or *Transaction*. The type of change represented by *XEvents* is directly related to the *XCommand* issued, e.g. an *XCommand* of type *Add* will result in an *XEvent* of type *Add*. *XEvents* also include the *XAddress* of the modified entity and information on the increased revision number after the *XCommand* has been applied.

Changelog

Xydra keeps a history of all *XEvents* that result from changes made to an entity. Each changelog entry includes the actor who invoked the respective change operation. It is therefore possible to revert any changes made to an entity: Each change can be reverted by executing the inverse type of *XCommand*, that is determined by the type of *XEvent*. A sequence of changes found in the changelog can then be rolled back by issuing inverse *XCommands* in reverse order. The existence of a changelog also helps to synchronize entities among a server and client by comparing their changelogs. More information on the synchronizer is provided in Section 2.2.

Transactions

XTransactions are extended *XCommands* that contain a sequentially ordered list of basic *XCommands* to be executed in a row. If all contained *XCommands* succeed, the *XTransaction* as a whole succeeds. If, however, an *XCommand* fails, then the *XTransaction* needs to roll back and undo any formerly executed *XCommand* of that *XTransaction*. In this way, *XTransactions* guarantee atomicity; i.e., they preserve an all-or-nothing property. Because *XTransactions* are *XCommands* themselves, they can be sent and used to synchronize operations between a client and server. As a side note, *XTransactions* could contain *XCommands* that are *XTransactions* themselves. In this case the structure is flattened and all nested basic *XCommands* are merged into one *XTransaction*.

Persistence

Xydra supports a persistence layer to permanently store application data. The persistence layer uses the Google App Engine data store described in Section 2.5. The persistence layer offers access to so called snapshots of *XModels*, that represent the state of an *XModel* for a given revision number.

Synchronizer

The Xydra synchronizer is capable of synchronizing a client-side *XModel* with a server-side *XModel*. For each *XModel* and containing *XObjects* it stores a synchronization revision number. The synchronization revision number determines the highest revision of an entity, that has been synchronized with the server.

In order for this synchronization to work, the synchronizer differentiates between local changes and remote changes.

Local changes are *XCommands*, derived from the changelog, that have been applied locally but that have not been sent to the server yet. Each *XCommand* affects a particular *XEntity* that is part of the *XModel* or affecting the *XModel* itself. During a synchronization step these local changes are sent as *XCommands* to the server. The server applies these *XCommands* and returns resulting *XEvents*. Some *XCommands* may fail due to a conflict. For those failures an empty result is returned. Besides the *XEvents* resulting from client *XCommands*, the server also includes any *XEvents* from *XCommands* that were issued by other clients or originated from server-side execution of application code.

The returned *XEvents* from the server after synchronization are called remote changes. From these remote changes *XCommands* are derived. The remote changes also include *XEvents* that are the result of *XCommands* the client sent to the server during synchronization. All *XCommands* derived from remote changes, leaving out the ones that originated from the client, are applied locally. The ones that originated on the client are already applied and are therefore left out. After a synchronization step on the client-side, the synchronizer sends out *XEvents* on all *XEntities* that were affected by synchronized *XCommands*. These events can be observed by a client-side application in order to react on remote changes.

After a synchronization step both client and server have an identical *XModel*.

2.3 Replication

An *XModel* that is present on a client is usually a replica of a server-side *XModel*. The replicated *XModel* is subject to synchronization by the Xydra synchronizer. In this section, the Xydra synchronizer is characterized in accordance to a classification of replication variants presented in [Ter08]. Definitions of the classification terms are also given in [Ter08].

The basic system model of the Xydra synchronizer assigns the server the role of a master and the clients the role of devices, therefore it can be characterized as device-master replication.

When exchanging updates, e.g. changes on an *XModel*, between the master and the device, the updates are represented by the operations, e.g. *XCommands* or corresponding *XEvents*. As the synchronizer sends *XCommands* and receives *XEvents*, the synchronizer is using an operation-sending protocol.

The Xydra changelog keeps track of operations, *XCommands* and *XEvents*, and revision numbers, it therefore can be characterized as a log-based recording of updates.

As the synchronizer stores a synchronization revision number for each *XModel* and *XObject*, the log-based protocol can utilize this knowledge vector to not send redundant updates between the devices and the master. Therefore, it can further be characterized as a knowledge-driven log-based protocol. A similar approach has been demonstrated by Bayou [TTP+95].

The ordering of updates is made possible by utilizing the revision number of each operation. Therefore, the synchronizer is further characterized as using update counters.

Conflicting updates are detected on the server by comparing the revision numbers of operations. The revision numbers of a sequence of operations thereby function as dependency checks on the update. Therefore, the detection can be characterized as made-with-knowledge conflict detection that utilizes dependency checks. Again, a similar approach has been used by Bayou [TTP+95].

When a conflicting update is detected by the server and the device is notified of it, the client-side synchronizer allows the registration of callbacks for conflict resolution.

To conclude, all operations are received by the server exactly once. The operations are ordered by revision numbers and can be executed in a deterministic way. Conflicting operations can be detected and rejected by the server. A rollback on the device is made possible by utilizing the changelog. The device can implement specific code to resolve conflicts. By means of this characterization and its effect, the Xydra synchronizer maintains eventual consistency.

2.4 HTML5 Web Storage

Within this work, the HTML5 Web Storage [Hic11], also referred to as local storage, feature is used. Web Storage is a browser key-value store that web applications can use to permanently store application data. The web storage feature is supported across modern browsers and allows the building of web applications that can work while the browser is offline.

2.5 Google App Engine

Google App Engine[1](GAE) is a Platform as a Service (PaaS) cloud service. It provides a server-side application execution environment for applications written in a variety of languages, including Java. GAE provides APIs for services such as file storage, database access, sending email and others that application can use. The services provided are highly scalable and do not need to be maintained by the application developer. Applications are bundled in containers that are automatically distributed and executed across multiple servers on increasing demand.

Xydra uses the GAE service's storage capabilities and other features. The Xydra data model is permanently stored within the GAE key-value datastore. By using the GAE service, applications built with Xydra benefit from GAE's scalability features.

1 Google App Engine - Google Developers, http://developers.google.com/appengine accessed July 24th, 2012

In Section 4.2 a comparison of the pricing model of GAE and other cloud service providers is given.

2.6 Google Web Toolkit

Google Web Toolkit[1](GWT) is a web application framework. GWT allows the building of web applications in Java that can be executed within Google App Engine at the server-side and also compiled to efficient JavaScript code runnable across browsers at the client-side. Developers can use many of the benefits of Java, such as being statically typed, and the Java ecosystem, including integrated development environments, in order to write their applications in a cost-effective way. In this work, GWT is used to compile the Java-based design solution and the Java-based Xydra framework into JavaScript in order to run it across modern browsers.

1 Google Web Toolkit - Google Developers, http://developers.google.com/web-toolkit/ accessed July 24th, 2012

3 Requirements

In this chapter requirements for building social web applications are discussed. Subject to the discussion are issues such as scalability, availability, and responsiveness among other requirements.

Scalability

When building social web applications, scalability of the application is of particular importance. A single social web application can have thousands or even millions of simultaneous users. Therefore, scalability is required in almost any aspect of the architecture.

Suppose a social web application exists in which users can publish articles as authors and subscribe to authors for new articles as readers. A group of co-authors may work together on a single article that is published to an audience of readers. Users can both be authors and readers at the same time.

Let n be the number of users, k the number of articles per user, s_w the number of co-authors working on a single article, s_r the number of readers per author. From this exemplary model regardless of social artifacts and numbers given by Facebook[1], a reasonable estimate of the size of the parameters is:

Parameter	Description	Typical	Upper bound
n	#users	1,000 – 1,000,000	1,000,000,000
k	#items per author	100 – 1,000	1,000,000
s_w	#co-authors	1 – 10	100
s_r	#readers per author	10 – 100	1,000,000

Table 3.1: Estimated figures for a typical social web application

Requirement 1: Scalability of the application is required with respect to the number of users and application specific parameters such as the number of items.

1 Key Facts - Facebook Newsroom, http://newsroom.fb.com/content/default.aspx?NewsAreaId=22 accessed July 19, 2012

Consistency

Considering the order of magnitude of the user base and the numerous author-reader relationships in the example application, global data consistency can be challenging while providing concurrent, real-time access to data objects. In practice, it is often only feasible to use relaxed guarantees on consistency such as eventual consistency [Bre12].

As an example for eventual consistency: For an application it may be tolerable that a newly published article is visible to all readers within a certain amount of time, e.g. 5 minutes, under normal system load. In exceptional cases though, no specific time frame can be determined, eventual consistency only guarantees that the article will be visible at some undetermined time in the future.

Requirement 2: Eventual consistency is required for application data.

Availability

In addition to consistency, availability of the application should be as high as possible. The user is able use the application while he can communicate with the server. Yet, the goal is to also support availability of the application in times the user is offline, e.g.: he experiences an intermittent loss of connection, or in cases of server outages. The user should be able to use the basic functionality of the application even though he can not directly synchronize his changes with the server while being offline.

Requirement 3: Availability of basic application functionality while the user is offline.

Synchronization, conflict detection, and resolution

When users work on application data while they are offline, the state of the application data will likely diverge from the state other users of the application see. Once the user reconnects, his changes should be synced with the server and the system should reconcile differences in state and do its best to resolve conflicting changes of other users [Can10].

For instance, suppose two co-authors editing a single article who are using the example application through an unreliable mobile network. At times, both authors may simultaneously be offline for a longer period of time. They can still edit the article locally and later need to synchronize the changes with the server. During synchronization, conflicting edits, e.g. one author first deleted an paragraph while the other subsequently added sentences to it, need to be detected and gracefully resolved. For instance, the latter author could be asked whether he wants to revert the deletion by the first author or whether he accepts his changes to be lost.

Requirement 4: Local changes are synced with the server, conflicts are detected and gracefully resolved automatically or with user intervention.

Responsiveness

Good responsiveness is a general requirement for a seamless user experience. It is particularly crucial due to the real-time nature of social web applications. Following a certain user activity, results of that action should be visible to the user within $0.1s$ such that the user has the perception that he were directly working on the application data [Nie93]. Providing such responsiveness can be challenging when the result of that action involves communicating with the server as round-trip times can already exceed the $0.1s$ bound. Therefore, the system should resort to working on local data directly accessible by the user and lazily propagate changes to the server in the background as much as possible. In addition to client-side application responsiveness, server-side response times for processing of application data should likewise be fast.

For instance, a newly created article may be instantaneously visible and modifiable by the user locally while a background task is synchronizing that article with the server. At times, this optimistic approach may lead to actions that were processed locally but couldn't be completed globally on the server. In this case, the system should handle such conflicts gracefully in ways described in the previous section on conflict detection and resolution.

Requirement 5: The application should respond to user activity within a timespan that is perceived by the user as responsive as working on local data.

Device independence

Social web applications should be compatible with any modern Web browser environment independent from the actual device or platform the browser is executed upon. A user may access a single application from his desktop computer and his mobile device, such as a tablet computer or smartphone. Therefore the application should gracefully adapt to the functionality provided by the browser, e.g. local storage capability.

Requirement 6: The application should be compatible to most of the modern browsers and therefore be mostly device independent.

Cost-effectiveness

Cost-effectiveness is largely affected by two main cost-drivers: development costs and cost of operation of an application. From a developer perspective, the developer should be assisted by tools such as a powerful development environment to be able to design and build social web applications with as little effort and time consumption as possible.

In addition to the development, the on-going maintenance of the application should also be cost-effective by following best practices and well-know patterns during development. Code base maintenance can be eased, when parts of the application code can be shared between server and client. This means, that the application business logic can be reused and does not have to be adapted and rewritten to be able to run on the client. Deployment of the application and its future releases should only require the end-user to reload the web application from its originating domain.

Another aspect in cost-effectiveness is the computational cost of the application. From an application provider perspective, the application will incur costs on computation, storage, and bandwidth. Therefore, at an early stage the application development should respect cost metrics and make use of the resources in an economical way. For instance, the amount of data that is transferred between the client and server should be limited to a minimum.

Requirement 7: The application should be cost effective in regards to development, maintenance and computational resources.

Data versioning and history

In order to alleviate support for consistency and conflict resolution, the underlying application data model should support versioning of the data entities. To support this, each data entity should keep a history of all changes made to itself.

Requirement 8: The data model should support versioning and keep a full history of changes on data entities.

Security

As a huge part of the functionality of the application should be available to the user even when the user can not establish a connection to the server, e.g. he is offline, it is required that the logic of the application is residing at least in parts on the client. The client-side part of the application may apply changes to the data which are later synced with the server. This immediately raises concerns about the validity and integrity of the changes to the data that the server receives from the client [Rit07]. When the client is allowed to change application data, the server must take measures to ensure these changes comply with a correct execution of the application, i.e. the data is a result of valid state transitions. Furthermore, the data must be validated to not contain harmful input that would compromise security. Finally, the server must ensure authenticity and check authorization of the client. All in all, this the principle to "never trust the client" must be enforced. Security, therefore, is an important factor.

Requirement 9: The server must authenticate and authorize the client. Synchronized data must be validated.

User interface and view binding

The user interface should instantly reflect changes made to the data model, referred to as model-view binding. The developer should be able to define a declarative layout which is translated to HTML with interactive JavaScript elements. The layout mainly consists of widgets that represent single application objects, e.g. an article, or can represent collections of such objects, e.g. a list of articles. These widgets should be bound to the application data. Whenever the user takes an action within the application which modifies the underlying application data, these changes should instantly be reflected by the widgets. For instance, a widget listing a collection of articles should directly display any new article that a user created either locally or which is originating from elsewhere from synchronization. In addition to this established data-binding UI-pattern, widgets should also reflect whether their underlying data objects are already synchronized with the server or only preliminarily created locally.

Developers should be assisted in binding widgets to the application data to alleviate the burden that is usually involved in keeping the view state in sync with the application data[1].

Requirement 10: A mechanism for the binding of user interface elements to the underlying application data is required. The elements should reflect the synchronization state of the application data.

Summary

In Table 3.2 a summary of the discussed requirements is listed.

Requirement	Description
R1	Scalability of the application is required with respect to the number of users and application specific parameters such as the number of items.
R2	Eventual consistency is required for application data.
R3	Availability of basic application functionality while the user is offline.
R4	Local changes are synced with the server, conflicts are detected and gracefully resolved automatically or with user intervention.
R5	The application should respond to user activity within a timespan that is perceived by the user as responsive as working on local data.
R6	The application should be compatible to most of the modern browsers and therefore be mostly device independent.
R7	The application should be cost effective in regards to development, maintenance and computational resources.
R8	The data model should support versioning and keep a full history of changes on data entities.
R9	The server must authenticate and authorize the client. Synchronized data must be validated.
R10	A mechanism for binding of user interface elements to the underlying application data is required. The elements should reflect the synchronization state of the application data.

Table 3.2: List of requirements for social web applications

1 Concepts of keeping a user interface in sync with an application data model are described by Martin Fowler in "GUI Architectures", http://martinfowler.com/eaaDev/uiArchs.html, accessed July 13, 2012

4 Analysis

This chapter analyzes traditional web applications and discusses the short-comings with respect to the new requirements of building cost-effective social web applications. The short-comings are discussed in order to contrast the traditional approach with the design of the approach presented in the following Design chapter.

The remainder of the chapter gives an overview of pricing models of three cloud service providers in order to draw conclusions on the cost-effectiveness of the designed solution, which is discussed in the Evaluation chapter.

4.1 Traditional web application

This section introduces traditional web applications which represent a simple form of web application.

A traditional web application embraces the server as the main component of the architecture that is holding the application state. The client browser merely presents the user interface and holds no application state.

As shown in Figure 4.1, the architecture can be decomposed into three layers, the presentation, application and persistence layer [ACKM10]. In the following, the layers for the traditional scenario are described in more detail.

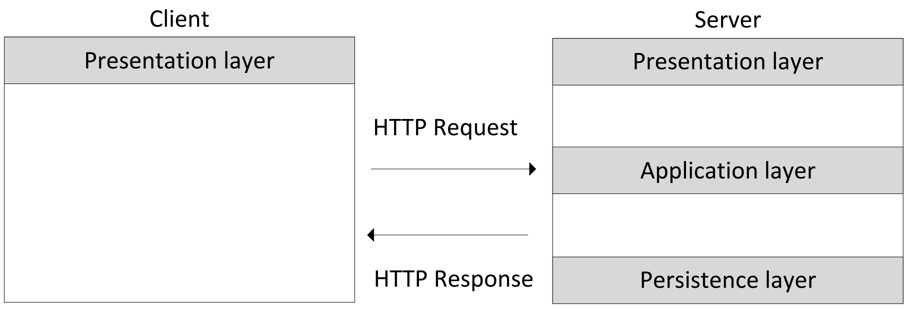

Figure 4.1: Visual overview of a traditional web application architecture.

Presentation layer. The presentation layer is present on both the client and the server.

The presentation layer on the client mainly consists of plain HTML pages that are returned by the server. User interaction with this page can be augmented with basic JavaScript. For instance, dialogs may inform the user of a missing but required input when submitting data to the server. The client-side presentation layer only provides minimal functionality.

The server-side presentation layer, however, is more complex and its main goal is to transform application layer data into a view that can be delivered as a page to the client. Therefore, it accounts for the visual representation of the application layer data. For instance, in order to meet this goal, templates can be defined that contain placeholders for application data attributes. A server-side render engine is often used to compile these templates and substitute the placeholders with actual application data. The result is a rendered page that can be delivered to the client.

Application layer. The application layer resides exclusively on the server. It holds the business logic of the application which defines the states of the application and the allowed transitions between them. The state is mostly represented by application level data objects. The data objects are composed of raw data that is permanently stored in the persistence layer. The application layer decides which data needs to be stored and when to hand the data down to the persistence layer. It also determines which data is handed over to the presentation layer.

The allowed state transitions are usually represented in application level methods that are invoked when the client fetches the corresponding Web resource through a HTTP request on a given URL.

The application layer accepts data from user input when the user invokes a particular application method. For security reasons, the application layer needs to authenticate the user and check if the user is authorized to invoke that method. It also needs to validate the input.

Persistence layer. Like the application layer, the persistence layer is also found exclusively on the server. The main goal of this layer is to store application data and provide access to it in a flexible and efficient way. For efficient storage, the data is often stored in a stripped down version as opposed to an enriched representation found in the application layer. Indexes are regularly used for fast access to that data, including access via queries. This functionality is often provided by including third-party relational databases or key-value stores into the architecture

Layer interaction

Following the previous description of the layers, the flow of application data in connection with the layers is described next.
Subject to the step-by-step walk-through is a view on the flow of data within a complete example, where application data is sent from client to the server which affects the server response as shown in Figure 4.2.

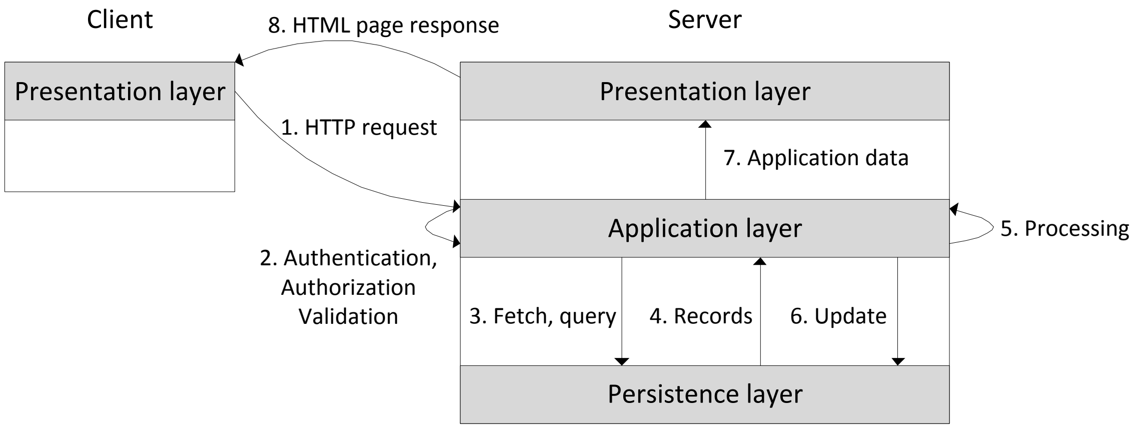

Figure 4.2: Interaction flow between the layers

Each user interaction with a transition from one application state to the next requires the sending of a request to the server which has a page with the new state as the response. As the application state fully resides on the server, a round-trip is necessary. A single state transition is the result from a method invocation by the client.

1. HTTP request. Initially, the client starts with requesting a Web resource on the server through a HTTP request by the client browser. A first request could be the initial page. Subsequent requests could carry user input from form fields, which are part of the presentation layer. At the server-side application layer these Web resource requests usually result in application layer methods to be executed.

2. Authentication, authorization, validation. The server-side application layer needs to authenticate the client if the requested resource has security constraints. It then needs to check whether the authenticated client is actually authorized for the method that is executed by the resource request. If there is input sent along the request, the server needs to validate that data.

3. Fetch, query. When the request is permitted and validated, in order to actually process the request the application layer likely needs to fetch records from the persistence layer that are subject to the method execution resulting from the request.

4. Records. The persistence layer returns resulting records upon a fetch request or query from the application layer.

5. Processing. When all data required by the request has been obtained from the persistence layer, the actual processing of the method takes place within the application

layer. During processing application data is likely modified resulting in a new state of the application.

6. Update. The application layer needs to store the newly modified data in the persistence layer. The new state is effectively permanently stored.

7. Page rendering. The application layer hands over relevant and updated application data to the presentation layer in order to compile the page that represents the result of the request originally invoked by the client. For that request, predefined templates with placeholders substituted by the application data can be compiled to a resulting HTML page.

8. HTML page response. The compiled page is returned to the client browser.

In a traditional web application this application flow or layer interaction is necessary for each interaction the client does. Most notably, a round-trip between client and server is required for each interaction.

Data transfer sizes and number of round-trips

In this section estimated quantitative results on the amount of data that is transferred during a typical user interaction with the application are given. In addition to that, the required round-trips are determined.

On each interaction the full page needs to be loaded. Let b be the size of the page HTML in bytes excluding the tasks, t_d the data size of one task, t_m the HTML markup to render a task, both in bytes. Let n be the number of tasks to be shown and k the number of interactions.

The total size of one page p in bytes is:

$$p = b + n(t_d + t_m) \tag{4.1}$$

The total amount of bytes s the client needs to receive from the server for all k interactions is:

$$s = kp \tag{4.2}$$

As each interaction requires a round-trip to the server, the number of round-trips l_{RT} is simply:

$$l_{RT} = k \tag{4.3}$$

Disadvantages of traditional web applications

Traditional web applications follow a server-centric and thin-client model. This means, that the client is dependent on the server in almost every aspect of the application except for displaying a single page as a single state of the application.

Therefore, traditional web applications have several disadvantages in respect to the requirements of social web applications:

Responsiveness. Because each interaction requires a round-trip to the server, results of a user action are not visible immediately but take at least one round-trip time. As an entire page is returned, the amount of bytes that are transferred is relatively high and does add to the overall response time. This has a particular impact on the responsiveness of mobile clients when bandwidth is usually small and latency is high.

Scalability. Due to the exclusively server-side execution of the application, scalability can be an issue as the server can become a bottleneck and performance can be low.

Availability. Moreover, when the server experiences an outage, the entire application can become unavailable.

Offline availability. Clients can not continue to use the application when they are offline or in case of a server outage. Thus the application is not providing offline availability.

Costs. The relatively huge data transfer sizes incur high traffic costs on the client- and server-side. The server-centric application execution also incurs high computational costs at the server-side.

As traditional web applications have several disadvantages, in practice, alternative approaches are commonly used. Alternative approaches are subject to the discussion in the following chapters.

4.2 Cloud service provider comparison

In order to draw conclusions on the cost-effectiveness of the solution presented in this work, this section gives a brief overview over key figures in cloud service pricing. Subject to the comparison are offerings by Amazon Web Services (AWS)[1], Google App Engine (GAE)[2] and Windows Azure[3]. The three vendors are reasonably the most relevant in the cloud service provider market. The pricing comparison is based on compute instance offerings, which are on a par with each other with respect to the computing performance. The costs of computation, bandwidth and storage for each type of instance are denoted in Table 4.1.

The costs of storage access are omitted within the comparison, as the figures depend either on the number of accesses or the time length of access with diverse pricing models among the providers and a high dependence on the particular application. The indicative storage access costs of a concrete example scenario, however, are given in Chapter 7.

Cost factor	**Amazon Web Services**	**Google App Engine**	**Windows Azure**
	EC2 Small Instance, Linux OS, 1.7 GB Memory, 1 CPU 1.2 GHz, 160 GB HDD	Front-end F2 Instance, 256 MB Memory, 1 CPU 1.2 GHz	Small Instance, Windows OS, 1.75 GB Memory, 1 CPU 1.6 GHz, 225 GB HDD
Computation	$ 0.08/hr	$ 0.16/hr	$ 0.12/hr
Bandwidth	$ 0.12/GB	$ 0.12/GB	$ 0.12/GB
Blob Storage	$ 0.125/GB, S3	$ 0.13/GB	$ 0.125/GB
Key-Value Storage	$ 0.25/GB, SimpleDB	$ 0.24/GB, Datastore	$ 0.125/GB

Table 4.1: Computing instance pricing comparison for Amazon Web Services, Google App Engine and Microsoft Azure.

According to the comparison, the costs for bandwidth and large binary storage (blob storage) are almost the same among the three particular instance offerings. While this also holds true for the key-value store of AWS and GAE, Windows Azure is offering a key-value store at approximately half the price.

However, there is a difference in the computational costs of the instances with Amazon having the cheapest offering and GAE being about double the price, while Microsoft Azure's computational costs are between the two.

The price difference can partly be explained by the different kind of service the providers offer. Amazon is an Infrastructure as a Service (IaaS) provider where a lot of responsibility including scalability and maintenance is left up to the developer. In contrast to this, GAE is a Platform as a Service (PaaS), where a lot of solutions to development issues including scalability and maintenance are already part of the platform. This does partly justify the higher instance computational costs with GAE. Moreover, the weaker performance characteristics of the instance with GAE are alleviated by the fact that the resources are exclusively available to the application. This means, the resources are exempt from powering an underlying operating system in contrast to AWS and Windows Azure, where the operating system consumes part of the available resources. Windows Azure is a self-described a hybrid of an IaaS and PaaS service which is also reflected by the computational costs which lie in between the other two offerings.

To conclude, the costs for computation, storage, and bandwidth among the three major vendors are basically on a par with each other. This allows for drawing more general conclusions on overall costs by the example cost calculation that is given within the Evaluation Chapter 7.

1 Amazon EC2 Pricing, http://aws.amazon.com/ec2/pricing accessed July 23, 2012
2 Google App Engine Pricing, http://cloud.google.com/pricing accessed July 23, 20122
3 Pricing Details: Windows Azure, http://www.windowsazure.com/en-us/pricing/details accessed July 23, 2012

5 Design of social web applications with Xydra

In this chapter the blueprint on how to design **S**ocial **w**eb **a**pplications based on **X**ydra (SWAX) is described. The chapter begins with an introductory overview on the architecture of SWAX applications.

5.1 Running example: Collaborative to-do list application

Throughout the remainder of this work, the design of SWAX applications is illustrated by means of a concrete example SWAX application. The application is a collaborative to-do list which allows the management of private tasks as well as the delegation of tasks to others. The basic features are as follows: First, new tasks can be created. Initially, a new task is marked as undone and can optionally be delegated. Second, tasks can be marked as done or be canceled. A delegated task can be accepted or rejected by the receiving user or it can be revoked by its creator. A delegated task can also be marked as done by its creator or marked as delivered by the receiver after he accepted the task.

Users can add comments to tasks. When appropriate, mails will be sent out to the users to inform them of important events regarding a task, such as a newly added comment or a change of a task's status.

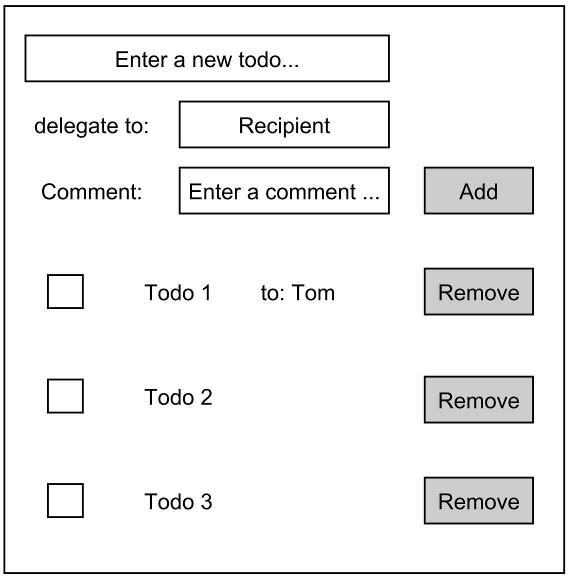

Figure 5.1: A conceptual illustration of the collaborative to-do list application.

5.2 Architecture overview

SWAX applications shift more functionality from server to the client. The client is now the primary component in the architecture which also holds application state as opposed to the traditional scenario described in Section 4.1. The server, however, merely serves as a data broker and persistence component with little business logic.

Like a traditional web application, SWAX applications can also be decomposed into a presentation, application, and persistence layer as shown in Figure 5.2. In contrast to the traditional architecture, the application layer as well as the persistence layer are now also placed on the client. The presentation layer is removed from the server and only present on the client.

In the following, the layers and their distinguished usage, as opposed to the traditional scenario, are described in more detail. First the client-side layers are explained followed by a description of the layers which are left on the server. Throughout the following sections the exemplary to-do list application is used to illustrate the design of a SWAX application.

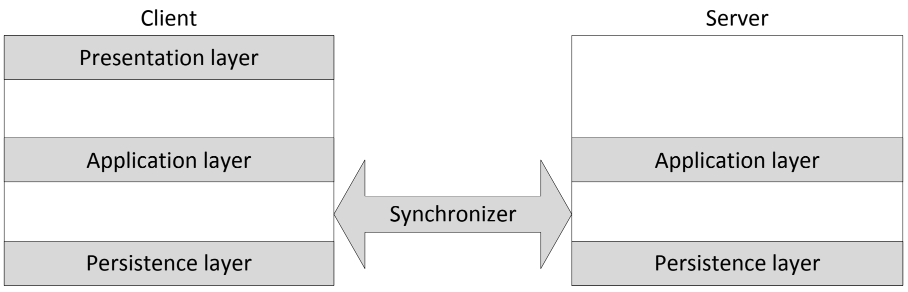

Figure 5.2: Visual overview of a SWAX application architecture

5.3 Application structure on the client

The client encompasses a presentation, application, and persistence layer to make the client largely independent from the server. This architectural difference towards the traditional scenario has the effect that the client now holds state across several user interactions. In the next paragraphs the entities of each layer are introduced followed by a discussion of key concepts of design, namely the state, separation of concerns, event propagation, and nesting.

Entities. SWAX applications use a different kind of entity set on the three layers that resemble the Xydra *XEntities*, *XModel*, *XObject*, and *XField*. The entities of one layer correspond to the entities in the other layers. Entities of an upper layer augment the entities of the underlying layer and help to separate concerns when designing SWAX applications. An overview of the layers and their entities is given in Table 5.1. The key principles of this concept are described next.

Layer	**Entities**		
Presentation layer	ModelWidget	ObjectWidget	FieldWidget
Application layer	AppModel	AppObject	AppField
Persistence layer	*XModel*	*XObject*	*XField*

Table 5.1: Overview of the layers and their entities, e.g. a ModelWidget has one to many ObjectWidgets. A ModelWidget is the visual representation of an AppModel.

State. As the client holds a local state within the application and persistence layer, the state will eventually diverge from the global server-side state until the state is synchronized. A state that is only known locally and not yet synchronized with the server is called a preliminary state. A state that has successfully been synchronized with the server is called persisted.

Part of the design of SWAX applications is to give the user feedback whether his local state is only preliminary or has already been permanently stored on the server. There can be entities that have been changed locally since the last synchronization and whose synchronization states therefore are preliminary. At the same time, entities which haven't been changed since the last synchronization keep the state persisted. However, an entity is considered changed and therefore has a preliminary state whenever any of its children has a preliminary state. This means, when an AppField is changed, the AppField, the parent AppObject and the parent AppModel are all set to a preliminary state at once.

Separation of concerns. Each layer has a specific purpose following a separation of concerns that is resembling concepts of the Model-view controller pattern [KP88, GHJV95]: the presentation layer maintains the view, the application layer is home to the business logic of the application and the persistence layer permanently stores the application state. Subject to each layer are entities that maintain an intra-layer and inter-layer relationship. The inter-layer relationship is realized by an event concept that is described next. This is followed by a description of the intra-layer relationship that is established through a nesting concept.

Event propagation. Xydra introduces an event concept. An *XEntity* can send out events when the *XEntity* changes. This event concept is used by the application layer in order to listen to changes within the persistence layer. Additionally, the event concept is also maintained between the presentation layer and application layer. Widgets of the presentation layer observe changes on the entities in the application layer. The details of this event concept are discussed in the following sections on each layer.

Nesting. Xydra also introduces a hierarchical nesting concept on its entities: An *XModel* has zero to many *XObjects*. An *XObject* has zero to many *XFields*. This nesting concept is helpful when designing applications based on Xydra. For this reason, the same nesting concept is maintained in the application layer (see Section 5.3.2) and presentation layer described in Section 5.3.3.

In the following the persistence, application, and presentation layer are described in respect to the key concepts of separation of concerns, event propagation, and nesting.

5.3.1 Persistence layer

The key purpose of the client-side persistence layer is to permanently store the application state.

The entities of the persistence layer are *XEntities*, namely *XModel*, *XObject*, and *XField*. As described in the Section 2.2 on Xydra, the nesting is as follows: an *XModel* has any number of *XObjects* as children. An *XObject* has any number of *XFields* as its children.

Events on these *XEntities* are propagated to the application layer as *XEvents*. It is the responsibility of the persistence layer to permanently store the *XEntities* across browser restarts. It utilizes the browser local storage capability for this purpose. In addition to that, it permanently stores any additional information that is necessary to rebuild the correct client-side runtime state of the application layer. Being able to store the application state and to restore the state after a browser restart are a main benefit of SWAX applications.

Apart from changes that are mirrored from the application layer and that need to be permanently stored, the persistence layer also needs to store changes originating from the server and that were fetched during synchronization (see Section 2.2 for a discussion of the synchronizer). The synchronizer fetches remote changes from the server that are yet unknown to the client. When these remote changes are fetched, they are applied within the persistence layer, *XEvents* are sent out which the application layer observes.

5.3.2 Application layer

The primary concern of the application layer is to provide the business logic of the application. The entities of the application layer are called AppEnties, namely AppModel, AppObject and AppField which correspond to the formerly described *XEntities* of the persistence layer. In accordance to the Xydra *XEntities*, the nesting is as follows: an AppModel has any number of AppObjects as children. An AppObject has any number of AppFields as its children. The AppEntities are described in more detail in the following paragraphs.

Events on these AppEntities are propagated to the presentation layer as AppEvents. The application layer observes *XEvents* on *XEntities* in the persistence layer, this includes synchronization state change events.

The application layer provides application methods that can be invoked by the presentation layer. In the to-do list example, the addition of a new to-do item is such an application method. It is the responsibility of the application layer to validate such a method invocation. This includes checking constraints on the method input parameters obtained by the view. For instance, a to-do item description may not be empty.

Upon successful validation, the application layer executes the invoked method. Subject to the method execution are the AppEntities.

Synchronization state

As described in the paragraph on state, Xydra *XEntities* can either be in a preliminary or persisted state. The kind of synchronization state is primarily a concern of the persistence layer, but it is also important to give the user visual feedback on the synchronization state within the presentation layer. To accomplish this, the application layer observes the *XEntities* and propagates synchronization state change events to the presentation layer.

In the following, the AppEntities are described in more detail.

AppField

An AppField can either represent the state of an *XField* or be stand-alone. An AppField is a single attribute. It defines the type of value it represents. For instance, the description of the to-do item is an AppField which has a string type and the done flag is of type boolean. AppFields can be children of AppObjects. Together they can be used to specify the schema of the application data. When data is validated for security reasons, it is validated against the specified schema. Whenever an AppField's value changes, an AppEvent is sent out.

AppFields can be stand-alone, which means they are not representing a single *XField* but rather represent values that are computed based on a function at runtime. This function may depend on the value of multiple other *XFields* based AppFields or stand-alone AppFields. For instance, the number of to-do items that are still open is a stand-alone AppField whose value is a function that depends on all currently present to-do items, in particular each item's done flag. As such, computed AppFields can observe other AppFields for changes and recompute or update their value accordingly.

AppObject An AppObject represents the state of an *XObject* and has AppFields as children. Depending on the application, an AppObject has a defined structure with a fixed set of AppFields. For instance, a to-do item is an AppObject of a fixed structure that has a description and a flag whether it is done or open. When creating AppObjects, the structural invariant, which is once defined, is validated.

AppModel The AppModel represents the state of an *XModel* and holds a collection of AppObjects. AppObjects can be added or removed and corresponding events, called AppEvents, are sent out. Logically, an AppModel is at the root of the application data model hierarchy and has AppObjects as children.

AppEvents The purpose of AppEvents is to notify the presentation layer of changes on AppEntities. The type of AppEvents are: 1. *Add* 2. *Remove* 3. *ValueChange* 4. *SyncState-Change*

AppEntity	**AppEvent**
AppModel	Add, Remove, SyncStateChange
AppObject	Add, Remove, SyncStateChange
AppField	ValueChange, SyncStateChange

Table 5.2: AppEntities and the AppEvents they can sent out

When an application method modifies any AppEntities at the application layer, these changes are mirrored at the persistence layer on the *XEntities* the AppEntities are representing. The application layer also observes *XEvents* on any changes on these *XEntities* that happen within the persistence layer during synchronization.

5.3.3 Presentation layer

The purpose of the presentation layer is to visually reflect the current client-side state of the application and to provide interface components the user can interact with.

The entities of the presentation layer are the ModelWidget, ObjectWidgets and Field-Widgets which correspond to the formerly described AppEntities of the application layer. In accordance to the AppEntities, the nesting is as follows: an ModelWidget has any number of ObjectWidget as children. An ObjectWidget has any number of FieldWidgets as its children. The nesting concept can also be seen as a composition of UI components, the widgets. Similar concepts of UI integration are described in [DYB+07].

The presentation layer has to goals, for one it observes AppEvents on the AppEntities in the application layer to visually reflect them. Second, it propagates user interaction on the view components to the application layer.

The interaction can invoke a client-side application method which can result in a state transition. Unlike the static and simple presentation layer in the traditional scenario, the presentation layer now provides broader functionality with extensive use of JavaScript. It dynamically adapts to state transitions happening on the client and most importantly without a round-trip to the server. The presentation layer mainly consists of the view which is bound to the underlying application data model. The data model defines the current state of the application and is part of the application layer which was described previously. The view updates automatically when the data model changes, this is referred to as model-view binding as realized by observing AppEvents. The view update process usually transforms the application data model into a visual representation which can include reformatting of the data. For this purpose, view templates containing placeholders for application data can be defined. The placeholders are substituted with the actual visual representation of the data on each update during the rendering process.

In the example to-do list view shown in Figure 5.3, the user can add a new to-do item or remove it. Initially, there is no to-do item, so the view shows an empty to-do list. The view provides a text input for the to-do item description and a button component for the user to add a new to-do item. Upon clicking the button, the view invokes the appropriate application layer method to add a new to-do item with the given description. The addition of the to-do item leads to a new application state as the application data model now contains the to-do item. The view is bound to these state changes and therefore updates its to-do list which now includes the visual representation of the to-do item.

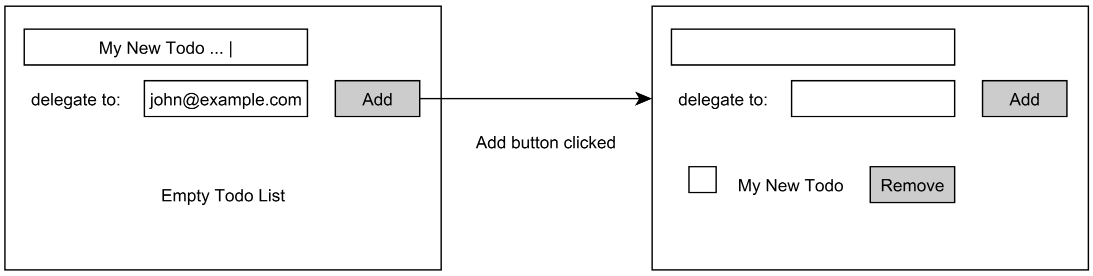

Figure 5.3: Example to-do list view. A new to-do item can be entered into the input component and added through the add button. The new application state with the added to-do item is reflected in the updated view.

Entities of the presentation layer. In the following, the presentation layer widgets are described. Figure 5.4 shows how these widgets are used.

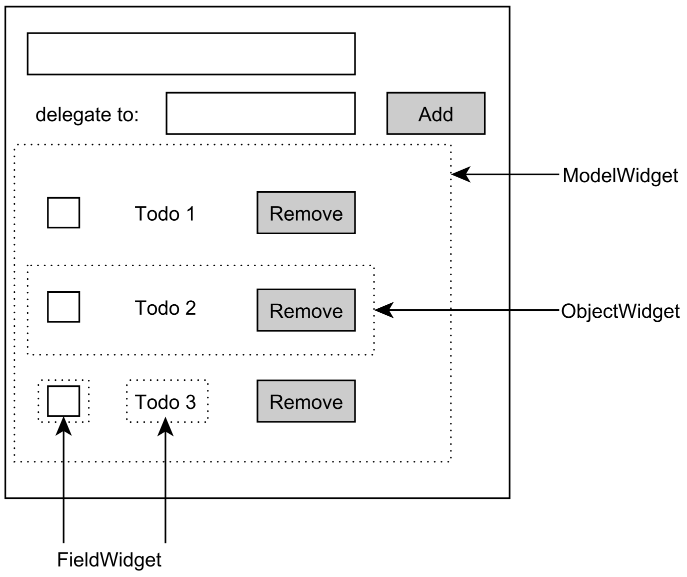

Figure 5.4: The use of widgets in the example Todo List View.

FieldWidget A FieldWidget visually represents an AppField. In the to-do list example, one FieldWidget is the to-do item's checkbox while another is its description. A FieldWidget knows how to translate an AppField property into a visual representation. For instance a boolean AppField value that marks whether a to-do item is open or is done is represented as a checkbox while the description string AppField is represented as a text node. In addition to translating AppField types to corresponding visual components, FieldWidgets can also format the output of an AppField. For instance, a FieldWidget can format a date

AppField relative to the local timezone.

Usually FieldWidgets are logically children of ObjectWidgets, but they can also be logically parent-less and be used as stand-alone representations of computed AppFields. For instance, a stand-alone FieldWidget could display the number of to-do items that are still open.

FieldWidgets can observe ValueChange and SyncStateChange AppEvents. When an AppEvent is observed, the widget updates its visual representation accordingly.

ObjectWidget An ObjectWidget is the visual representation of an application level AppObject. Logically, the parent of an ObjectWidget is a ModelWidget and it has FieldWidgets as children. The ObjectWidget as being the container to FieldWidgets determines how its child FieldWidgets are visually arranged.

ObjectWidgets can observe Add, Remove, and SyncStateChange AppEvents. When an AppEvent is observed, the widget updates its visual representation, it adds or removes FieldWidgets or updates its sync state respectively.

ModelWidget A ModelWidget is a collection of ObjectWidgets. In the to-do list example, the list of to-do items is a ModelWidget and each to-do item is an ObjectWidget. A ModelWidget knows how to display a list of ObjectWidgets. A ModelWidget is bound to an application level AppModel. Whenever the AppModel changes, the ModelWidget updates accordingly. The changes the ModelWidget observes are the addition and deletion of items within the AppModel. On addition of an item, the ModelWidget adds a new ObjectWidget to its collection and on deletion, it removes it.

ModelWidgets can observe Add, Remove, and SyncStateChange AppEvents. When an AppEvent is observed, the widget updates its visual representation, it adds or removes ObjectWidgets or updates its sync state respectively.

Each type of widget can observe a synchronization state and will be notified by the application layer on synchronization state changes. For instance, a to-do item that has been added locally is represented by a Xydra *XEntity*. As long as the entity which represents that to-do item has a preliminary synchronization state, the visual representation of the to-do item is grayed out in the view until its entity has a persisted state.

5.4 Application structure on the server

The server encompasses an application and persistence layer and has no presentation layer.

5.4.1 Application layer

The server-side application layer complements the client-side application layer. When the client synchronizes with the server, the changes that happened on the client are re-executed on the server. As the changes are the result of a client-side method invocation within the client application layer, the server invokes similar methods in its application layer. The concrete implementation of that method, however, can differ. For instance, a method to add a to-do item can result in an email being sent, which can only be done on the server. The server execution of the method may also have other side effects, for instance, the to-do application could be extended to allow the delegation of to-do items to someone else. In this case, the server would modify the other user's application state on behalf of the client.

Subject to the server-side method invocation are AppEntities that can be modified. Like the AppEntities on the client-side, these AppEntities are based on *XEntities* such that changes on them are mirrored onto *XEntities* within the persistence layer.

Security. When the server receives changes from the client he cannot trust the data, as a malicious client could send arbitrary data, in particular data that could compromise the server. The data exchange is the responsibility of the synchronizer. Checking that the data meets security constraints is the responsibility of the server-side application layer. The application layer needs to perform the following security checks:

1. The application layer endpoint of the synchronizer must authenticate the client.
2. The changes must match the Xydra data structure, i.e. they are a set of *XCommands*.
3. The semantics of the changes must be interpreted, i.e. which method should be invoked with which parameter values. This is described in more detail in Section 5.5 on remote method invocations.
4. The parameter values need to be validated to conform to the application data schema, e.g. validating the type of a parameter value as well as its content in respect to its syntax and semantics.
5. The application layer must check if the client is authorized to invoke the particular method with the given parameter values.

5.4.2 Persistence layer

The server-side persistence layer stores *XEntities* that are are represented by AppEntities and were modified by the application layer. It uses the key-value store that is provided by Google App Engine for storage.

5.5 Remote method invocation and synchronization

While the synchronizer synchronizes changes, it is also used to invoke remote methods on the server. Traditionally, when the client wants to invoke a remote method on the server it can typically issue a HTTP request on a webservice offered by the server. For instance, one such method could be a new user sign up and the sending of an email on behalf of the client as part of an application workflow. The business logic to create a new user will likely reside on the server-side. Sending an email also requires a server-side component.

Given the Xydra application structure, however , the server does not offer any additional web service besides the synchronizer endpoint. In fact, the semantics of a method invocation are encapsulated within changes the synchronizer will sent to the server. This is realized by establishing a simple protocol between the client and server which is using the synchronizer as the transport: Changes at the persistence layer that result from a single application layer method invocation at the client browser are grouped within a transaction. One transaction with its included commands therefore represents a single method invocation. The synchronizer then sends these transactions to the server. Now, at the server, each transaction can be interpreted as a single semantically coherent unit that represent a method invocation on the client. The server inspects each transaction at synchronizer endpoint and derives the corresponding application level method from it. For instance, a Xydra *Add* command of an *XObject* with additional commands that add and set *XField* values within one transaction could be translated to an application level method, e.g. a method for creating a new instance of an todo-item with given parameters. When the method is determined and it requires parameters to be passed on invocation, these parameter values need to either be derivable from commands within the transaction or already present on the server. In the former case, a lightweight method helper routine can fetch the method parameter values from the transaction: For instance, it could look for a command which adds an *XField XValue* that holds the description of the to-do item to be created. When all parameters can be extracted, the method invocation is passed to the server-side application layer for invocation.

In essence, the server-side execution of the method will generate the same changes to the application data as the client already generated locally, provided the method invocation was successful. There may be additional side effects on the overall application state that should not be visible to client, e.g. administrative maintenance flags. This private data will be stored outside of the scope of the client data model. The synchronizer will finally synchronize the new state with the client.

The client can choose when he wants to synchronize his state with the server. This allows for aggregating several method invocations and saves on dispensable round-trips to the server. Furthermore, as synchronization can be delayed at will, the client can continue to execute its application even though the client may be offline. For efficient synchronization, an application developer can decide on a heuristic method on when the client should synchronize, e.g. at a fixed time interval or after a certain number of method invocations.

5.6 Interaction control flow

Following the previous description of the layers, the flow of application data in connection with the layers is described next in a similar manner as the previously described walk-through of a traditional web application in Section 4.1.

As depicted in Figure 5.5, the client and server communicate via the synchronizer. The synchronization takes place asynchronously to the rest of the interaction control flow.

1. Sending *XCommands*. When the synchronization starts on the client, the synchronizer sends recently applied local *XCommands* to the server.

1. Receiving *XCommands*. After the server processed the *XCommands*, the resulting *XEvents* are returned.

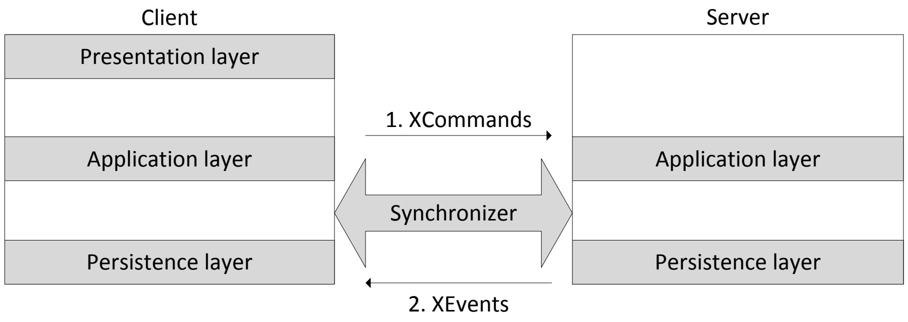

Figure 5.5: SWAX application interaction flow between client and server

The *XCommands* that are transported by the synchronizer result from user interaction with the application on the client. The interaction control flow on the client is depicted in Figure 5.6 and is described as follows:

1. Method invocation. User interface components within the presentation layer allow the user to specify input and to invoke application layer methods, e.g. by clicking a button.

2. Validation. The user input that is part of a method invocation is validated within the application layer.

3. Processing. After the input is validated, the invoked application layer method is executed as a *XTransaction* that is grouping *XCommands*. By grouping the changes in a transaction, the method invocation semantics are preserved for later synchronization. This processing step modifies the local state,e.g. the AppEntities and their corresponding *XEntities*, of the application:

4. *XCommands*. During execution of the method, the *XTransaction* is applied on the *XModel* within the persistence layer.

5. *XEvents*. The persistence layer returns resulting *XEvents*.

6. AppEvents. The application layer receives the *XEvents* and transform them to AppEvents. The presentation layer is notified of these AppEvents and updates its view accordingly. The result of the method invocation is now visible and the method invocation semantics are stored in the persistence layer.

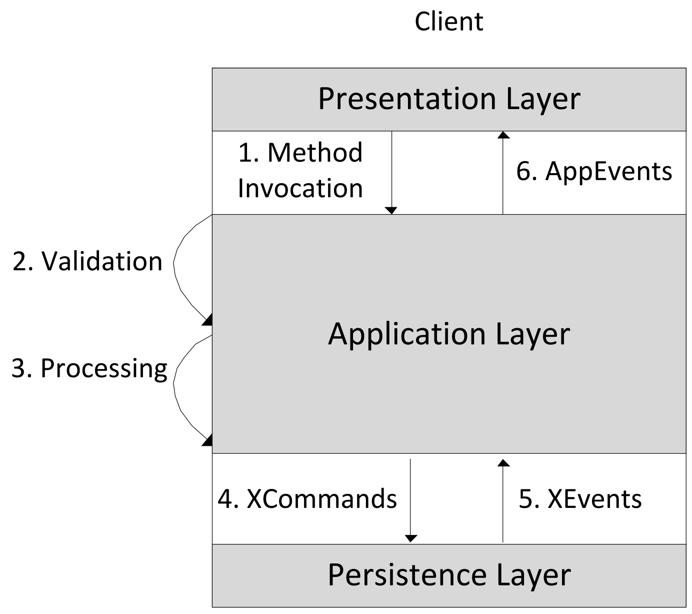

Figure 5.6: SWAX application interaction control flow on the client

The synchronizer asynchronously synchronizes the state changes between the client and server. It sends the client-side *XTransactions* that were the result of method invocations to the server. The interaction control flow on the server is depicted in Figure 5.7:

1. Method invocation. The server interprets the *XCommands* that are grouped to *XTransactions* and determines the application layer method that needs to be invoked on the *XModel*. There can be multiple *XTransactions* being received and as described in Section 5.5 each *XTransaction* corresponds to one application layer method. The methods are invoked one after the other. In this case, only a single *XTransaction* and method invocation is assumed. The server checks if the client is authorized to invoke each method.

2. Validation. The input contained in the *XCommands* is validated in respect to the method that should be invoked for security reasons .

3. Processing. After successful validation, the application layer method is executed. This step modifies the the server-side state of the *XModel*. The server-side method execution can have additional side effects such as sending an email.

4. *XCommands*. During execution, the *XCommands* are applied on the *XModel* within the persistence layer.

5. *XEvents*. The persistence layer returns resulting *XEvents*. The *XEvents* are then returned to the synchronizer for delivery to the client.

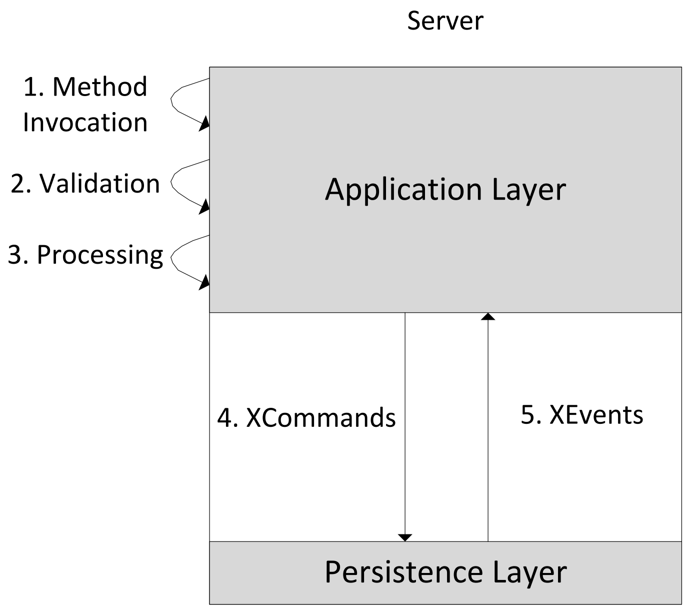

Figure 5.7: SWAX application interaction control flow on the server

6 Implementation

This chapter describes the modeling of the blueprint for SWAX applications. It then presents two prototype implementations of the to-do list application described in Section 5.1.

6.1 Blueprint for SWAX applications

Figure 6.1 depicts the UML class diagram that illustrates the SWAX design with the relationships of the classes and methods. The classes and their arrangement resemble the key concepts of the application structure presented in Section 5.3.

Regarding the concept of separation of concerns, each of the three entities of the different layers are arranged in three separate columns within Figure 6.1. For instance, the *XEntities* of the persistence layer, namely *XModel*, *XObject*, and *XField* are vertically aligned in one column. The next column contains the AppEntities of the application layer: the AppModel, AppObject, and AppField. The widgets of the presentation layer, namely ModelWidget, ObjectWidget, and FieldWidget are arranged in the rightmost column.

Within one column, the intra-layer relationship of entities, introduced as the nesting concept, is illustrated. Regarding the nesting concept, the classes representing these entities have composition relationships. Within the diagram the nesting concept is illustrated as composition relationships that are drawn vertically in each column connecting the entities. For example, an *XModel* has a composition relationship to *XObjects*. The multiplicity of the composition is that one *XModel* contains zero or more *XObjects*. In addition to that, as another composition illustrates, an *XObject* contains zero or more *XFields*, respectively. Other corresponding composition relationships can be observed in the application layer entities: between the AppModel and AppObject, or the AppObject and AppField. Finally, the multiplicity of the composition relationships for the presentation layer entities is: one ModelWidget to zero or more ObjectWidgets, and one ObjectWidget having zero or more FieldWidgets.

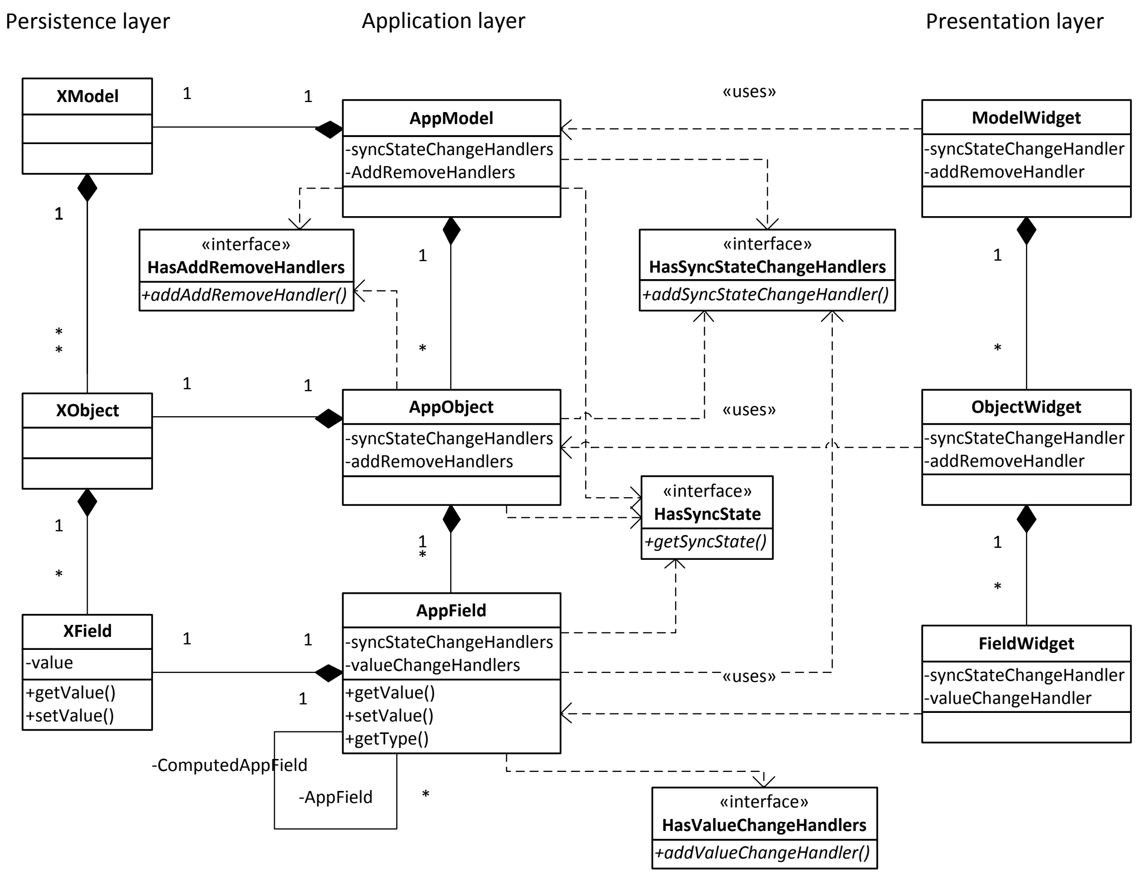

Figure 6.1: The UML class diagram illustrating the SWAX design.

In addition to the arrangements to columns representing the layers, the rows illustrate the event propagation that represents the inter-layer relationship of entities. For instance, an AppModel maintains a one-to-one composition relationship to an *XModel*. An AppObject has a one-to-one composition relationship to an *XObject* and an AppField a one-to-one relationship to an *XField*. The composition relationship is used as there is a strong dependency between the entities of the application layer and their respective entities in the persistence layer. In contrast, the entities of the presentation layer are only loosely coupled with their corresponding application layer entities as the widgets solely listen for AppEvents on the AppEntities. Therefore, the relationship is illustrated as dependencies instead of compositions. For instance, the ModelWidget uses the AppModel, an ObjectWidget uses an AppObject, and an FieldWidget uses an AppField.

For event propagation between the application layer and presentation layer to work, event handlers and corresponding interfaces have been implemented. The ModelWidget and ObjectWidget both have a syncStateChangeHandler that is notified of a SyncStateChange AppEvent. Furthermore, they have handlers for the Add and Remove AppEvent. The

FieldWidget also has a syncStateChangeHandler but has no addRemoveHandler. Instead, as FieldWidgets represent AppFields, which hold values, they have a valueChangeHandler and are notified of ValueChange AppEvents. In order to indicate that the application layer entities notify the widgets for the particular AppEvents, the HasAddRemoveHandlers, HasSyncStateChangeHandlers and HasValueChangeHandlers interfaces have been defined, which are implemented by the AppModel, AppObject and AppField. Furthermore, the HasSyncState interface indicates, that an entity has a synchronization state. The HasSyncStateChangeHandlers as well as the HasSyncState interface are implemented by all three AppEntities. HasAddRemoveHandlers is implemented by the AppModel and AppObject, and HasValueChangeHandlers is only implemented by an AppField.

Finally, an AppField can represent an *XField*, holds a value of a certain type and offers methods to get and set this value. As illustrated in the UML diagram, AppFields can have a one-to-many association relationship to other AppFields. This is how ComputedAppFields are realized, which are AppField whose value is computed at runtime and is depended on the value of other AppFields.

6.2 Prototypes

The first prototype called one-way synchronization is a commercial to-do list application that uses the Xydra data model and its versioning capabilities for the data exchange between client and server. The second prototype called Swaxodo is an experimental implementation of the to-do list application similar to one-way synchronization but that fully incorporates the SWAX applications design. As such, it represents itself as a type of Rich Internet Applications (RIA) [RECC^{+}12]. Both prototypes are implementations of the collaborative to-do list application described in Section 5.1. They are written in the Java programming language and the client-side component is compiled as a JavaScript web application using the Google Web Toolkit. The server-side component is a Java application that uses Xydra and is running on top of Google App Engine.

6.2.1 One-way synchronization

The one-way synchronization is a commercial application that implements the collaborative to-do list with a partial implementation of the SWAX application design. The conceptual interaction flow is depicted in Figure 6.2. The application uses a client-side Xydra *XModel* for application data objects. The key difference to the Swaxodo implementation is that synchronization takes place only in the direction from the server to the client. There are two variants of the one-way synchronization prototype for further evaluation. The difference is that one variant stores the Xydra *XModel* in local storage and the other variant does not leverage the local storage. The variant without local storage support is technically representative of commonly used web applications utilizing asynchronous requests (AJAX) [Pau05]. The application flow is as follows:

1. When the user interacts with the application, a request is sent to the server. Along the request, the client submits its highest known revision number of the *XModel*.
2. When processing the request at the application layer, *XCommands* are issued on the *XModel* which increases its revision number. The formerly received client revision number of the *XModel* enables the server to determine the delta changes, i.e. the *XEvents* that happened from the highest known client revision number up to the latest server-side revision number.
3. The *XCommands* result in *XEvents* on the server-side *XModel*.
4. These resulting *XEvents* are returned to the client.
5. The client updates its local *XModel* with the returned *XEvents*. In the local storage variant of the prototype it stores the updated *XModel* in local storage.
6. The client finally displays the new to-do list that the *XModel* represents.

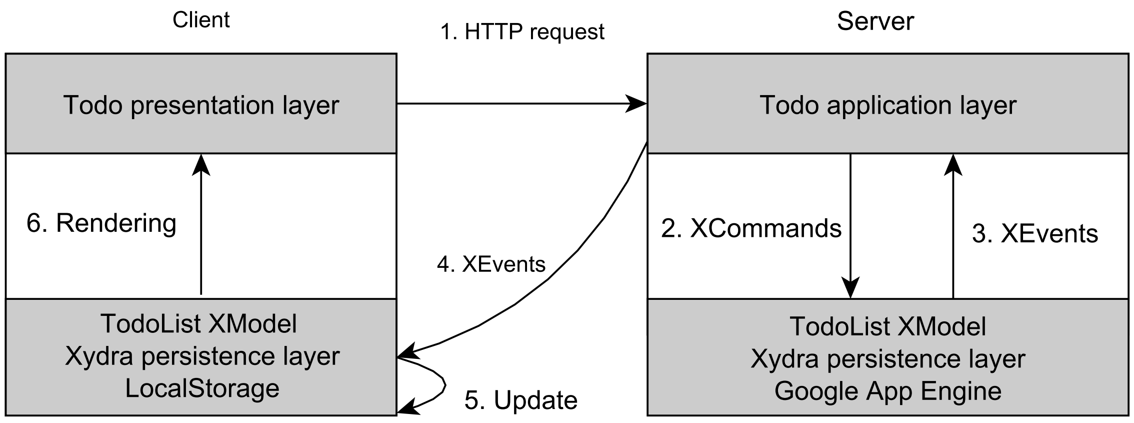

Figure 6.2: Interaction flow of the one-way synchronization application

6.2.2 Two-way synchronization with Swaxodo

Swaxodo is an experimental prototype implementation of the collaborative to-do list built as a SWAX application which serves as a full realization of the SWAX design concept. The key difference to the one-way synchronization implementation is that Swaxodo is a complete realization of the SWAX design concept, e.g. it provides a client-side presentation, application, and persistence layer. Furthermore, it fully utilizes the Xydra synchronizer. This means, that in contrast to the one-way synchronization application that only received *XEvents* from the server and sent application level request in the opposite direction from the client to the server, Swaxodo is a two-way synchronization application. In the following, a typical application flow is described:

1. When the user interacts with the application, the application level methods the user invokes are no longer sent to the server as requests but they are rather locally processed within the client-side application layer.

2. When locally processing the user interaction at the application layer, the application layer method validates the input and issues corresponding *XCommands* on the *XModel*. These *XCommands* are grouped into transactions that are corresponding to application level methods. The issued *XCommands* result in a higher revision number of the *XModel*. Each applied *XCommand* results in a *XEvent* that the application layer is notified of. The changed entities have a preliminary synchronization state and the *XModel* is persisted to local storage.

3. The result of the processing of the user interaction is immediately shown in the application user interface.

4. Asynchronously, at a fixed time interval or after a number of interactions, the synchronizer sends the local changes that were grouped into transactions to the server.

5. The server authorizes the client and interprets as well as validates the transactions. The client changes are merged with the server-side state of the *XModel*. Conflicting *XCommands* are detected. Further server-side *XCommands* are can be applied to the *XModel*.

6. The *XCommands* result in *XEvents* being returned by the synchronizer from the server-side state of the *XModel*.

7. The client updates its local *XModel* with the returned *XEvents* and stores the *XModel* in local storage.

8. The application layer is notified of the returned *XEvents*. The formerly preliminary local changes now have a persisted synchronization state.

9. The persisted synchronization state is immediately shown in the application user interface. Further received changes from the server may result in additional components to be shown.

As a summary, the application level methods the user invokes are no longer sent to the server as requests but they are rather locally processed within the client-side application layer resulting in changes within the client-side persistence layer. The synchronizer then takes these local changes grouped to transactions and sends them to the server as *XCommands*. The server executes the *XCommands* and returns resulting *XEvents*. The direction from the server to the client is similar to the one-way synchronization implementation. The interaction flow of Swaxodo is a direct implementation of the control flow illustrated in Section 5.6.

7 Evaluation and Related Work

In this chapter the design of social web applications based on Xydra (SWAX) is evaluated. Given each of the requirements of social web applications described in Chapter 3, this chapter discusses to what extent SWAX applications meet the requirements. Subject to this discussion are the two prototype implementations introduced in Section 6.2. The chapter continues with cost estimations of SWAX applications in comparison with other approaches. The chapter concludes with a review of other frameworks comparable with SWAX and a feature comparison between them.

7.1 Comparison with requirements

This section discusses to what extent SWAX applications meet each of the requirements described in Chapter 3.

Scalability

SWAX applications are primarily running on the client with the server merely functioning as a data broker and persistence layer. This is comparable to the deployment and scalability characteristics of an installable desktop application. Therefore, the client-side component is negligible in terms of scalability, because each user of the application runs the application locally and provides his own computational resources.

However, scalability of the server-side component of the SWAX application is of importance, as it is largely affected by the number of users and application items. In particular sharing items between users in social web applications is a critical component, as the number of possible user-to-user relationships, i.e. those relationships where items are shared, grows in $O(n^2)$ with n being the number of users. In addition to that, the popularity of single items or users often times follows a power law distribution so the system must be able to handle load peaks and bursts in distribution of items.

The server-side component of SWAX applications is based on Xydra utilizing the Google App Engine. Google App Engine allows the dynamic addition of computational resources, called instances, when the load of the application increases. In addition to that, it provides a scalable datastore. Xydra allows applications to utilize the dynamic instances and scalable datastore feature and provides functionality for concurrent execution of application instances. Therefore, it is possible to write scalable Xydra applications on top of Google App Engine.

As a result, SWAX applications provide scalability in respect to the number of application users and data items and meet the requirement.

Consistency

The Xydra synchronizer is capable of providing eventual consistency between client and server as described in Section 2.3. The server thereby has the role of the master that can detect conflicting commands/updates submitted by the client and reject them. The final state on the server is propagated to the client. Therefore, SWAX applications meet the consistency requirement.

Availability

As per design, SWAX applications can be used even when the user is experiencing intermittent connectivity, i.e. is offline, or in the case of server outages. The presence of a presentation, application, and persistence layer render it possible that the client-side application can be used independently of external factors such as server outages or loss of connection to the server. In literature, this is known as optimistic replication [GH96] and has been demonstrated by [SKK$^+$90].

SWAX applications can even be used across browser restarts. Changes to the application state which have been applied locally are persisted by SWAX applications within the browser's local storage. A similar approach has been demonstrated by [Can10].

As soon as the server can be reached again by the client, the changes are synchronized with the server. However, the availability of application functionality sometimes is restricted or limited. For instance, features that inherently require a server-side execution such as sending an email can only be queued locally for later execution. Nonetheless, principle functionality can be provided and the user can continue to use the application while the progress of application state is persisted locally.

Therefore, SWAX applications meet the availability requirement.

Synchronization, conflict detection and resolution

SWAX applications use the Xydra synchronizer in order to synchronize the client-side state with the server-side state. In this process, Xydra can detect client changes that are in conflict and are rejected on the server. Therefore, client changes can either be successfully applied or can fail due to a conflict. Conflict detection happens on the server and is propagated to the client. Client-side application code can include application specific conflict resolution routines.

SWAX applications can use synchronization callbacks in order to provide a client-side conflict resolution. Concrete conflict resolution techniques are not part of the SWAX design as they are often application specific. Yet, Xydra enables custom resolution techniques to be adapted by the application developer.

As a summary, SWAX applications meet the synchronization requirement. Conflicts are detected at the server-side. Conflict resolution techniques, however, are unadressed, but common resolution techniques are compatible with SWAX.

Responsiveness

Due to the client-side execution of the business logic, the results of user interactions can preliminarily be determined locally. Therefore, no server round-trip is necessary in order to complete an action and to show its preliminary results. Consequently, the user perceives the application as being executed locally with no latency. Effectively, responsiveness is good and is mostly limited by browser execution performance. The savings in round-trips are discussed in Section 7.2.

As a result, SWAX applications meet the responsiveness requirement.

Cost-effectiveness

Cost-effectiveness in regards to the initial development of SWAX applications is rendered possible due to the assistance in development by powerful tools like the Eclipse IDE and the Google Web Toolkit. Java has an advantage over JavaScript in development as it is a statically typed language.

SWAX applications allow the reuse of application layer code on both client and server which can lower development time and costs. Furthermore, as SWAX applications are regular web applications the application as well as new versions can easily be deployed whenever the client requests the web application.

SWAX applications incur costs in terms of computational resources, network usage, and storage. The client-side part of SWAX applications saves costs for computational resources on the side of the application provider. As the application and presentation layer are present on the client, the server is unburdened from the computational cost-incurring charge of that part. Still, a part of the application layer and also the persistence layer remain on the server and incur costs on computational resources and storage. The savings on computational costs are part of a cost estimation in Section 7.2.3.

SWAX applications help in minimizing traffic costs as the synchronizer persists application data locally and only submits and receives changes and not the entire application data. Furthermore, the changes do not include any presentational markup which also helps in minimizing traffic. The network usage of SWAX applications is discussed in more detail in Section 7.2.

In summary, SWAX applications provide cost-effectiveness in regards to development, maintenance and deployment of applications. The SWAX design allows economic usage of computational, storage, and network resources. The cost-effectiveness requirement is fulfilled.

Device independence

The Google Web Toolkit allows to compile SWAX applications to packaged JavaScript applications which are compatible with most modern Web browser environments including mobile devices. SWAX applications can detect supported features like local storage support and adapt to the availability of these features.

Consequently, the usage of the Google Web Toolkit allows SWAX applications to be device independent.

Data versioning and history

As discussed in Section 2.2, Xydra supports versioning of its data model and keeps a changelog with increasing version numbers as a history on each entity.

SWAX applications can use Xydra support for versioning and history, as such the requirement is fulfilled.

Security

The security of SWAX applications is primarily a concern of the server-side application layer as the synchronizer receives client changes as input that cannot be trusted. A key benefit of a server-side application layer is that the business logic is running in a privileged environment that can be trusted as opposed to the client. The application layer provides functionality to check the structure and interpret the application level method the client changes do represent. After determining the method and input parameters, the application layer can check the access control rights of the client. The initial authentication of the client is orthogonal to the design of SWAX applications. However, common methods for authentication, such as HTTP authentication, are compatible to SWAX.

Consequently, SWAX applications support security for authorization and validation at the server-side. Authentication of the client is not part of the design, but common techniques are SWAX compatible. Therefore SWAX applications meet the security requirement.

User interface and model-view binding

User interfaces in SWAX applications can be built using widgets of the presentation layer (see 5.3.3). The widgets can be bound to application layer data models. This view binding allows the user interface to instantly reflect changes made to the application data model. These changes can either be in a preliminary or persistent synchronization state and the widgets can give visual feedback on the state.

Therefore, SWAX applications provide widgets for the user interface that support model-view binding. The synchronization state of the application data is represented by the widgets. The user interface and model-view binding requirement is fulfilled.

Summary

SWAX applications meet the requirements as described in Chapter 3 to a large degree. SWAX applications are scalable and provide availability during loss of connection or server outages. The client-side execution makes SWAX applications responsive. Furthermore, applications can be developed, operated and maintained in a cost-effective way. In particular, SWAX minimizes traffic and computational costs on behalf of the application provider. SWAX provides widgets for the user interface including model-view binding. Provided by the Google Web Toolkit, SWAX applications are device independent and run in any modern browser environment.

Provided by Xydra, SWAX also supports versioning and history on application data. SWAX allows the authorization and validation of changes during synchronization and conflicts can be detected.

Authentication and conflict resolution, however, are unaddressed in the SWAX design. Yet, compatible techniques to these features can be implemented by an application developer.

In relation to the disadvantages of the traditional approach as discussed in Section 4.1, the important benefits of SWAX applications are the better responsiveness, higher scalability, availability, and the decreased costs. A more detailed evaluation of the reduction in round-trips and data transfer sizes and the resulting responsiveness as well as decreased costs is subject to the next section.

7.2 Cost estimations

According to the analysis of network usage and round-trips of a traditional web application in Section 4.1, this section analyzes the one-way synchronization application (see 6.2.1), a partly realization of the SWAX design, and Swaxodo (see 6.2.2), a full realization of the SWAX design, in a similar manner.

7.2.1 One-way synchronization

As an improvement to the traditional web application, in the one-way synchronization application the page is only requested once and does only contain the framing presentational markup without the actual application data. There are two variants of the one-way synchronization approach: One variant leverages the local storage capability of the client whereas the second variant does not utilize local storage. The latter variant is representative of web applications commonly used today.

Right after the initial page load the application data is received as a Xydra *XModel* from the server via an asynchronous JavaScript request (AJAX). The initial response contains the full state of the application data, e.g. the entire to-do list, leaving out the presentational markup. Each subsequent response only contains the data that is relevant to the request, e.g. a single task.

The presentation layer resides on the client and no longer on the server. The presentation layer uses the application data and renders a corresponding view from it. This way the page is dynamically adapted on each interaction.

According to the key difference to the traditional scenario, the page markup is only loaded once and each subsequent interaction takes places through asynchronous JavaScript requests. Each JavaScript request only returns the task data t_d, leaving out the markup to render a task t_m. The task markup is only transferred once at the initial page load.

The variant with local storage support does only need to request the entire to-do list once as the to-do list thereafter is permanently stored in the local storage.

Therefore, the total size of the initial page p, assuming an empty local storage, in bytes is:

$$p = b + t_m + n * t_d \tag{7.1}$$

For the variant with local storage support and assuming that the to-do list is already present in local storage, the size of the initial page p^* is:

$$p^* = b + t_m \tag{7.2}$$

As such, the total amount of bytes for all k interactions is:

$$s = p + (k-1)t_d \tag{7.3}$$

As each interaction requires a round-trip, the number of round-trips for k interaction is:

$$i_{RT} = k \tag{7.4}$$

7.2.2 Swaxodo

The main difference of Swaxodo in comparison with the one-way synchronization approach is the offline availability of applications. As a result, dispensable round-trips to the server can be saved.

Initially, assuming an empty local storage the to-do list needs to be received once entirely. Like the one-way synchronization approach the bytes transferred can be stated as:

$$p = b + t_m + n * t_d \tag{7.5}$$

Assuming that the to-do list is already present in local storage, the size of the initial page p^* similarly to the one-way synchronization with local storage approach is:

$$p^* = b + t_m \tag{7.6}$$

And similarly, the total amount of bytes for all k interactions is:

$$s = p + (k-1)t_d \tag{7.7}$$

As Swaxodo permanently stores application data in local storage and is able to rebuild the application state after browser restarts, which essentially is part of the offline capability, no round-trip to the server is necessary but can optionally be performed to fetch new changes from the server. Therefore, no data transfer needs to take place after a browser restart. Typically though, after a browser restart the page according the formular p^* will be loaded.

Swaxodo applications do not require an immediate round-trip to the server in order to display results of an interaction, as the application is executing locally. Round-trips are only necessary when the client synchronizes its state with the server, which is happening asynchronously to the user interaction and can be delayed until after k interactions. The synchronizer can aggregate k changes made by the client or within a certain time interval, to avoid dispensable round-trips. The strategy that is used to synchronize aggregated interactions can be customized. In this case it is assumed that one synchronization takes place at beginning of a page load and a second synchronization after k interactions.

Therefore, the number of required round-trips for k interactions with delayed synchronization is simply one (optional) round-trip at the beginning of a session of k interactions and one round-trip at the end of a session.

$$j_{RT} = 2 \tag{7.8}$$

7.2.3 Example scenario and costs

This section presents an estimated calculation and comparison on basis of the previously discussed formulas among the three approaches. With the help of an example scenario, the calculations and estimations of data transfer sizes, round-trips, and service provider costs are determined. The estimated application parameters are derived from benchmarks conducted on real-world data from the one-way synchronization approach. Furthermore, the number of users and the average number of items per user is derived from the estimated figures on social web applications given in Table 3.1.

Session. Part of the following scenario is the notion of sessions. A session is a sequence of n consecutive interactions a user can make without closing the browser. For instance, a session has ten interactions when the user creates seven tasks, delegates one task to someone else and marks two of them as done in a row.

Scenario. The example scenario is based on the collaborative to-do list application (see 5.1). The application is assumed to be used by 1,000,000 users with 500 tasks per user. During a session, the user is presented a page with 50 tasks as a subset of the total number of tasks. The values of the parameters are as follows:

Parameter	Description	Size
m	#users	1.000.000
n_p	#tasks per page	50
n_{total}	average total #tasks per user	500
b	Page markup size	60,000 bytes
t_m	Task markup size	2,700 bytes
t_d	Tasks data size	300 bytes

Table 7.1: Parameters of the to-do list application scenario[1]

Data transfer sizes

Given these parameters, the figures are estimated according to four consecutive sessions (S_0, S_{10},S_{100} and S_{1000}) with increased numbers of interactions (0 - 1000 interactions respectively). Session S_0 is the initial page load with an empty local storage. Session S_{1000} with 1000 interactions is a theoretical example for illustration purposes and has little practical relevance. At the end of each session, a browser restart is assumed in order to illustrate the effect of the local storage usage.

Given the formerly discussed formulas, following figures on data transfer sizes, without utilizing compression, for the three approaches are determined.

Approach	Session [bytes]	S_0	S_{10}	S_{100}	S_{1000}
Traditional approach		210,000	2,100,000	21,000,000	210,000,000
One-way synchronization w/o local storage		77,700	80,700	107,700	377,700
One-way synchronization with local storage		77,700	65,700	92,700	362,700
Swaxodo		77,700	65,700	92,700	362,700

Table 7.2: Data transfer sizes of each session for a traditional web application, the one-way synchronization approach and Swaxodo.

The data transfer sizes for the traditional approach grow linearly in the amount of interactions. The one-way synchronization approach and Swaxodo have the same transfer size for the initial page. The one-way synchronization approach without local storage needs to reload the tasks at the beginning of each session, therefore the data transfer sizes for each session have a factor added. Most notably, one-way synchronization with local

1 The sizes of page and task markup as well as tasks data sizes were measured in benchmarks conducted on real-world data from the one-way synchronization approach.

storage and Swaxodo, which also leverages the local storage, do not need to request the entire todo-list after a browser restart. Therefore, savings in data transfer size for one-way synchronization with local storage and Swaxodo can be observed for the session S_{10} and the subsequent sessions. Session S_{10} is reasonably the most common kind of session a user will perform. For this particular scenario in session S_{10}, 18.6% in data transfer sizes are saved when comparing one-way synchronization with local storage and Swaxodo to the one-way synchronization approach without local storage.

Round-trips

Similar to the data transfer sizes, the number of round-trips for the three approaches is determined in the following table. In this scenario, Swaxodo will synchronize its application state with one round-trip for every ten interactions.

Approach	**# Round-trips**	S_0	S_{10}	S_{100}	S_{1000}
Traditional approach		1	10	100	1,000
One-way synchronization w/o local storage		1	10	100	1,000
One-way synchronization with local storage		1	10	100	1,000
Swaxodo		1	2	10	100

Table 7.3: Round-trips of each session for a traditional web application, the one-way synchronization approach and Swaxodo.

As Table 7.3 shows, all approaches but Swaxodo use a round-trip to the server for each interaction. In contrast, Swaxodo allows the application to execute locally without the need for a round-trip. Swaxodo can use a heuristic method to determine when a round-trip should be performed and in this scenario, every ten interactions a round-trip is performed to synchronize the state. In principle, SWAX applications can be used offline and can delay a synchronization round-trip as long as the client eventually can establish a connection to the server. The trade-off hereby is that fewer round-trips increase the probability of conflicts.

Cloud service costs

In the following section the costs of computation, bandwidth, and storage are determined for the scenario for all three approaches. The cost are determined on the basis of the pricing model of the Google App Engine that is given in Section 4.2.

Storage costs.

The costs for storage are roughly the same for all three approaches, as the server-side persistence of data is the same for all approaches. They are therefore negligible for this evaluation and only given as an indication. The scenario leads to storage costs of

\$36/month. Assuming that each interaction yields about 10 write operations, the costs for storage access are about \$1000/month.

Bandwidth costs. The costs for bandwidth, however, are different for the approaches. Assuming that an average user performs 10 sessions each with 10 interactions (similar to S_{10}) per month, the costs for bandwidth are as follows:

Approach	**Bandwidth costs**
Traditional approach	\$2520/month
One-way synchronization w/o local storage	\$96.84/month
One-way synchronization with local storage	\$78.84/month
Swaxodo	\$78.84/month

Table 7.4: Bandwidth costs for a traditional web application, the one-way synchronization approach, and Swaxodo.

As can be seen in Table 7.4, the cost for bandwidth with the traditional approach are prohibitively high. Therefore, today, the traditional approach is rarely used in social web applications. Approaches similar to the one-way synchronization w/o local storage are more common in practice.

Computational costs. The computational costs are determined by the number of interactions and the required time to process that request on the server. An average response time of $200ms$, a uniformly distributed load and an average usage of 10 sessions each with 10 interactions per month per user is assumed for the calculation in order to derive indicative computational costs.

Approach	**Computational costs**
Traditional approach	\$444.44/month
One-way synchronization w/o local storage	\$444.44/month
One-way synchronization with local storage	\$444.44/month
Swaxodo	\$88.88/month

Table 7.5: Computational costs for a traditional web application, the one-way synchronization approach, and Swaxodo.

As Table 7.5 shows, all approaches but Swaxodo incur the same computational costs per month as they all use a round-trip to the server for each interaction. Swaxodo, however, saves dispensable round-trips and therefore incurs lower computational costs. The savings are largely dependent on the used synchronization heuristic, e.g. after a fixed time interval or after ten interactions.

Overall, the driving cost factors for this scenario are storage access and cost of computation. Even though there are no improvements in reducing the costs regarding storage and storage access, Swaxodo and the one-way synchronization with local storage approach allow savings in bandwidth costs compared to one-way synchronization without local storage. More importantly, the reduction of the computational costs which has a large overall effect on savings is enabled by Swaxodo as round-trips are minimized. Table 7.6 summarizes the overall costs in this example scenario.

Approach	Total costs
Traditional approach	$4000.44/month
One-way synchronization w/o local storage	$1577.28/month
One-way synchronization with local storage	$1559.28/month
Swaxodo	$1203.72/month

Table 7.6: Overall costs including storage, computation, and bandwidth for a traditional web application, the one-way synchronization approach, and Swaxodo.

7.2.4 Summary

Swaxodo as an example for an experimental SWAX application provides significant savings in data transfer sizes and needed round-trips. In respect to the data transfer sizes, Swaxodo and one-way synchronization with local storage reduce data transfer sizes by about 20% in comparison with other approaches for common use cases.

The number of required round-trips is linear in the number of interactions for all approaches but Swaxodo. With Swaxodo, it is subject to a chosen heuristic method. In the given scenario, round-trips were reduced by a factor of 5 - 10. The heuristic method used and the resulting reduction depends on the particular use case and a trade-off exists regarding potential conflicts.

As a result of the reduction in data transfer size and round-trips, significant savings on behalf of the application provider are possible. Therein, Swaxodo is the only solution that allows the reduction of a driving cost factor of computation. In this example computational costs are reduced by a factor of 5. The factor is determined by the aggregation of round-trips according to the synchronization heuristic, e.g.: factor 5 - 10.

7.3 Comparison with other frameworks

This section discusses the related web application frameworks Firebase, Meteor, and Derby. Firebase, Meteor, and Derby present complete solutions for building web applications with synchronization. They have recently been introduced to the market, are under active development and received a remarkable coverage among the tech-scene. Thus they present themselves as popular representatives among related frameworks. On grounds of this relevance, Firebase, Meteor, and Derby have been chosen to be subject to the comparison with SWAX within the remainder of this section.

Firebase

Firebase[1] is a commercial cloud service provider that allows client-side web applications to store and retrieve data on the Firebase backend as well as exchange data with other client instances. Firebase provides an API with a focus on real-time synchronization of changed data objects.

Persistence layer. Data records in Firebase are schema-less and can be grouped to lists. Firebase allows the issuing of queries against the available data objects. Firebase applies changes to data objects locally and asynchronously propagates these changes to the Firebase backend and from there to other client instances. Changes to data objects produce new snapshots of these data objects. In order to set a value of a data object, Firebase offers a `Set` method that overwrites any value regardless of its previous state. In order to provide concurrent updates of data objects with transaction semantics, Firebase offers a `Transaction` method. In principle, Firebase applications can be used offline.

Application layer. Firebase can be used with most modern browsers including mobile browsers and is installed by including a JavaScript file. Firebase applications are executed exclusively on the client. Firebase offers a persistence layer for applications but the design of the application layer is up to the application developer.

Presentation layer. Firebase does not provide a way to bind the data to a view itself, but it is compatible to other third party solutions for model-view binding. At the time of writing, Firebase doesn't offer authentication and authorization of clients. Instead, Firebase states that all data within Firebase is publicly accessible and access control is planned to be added in a future version. By design, there is no server-side application layer for the application developer to deploy.

Comparison with SWAX. Similar to SWAX, Firebase does provide synchronization of changes with snapshots and transactions at the persistence layer as well as an offline capability. The data model, however is schema-less and queries can be issued whereas SWAX applications use a application specific schema and provide no queries. Firebase provides a focus on real-time synchronization of changes. Synchronization with SWAX applications take place at fixed intervals or by other strategies. A key difference to SWAX is that Firebase does not offer a server-side application layer and therefore it is not possible to execute parts of the application in a privileged server-side environment. This makes it

1 Firebase, http://www.firebase.com accessed July 19, 2012

difficult for Firebase to provide security whereas SWAX applications can authorize and validate data at the server-side. Furthermore, Firebase does not provide a presentation layer. SWAX provides widgets with a binding on the application data model within the presentation layer.

Overall, Firebase does provide a powerful persistence layer with real-time synchronization but is missing features in the other layers when compared to SWAX applications.

Meteor

Meteor[1] is an open-source web application platform with a lightweight server component using Node.js and a focus on a client-side application execution.

Persistence layer. Meteor provides a persistence layer as a schema-less server-side key-value store. The client browser has an in-memory cached database with a subset of the same API as the server database. This client-side caching database replicates data objects the client is interested in. Similar to Firebase, data objects can be grouped to lists and queries can be issued against the database. Read and writes of data objects are executed on the local cache first and are asynchronously synchronized with the server. This is self-described as latency compensation as the application execution can make progress without a synchronous round-trip to the server. Meteor has no notion of snapshots or versioning of data and only provides write access to data in a way that will overwrite any previous object value. It is the responsibility of the application developer to detect and resolve concurrent updates and provide versioning of data objects. The persistence layer allows applications to be executed while the client is offline.

Application layer. Meteor allows the definition of callbacks on the database API methods that allow the client to react on the outcome of the synchronization with the server. Customized server-side application code can in principle provide access control and validation, because application code on the server is running in a privileged environment. Implementing access control and validation is the obligation of the application developer. As Meteor applications are written in JavaScript for the client and the server-side they are compatible with any modern browser environment including mobile platforms.

Presentation layer. In addition to the functionality concerning the persistence layer, Meteor provides a comprehensive model-view binding. Templates for the view can be defined with placeholders for data objects that automatically update when the model changes. The view is also updated when the template changes. The framework offers a packaging mechanism that allows to include popular third-party modules to ease development of applications.

1 Meteor, http://meteor.com accessed July 19, 2012

Meteor is still in an early stage of development and currently lacks a sophisticated access control mechanism. The authors claim to add authentication and authorization in a planned future release.

Comparison with SWAX. Within the persistence layer, Meteor in contrast to SWAX does not provide snapshots or versioning of data objects, has a schema-less data model, and does not detect conflicts. It, however, provides queries opposed to SWAX. Both provide asynchronous synchronization of client-side and server-side state. Similar to SWAX, Meteor has a server-side application layer that can be used to authorize clients and validate application data as well as execute server-side only functionality like sending emails. In addition to this, application code is reused on client and server. The presentation layer in Meteor provides similar functionality as SWAX.

Derby

Derby[1] is an open-source model-view controller framework similar to Meteor with a focus on real-time synchronization and offline capability of client-side browser applications. Applications are written in JavaScript such that code can be reused on the server-side in Node.js as well as the browser.

Persistence layer. Derby's persistence functionality uses Racer, a real-time model synchronization engine that can permanently store data to a MongoDB key-value store at the server-side. Racer provides synchronization features with a selection of conflict resolution techniques, e.g. Operational Transformation and Diff-match patch. In order to facilitate conflict resolution, Racer makes use of transactions and versioning. The conflict resolution functionality supports Derby applications to also work offline. Like Meteor, common model access methods including queries are shared between the client and server and callbacks on access methods can be defined.

Application layer. Derby organizes models into hierarchical namespaces called paths which help to structure and access models and to declare private attributes that are not synced.

Presentation layer. Derby provides model-view binding and templating of the user interface. Moreover, views which are determined by routes can be rendered server-side which improves the initial load time of the application. It also gives non JavaScript-enabled clients, e.g. search engines, access to the application. Derby itself does not support access control with authentication or authorization and validation of a model. It is the responsibility of the application developer to implement this functionality within the privileged application environment on the server.

1 Derby, http://derbyjs.com accessed July 19, 2012

Derby is currently alpha software and its model conflict detection and resolution is self-described in a preliminary state. The authors state that it is undergoing major development.

Comparison with SWAX. Derby's strength lies within the persistence layer conflict detection and resolution functionality. Conflict resolution is not a part of SWAX. Like with SWAX, application data is synchronized, but in real-time whereas SWAX synchronizes at fixed intervals or uses other strategies. Like SWAX, Derby has a server-side application layer, but authorization and validation unlike SWAX are orthogonal to Derby. A key difference to SWAX in the presentation layer is that Derby allows to render a view on the server. The model-view binding functionality, however, is comparable to SWAX.

Summary

Firebase, Meteor, and Derby are three frameworks with similar goals and feature set as SWAX. Firebase has a slightly different focus on the client, as a server-side application layer is missing. Derby and Meteor have a similar architecture and functionality in common. Table 7.7 summarizes the feature comparison of the frameworks.

Description	Firebase	Meteor	Derby	SWAX
R1 Scalability	+++	+++	+++	+++
R2 Consistency	+++	+++	+++	+++
Hierarchical model	-	-	++	+++
R3 Offline availability	+++	+++	+++	+++
R4 Synchronization	+++	+++	+++	+++
R4 Conflict detection	++	-	+++	+
R4 Conflict resolution	-	-	+++	-
R5 Responsiveness	+++	+++	+++	+++
R6 Device independence	+++	+++	+++	+++
R7 Cost-effectiveness	+++	+++	+++	+++
Privileged environment	-	+++	+++	+++
Remote method invocation	-	+++	+++	+++
Queries	+++	+++	+++	-
Transactions	+++	-	+++	+++
R8 Versioning	++	-	-	+++
R9 Authentication	-	-	+	+
R9 Authorization	-	-	+	+
R9 Validation	-	+	+	+++
R10 Model-view-binding & Templating	-	+++	+++	+++
Server-side rendering	-	-	+++	-

Table 7.7: Feature comparison of Firebase, Meteor, Derby and SWAX

The frameworks are all under intensive, active development. A main issue of current development among the frameworks is authentication and authorization of clients as well as validation of the data. This is particular important as large parts of the application run on the client and a malicious client could alter this part and the data in order to compromise security. SWAX allows to validate data and has support for authorization. SWAX is compatible to common authentication methods such as cookie-authenticated sessions.

Firebase, Meteor, and Derby support queries on data objects. Providing queries and maintaining scalability is a challenge and it remains to be seen whether the frameworks' query functionality scale up to the size of popular social web applications. Queries are not part of Xydra currently and therefore are not provided by SWAX.

Even though Derby supports common solutions for conflict resolution, in practice, it is unclear whether these solutions are suitable for generally maintaining consistency of the data because the technologies are partly heuristics or need complex customization for a specific use case. In principle, SWAX would be compatible with customized implementations of conflict resolution techniques, but they are not part of the current design for their lack of generality.

One differentiating feature for Derby is its server-side rendering of pages, which is reducing the initial page load time. This is not part of SWAX but as SWAX focuses on offline availability of the application, initial page loads are less frequent.

SWAX has a focus on offline availability of the application, including browser restarts. This is partly realized through permanently storing the application state to local storage. The other frameworks would in principle be compatible with a similar approach, but they are not providing it as part of their current functionality.

All in all, the strength of SWAX lies in its offline availability of the application and the server-side validation of data. Furthermore, it uses a hierarchical data model with versioning and transactions provided by Xydra. Other functionality such as model-view binding and cross-platform compatibility that are in common with the related frameworks render SWAX a competitive solution in building social web applications.

7.4 Summary

SWAX applications are to a large degree in compliance with the requirements of social web applications. Important benefits of SWAX are the responsiveness, scalability and offline availability as well as the cost-effectiveness of applications.

As the the costs evaluation has shown, SWAX significantly reduces the bandwidth and more importantly computational costs on behalf of the application provider.

In regards to the comparison with related frameworks, SWAX presents itself as a competitive solution.

8 Conclusion and Outlook

This chapter gives a resume of the solution and findings presented in this work and a future outlook.

8.1 Summary

Chapter 1 introduced the problem of building social web applications. It presented the goal of this work: designing and evaluating a new approach in building social web applications based on Xydra in a cost-effective way.

Chapter 2 gave a brief overview on social web applications. In this chapter a comprehensive discussion of the Xydra framework was given and the replication functionality of the Xydra synchronizer was characterized. This is likely the first overview of Xydra in the academic literature. The chapter closed with an introduction to basic technologies used.

Chapter 3 listed requirements for social web applications. Therein, scalability, responsiveness, and cost-effectiveness have been presented as important requirements.

Chapter 4 analyzed short-comings of traditional web applications and gave an analysis on pricing of cloud-service providers.

Chapter 5 presented the design of SWAX. The design of SWAX is a central part of this work. The overall architecture and detailed description of the SWAX application structure was shown. The design was illustrated with the help of a collaborative to-do list application as an example for a social web application.

Chapter 6 gave an overview of two implementations of the collaborative to-do list application. The major part of that chapter described the experimental Swaxodo application. Swaxodo is a reference implementation that was used to evaluate the SWAX design.

Chapter 7 evaluated SWAX in three ways. First, it was determined, whether SWAX meets the requirements of social web applications. Second, with the help of Swaxodo, data transfer sizes and round-trips as two important factors for cost-effectiveness and responsiveness were measured and discussed. Third, SWAX was compared with related solutions.

8.2 Conclusion

The goal of this work was to evaluate the Xydra framework for social web applications. This work presented the SWAX blueprint design, the main contribution of this work that uses Xydra. SWAX meets the requirements of social web applications to a large degree as shown in the evaluation on the basis of a reference implementation. The key benefits are that SWAX applications are responsive, work offline, can easily be developed thanks to the blueprint design and are overall cost-effective as data transfer sizes are small and dispensable round-trips are avoided. A limitation of Swax in comparison with other solutions is the lack of queries on the data model. Unaddressed in this work are ways of resolving conflicts. Yet, resolving conflicts in a generic way remains an open issue in the data synchronization domain. Often times, customized strategies are required as more general approaches often rely on heuristics which can be prone to errors. Swax is in principle compatible to customized resolution techniques. Furthermore, authentication of the client is not part of this work, but commonly applied techniques such as cookie-based authentication are possible. According to the results of the evaluation on the basis of the Swaxodo prototype implementation, cost estimations and the comparison with other related frameworks, it can be concluded that Xydra is well suited for building social web applications.

8.3 Future work

The suitability of Xydra for social web applications raises the immediate question whether Xydra is also suitable for building of desktop applications. Xydra with its synchronization capability presents itself as a promising candidate for a more generic desktop synchronization solution.

Furthermore, it would be interesting to evaluate solutions to extend the current synchronization functionality to also support real-time synchronization of data objects.

Finally, with the advent of radically new browser technologies such as peer-to-peer communication it remains to be seen to what extent Xydra can be used to manage and synchronize data on the basis of peer-to-peer replication.

Bibliography

[ACKM10] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[BE07] D. Boyd and N. B. Ellison, “Social network sites: Definition, history, and scholarship,” *Journal of Computer-Mediated Communication*, vol. 13, no. 1-2, Nov. 2007. [Online]. Available: http://jcmc.indiana.edu/vol13/issue1/boyd.ellison.html

[Bre12] E. Brewer, “Cap twelve years later: How the "rules" have changed,” *Computer*, vol. 45, no. 2, pp. 23 –29, feb. 2012.

[Can10] *Automated object persistence for JavaScript*, ser. WWW ’10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772711

[DYB$^+$07] F. Daniel, J. Yu, B. Benatallah, F. Casati, M. Matera, and R. Saint-Paul, “Understanding ui integration: A survey of problems, technologies, and opportunities,” *IEEE Internet Computing*, vol. 11, no. 3, pp. 59–66, May 2007. [Online]. Available: http://dx.doi.org/10.1109/MIC.2007.74

[GH96] J. Gray and P. Helland, “The dangers of replication and a solution,” in *In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996, pp. 173–182.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[Hic11] I. Hickson, “Web storage,” W3C, Candidate Recommendation, Dec. 2011, http://www.w3.org/TR/2011/CR-webstorage-20111208/.

[KJL10] W. Kim, O.-R. Jeong, and S.-W. Lee, “On social web sites,” *Inf. Syst.*, vol. 35, no. 2, pp. 215–236, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.is.2009.08.003

[KP88] G. E. Krasner and S. T. Pope, “A cookbook for using the model-view controller user interface paradigm in smalltalk-80,” *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, Aug. 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=50757.50759

[Nie93] J. Nielsen, *Usability Engineering.* San Francisco, CA, USA: Morgan

Kaufmann Publishers Inc., 1993. [Online]. Available: http://portal.acm.org/citation.cfm?id=529793

[Pau05] L. Paulson, "Building rich web applications with ajax," *Computer*, vol. 38, no. 10, pp. 14 – 17, oct. 2005.

[RECC$^+$12] R. Rodríguez-Echeverría, J. M. Conejero, P. J. Clemente, J. C. Preciado, and F. Sanchez-Figueroa, "Modernization of legacy web applications into rich internet applications," in *Proceedings of the 11th international conference on Current Trends in Web Engineering*, ser. ICWE'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 236–250. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27997-3_24

[Rit07] P. Ritchie, "The security risks of ajax/web 2.0 applications," *Network Security*, vol. 2007, no. 3, pp. 4–8, Mar. 2007. [Online]. Available: http://www.sciencedirect.com/science/article/B6VJG-4NGKDY6-4/2/8945a7ec84b9063a377b3e0fc6168952

[SKK$^+$90] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, "Coda: a highly available file system for a distributed workstation environment," *Computers, IEEE Transactions on*, vol. 39, no. 4, pp. 447 –459, apr 1990.

[Ter08] D. B. Terry, *Replicated Data Management for Mobile Computing*, ser. Synthesis Lectures on Mobile and Pervasive Computing. Morgan & Claypool Publishers, 2008.

[TTP$^+$95] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, pp. 172–182, Dec. 1995. [Online]. Available: http://doi.acm.org/10.1145/224057.224070

[VWV07] G. Vickery and S. Wunsch-Vincent, *Participative Web and User-Created Content: Web 2.0, Wikis and Social Networking*, 1st ed., ser. OECD Publications. Paris: OECDpublishing, 10. 2007, http://www.oecd.org/document/40/0,3746,en_2649_34223_39428648_1_1_1_1,00.html [Stand: 09. 03. 2011]. [Online]. Available: http://www.oecd.org/document/40/0,3746,en_2649_34223_39428648_1_1_1_1,00.html

List of Figures

List of Tables