Offline Data and Synchronization for a Mobile Backend as a Service system.

Master's Thesis

Johan Laanstra

Offline Data and Synchronization for a Mobile Backend as a Service system.

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Johan Laanstra
born in Franeker, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Offline Data and Synchronization for a Mobile Backend as a Service system.

Author: Johan Laanstra
Student id: 1509268
Email: j.p.laanstra@student.tudelft.nl

Abstract

Because of the increase in the number of connected devices used, developers are increasingly building mobile applications. These applications primarily connect users to information that comes from the internet. Mobile Backend as a Service ((M)BaaS) providers started providing developers with hosted backends that are easy to set up and can be used to store data for the application.

Mobile connections are not always reliable or available. Therefore the developer needs to do significant work to make the application work under these conditions. Hosted backends allows for the opportunity to offer a solution for the unreliable connection problem as part of their software development kits (SDK).

In this thesis we design a model for an (M)BaaS system to offer offline data and synchronization support, while preserving as much of the functionality that is offered by such a system. In addition, the model offers flexible conflict resolution and optimal use of data stored offline.

The model is implemented on top of Windows Azure Mobile Service (WAMS), which is the (M)BaaS system offered by Microsoft. This implementation shows that the model works and that it can be successfully applied to an (M)BaaS system.

Our main contributions are a synchronization model for an (M)BaaS system, including an implementation that can be used with the existing WAMS .NET SDK on the Windows platform, the flexible conflict resolution options and the ability to answer queries locally using the cached data from other requests.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, HGL SE, Delft University of Technology
University supervisor:	Dr. Ir. F.J.H. Hermans, UD SE, Delft University of Technology
Company supervisor:	J. Allor, Development Manager, Microsoft
Committee Member:	Dr. A. Iosup, UD PDS, Delft University of Technology

Preface

This thesis was written as part of my Master of Science degree at the Delft University of technology.

As part of this thesis I did a three-month internship at Microsoft's main campus in Redmond in the Windows Azure Mobile Services team. This has been an amazing experience and has lead me to decide to go back to work full time at Microsoft. The Mobile Services team is a great team with a lot of passionate people working on improving the product every day. I want to thank everyone at Microsoft who has helped making the internship possible, especially my manager Jason Allor, my mentor Randall Tombaugh and Yavor Georgiev. I know managing me can be hard, but they did an awesome job and I learned a lot.

In this thesis you will find the results of the project I started at Microsoft during the internship. I want to thank Erik Meijer for his advice during the internship and Arie van Deursen for his excellent guidance, support and feedback on this thesis. Looking back at the years I spent in Delft, I have had a great time during my study. I want to thank everyone who has helped make this possible, including my parents and all my friends.

Johan Laanstra
Delft, The Netherlands
February 27, 2014

Contents

Chapter 1

Introduction

The number of connected devices in the world is rapidly increasing. A report from IDC estimates that the total number of shipped connected devices will grow with 58.1% between 2013 and 2017 [1]. This number indicates that the total number of connected devices will be growing rapidly. These devices are vastly different from desktop computers, because they are meant for different purposes. These connected devices are meant to connect people to information. This information from their social networks, their work information and their email. Also in contrary to desktop computers these devices are usually taken all over the world instead of standing at a desk.

A different kind of device needs different kinds of applications running on it. These mobile applications generally focus on doing one thing and doing that one thing very well, in contrary to desktop applications that have a large feature set that has increased over years to fulfill the needs of different users. As the devices are mostly touch-based, always-on and taken everywhere on the world, applications should take these conditions into account, when providing the user with information. The content of the application should refresh itself periodically and provide information that is relevant in the region where the user currently resides. The application should be easy to control using touch and take care of unreliable connectivity while operating.

These mobile applications are no longer only built by development teams at companies consisting of experienced software engineers. Basically anyone with a computer and an idea can start building an app and publish his or her creation into one of the many app stores available for the different platforms. Many of those applications connect to backend services and data sources available on the web to provide their users with information. While most developers are able to build a mobile application, building a scalable backend for those applications is a different story. These developers do not necessarily have the knowledge to setup such a backend.

[1] http://www.idc.com/getdoc.jsp?containerId=prUS24314413

1.1 Backend as a Service

In a backend, security and scalability are more important than they are for a mobile application. Because of the increasing amount of developers focusing on mobile applications, several companies started offering scalable and flexible backends for developers to build their applications on top of. These offerings are called a (Mobile) Backend as a Service ((M)BaaS) and include features ranging from storage to push notifications, user management and integration with social media services [22]. The goal of these BaaS providers is to make it very easy for developers to setup and operate a backend for any application. Most mobile applications will need a similar feature set for their backend and BaaS providers offer a consistent and unified way to access these different services via their Software Development Kit (SDK). With those offerings you can have a running backend in seconds.

Examples of some of the big (M)BaaS providers are StackMob [2], Parse [3], Kinvey [4], Appcelerator [5], Google Mobile Backend Starter [6], Windows Azure Mobile Services [7], and there are many more. They all offer similar functionality but some of them focus on specific functionality such as storage or location services.

1.2 Not So Connected Devices

Having a scalable and reliable backend for a mobile app is important, but if the network connection is not working, it does not buy you much. While it does not happen often on wired connections, wireless and broadband connections are not always reliable and available. Therefore it happens occasionally that an app cannot reach its backend. We do not want our application to stop working under these conditions. If there is no network connection we cannot perform operations that require a connection such as logging in a user, but we can show an already logged in user the data of the previous session. We can also allow him to make changes to data. These changes can then be synchronized as soon as the connection is restored.

Support for offline data is frequently requested by developers that build applications on top of (M)BaaS systems. For the Windows platform they only have the option to build a custom solution for their application. This offers opportunities for (M)BaaS providers to offer a solution that works for most or all of the scenarios. (M)BaaS offerings have the advantage that they control both the client and the server part of their offering. Because (M)BaaS offerings have a clear goal and feature set to support these goals, a synchronization solution can be built specifically to support these goals and to work with the feature set that the solution provides. An example of a feature that a synchronization solution can use is the querying of a remote data source. This is offered by almost all providers and a synchronization solution could add support for offline querying. Some of the (M)BaaS

[2] https://www.stackmob.com/
[3] https://parse.com/
[4] http://www.kinvey.com
[5] http://www.appcelerator.com
[6] https://cloud.google.com/developers/articles/mobile-backend-starter
[7] http://www.windowsazure.com/en-us/solutions/mobile/

providers offer some offline functionality, but currently no offerings exist for the Windows platform.

1.3 Goal of this Thesis

The goal of this thesis is to investigate how we can add offline data support and data synchronization to an (M)BaaS system, while keeping as much of the feature set working when offline. Having such a system is useful for situations where we lose the internet connection, but we do not want an application to stop working. We will look at the different ways this can be implemented on an (M)BaaS system and create an implementation on top of Windows Azure Mobile Services (WAMS). The implementation on WAMS will service as a proof of concept of how we can extend an (M)BaaS system with offline data support and data synchronization.

This research was conducted as part of an internship at Microsoft. The goal of this research was to explore the possibilities of extending the existing Mobile Services .NET SDK with offline data support and data synchronization. This should be done without making modifications to the existing SDK and runtime to make the project usable outside of this research and to not be dependent on the release schedule and plans of WAMS. The solution has the following goals:

- **G1 Offline data support and data synchronization.** The solution should offer offline data and data synchronization support for an (M)BaaS system.
- **G2 Extend the Mobile Services .NET SDK.** The implementation on WAMS should extend the Mobile Services .NET SDK without requiring changes to the SDK or runtime.
- **G3 Consistent data.** The offline data solution should have at least the same data consistency as (M)BaaS system would have without the offline data solution.
- **G4 Flexible conflict resolution.** The solution should be as flexible as possible to do conflict resolution. Conflict resolution should be possible both on the client and on the server, depending on what the developer needs.
- **G5 Optimal use of cached data.** The data cached locally on a device for certain requests may overlap with data required for other requests when the network connection is not available. In the optimal situation we could reuse the cached data for other requests providing the data is applicable to the new request.
- **G6 Fast response times.** The solution improves response times without a network connection, since in these situations the application would currently stop working. In addition we do not want the response times to be significantly slower when a network connection is available.
- **G7 Decreased network usage.** Due to the caching of data, for previous requests we do not want to send the same data over again if that is not needed. Instead we would like to only transmit the changes since the previous request.

This thesis is organized as follows. Chapter 2 will provide an overview of Windows Azure Mobile Services (WAMS) and other tools and techniques used in this thesis to im-

plement data synchronization on WAMS. Related work on data synchronization will be the topic of chapter 3. Chapter 4 will discuss the problems of data synchronization in more detail. The design of the synchronization solution for an (M)BaaS system will be discussed in chapter 5 followed by the implementation of the solution on top of WAMS in chapter 6. How we will evaluate the solution will be discussed in chapter 7 followed by the results of the evaluation in chapter 8. Chapter 9 will discuss offline data offerings from similar products on the market including a comparison with the solution proposed in this thesis. The results and the comparison will be discussed in chapter 10 and this thesis will be concluded in chapter 11.

Chapter 2

Background

The solution in this thesis was built on top of the backend services offered by Windows Azure. This chapter discusses Windows Azure and Windows Azure Mobile Services. Ii also contains information about the tools and techniques used by the solution. Subsequent chapters will reference back to this chapter for more information when these technologies are used or applied.

2.1 Windows Azure

Windows Azure is Microsoft's cloud platform. It offers services ranging from raw virtual machines to push notifications for mobile devices. Windows Azure was released on February 1, 2010. It offers both Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) services including support for many programming languages such as C#, Node.JS, PHP, and Java. Specific services that are offered by Azure include [1]:

- Web application hosting
- SQL databases
- Media streaming
- Virtual machines
- Storage
- Access control
- Hosted backends

2.2 Windows Azure Mobile Services

Windows Azure provides (M)BaaS services, called Windows Azure Mobile Services (WAMS). The goal of WAMS is to provide you with an easy scalable backend for your mobile application that is managed for you. When WAMS was released at the end of June

[1] This is not a complete list. For more info look at `http://www.windowsazure.com` or download the Azure Poster at `http://www.microsoft.com/en-us/download/details.aspx?id=35473`.

2013 and came out of preview, it had the following distinctive features to use in a mobile app which are also shown in figure 2.1:

- Store data in a SQL Azure database hosted on Azure that is automatically setup and exposed via REST API's.
- Host custom REST APIs (without the need for a SQL database).
- Authenticate users using the available authentication providers which include Microsoft Account, Google, Facebook and Twitter.
- Send push notifications to Windows, Android and iOS devices.
- Connect to other services in the backend using Node.JS.

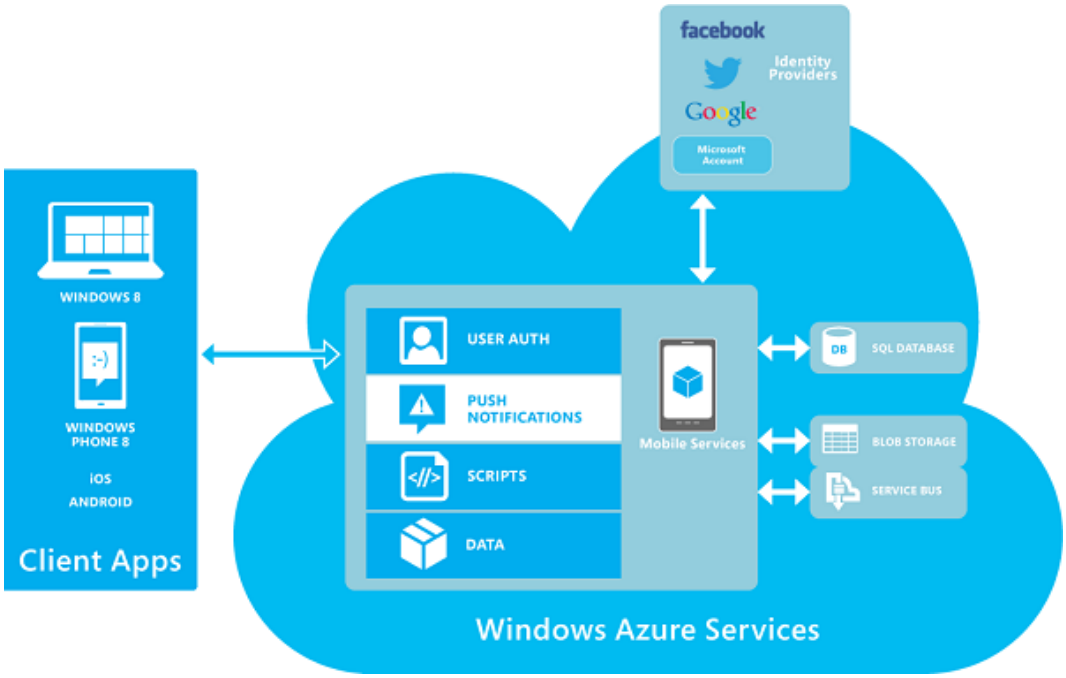

Figure 2.1: Overview of Windows Azure Mobile Services [2].

Windows Azure Mobile Services exists out of a backend service and a client SDK that is part of the application built by the developer. These two parts work together to offer all the functionality as easy as possible.

- **Backend application.** The backend application offers all the services. This is the real backend and requires a SQL Azure database to work [2]. This is where APIs can be hosted, where the server scripts are executed and where data is stored.

[2] http://www.windowsazure.com/en-us/pricing/details/mobile-services/

```
1 function insert(item, user, request) {
2   request.execute();
3 }
```

Listing 2.1: Default table script for an insert operation.

- **Client SDK.** The WAMS client SDKs are available for almost every platform to maximize cross platform usability. This SDK is used to make connecting to the backend as easy as possible. It can send queries to the backend, make calls to APIs and authenticate users.

2.2.1 Backend Application

The backend application is built mostly in JavaScript using Node.JS which is discussed section 2.4. More specifically, the server runtime is built on top of Express [3], a web application framework built on top of Node.JS. The backend makes sure that logging information ends up at the correct place, the database connection is set up, requests are authorized when authentication is enabled and many other things and requires a SQL Azure database to operate. This database is also used to store data in the mobile service. Every mobile service has a single database associated with it.

The ability to store data in a SQL Azure database includes more than simply exposing database tables on the internet. The CRUD (Create, Read, Update, and Delete) operations for the database are exposed as REST APIs called Table APIs. For each operation on a table, a custom JavaScript script can be run on the server before the request is executed and the data gets changed in the database. This allows one to for example send a push notification for a new insert or remove associations with other services when an entry is deleted. An example of the default script for an insert operation can be seen in listing 2.1. The default scripts for other operations are similar to the insert script.

These script do not have global or shared state. After execution of the script and completion of the operation, all state is destroyed. It is possible to do almost anything before and after a CRUD operation and the response can even be adjusted to the way the user wants. Nevertheless, the SDKs provided for Mobile Services expect certain responses for certain requests and therefore the responses need to be conform these expectations. For example, for tables the only valid request and response format is JSON.

Custom APIs extended the possibilities of what one can send and receive from the server compared to the Table APIs. This flexibility enabled a lot more scenarios about what is possible with Mobile Services. The SDK support for this feature is a lot simpler because less assumptions can be made about the functionality of such an API. Custom APIs have no database requirement although one can still access the database where needed. In custom APIs any external service can be used and one can return any response required. The SDK still expects the format to be JSON, but this is only an SDK requirement instead of enforced by the server.

[3] http://expressjs.com/

```
1 http://services.odata.org/OData/OData.svc/Products?$filter=
    Price gt 5
```

Listing 2.2: Example of an OData query for products with a price greater than 5.

2.2.2 Client SDKs

The client SDKs are focused on enabling the backend functionality as easy as possible on the platform they target. The SDKs are unobtrusive for an existing application and usually allow working with native objects of the platform of choice, including queries over these objects. The SDKs offer facilities to send queries to the server and to make requests to custom APIs hosted in the backend.

Authentication is another things the SDK makes really easy. Developers no longer have to think about the details of OAuth, the web standard for authentication these days. The WAMS client SDKs take care of all the specifics and offer by default the option to authenticate with Microsoft Account, Google, Facebook or Twitter. If one wants to authenticate with something else this can be done by retrieving a JSON Web Token [4] from the authentication provider and use this token to set the user on the client.

2.3 OData

The WAMS Table APIs allow for a rich query syntax specified as part of the URI. This query syntax consist of a subset of the OData query language that is part of the Open Data Protocol (OData) [5]. OData was created to offer a consistent way to create and consume APIs that offer data. OData is consistent with the way the web works, because it uses URIs for resource identification and has an HTTP-based, uniform interface for interacting with those resources. OData was originally built by Microsoft.

The protocol is built around collections of entities. These entities can be added, removed, changed and requested by specifying queries as to what data should be returned. There is also the possibility to specify relations between different collections of entities. To enable clients to gain information about the entities in an OData web service, OData provides the metadata document. This document describes the entities, their properties and the relations between them. Based on this metadata document client SDKs can derive a model of the web service for use in the application.

OData supports a rich query language to be specified in the URI of a request to specify which data to retrieve, called the OData query language. It allows for filtering, ordering and limiting the result set for a request. Consider an OData web service that exposes a collection of `Product` instances. Using this web service we can request all products with a price greater than 5. A possible URI for this request is shown in listing 2.2.

WAMS supports a subset of the OData query language to query for data in a mobile service. These OData queries are supported by both the Table APIs and the Custom APIs.

[4] `http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-13`

[5] `http://www.odata.org/`

A URL used by an OData service has at most three parts: the URI root, a resource path and query options as can be seen in figure 2.2.

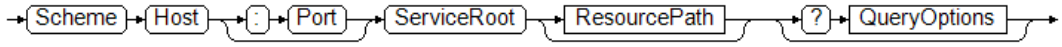

Figure 2.2: Different components of an OData URI[6].

Mobile Services does not use the "ServiceRoot" part of the URI. Based on the Augmented Backus Naur Form (ABNF) for OData v3 URIs [32], the `odataRelativeUri` rule is supported, which is equal to the "ResourcePath" and the "QueryOptions" part of figure 2.2 combined. As long as the resource path points to either a table URI such as `/tables/products/` or a URI to a custom API such as `/api/products/`. WAMS support for OData queries (`queryOptions` rule in ABNF and "QueryOptions" in figure 2.2) can only exist of either the `customQueryOption` rule which indicates normal URI parameters or the following subset of rules of `systemQueryOption` which indicate the different OData query URI parameters and always start with a $ sign:

- filter
- orderby
- skip
- top
- select
- inlinecount

There is no supported in WAMS for the "expand" option or relations between data.

2.4 Node.JS

Node.JS is a software platform meant to make building server side applications in JavaScript very easy. It is built on top of Google's V8 JavaScript engine [35]. Node.JS abstracts the threading for web applications away and because it is written in JavaScript, web developers do not have to learn a new language. They only have to get familiar with the Node.JS libraries. Node.JS can be run independently from a web server such as Apache, because it includes a built-in HTTP server library.

Node.JS was created by Ryan Dahl who was searching for a way to bring event driven programming to the web server. Other languages and frameworks mostly used blocking I/O which conflicted with Dahls event driven approach. When Google opened up its V8 JavaScript engine this offered the perfect opportunity to use JavaScript on the server. Since JavaScript was essentially devoid of server-side code, Dahl could write the code for I/O from scratch and use his event-driven approach to do that.

Listing 2.3 shows a simple web server written in Node.JS that responds with "Hello World" for every request.

[6] `http://www.odata.org/documentation/odata-version-3-0/url-conventions/`

```
1 var http = require('http');
2 http.createServer(function (req, res) {
3   res.writeHead(200, {'Content-Type': 'text/plain'});
4   res.end('Hello World\n');
5 }).listen(1337, '127.0.0.1');
6 console.log('Server running at http://127.0.0.1:1337/');
```

Listing 2.3: Simple web server responding to every request with "Hello World".

The WAMS server runtime runs on express.js. Express.js is a web application framework built on top of Node.JS. It provides functionality to build an application using the Model-View-Controller architecture and helps with everything from routing to handling request and views.

2.5 C#

The solution in this thesis will be built on top of the WAMS client SDK for the Windows platform. This SDK is written in C#. C# is a general purpose, object oriented, programming language created by Microsoft as part of its .NET initiative. It was announced in July 2000. Among the design goals were simplicity, familiarity for C and C++ developers and embracing Object Oriented Programming principles. The designers of C#looked at many of the existing programming languages available at the time with the same core functionality they wanted for C#. While some people would say that C# was built as a Java clone, it initially had a lot more resemblance to C++ [30]. Most of the operators, keywords and statements were directly borrowed from C++ as well as operator overloading and enums which are not available in Java [7].

C# supports most of the concepts and features that are generally defined as Object Oriented as identified by Armstrong [9] and Pierce [31].

- **Inheritance (subtyping).** C# supports both specification inheritance and implementation inheritance. A class in C# can only inherit from a single subclass (implementation inheritance), but it can implement multiple interfaces (specification inheritance).
- **Encapsulation.** Encapsulation is the ability to hide the internal implementation of a class and only expose its public interface. This is fully supported in C# by using the various access modifier keywords available (private, protected, internal, public).
- **Polymorphism.** C# supports both ad-hoc polymorphism and parametric polymorphism. Ad-hoc polymorphism is also known as method overloading and operator overloading. When a class has multiple methods with the same name the method to be called is determined based on the arguments provided. C# has parametric polymorphism in the form of generics. Generics allow a type to be written generically with respect to a type. Using generics it is possible to specify that values should have

[7] Actually enums are available in Java, but they are just classes.

the same type without explicitly defining the type. Co- and Contravariance is also supported for generics.
- **Static Single Dispatch.** The decision which method to call is made based on the compile-time types of the arguments and the object the method is called on. When using the dynamic keyword it is possible to do dynamic dispatch and multiple dispatch where the method to call is based on the types of the arguments and the object at runtime.
- **Classes and Objects.** C# has classes and objects. A class is simply a description of the properties of an object. All concrete objects in C# will be either null, or a class, delegate, array, struct, enum or value type, which means that all object instances derive from object and are therefore objects. This is different from for example Java, where the basic types such as "int" do not inherit from "object".

A detailed overview of the C# language is outside the scope of this thesis. There are excellent resource available to learn more about C#, for example C# 5.0 Unleashed [16].

2.6 JSON

JavaScript Object Notation (JSON) [15] is a data format primarily used to exchange structured data, in serialized human readable text form, between a client or web application and a server. The format is language independent. For almost every programming language there exist libraries to parse and generate JSON. JSON can represent four primitive types as shown below:

- **Number**, double-precision floating-point format as in JavaScript.
- **String**, double-quoted Unicode strings that support escaping by using a backslash.
- **Boolean**, either true or false.
- **null**.

Next to the primitive types JSON also supports two structured types:

- **Array**, an ordered, comma-separated sequence of values enclosed in square brackets. JSON allows the values in the array to be of different types.
- **Object**, an unordered, comma-separated collection of key-value pairs enclosed in curly braces. The ':' character separates the key and the value and the keys must be distinct strings.

Whitespace can be inserted between any pair of tokens. JSON supports defining the structure of a document using JSON Schema, just as XML Schema defines the structure for an XML document. Listing 2.4 shows an example of s JSON document.

```
{
  "employees": [
    {
      "firstName":"John",
      "lastName":"Doe"
    },
    {
      "firstName":"Anna",
      "lastName":"Smith"
    }
  ]
}
```

Listing 2.4: JSON document.

2.7 SQLite

SQLite [8] is becoming the standard for local databases on mobile devices. It is supported on all major platforms including Windows, Windows Phone, Android and iOS. On a mobile device there is no need for the advanced scalability, resiliency and distributed functionality that is part of databases like Microsoft SQL Server or MySQL. SQLite is largely self-contained and requires very minimal support from the operating system to do its work. It needs zero configuration and does not use a database server.

In SQLite every database operation happens in a transaction. All changes and queries are Atomic, Consistent, Isolated, and Durable (ACID). It implements most of the SQL92 standard [9]. Unsupported SQL features in SQLite can be found at the SQLite website [10] and include:

- RIGHT and FULL OUTER JOIN.
- Complete ALTER TABLE support.
- Complete trigger support.
- Writing to VIEWs.
- GRANT and REVOKE.

In contrary to most databases SQLite uses a dynamic typing schema. The datatype is associated with the values instead of with the table or column. 5 different datatypes or storage classes are distinguished:

- **NULL.** The value is a NULL value.
- **INTEGER.** The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.

[8] http://www.sqlite.org
[9] http://www.contrib.andrew.cmu.edu/ shadow/sql/sql1992.txt
[10] http://www.sqlite.org/omitted.html

- **REAL.** The value is a floating point value, stored as an 8-byte IEEE floating point number.
- **TEXT.** The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- **BLOB.** The value is a blob of data, stored exactly as it was provided.

Any column in SQLite can store values of any storage class except for the 'rowid' column that every table has by default. Each column in a SQLite database is assigned a type affinity that indicates what the recommended storage class is for that column. SQLite supports the following type affinities:

- **TEXT**
- **NUMERIC**
- **INTEGER**
- **REAL**
- **NONE**

SQLite determines the affinity of a column based on the specified type name for a column when the column is created. For example if the type name contains the string 'INT' it will be assigned INTEGER affinity. More information about the SQLite type system and how type affinities are derived can be found at the SQLite website [11].

[11] `http://www.sqlite.org/datatype3.html`

Chapter 3

Related Work

Before we will take a careful look at the problem statement, in this chapter we give an overview of work done by others on data synchronization and offline data support. Keeping data offline and the synchronizing of changes made to data in offline scenarios has been a problem that many studied or tried to solve in their products or in standards.

This chapter will discuss different synchronization techniques and conflict resolution including some applications of the techniques. It will also dive into the theoretical limits that a synchronization solution has.

3.1 Synchronization Techniques

Literature about data synchronization differentiates between synchronization of ordered data and unordered data. Synchronization of unordered data is known as the set-reconciliation problem. Several solutions have been developed to deal with this problem. The applicability of these solutions depends on the chosen distributed model. Most of these solutions work very well in a hub-and-spoke model where data is synchronized through a hub, a central server, but when applied to a peer-to-peer model most solutions need changes to account for the fact that their changes need to be synchronized with multiple clients. Some of the strategies developed include:

- **Wholesale synchronization [34] [8].** When data is synchronized, one of the devices involved in the synchronization operation sends all the data it has to the other device for local comparison. This tends to be very inefficient because in most cases there will be only a few changes made to the data, while all the data is sent over the network for comparison. It has the advantage that it is simple to implement and guarantees that all changes are transmitted.
- **Status flag synchronization [34].** With status-flag synchronization a client maintains information about the data in the form of status flags. These flags indicate if an item was created, modified or deleted. When synchronizing, a client can simply send those items who have their flag set. This is more efficient than wholesale synchronization. If synchronization is performed with multiple clients, this does not work

well. In addition to recording which data has been changed, we also need to maintain information about with whom we have synchronized the changes.

- **Timestamp synchronization [34].** When timestamp synchronization is used, every client maintains information about when the data was last changed and a timestamp per client when the last synchronization with this client took place. This includes a timestamp and information about when the data was inserted, modified or deleted. During synchronization only changed items since the last synchronization have to be sent. This is an improvement over status flags, since we do not have to maintain information about what change has been synchronized with whom. It may however be very inefficient in situations where two clients are both fully synchronized with other clients, but are synchronizing with each other for the first time. In this case they will both send all data, while there are no changes between them.
- **Mathematical synchronization [34] [29] [14].** This mechanism makes use of mathematical properties of the data that needs to be synchronized. Minsky and Trachtenberg use characteristic polynomials to represent their sets of data as proposed by [25] and then send enough data over to reconstruct the polynomial and find the changes. Mathematical synchronization using characteristic polynomials requires knowledge of the number of changes upfront. Minsky and Trachtenberg use a probabilistic scheme to determine an upper bound on the number changes, which requires multiple requests to be send to determine all changes with very high probability.
  Choi et. al. use a Message Digest to determine changed data that needs to be synchronized [14]. Synchronization based Message Digest is another form of mathematical synchronization. Their solution is independent of vendor specific database features and only uses standard SQL operations. Nevertheless their solution requires several tables on the server and is highly dependent on the relational database model, by using foreign keys and JOINs. In addition they have a table per client on the server.
- **Log synchronization.** Log synchronization is used a lot in databases. Every change is executed as a transaction and also saved in the log. These logs are than synchronized with other clients. This approach is also frequently taken by source control solutions. When the log is synchronized, each operation is replayed on the other client. The log approach is also taken by Core Data and iCloud Sync on iOS [1]. Logs containing the changes can grow significantly as they store all operations in addition to the normal data.

Synchronization of ordered data is a different problem and more difficult. For ordered data not only the content of the data is important, but also its location in the sequence. The simplest example of ordered data is text. Text is ordered data, because the string "a b c" is different from "b a c". The set existing of the strings "a", "b", "c" would be the unordered variant. Ordered data synchronization concerns usually keeping files or text synchronized and is outside the scope of this thesis.

Data synchronization can work in both directions or only in a single direction [5]. Synchronization in both directions is called bi-directional synchronization. When data is read-only on a client and changes only occur on the server, we only have to retrieve changes from the server. This is called download-only synchronization. When items never change

and are only used and modified by a single client, we only have to send new items to the server, called upload-only synchronization.

3.2 Conflict Resolution

When two or more clients synchronize data, it may happen that they want to synchronize conflicting changes. Consider the following sequence of events, which is also shown in the revision diagram in figure 3.1 [11]:

- Client A retrieves item 1 from the server.
- Client B retrieves item 1 from the server.
- Client A and B go offline.
- Client A makes a change to item 1 and comes back online to synchronize.
- Client B makes a change to the same property as Client A and comes back online to synchronize.

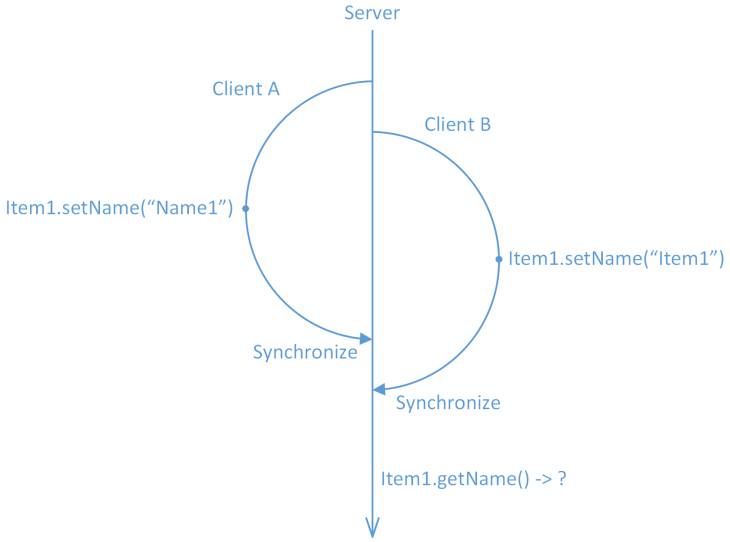

Figure 3.1: Revision diagram of Client A and Client B both modifying the same property of "Item1" while offline, resulting in a conflict.

When client B tries to synchronize we have a conflict and conflict resolution needs to be performed. Several general conflicts resolution policies have been identified in literature [24], besides the custom policies that can be used for specific applications and in certain scenarios:

- **Originator Wins.** Take the data item of the originator.
- **Recipient Wins.** Take the data item of the recipient.
- **Client Wins.** Take the data item of the client.

- **Server Wins.** Take the data item of the server.
- **Recent Data Wins.** Take the data item which has been updated recently in time.
- **Duplication Apply.** The requested modification is applied on a duplicated data item while keeping the existing data item.

3.3 The CAP Theorem

The CAP theorem (also known as Brewer's conjecture) for a distributed system states that it is impossible for such a system to provide simultaneously at all times the following capabilities [20].

- **Consistency.** All nodes see the same data at the same time.
- **Availability.** The service is available and responds to requests, either with success or failure.
- **Partition Tolerance.** If two different parts of the system can no longer communicate with each other, the system will still respond to requests correctly.

The theorem was proven in 2002 when Seth Gilbert and Nancy Lynch of MIT published a formal proof of the CAP theorem [20]. The theorem in practice means that if a system is needed that scales over multiple servers and has 100% availability we cannot achieve 100% consistency at every moment in time for every request made to the service, because communication channels can break. In this case we have achieved (C)AP. To achieve 100% consistency we can sacrifice availability, by stopping to answer requests until the total system is again consistent. In this case we have achieved C(A)P. The simplest way to sacrifice partition tolerance is by not having it. Most web applications still run on a single server with a single database and therefore are not partitioned at all (CA(P)). This is also the case for WAMS when you create a new mobile service [1]. In the case that this server goes down, it is not available, but it also will not cause inconsistencies. This state is equal to a none-existing system. Running on single server or a single database comes of course with significant scaling limits, but for small applications this is usually acceptable.

The CAP theorem does not mean that choosing partition tolerance and availability precludes consistency. It merely states that one cannot have 100% consistency all the time. Consistency exists in many forms and such a system can be designed to offer very good consistency for a single user or eventual consistency [36]. Eventual consistency is a weak consistency model that states that when a data object is modified, eventually all readers will read the modified item. A situation that weak consistency can result in is when two customers order the same item and there is only a single item in stock. In this case we can transform the second into a back-order.

3.4 Database Synchronization

The synchronization techniques from section 3.1 have been widely applied to synchronization of databases [19] [14] [8] [34]. Many database vendors have recognized the need for

[1] Although WAMS can run on multiple server, it uses a single database.

data synchronization and tried to make it easy by incorporating specific features to do synchronization into their products. This section will discuss some of these features that exist in database systems today.

3.4.1 Replication

Replication is a technique used in databases to build a change log that can be replayed on a different server to create an exact copy of a master database. Replication can be synchronous or asynchronous and is a form of log synchronization. It can be used to scale out and thereby to increase performance or it can be used as a backup for an existing database.

Most modern databases support replication in some form. These include MySQL [2], SQL Server [3], and Oracle Database [4].

3.4.2 Change Tracking

Some databases including Microsoft's SQL Server offer a feature called "Change Tracking" which can be enabled on a per table level [5]. When enabled on a table, every change to a table is recorded in an internal change tracking table. Information about changes can be retrieved from the database by using special database function. Among the information that is tracked is the SQL operation (insert, update, delete), but also which columns have changed.

Change Tracking can be configured to clean up the data after a certain period, but change tracking does not track changes to the schema of a table.

3.4.3 Timestamp Data Types

Microsoft's SQL Server offers another feature useful for synchronization scenarios. It has support for a "timestamp" data type called "rowversion" [6]. This datatype does not store any time related information, but serves as an incremental number that can be assigned to stored data items. It is automatically updated when a new item is inserted or when an existing item is changed. "Rowversion" values are unique per database and SQL Server also stores the latest assigned number in a database.

3.5 Bloom Filters

Bloom filters can be used for data synchronization as noted by Byers [12]. The property that makes them useful is that they can be used to test set membership without storing all the elements in memory. Bloom filters work by using a bit array of m bits, all set to 0, and k hash functions. If an element is in the set it is fed to each of the k hash functions to get k

[2] http://dev.mysql.com/doc/refman/5.0/en/replication.html
[3] http://technet.microsoft.com/en-us/library/ms151198.aspx
[4] http://docs.oracle.com/cd/E16655_01/server.121/e18895/toc.htm
[5] http://technet.microsoft.com/en-us/library/bb933875.aspx
[6] http://technet.microsoft.com/en-us/library/ms182776.aspx

array positions. The bits at those positions are set to 1. To test if an element is part of the set we can simply check if the array positions are 1. An example of a Bloom filter is shown in figure 3.2. Bloom filters allow for false positives, i.e. the filter indicates that an item is included that is not, but they do not allow for false negatives, i.e. the filter indicates an item is not included while it actually is. Bloom filters have a strong space advantage over other data structures for representing sets, because those other data structures usually require the actual elements to be available in memory.

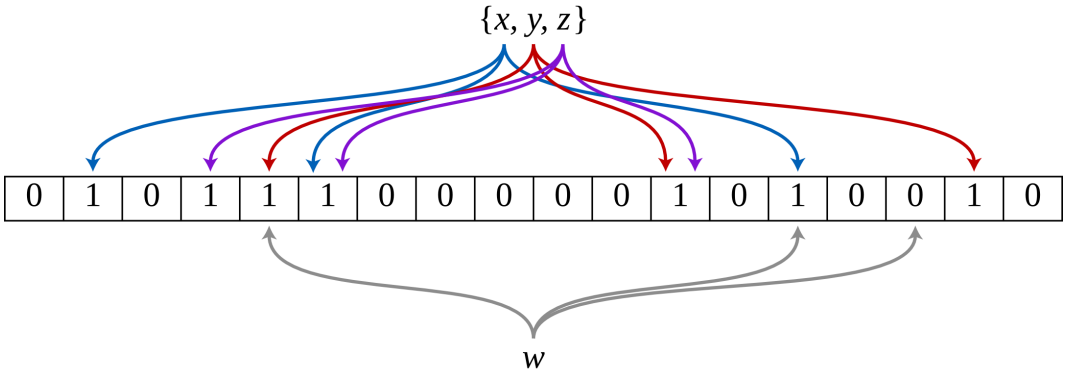

Figure 3.2: Example of a bloom filter. This filter contains x, y, and z. It does not contain w, because one of the hash positions points to a zero. [3].

Bloom filters can be categorized as a form of mathematical synchronization and the synchronization works by a client A sending his filter to a client B. Based on the received filter, client B can determine which elements are not on available on client A, the client that has sent the filter, and it can send those elements to client A. Client B can also unintentionally think that some elements are already on Client A, which will therefore not be sent to client A. Byers et al. argue that this is not always a problem, because the false positive ratio can be kept low and parts of the missing data can be reconstructed based on the available data, especially for encoded content [12]. The chances of missing data can be lowered even more when synchronization is performed with multiple clients making this technique more useful in peer-to-peer based synchronization.

3.6 Cloud Data Types

Burckhardt et al. propose the use of special cloud data types at the programming language level [11]. These cloud types should be an abstraction layer to allow the developer to focus on the client code that uses them. These cloud types are automatically persisted locally and remotely and are designed to behave predictably under concurrent modification.

These special data types offer higher level operations with better conflict resolution semantics. For example their cloud integer type does not only define absolute change operations such as “get” or “set”, but also relative change operations such as “add”. Their

```
1 ETag: "loremipsum"
```

Listing 3.1: ETag header.

```
1 If-None-Match: "loremipsum"
```

Listing 3.2: If-None-Match header.

model is highly dependent on revision graphs, similar to version control systems. Changes can only be applied to the fork this revision was based on. They also define the conflict semantics for each of their cloud types.

3.7 HTTP ETags

The HTTP specification developed by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C) provides some mechanism for synchronization and caching, namely ETags (entity tags) [17]. The ETag is one of the mechanisms provided in HTTP for cache validation, which allows a client to make conditional requests based on the ETag of the value it currently possesses. The ETag header can simply contain a version number or a hash of the content or the timestamp. The format and requirements of the ETag header are not part of the HTTP specification and are therefore solution specific.

In a typical scenario the web server assigns an ETag to a resource and places this ETag in the ETag header of the HTTP response as in listing 3.1.

When the client receives the resource with an ETag header it may decide to cache it. Later when the resource is again required, the client can include the ETag value in the "If-None-Match" header as in listing 3.2.

When the server receives an "If-None-Match" header, it can optionally check if the value still matches the current ETag and return an HTTP 304 "Not Modified" status. If the values did not match, the normal response is returned as if there were no ETags used.

The client could also send an "If-Match" header similar to the "If-None-Match" header to indicate the request should only be executed when the ETag value is matched. For example to only update or request specific resources. Both "If-None-Match" and "If-Match" can take multiple values as a comma separated list.

Chapter 4

Problem Statement

In chapter 3 we have seen work done by others on data synchronization and offline data. In this chapter we discuss the problem of data synchronization and cached data for offline usage in more detail. Data synchronization tends to be hard to implement and requires a developer to have knowledge about a variety of system and platforms. Part of this chapter is to see how these problems map to an (M)BaaS system.

The goal of this chapter is to identify the problems that an offline solution can solve, but also the possible problems an offline solution brings to the surface and that need to be taken care of.

4.1 Reasons for Offline Data

In an (M)BaaS system, multiple clients usually exchange data with a single server. An (M)BaaS system can therefore be considered a hub-and-spoke distributed system, where the backend server is the hub and the clients are spokes. In every connected system it can happen that the network connection disappears for a certain amount of time. This becomes problematic as soon as the network is needed, when it is unavailable. The way the system handles this situation, depends on its connectivity model. Giguere identifies three such models [19]:

- **Always connected.** The application requires a connection. Not being able to connect to a resource is a fatal error. This is the situation that is currently offered by WAMS, offering availability and consistency by using a single database. In this model there is no partitioning possible and availability and consistency of the server are preferred as discussed in section 3.3. This is the least ideal situation, since the app cannot run without a connection and is unusable in that case.
- **Always available.** The application continues to work, but may show error messages. This situation is similar to the above situation, but does not cause the application to stop working. Instead the current state is preserved and the application waits for the network to come back. Giguere [19] does not state if changes can be made or not, but we would state this is not possible in this model, to make the differences with the occasionally connected model clear. This model changes the consistency guarantees

by allowing partitioning at the cost of consistency. Although data cannot be modified, data is also not updated with changes made on the server.

- **Occasionally connected.** The occasionally connected model recognizes that connections to resources are occasionally available. The application continues to work and synchronizes changes when the connections are available again. This is the most ideal situation. All functionality continues to work with the local data, although the data is not updated on the server and the client until the network is back online. The consistency guarantees are the same as those of the "Always available" model. Increased availability and partitioning is offered at the cost of consistency.

The occasionally connected model is desirable for many modern mobile applications to offer the best experience for users. This model requires logic on both the client and the server to work correctly. Getting this logic right is hard, even for experienced developers. Typical challenges identified by Burckhardt et. al. include [11]:

- **Data Representations.** Building mobile applications usually includes the building of web services using databases and sending data to mobile clients. There is a constant change in the representation of the data, varying from SQL to JSON to plain objects as represented by the language of choice. This forces application programmers to deal with subtle platform differences between clients and servers.
- **Consistency.** Multiple clients can modify local data while disconnected. This can lead to conflicts when these changes are synchronized and conflict resolution code needs to be written to handle these conflicts and to make sure we do not lose changes.
- **Offline Changes.** Supporting modifications in offline mode requires a log of changes made offline. These changes need to be stored and synchronized to the server when the application is reconnected. These logs can grow rapidly and resolving conflicts for a large number of changes can be a difficult problem.

Most (M)BaaS systems take care of the representation of data already in their frameworks and offering a solution for this challenge is one of their goals. The other two challenges are hard to get right and expensive to build and it is therefore not surprising that lots of applications do not offer this. Solving this in a flexible and general way can be one of the reasons for developers to choose an (M)BaaS provider for their applications.

4.2 Data Conflicts

In an (M)BaaS system, the clients are independent and do not share a common clock. Therefore changes made on different clients are considered concurrent. The server is unable to determine for two different changes coming from two different clients which one occurred first. By default the server does not do anything to handle concurrent updates and the last write will be preserved, effectively resulting in a "last write wins" strategy. This means that the client that sends an update to the server last has its change applied.

When two changes are made to two different data items, the changes can directly be applied without the chance of data loss. As soon as the changes are made to the same

data item, we are likely to lose information, because the server will apply the changes sequentially and will therefore overwrite one of the changes with the other if we do not take precautions. Note that in the case of two normal connected clients that do not have offline data support, it is also possible that they edit the same data resulting in a conflict. When such a state is reached, either data is lost or we have a conflict and we need to resolve it.

When an application depends on remote data, we can keep parts of the data locally on the client to accommodate for situations where the application loses the network connection. When we are offline, the chance of a conflict occurring is even more likely. The user can make changes to "stale data", data that has already been changed on the server. When this client synchronizes we have the same problem as above that data loss can occur and the data item is in conflict state.

4.3 Caches are not Transparent

In addition to conflicting changes made to data, implementing an efficient and transparent cache is not trivial. We define a transparent cache to be a cache where data is inserted independently of the request that was used to retrieve it.

Suppose we have product data on the server and we request all products with a price lower than 20 euros. The server returns the products and we cache them locally in case the application goes offline. When the application is offline and again requests all products with a price lower than 20 euros, the cached result will be returned, but when all products with a price lower than 10 euros are requested no data is returned, since there is no cached response. Although there is no cached response, all data required to respond to this request is already cached, because the set of products with a price lower than 10 euros is a subset of the products with a price lower than 20 euros.

The problem is that caches are usually not transparent. Those caches are implemented as a one-to-one mapping between a request and a response. When there is no mapping for a request there is no response. This approach leads to a less than optimal use of data in the cache and can also cause a cache to be filled with lots of duplicate data items when the responses for certain requests have a high overlap.

A second problem arises when an item, which is cached as part of the response for a request, is modified. The old item is still part of the cached response for the request. This can lead to an application showing an outdated item that the user has clearly updated recently. For example a deleted item can still show up in a list of items, because it still exists in the cache.

4.4 Duplicated Data Retrieval

When an application is closed, any data retrieved by an application and stored in memory is usually destroyed. When the application is started again, the same data or parts of the same data need to be retrieved again.

This “throw away and retrieve again next time” approach can have a large impact on the data plan of a user and also on the load on the server that makes the data available. A

simple solution would be to store the response on disk and load it from disk the next time the application is started. Although this seems like a simple solution to implement, it is not that easy, because there might still be new data on the server that needs to be retrieved. In that case the two responses need to be merged and a single call now results in two responses, one from the cache and one from the server.

The logic to implement this reliable and correct can be difficult to get right and this might be a reason that many applications do not offer this functionality.

4.5 Summary

Based on the problems identified above, the goal with the solution in this thesis is to develop an offline data and synchronization solution that addresses at least the following challenges:

1. **The occasionally connected model.** The occasionally connected model, which is preferable for mobile applications, offers challenges with respect to consistency of the data. Data needs to be synchronized and changes need to be tracked reliably.
2. **Conflict resolution.** Because of changes made when offline and on multiple clients, conflicts can occur during synchronization and can lead to data loss.
3. **Cache transparency.** The data cached locally is associated with specific requests and not reused for other requests. Cached responses may overlap with data required for other requests, but cannot be reused because the client has no knowledge of this.
4. **Duplicated data retrieval.** Due to application restarts we might use more bandwidth than necessary to retrieve the same data over and over again.

Chapter 5

Offline Data and Synchronization Design

So far we have identified the problems that we want to solve with the solution in this thesis in chapter 4. In this chapter we will discuss the design of a solution that tackles these problems. This includes choosing a synchronization model that works to solve the problems on an (M)BaaS system, where we have a central server to synchronize with. We will refer to this solution as "BaaSSync".

The result of this chapter will be a detailed design that we can apply onto Windows Azure Mobile Services in chapter 6.

5.1 Choosing a Synchronization Model

At the core of the synchronization solution is the synchronization model. Several synchronization models are available and have been discussed in section 3.1. We will list their advantages and disadvantages below. They can also be seen in table 5.1.

Wholesale synchronization is the easiest option, but requires unnecessary amounts of bandwidth, since we need to send all data every time we synchronize. On a mobile device this data usage might not only have an impact on the data plan of the user, but also on the energy usage of the device.

Mathematical synchronization can be computational intensive. In the case of using polynomials it requires multiple request to the server to reach very high probability of synchronizing all changes [34]. On mobile devices with slow connections we want to keep the number of requests as low as possible. These requests also increase the amount of API calls a client makes and therefore has impact on the number of clients an application can handle on a subscription. The approach taken by Choi et. al. [14] requires large amounts of tables stored on the server. The exact amount is linear dependent on the number of clients.

Status flag synchronization works well to keep track of locally made changes on a client, but on the server, keeping track of status flags for every client can quickly grow out of control and requires every change on the server to update all status flags for every client. This is clearly not a viable solution on the server side, since the amount of space used

increases linearly with the number of clients. On the clients, status flags allow us to track multiple changes to a single object as a single change.

Timestamp synchronization works well to keep track of changes in general. A disadvantage is that the type of change is not recorded and that delete operations are tricky. An option to track deletes is to clear the timestamp when an item is deleted, but this causes the timestamp information to be lost. However, timestamps work significantly better for a central server, because only a single timestamp per data item needs to be saved to make it work. For clients it is also very important that their clocks are correct for timestamp synchronization to work correctly.

Log synchronization works for keeping track of local changes made while offline, but will use additional disk space for storing the changes. It also does not allow us to collapse multiple changes to a single object into a single change. On the server we will still need to maintain information about then the log was last synchronized with a client. Log synchronization has the advantage that the changes can be replayed exactly the same on the server.

	Bandwidth efficiency	Storage efficiency	Track server changes	Track local changes	Computational complexity
Wholesale synchronization	-	+	N/A	N/A	+
Mathematical synchronization	+/-	+[1]	N/A	N/A	-
Status flag synchronization	+	+	-	+	+
Timestamp synchronization	+	+	+	+/-	+
Log synchronization	+	-	-	+	+

Table 5.1: Advantages and disadvantages of different synchronization models. "Track server changes" includes tracking changes per client. Wholesale synchronization and mathematical synchronization do not track changes. They determine changes as part of the synchronization process.

We want both our download synchronization step and our upload synchronization step to be as efficient as possible with respect to storage, bandwidth usage and number of requests. We therefore choose a variation of timestamp synchronization for download synchronization and status flag synchronization for upload synchronization. Both methods have negligible computational complexity and are guaranteed to find the precise changes made locally and remotely and therefore allow us to send and receive changes by using a single request each. The synchronization model is shown in figure 5.1.

Instead of real timestamps we use vector timestamps or Lamport timestamps [21].

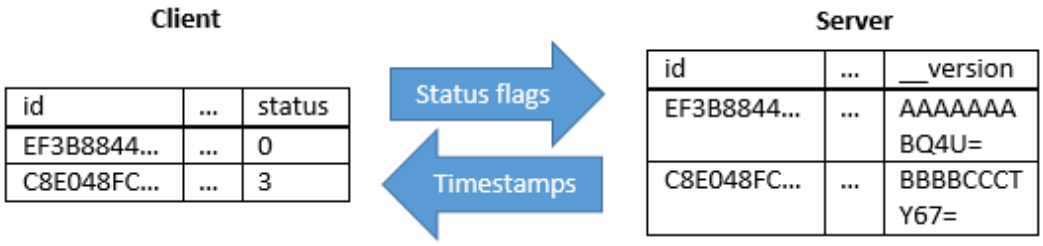

Figure 5.1: The synchronization model using timestamp and status flag synchronization.

These timestamps are incrementally generated on the server for each change to an item and should be unique per table. On the server we can use the latest assigned timestamp to determine which elements have changed since the last synchronization. These timestamps are further discussed in section 5.2.2.

To determine the local changes made on the client, we use status flags. These flags can indicate if an item was unchanged, inserted, modified or deleted. Local changes are then very easy to find. They are all the items whose status flag is not equal to "unchanged". Multiple changes to the same item will result in a single change being recorded and the same holds for updates and deletes to a locally inserted item.

5.2 A Database Model for Synchronization

For the chosen synchronization model to work correctly, some changes have to be made to the structure of the data we store and the information we record for each data item.

5.2.1 Distributed Identity

To be able to support creating new data items locally on a client when it is offline, we need to use an identity that can easily be created on a client while offline, without assigning the same identity twice on two different clients. A traditional incremental number identity such as integers is not suitable in this case because we cannot determine the next integer without negotiating it with all the available clients, requiring them to be online. A distributed identity is needed that can be generated independently and uncoordinated on a device, with a vanishing small probability of generating a duplicate.

Object identities can usually be divided into two categories. They are either natural identities or surrogate identities. Natural identities or identifiers are part of the data itself, for example a Dutch address is uniquely identified by a combination of postal code and house number. Windows Azure Mobile Service by default uses an "id" column, which is a surrogate key. Surrogate keys are not part of the data itself. Since we do not know if the data that will be inserted has a natural key we force a surrogate key for our distributed identity. There are several options for distributed identity, each with advantages and disadvantages [26].

- **GUIDs.** GUIDs or UUIDs are 128-bit identifiers and are commonly displayed as 32 hexadecimal digits with groups separated by hyphens, such as "{E3FCF896-7D3E-4298-865B-9B817DEC5094}". The UUID format has been standardized by the Internet Engineering Taskforce (IETF) in rfc4122 [23]. GUIDs are usually generated from random numbers and sometimes have the first six bits fixed. GUIDs are never 100% guaranteed to be unique. This can simply be proven by using the pigeonhole principle. As soon as we generate $2^{128}+1$ GUIDs, the probability of a duplicate would be 1.
  Note that the generation of 2^{128} GUIDs on a billion computers that generate a billion GUIDs per second, would still take approximately 10790283070806 years to complete. This is more than 700 times the age of the universe. Much more interesting is the number of GUIDs we need to create to reach a 50% chance of having one duplicate or collision. This problem is similar to the problem of finding the number of people needed to have a probability of 50% that two of them have the same birthday. It turns out that this number is much lower than many expect and the phenomena is therefore known as the birthday paradox [10]. To calculate the number of GUIDs needed to have a probability of 50% to have a duplicate, we can use an approximation formula for the generalized birthday problem (5.1) where $p(n,d)$ is the probability of having at least one duplicate in n elements out of a set of d elements.

$$p(n,d) \approx 1 - e^{\frac{-n(n-1)}{2d}} \tag{5.1}$$

$$p(n,2^{122}) \geq \frac{1}{2} \Rightarrow n \approx 2714922669395445312 \tag{5.2}$$

  Substituting 2^{122} for the number of elements d (first six bits fixed), we get an n of 2714922669395445312 (5.2). If we assume a world population of six billion people, each one of them could have 452487111 GUIDs and the probability of a single duplicate would still be only 50%.
- **Time-based/network-based identifiers.** Identifiers can be generated based on a combination of the current time and the MAC address of the machine. MAC addresses are assigned to network adapters by the manufacturers and they use a system that ensure the addresses are unique. Therefore an identifier containing this address would be globally unique. However especially in virtualized environments MAC addresses are generated locally and therefore not necessarily unique anymore. A second problem is the case where multiple processors generate an identifier at the same time, effectively resulting in the same identifier. This can happen when multiple applications run on the same device or when the system time is changed.
- **Hierarchical identifiers.** Hierarchical identifiers usually consist of a number generated locally on a device combined with a global client identifier. The global identifier is assigned by a central server to the client. As long as both parts are guaranteed to be unique, the resulting number will be globally unique. In this case the server needs to send the global id to the client, because the client needs to be able to generate the identifier locally. This adds a challenge to the server, because it now has to track

clients. This tracking is fragile because not every client platform offers the capabilities to uniquely identify an application or identities might change due to re-install, removal of temporary data and other events.

In BaaSSync we use GUIDs, since both of the alternatives have problems that are not easily solvable and GUIDs are the most robust. The chances of a collision are negligible.

5.2.2 Object Versioning

Each item will be assigned a vector timestamp, which can be seen as the current version of the item and is unique per table. In the rest of this thesis will use the term "version" to reference to these vector timestamps. These timestamps will have two purposes:

- Detect changes for download synchronization.
- Detect conflicts when an item is modified or deleted.

When download synchronization is performed, we include the most recent assigned timestamp in the response for a request. The next time the client sends the same request, it can include this version number in the request to receive only the changed items since the last time the request was sent. This includes additions, modifications and deletes.

When an element is changed we can use the timestamp to check if the item has not been changed on the server by other clients while this change was made. In other words, is the current element on the server the same as the element known by the client when the change was made?

5.2.3 Tracking Deletes

When an item gets deleted, we cannot simply remove that item from the server. There might still be clients who are working offline with this item and need to be notified of the deletion when they synchronize. Instead of removing a deleted item from the database we have to put it into a deleted state indicating the item was removed. This can be done by recording its state in a special flag on the server.

5.3 Flexible Conflict Resolution

As discussed in section 4.2, conflict resolution is a very important aspect of any synchronization solution. Conflicting modifications are detected by checking if the versions of the incoming item and the item in the database match or not. When they are different a conflict occurs. Modification of a deleted item is detected by checking if the item is in the "deleted" state. In section 3.2 several different strategies are listed that can be used to resolve conflicts. These strategies are useful in a variety of situations, but not one of them can be considered to be always the best strategy, since they in some situations can lead to data loss or duplicated data.

Our proposed solution, BaaSSync, only handles semantic conflicts and the use of the word "conflict" in the rest of this thesis will refer to this type of conflict. Natural conflicts

are not handled. For example two changes to the same property of an item by two different clients are detected as a semantic conflict, but two insertions of the same address in a table of addresses are not detected and are considered to be a natural conflict of which the system has no knowledge. The cases of natural conflicts can be handled by writing custom logic on the server and their occurrence is not dependent on support for data synchronization and offline data.

In the case of offline clients adding, modifying and deleting data items, conflicts only occur in some cases. The creation of a new item on a client that is offline never leads to a conflict on the server since the item is new. Therefore offline insertion is always safe for conflicts. Modification of a data item on a client that is offline, leads to a conflict if the item on the server is modified before the offline client synchronizes its changes. This conflict is an update-update conflict. In this case a conflict resolution strategy needs to be applied on the server or the server can reply with a conflict response and allow the client to solve the conflict locally. Resolving a conflict locally on the client is useful because it allows cooperation from the user.

The deletion of an item leads to a conflict when the item has changed on the server and gets deleted on a client that is offline or when it is deleted on the server while it gets modified on a client that is offline. The first situation is an update-delete conflict and the second a delete-update conflict. Multiple deletes are a delete-delete conflict, but are in general not considered a conflict because the end result is the same. The way these conflicts are handled is the same as the case of conflicting modifications.

5.4 Transparent Cache

A persistent cache is crucial for any offline data solution. Besides making data available when offline, it can also help saving bandwidth by using data from the cache instead of sending requests for it multiple times as described in section 4.4.

To make the cache as useful as possible, the cache should be a transparent cache that does not have the problems described in section 4.3. To achieve this, we can insert data items separately in a local database and use this local database as the cache. Using a database as the cache has the advantage that items are no longer stored per request, but independently of the request. The database also helps by storing every item only once. Storing the same item multiple times will preserve only the last stored item.

The database allows us to execute queries directly on the local database. Every (M)BaaS provider supports some form of querying and a database allows us to continue to support queries locally and while we are offline. The database also ensures the result includes all data we have available including data that has been stored in the context of another query. Furthermore, it facilitates applying changes to the items in the database and including these changes in the query results. This guarantees us that we never return stale data that has already been changed on the same client.

5.5 Synchronization Algorithm

This section describes the algorithm that is used to synchronize changes. The synchronization algorithm keeps the local database synchronized with the remote server. When a request is made the algorithm executes the following steps:

- Synchronize all items whose status flag is not “unchanged”.
- If the request is a read request, send the request to the server including the last version that is stored for the request URI and insert the response into the local database.
- If the request is not a read request, send the request to the server and insert the result into the local database.
- Replay the request to the local database and return the result.

In the case when we are offline, the steps that send the request to the server are omitted.

For subsequent requests, the version that is received with the last request is included in the URI. This allows the server to only send modifications since the last time the request was sent. Although the algorithm is only sending changes for the same request, it may send duplicate items when the result sets of two requests overlap between two different requests. We do not know which items are already cached for other requests, so the responses for different requests might overlap in the response data returned.

Because conflict detection is done on the server, the synchronization algorithm first pushes changes to the server, before pulling changes from the server. This sequence is necessary, because, if we first pulled changes from the server, we could overwrite local changes, which could have been conflicted changes on the server.

When the application is offline, the algorithm needs to handle the following situations that can occur when changes are made in offline mode.

- A new item is created locally and then updated. After the update the item should still have the “Inserted” status instead of been changed to “Changed”. This is needed since the item is still a new item for the server.
- A new item is created locally and then deleted. After the delete the item should be deleted and the server should never know about its existence. We need to delete the item locally since we cannot send a delete request for an item that the server does not know about.

5.6 Eventual Consistency

In section 4.1 we saw that adding offline data support to an (M)BaaS system and thereby making it an occasionally connected system, gives increased availability and is trivially causing partitioning to occur. But being unable to communicate with the server comes at the cost of consistency. Inconsistencies can occur between the local cached data and the data stored on the server. These inconsistencies do not cause invalid responses, but the responses can contain stale data. Therefore, BaaSSync changes the consistency model from a strong consistency model to a weak consistency model. This means we have to drop at

least part of goal **G3**. We cannot have the same consistency guarantees in BaaSSync as in the normal WAMS offering.

Nevertheless there are some properties we can guarantee about the consistency model for our solution. Assuming every client will eventually come back online and synchronize, we reach eventual consistency [36]. When no other changes are made to an item, eventually all clients will see the same item. We can make the consistency model even stronger by guaranteeing that changes made by an offline client to its local data will be reflected in subsequent reads and queries on that same client. A transparent cache is key in offering this functionality. This type of consistency where subsequent reads include changes made by the same client, is called read-your-writes consistency. In addition the offline system also offers monotonic-read consistency. When an updated value has been seen by a client it will never return older values of that same item. Vogels identifies these two consistency guarantees as the most desirable [36].

5.7 Summary

In this chapter we discussed our design for a synchronization solution for an (M)BaaS system called "BaaSSync". This solution will have the following characteristics:

- **Timestamp synchronization combined with status-flag synchronization.** As discussed in section 5.1, BaaSSync will use timestamp synchronization on the server and status-flag synchronization on the clients.
- **Database model.** In section 5.2, we identified the properties that our data items need to have. Each data item will have a GUID identifier and a version timestamp to identify changed items and to detect conflicts. In addition, the server will record if an item is deleted and the client will record a flag indicating the status of the item with respect to the corresponding server item.
- **Conflict detection and resolution.** Flexible conflict resolution options are at the core of the solution as we have seen in section 5.3. BaaSSync allows resolution to happen on the server or on the client. The detection of a conflict always takes place on the server.
- **Database cache.** To allow local querying and cache transparency, we discussed in section 5.4 to use a database as the cache.
- **Synchronization algorithm.** Our synchronization algorithm discussed in section 5.5 keeps our local database synchronized with the backend and always pushes changes first to detect conflicts. When synchronization is completed it replays the request on the local database.
- **Read-your-writes consistency.** The database cache can be used to offer a specific form of eventual consistency, called read-your-writes consistency as discussed in section 5.6. This allows offline clients to immediately see their changes in the result sets.

Chapter 6

Implementation on Windows Azure Mobile Services

In chapter 5 we developed a solution to add offline data and data synchronization to an (M)BaaS system called "BaaSSync". We defined the steps the algorithms should take, the conflict resolution options and the working of the local cache.

In this chapter we will implement BaaSSync on top of Windows Azure Mobile Services (WAMS). The source of the implementation is publicly available on Github[1] under the Apache license. The implementation includes SQL scripts to change the database tables into the correct schema, server scripts and an extension for the Mobile Services .NET Software Development Kit (SDK).

6.1 Overview

The implementation of BaaSSync exists of two parts. A high level overview of the components can be seen in figure 6.1.

- **Server-side table scripts.** These scripts handle the synchronization on the server. This includes all requests made to a table with synchronization enabled. This is where conflict detection happens and conflict resolution can be done.
- **Client-side extension.** The client extension library hooks into the WAMS .NET SDK to intercept requests and forward them to the synchronization layer. This extension library is used by an application to extend the functionality of the WAMS .NET SDK. The client extension library is also the part of the implementation that initiates synchronization and caches data locally.

[1] `https://github.com/jlaanstra/azure-mobile-services`

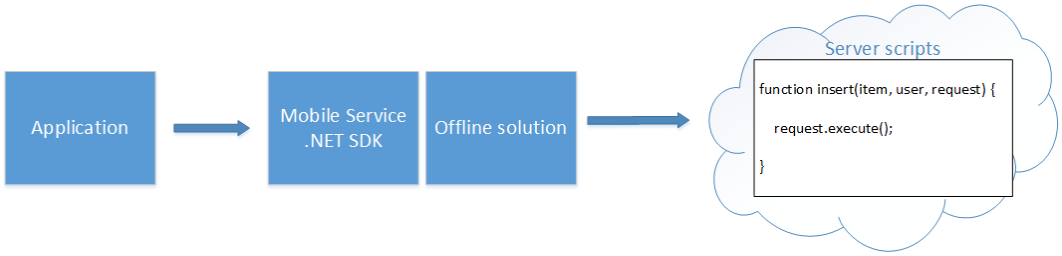

Figure 6.1: High level overview of the different components of the BaaSSync implementation.

6.2 Database Schema

To implement the synchronization model discussed in section 5.1, we have to change the database schema as provided by WAMS when a new service is created. A script to perform the changes is provided as part of the implementation on Github.

For the synchronization solution we need an "id" column that can store GUIDs, a "version" column to store the current version of an item and an "isDeleted" column to keep track of deleted items.

In November 2013, the WAMS team released a feature called "Optimistic Concurrency" (OC) [2]. This feature allows for detection of multiple updates to the same item and therefore allows the detection of update-update conflicts. As discussed in section 4.2, conflicts can also occur in a situation where all clients are connected and the OC feature allows detection of these cases, but only for an update-update conflict.

Before the release of OC, entries in a WAMS table had a unique integer identity. The OC feature added string identities, which we can now use to store our GUID ids.

To keep track of the version of an item in a WAMS table we add a column to every item that contains this value. The SQL Server database that is used by WAMS has a datatype that is exactly meant to provide this type of functionality, called "rowversion". The "rowversion" datatype in SQL Server consists of 64 bits and is unique per database. Each time an object in the database is updated, the database automatically generates a new version number for the object. These version numbers are only generated on the server. SQL Server also records the latest assigned version to a database. We can use this latest version number as the latest version assigned to an item in the database and therefore as the version of the whole database. This version number will be included in the response for a request as discussed in section 5.2.2. This "version" column was also added by the OC feature.

As discussed in section 5.2.3, we cannot remove items when they are deleted. Instead we put those items in a deleted state by adding an additional "isDeleted" column with a boolean datatype. This boolean will indicate if an item was deleted and these items have to

[2]More info about the Optimistic Concurrency feature can be found at `http://blogs.msdn.com/b/carlosfigueira/archive/2013/11/23/new-tables-in-azure-mobile-services-string-id-system-properties-and-optimistic-concurrency.aspx`

be filtered out of the response when necessary. The OC feature did not include this change to the database schema. Therefore the implementation on Github includes a SQL script to make this change to the database.

After the required changes are made to the database schema to support the synchronization model, the basic schema will look like the schema in figure 6.2.

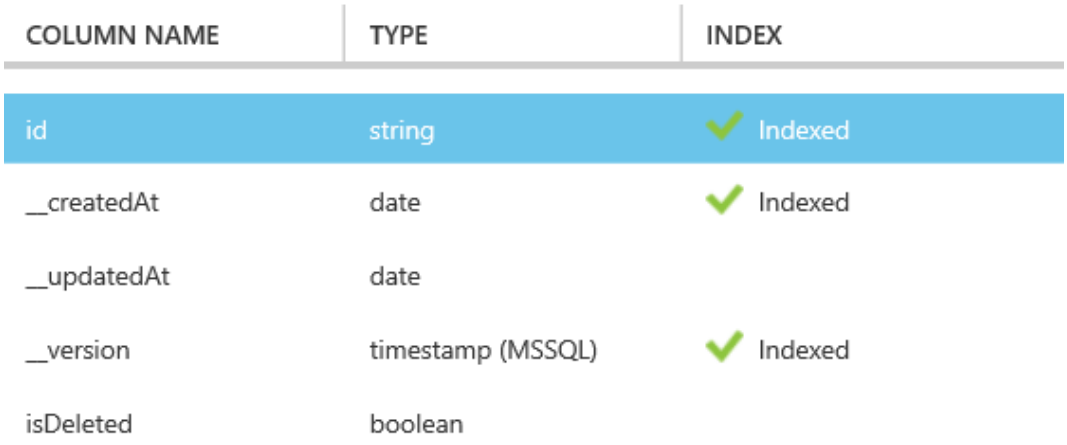

Figure 6.2: Database schema with synchronization changes applied.

6.3 Server-side Scripts

As explained in section 2.2, server scripts can be used to execute arbitrary code before and after sending the request to the database. This is useful for BaaSSync, since it allows us to put the server part of the solution in those scripts. We can use the scripts to check if the incoming requests are valid, to check if conflicts occur and to make sure delete operations do not actually delete items but only change the value of the "isDeleted" column. It also allows us to use a different response format to include additional info regarding the result of the operation, for example when a conflict occurs.

6.3.1 Response Format

The default response format used by WAMS tables depends on the operation that is executed and can even depend on specific parameters sent by the request. Read operations for example return an array while insert or update operations return a single item as the result.

The response formats that return an array or a single item do not easily allow the inclusion of additional information regarding the request. Therefore the implementation of BaaSSync changes the response format that is sent by the server to be the same for all operations independent of parameters sent by the request and to include a top level object

```
{
  "__version": "",
  "results": [],
  "deleted": [],
}
```

Listing 6.1: The new response format used by the implementation of the BaaSSync solution. The root is a JSON object containing a "results" array, a "deleted" array with ids of deleted items, a "__version" property and optionally additional properties.

```
https://syncservice.azure-mobile.net/tables/products?version
    =AAAAAAAA5Xo%3D
```

Listing 6.2: Read request with version parameter.

that can include additional information regarding a request. The response format used by BaaSSync is shown in listing 6.1.

The response can contain a count property if this is requested in a query using the "inlinecount" feature of the OData query syntax that is discussed in section 2.3.

WAMS has the concept of "system properties". These properties start with two underscores and are unique for each item. By default they are not included in the responses. These properties include the "__version", "__createdAt" and "__updatedAt" properties. They normally can be requested explicitly by specifying in the URI of a request that they should be included. The BaaSSync implementation changes this to always include these properties, since the "__version" property is used by BaaSSync to detect conflicts.

6.3.2 Read Script

The read script has been modified to support our bidirectional synchronization model. Read requests can specify a "version" parameter that indicates the latest version assigned in the database at the time this request was last sent to the server. By indicating this version in the URI, they can request only the changes since that version. An example of such an URI is shown in listing 6.2. Accessing URI parameters in server scripts is very easy, which is the reason why this version is put in the URI.

The read script will modify the query sent to the database to filter out all items with a version number earlier than the one specified. The ids of deleted items will be included in the deleted array of the response. The "isDeleted" property of the data items is a server-only property and will be deleted from every item, before the response is sent.

6.3.3 Insert Script

The insert script adds a check to see if the incoming data item does not already have a "__version" property specified or if it is trying to set the "isDeleted" property. New items

are unknown by the server and will therefore not have a version. Also, the "isDeleted" property can only be set by the delete script on the server.

The insert script also handles the case where we have an id collision by returning an error. We cannot easily assign a new id to the item on the server, because this might break mappings on the client using the old id. Although we discussed in section 5.2.1, that the chance of a GUID collision is extremely small, we still guard against this possibility. The rest of the work in the insert script consists of building the response and removing the "isDeleted" property.

6.3.4 Update Script

The update script performs similar checks as the insert script, but in this case it requires the incoming item to have a "__version" property. It then continues to execute the request. This can result in two situations:

- When the update completes without a conflict, it builds the response just like the other scripts and includes the modified item in the "results" array of the response.
- When the update causes a conflict, the conflict function passed into "request.execute()" gets called which triggers conflict resolution. After the conflict has been resolved on the server, the response is built just like in the other case. Optionally the item is sent to the client for resolution. Conflict resolution is discussed in more detail in section 6.5.

6.3.5 Delete Script

The delete script never actually executes the delete request, because it never deletes data. The delete script executes a SQL statement that changes the value of the "isDeleted" column to "true". This can result in two situations similar to the update script:

- There was no conflict and the id of the deleted item is sent back in the "deleted" array of the response.
- The delete operation results in a conflict and conflict resolution is triggered. Note that deletes can result in conflicts in the same way as updates can as discussed in section 4.2. After the conflict has been resolved on the server, the response is built just like in other cases. Optionally the item is sent to the client for resolution. Conflict resolution is discussed in more detail in section 6.5.

Although data is not deleted by the script, for sensitive information it is possible to remove all the object data from the database. As long as the "id", "version" and "isDeleted" columns are left untouched, the solution can continue to work normally.

6.4 Client-side Extension

The client-side extension library in the BaaSSync implementation has been written only for the WAMS .NET SDK. It hooks into the Mobile Services SDK in a way that is offered by

```
1 // Configure your mobile service
2 MobileServiceClient MobileService = new MobileServiceClient(
      Constants.MobileServiceUrl, Constants.MobileServiceKey);
3 // Enable offline data for tables
4 MobileService = MobileService.
      UseOfflineDataCapabilitiesForTables(new
      WindowsNetworkInformation());
```

Listing 6.3: Adding the BaaSSync implementation to an existing `MobileServiceClient` can be done with the single line of code at line 4. The other line of code would also be needed when the solution proposed in this thesis is not used.

most of the other SDKs by using an `HttpMessageHandler`. This is one of the few extension points that is offered by the SDK out of the box.

Adding BaaSSync to an application that already uses WAMS is straightforward and requires only a single line of code as shown in listing 6.3.

6.4.1 `CacheHandler`

The Mobile Services .NET SDK supports an extension point via an HTTP pipeline. This pipeline was originally custom built and later migrated to make use of the built-in HTTP pipeline of the `HttpClient` class [3]. The `HttpClient` class uses `HttpMessageHandler` instances to process requests as can be seen in figure 6.3.

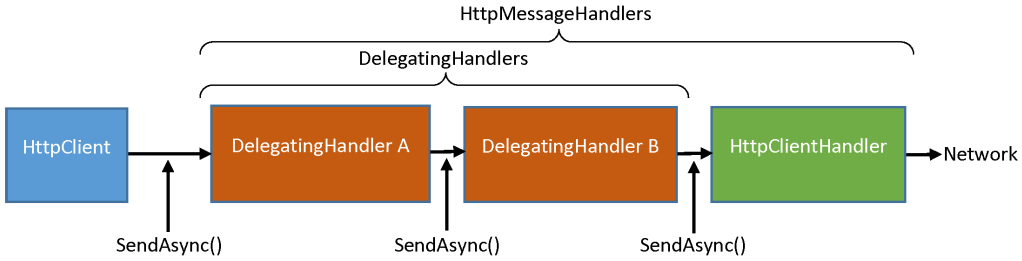

Figure 6.3: Architecture of the `HttpClient` class. It uses an `HttpClientHandler` by default and can be extended with one or more `HttpMessageHandler` instances to modify how requests are handled.

The default handler that is used by the `HttpClient` class is an instance of `HttpClientHandler`, which sends the request over the network and gets the response from the server. Between the `HttpClient` class and the `HttpClientHandler` class one can

[3] `http://www.johanlaanstra.nl/2013/03/13/preview-of-the-new-managed-client-for-windows-azure-mobile-services/`

insert custom message handlers to change how requests are handled. These message handlers will in most cases be one or more instances of the `DelegatingHandler` class. These `HttpMessageHandler` instances are plugged together in a "Russian doll" model where each handler delegates to the inner handler until the last one hits the network or a previous one short-circuited the path.

The synchronization library uses this `DelegatingHandler` pipeline to hook into the Mobile Services SDK. It inserts an instance of the `CacheHandler` class that intercepts the requests and determines if the current request should be send to the cache or to the server. The actual cache implementations are provided by implementations of the `ICacheProvider` interface. The `CacheHandler` instance also provides the cache implementations with access to the network to push and pull changes to and from the server via an instance of the `IHttp` interface.

6.4.2 `ICacheProviders`

The cache implementation used by the `CacheHandler` can be one or more classes implementing the `ICacheProvider` interface. When multiple cache providers are specified, the first cache provider that returns true for `ProvidesCacheForRequest(Uri)` method will be used as the cache for the request. This method is an implementation of the Strategy pattern [18]. The interaction between the `CacheHandler` and the `ICacheProvider` instances is shown in figure 6.4. The definition of the interface can be seen in listing 6.4.

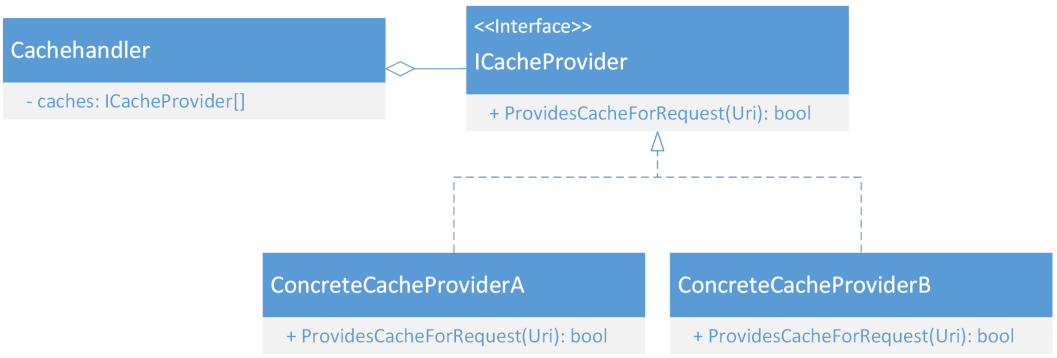

Figure 6.4: The `CacheHandler` instance implements the Strategy pattern by using one or more `ICacheProvider` instances (strategies) for offline data support. A request will be handled by the first `ICacheProvider` instance that return "true" for the `ProvidesCacheForRequest(URI)` method.

The `ICacheProvider` interface can be used to implement a variety of different caches. The implementation of BaaSSync on WAMS comes with three cache providers out of the box.

- **DisabledCacheProvider.** This cache provider provides no caching functionality at all and makes it look like there is no caching solution used. There is really no reason

```
1  public interface ICacheProvider
2  {
3    bool ProvidesCacheForRequest(Uri requestUri);
4
5    Task<HttpContent> Read(Uri requestUri, IHttp http);
6
7    Task<HttpContent> Insert(Uri requestUri, HttpContent
       content, IHttp http);
8
9    Task<HttpContent> Update(Uri requestUri, HttpContent
       content, IHttp http);
10
11   Task<HttpContent> Delete(Uri requestUri, IHttp http);
12
13   Task Purge();
14 }
```

Listing 6.4: ICacheProvider interface.

to use this cache provider and it merely exists to make sure the implementation is flexible enough to support a variety of different kinds of cache providers.

- **MemoryCacheProvider.** This cache provider caches responses for a certain amount of time, but only for read requests. Requesting the same URI twice in a short time would result in the cached response being returned without hitting the network.
- **TimestampCacheProvider.** This cache provider implements the BaaSSync solution proposed in chapter 5. More specifically, it contains the part of the solution discussed in chapter 5 that runs on the client.

The `TimestampCacheProvider` implements the synchronization algorithm discussed in section 5.5. The responsibilities for storage, network information, network access and uploading and downloading of changes are moved into separate classes and used by the `TimestampCacheProvider` class via the `IStructuredStorage`, `INetworkInformation`, `IHttp` and `ISynchronizer` interfaces. Local conflict resolution can be offered via the `IConflictResolver` interface. The architectural overview of the `TimestampCacheprovider` class and the interaction with the other interfaces can be seen in figure 6.5.

The `TimestampCacheProvider` class implements the synchronization algorithm as discussed in section 5.5. The exact steps it executes per operation are documented in appendix A. The `TimestampCacheProvider` class can access the network via the `IHttp` instance that is passed to it by the `CacheHandler`. This `IHttp` instance is created by the `CacheHandler` for each request. This instance is an implementation of the Adapter pattern [18] since the implementation of the `IHttp` interface gives access to the `SendAsync(HttpRequestMessage)` method of the `CacheHandler`. The pattern can be seen in figure 6.6. The `IHttp` interface also includes the original performed request and allows for storing the response for that request.

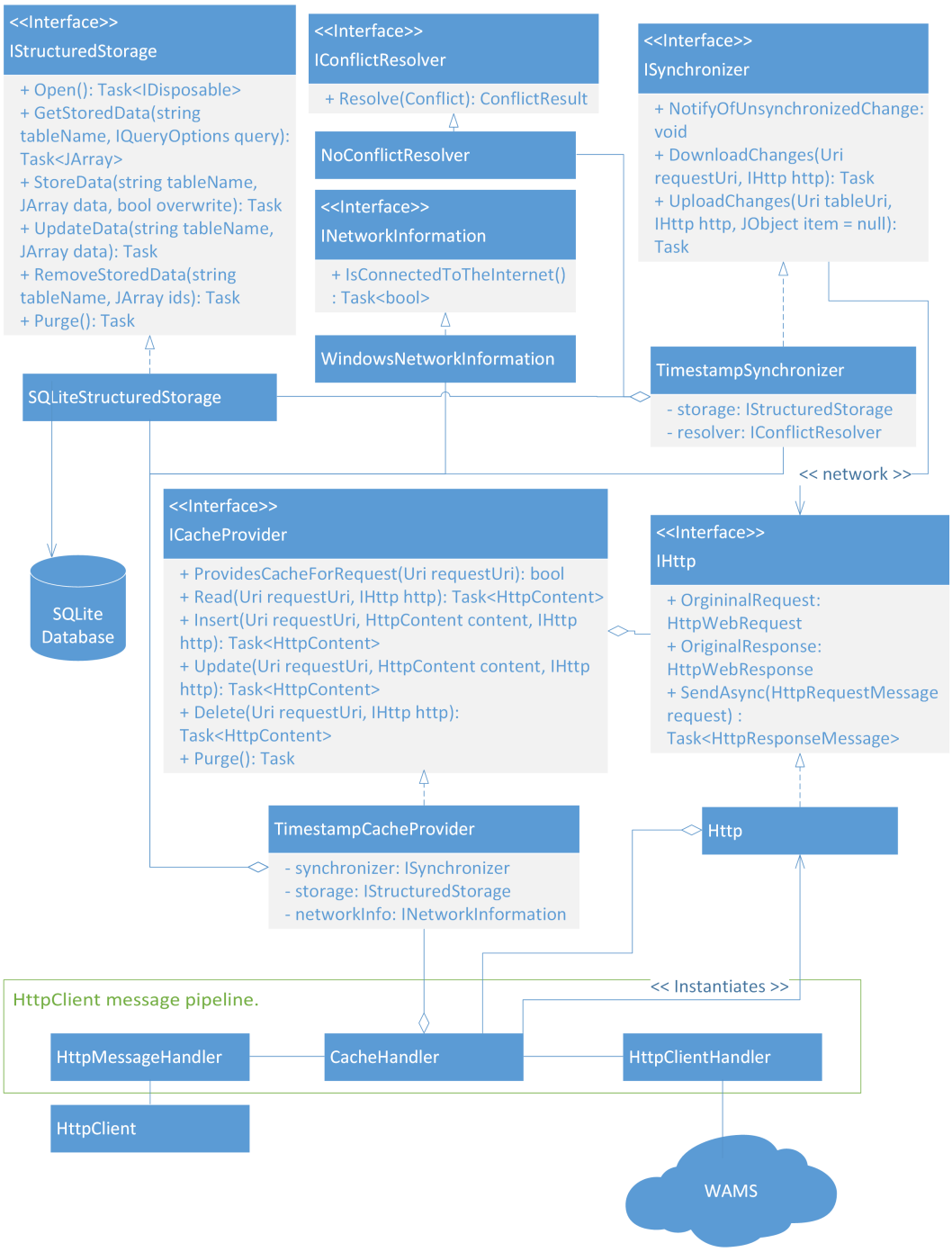

Figure 6.5: Architectural overview of the `TimestampCacheProvider`.

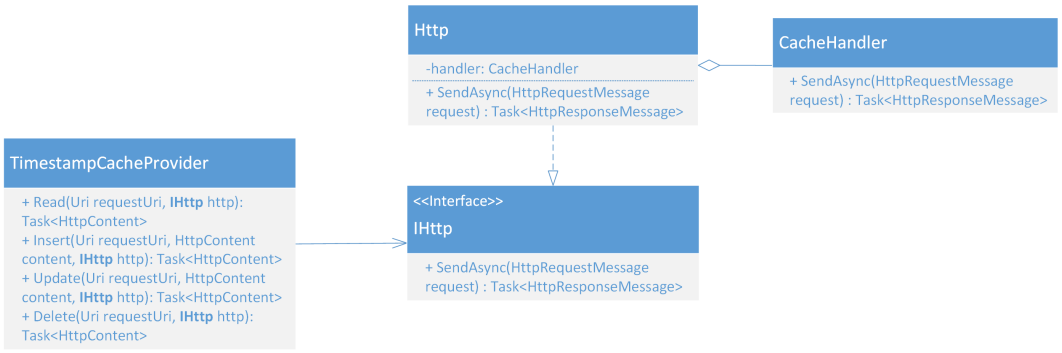

Figure 6.6: Access to the network is implemented via the `IHttp` interface (Adaptor) and an adapter class, `Http` (Adaptee), that allows access to the network via the `CacheHandler`.

The provider can access storage via the instance of `IStructuredStorage` interface that is passed to it and can acquire information about the current state of the network via the instance of the `INetworkInformation` interface. These two interfaces allow for storage specific classes and platform specific classes. For SQLite the `SQLiteStructuredStorage` class is provided which is a form of the Adapter pattern [18]. This same pattern is used to provide network information of different platforms.

The architecture of the library is highly interface based and built with the principles of Object-Oriented Design in mind as identified by Martin [27]. It separates the different responsibilities between different classes and has been designed to allow for implementations to be substituted by others. The interfaces are designed to be open for extension and closed for modification. To that end we particularly use C#'s extension methods. They allow for extending existing interfaces or classes while preventing the modifications of the internals of a class, because they can only access public APIs. Extension methods can be seen as an implementation of the Decorator pattern [18], although strictly they are not, because extension methods affect all instances.

The implementation makes enabling offline data support as easy as possible as we have seen. However, we acknowledge that some developers need more fine grained control over the synchronization setup. Therefore, instead of adding offline data support with a single line of code, more complex control is possible by letting the developer choose and instantiate the classes himself.

6.4.3 Local Database

As we have discussed in section 5.4, we will use a database to provide caching functionality. It allows us to make use of the query capabilities for local querying and it is persistent between application sessions.

On the Windows platform we do not have many database options to choose from. SQL Compact Edition [33], the mobile version of Microsoft's SQL Server, is no longer supported

and does not work in the new Windows Store applications [4]. The database option that is recommended by Microsoft for Windows Store applications, is SQLite, discussed in section 2.7. Choosing SQLite as the database for the cache has the advantage that it is very fast and that it works on all the major platforms including the Windows platform.

SQLite offers almost all SQL features which we will use to support local querying.

The local database has a "status" column that indicates the status of each item. This status property indicates if an item has been locally inserted, modified or deleted or if it is unchanged.

The schema to use for the tables in the local database is derived by making use of certain behavior of the WAMS client SDKs and server runtime. The WAMS client SDKs serialize data they received from the application into the JSON format, which is discussed in section 2.6. This serialization step does not allow for serialization of nested objects or arrays and by doing so allows the server runtime to make the assumption that items will only exist out of simple data types. These simple data types can be easily stored in a database. We can use this same approach on the client to derive a database schema for the local database.

Because WAMS uses a subset of the OData query language for querying as described in section 2.3, we can easily support the same queries offline as online. OData has been designed to map easily to SQL and we can use an LL(1) parser to translate this syntax into SQL queries that the local database understands. This allows us to support all queries locally in the same way we can query the REST table APIs.

6.5 Conflict Resolution Strategies.

Resolving conflicts is a two-step process. A conflict first needs to be detected and then needs to be resolved.

Conflict detection needs to happen on the server, as this is where items are inserted, updated or deleted. Conflict detection on the client is broken by design, because, at the moment a client receives an item to do conflict detection, this item can already have been changed by another client on the server and therefore be outdated.

Conflict resolution on the other hand can be done either on the server or on the client. The choice between conflict resolution on the server or on the client can be made by the developer of the application. This choice has to be made by the scripts on the server, since that is where the conflicts are detected.

When conflict resolution is taking place on the server, a special conflict resolution function is called. For some built-in strategies these functions are provided, but essentially any function can be used as long as the function behaves according to the contract that takes two items and returns the items that need to be preserved. The resulting items are send back to the client and the response includes an additional property "resolvedConflict" that states the policy that was used to resolve the conflict. When conflict resolution is taking place on the client, the server sends back a special conflict response that includes information regarding the occurred conflict. These conflicts can be resolved at the client by an instance of the `IConflictResolver` interface. This interface contains a single method

[4] http://connect.microsoft.com/SQLServer/feedback/details/776328/port-sql-compact-to-windows-rt

`Resolve(Conflict)` that takes a conflict and returns an instance of the `ConflictResult` type indicating which items need to be preserved. The conflict response includes the version of the server item that needs to be set on the preserved item when the conflict is resolved. This version can then be used by the server to check if the item has changed since the conflict was sent to the client and has led to yet another conflict.

Different strategies can be used for conflicts resulting from a change or from a delete. Some of the conflict resolution strategies as discussed in section 3.2 do not apply to BaaSSync or are the same as others. These strategies are listed below:

- **Originator Wins.** Data always originates from a client and we do not support peer-to-peer synchronization, so this strategy would be the same as "Client Wins".
- **Recipient Wins.** Data always originates from a client and we do not support peer-to-peer synchronization, so this strategy would be the same as "Server Wins".
- **Recent Data Wins.** Within the current implementation of BaaSSync, the "__updatedAt" values are not updated locally. This means that this strategy will be equal to "Server wins". Note that this can change in the future. It is recommended not to use this strategy for critical data, since the local time of a client can be significantly of, causing data loss.

For the remaining strategies, built-in functions are provided to use them. There are functions for "Client Wins" (conflicts.clientWins), "Server Wins" (conflicts.serverWins) and "Duplication Apply" (conflicts.duplicationApply). All built-in strategies do not assume that an update-delete or a delete-update conflict should result in the deletion of the item. Instead they are handled the same as an update-update conflict.

Because of the built-in conflict resolution in the BaaSSync solution, the `MobileServicePreconditionFailedException` is no longer thrown by the SDK. All cases where this exception could be thrown are now handled by the BaaSSync implementation. This exception is normally used by the OC feature to detect concurrency problem when updating items. For resolved conflicts on either the client or the server the implementation throws a `MobileServiceConflictsResolvedException`. The reasons behind this are explained in section 6.6.3.

6.6 Implementation Challenges

6.6.1 Type Information in SQLite

As described in section 2.7, SQLite uses a dynamic typed schema and assigns a type affinity to columns. This means we can by default not rely on SQLite's type information to recreate the exact same JSON objects as we stored in it. This problem can easily be seen when trying to store booleans. SQLite stores booleans as 0 (false) and 1 (true). When we retrieve the data from the local database there is no information if this value should be a boolean or an integer. The same holds for datetime values. The only way to determine the type would be to copy the type affinity logic of SQLite in the implementation, which will break when SQLite changes the type affinity logic.

To solve this problem, type information needs to be added somewhere in the SQLite database. We decided to encode the .NET type into the type name of the column. In this way we can save the types of the properties of the JSON objects in the columns. Since SQLite also determines the type affinity based on the type name of the column, the type name will exist of two parts separated by an underscore [5].

- **Type affinity prefix.** This prefix forces a specific type affinity on the column.
- **The .NET type name.** This type string can be easily parsed back to the actual .NET type.

The type affinity prefix is based on the .NET type stored and is hard-coded in the application. If we want to store a boolean in SQLite, we would choose "NUMERIC" type affinity and the type name of the column would be "NUMERIC_System.Boolean".

6.6.2 Local Queries

In appendix A, step 5 of the read algorithm, the OData query string of the request is parsed into an expression tree. The OData queries can be easily parsed by an LL(1) recursive decent parser.

"ODataLib" [6] would be a great tool to use, since it is specifically built for this purpose. On of the limitations of "ODataLib" is that it requires strong typed classes to define a strong typed expression tree. At the level where BaaSSync is implemented, we do not have information about the model of the data via strong typed classes. This effectively prevents us from using "ODataLib" and therefore we have to build a parser and a custom expression tree ourselves.

To parse the OData queries in the URI, we use our own parser that is based on the parser of "ODataLib" and produces an `ODataExpression` class that contains the expression tree. This expression tree is passed to the `IStructuredStorage` interface. The `SQLiteStructuredStorage` class uses a `SQLiteExpressionVisitor` class that extends the `ODataExpressionVisitor` class to visit the expression tree and translate it into SQL queries that the SQLite database understands. The `SQLiteExpressionVisitor` understands both the expression tree and SQLite statements and can therefore build a SQL statement out of it. This expression visitor design is based on the Visitor pattern [18]. How the Visitor pattern is implemented by the `ODataExpression` and `ODataExpressionVisitor` classes can be seen in figure 6.7.

Other storage implementations can create their own extension of the `ODataExpressionVisitor` class to translate the expression tree.

6.6.3 Conflict Resolution Special Cases

An update operation executed via the Mobile Services SDK expects a single item to be returned as the result. As long as no conflicts occur, this assumption is correct and the

[5] In fact this could be any character that is not part of a type affinity and valid in a SQLite type name, since we will split on the first occurrence of the character

[6] http://odata.codeplex.com/wikipage?title=ODataLib

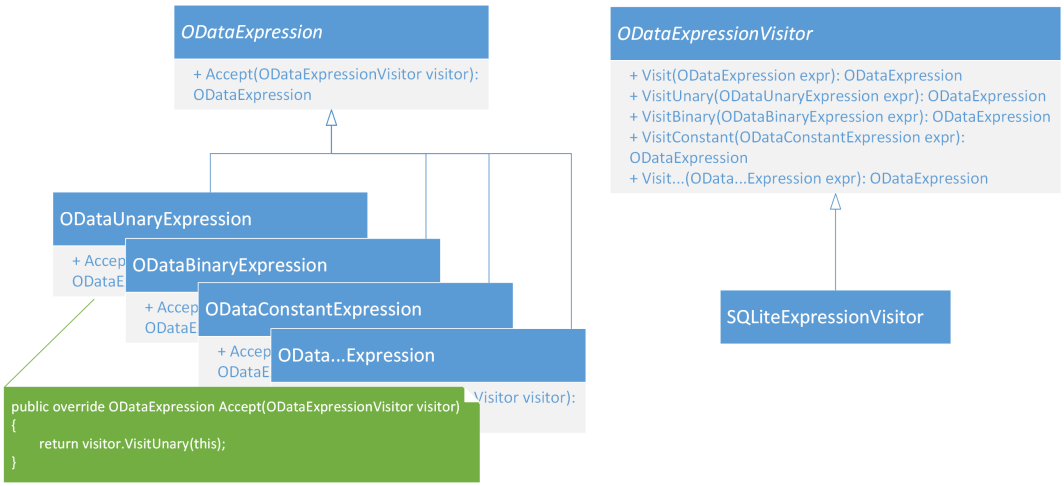

Figure 6.7: The translation of queries is performed by instantiating an `ODataExpressionVisitor` class and visiting each part of the `ODataExpression` to build for example a SQL statement.

synchronization layer can return a response as expected by the SDK. As soon as a conflict occurs, this is not necessarily the case.

In the case of a delete-update conflict, the result of an update operation can be that the item gets deleted and no item is returned or that multiple items are returned. This causes incompatibilities with the SDK, because it expects a single item. Therefore in the case of resolved conflicts either on the client or on the server, indicated by the "resolvedConflict" property in the response, the synchronization layer will throw a `MobileServiceConflictsResolvedException`. In this way we can work around the fact that the SDK expects a certain format.

Chapter 7

Evaluation

Now that the implementation of the BaaSSync solution is completed in chapter 6, we can continue with the evaluation of the implementation and the solution.

In this chapter we will describe the methodologies that will be used to evaluate the solution. BaaSSync will be evaluated based on if we reached the goals set in section 1.3. These goals include evaluating the consistency of the algorithm and the data **G3**, evaluating the conflict resolution strategies **G4**, answering queries that we have not seen before with data from other queries **G5**, having fast response times in both online and offline mode **G6** and saving bandwidth **G7**.

We already stated in section 5.6 that we could not meet our strong consistency goal **G3**, which we have by default without using BaaSSync. The results of the evaluation can be found in chapter 8.

7.1 Unit Testing

We will use unit tests to verify that our implementation of BaaSSync works as expected and that the quality of our code is high. Although it is difficult to unit test every part of the implementation, especially the I/O parts, we will write unit tests whenever possible.

These unit tests primarily serve to verify that the individual components work correctly and to detect regression when parts of the code change. They test if the individual components handle the different synchronization cases correctly, including no connection and conflicts. The goal for code coverage is, to cover the optimistic execution path. This means we will not check if every incorrect argument passed to a method will throw an `ArgumentException`, but if correct arguments result in the correct response. The mocking framework "Moq" is used to create simulated dependencies of the classes and interfaces an object depends on. These mock objects mimic the behavior of real objects where necessary. We also use the ability of "Moq" to verify if certain method calls occurred.

The integration of the different components including the interaction between the client extension library and the server scripts will be tested separately with integration tests described next.

7.2 Integration Testing

A special test application will test the integration of the BaaSSync implementation with the Mobile Services .NET SDK and the corresponding server part. This application allows us to test the server scripts including the different conflict resolution strategies. It will also test the synchronization algorithm as a whole and check if the data that is returned is correct and expected. This test application will use a list of "Product" objects which will be inserted, modified and deleted. Every type of conflict will be created by sending some requests multiple times while modifying different properties.

Integration tests will be created per goal. There will be groups of tests to verify each goal:

- **Consistency tests.** Does the proposed solution offer read-your-writes consistency and monotonic-read consistency?
- **Conflict resolution tests.** Do conflicts get detected and are the different conflict resolution strategies working correctly?
- **Usage of cached data.** Does BaaSSync make optimal use of locally cached data both for previously seen and new queries?
- **Response times tests.** Response times will be measured for all tests. In addition to that there will be specific tests to compare response times for different operations.
- **Bandwidth usage tests.** Bandwidth consumption will also be measured for all tests. In addition there will be specific tests to evaluate if bandwidth is saved in cases where this would be expected.

The integration test application is part of the implementation and can also be found on Github.

7.3 Response Times Tests

For all requests the test application above executes, response times will be measured. We expect to see a vast improvement in response times for read requests as more request can be answered from the cache, especially on a 3g or slower connection. On WiFi the differences might be smaller. Requests other than read will still go to the server in the same way as before so except for the case when there is no network connection we do not expect an improvement and because we are performing extra work we might actually see slightly worse performance.

To measure response times we will use the built-in "StopWatch" class, which uses the high-precision performance counters in Windows. To simulate different types of connections, we will use version 3.8.3 of the Charles Web Debugging Proxy [1], which is able to simulate different connection speeds and delays. We will tests the response times with each of the following settings in Charles:

[1] `http://www.charlesproxy.com/`

- **56k.** The 56k setting in Charles is used to simulate a very slow connection similar to a 2G connection.
- **3G.** The 3G setting in Charles is used to simulate a 3G connection as found on most mobile phones.
- **WiFi.** The same tests are run without a throttle setting to test a normal WiFi Broadband connection.

7.4 Bandwidth Consumption Tests

Testing the amount of bandwidth that is saved with BaaSSync is easy to test for predefined scenarios. The hard part in testing this is coming up with test scenarios that represent an actual user scenario. Because of different types of applications and the data they store on and retrieve from the mobile service, there is no general test case that can give us information about the bandwidth usage in general. Therefore the bandwidth tests will focus on the individual operations to get a sense of the actual savings for individual operations. We will measure the bandwidth consumption of all our tests and also perform specific tests that we expect to save bandwidth when using BaaSSync. To measure the differences we will run the tests against a mobile service with offline support and against a mobile service without offline support. Expectations are that the bandwidth consumption will be significant for read requests while for requests other than read it will be approximately the same.

To measure the bandwidth used by an application we will also use the built-in performance counters in Windows which can be used to measure the bandwidth usage of an individual application.

Chapter 8

Results

In this chapter we will discuss the results of the evaluation discussed in chapter 7. As part of the evaluation an integration test application was built which can be seen in figure 8.1. This application uses the test framework that is also used by the WAMS .NET SDK to do integration testing.

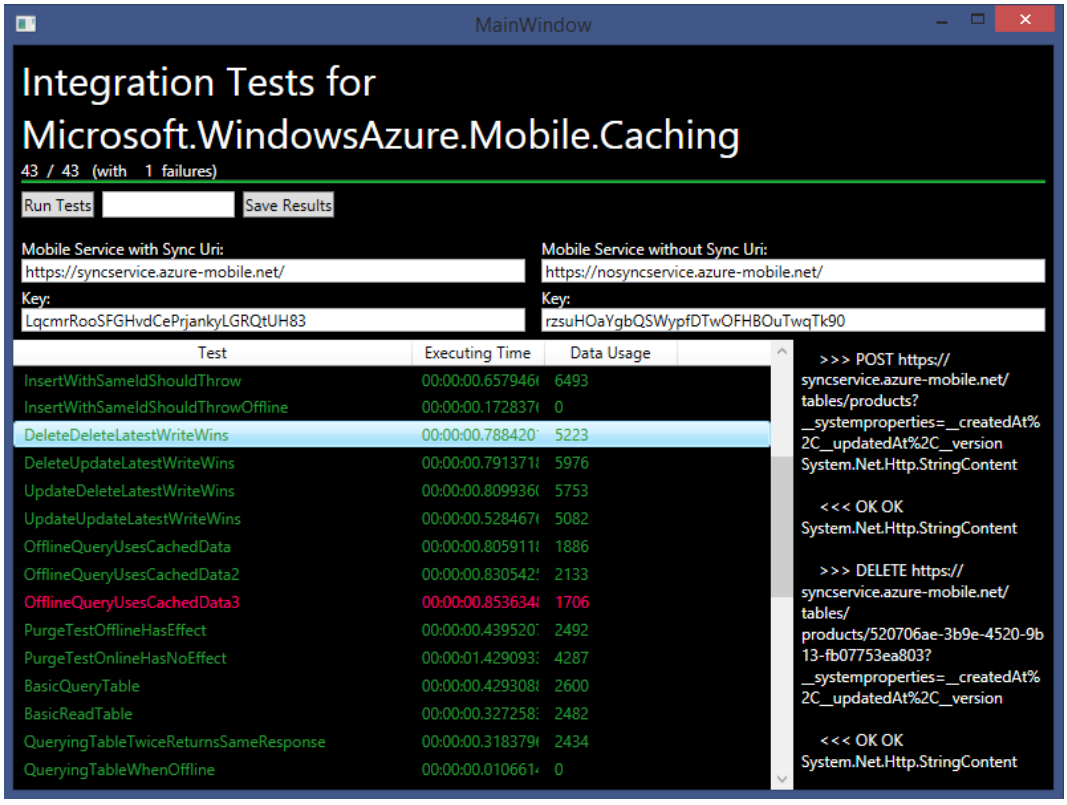

Figure 8.1: Integration test application for the offline data implementation on WAMS.

For the implementation discussed in chapter 6, a total of 108 unit tests were written. For the classes that were unit tested, we reached a code coverage of 84.35%. The integration test application contains 114 tests including 45 tests for response time testing and 26 tests for bandwidth usage testing.

8.1 Consistency

Using integration tests we verified that the implementation offers monotonic-read consistency for read requests and queries. In addition for each type of change the tests indicate that these changes are immediately reflected in read requests and queries. This means our implementation offers read-your-writes consistency.

According to the tests, reads from a table have been able to read new inserted items, updated items and deleted items.

8.2 Conflict Resolution

Our conflict resolution tests indicate that we are able to successfully perform conflict resolution on the server as well as on the client.

For each strategy we simulated each of the four possible conflicts types to ensure the server scripts send the correct response and the client handles this response correctly. Four different strategies have been tested, including “last write wins”, “server wins”, “duplication apply” and conflict resolution on the client. On the client we also applied a “last write wins” policy.

8.3 Cache Transparency

We should be able to use all data stored to answer any query or read operation. To test this we execute a query while we are connected. Then we disconnect and execute a different query. Finally we check if the results set contains the items we expect based on the results of the query that was performed while connected.

Several queries have been executed to verify that we can indeed answer queries we have not seen before by using cached data. The tests indicate that this is indeed the case.

8.4 Response Times

The results of the response times for read requests on a 2G, 3G and WiFi connection as well as for other operations can be found in appendix B.

In the results for read requests on a 3G connection, as shown in figure 8.2, the peak for the first read request to a BaaSSync enabled service is immediately apparent. This peak is caused by the fact that when the first request is executed, the objects coming from the server are stored first in the SQLite database. Although these objects are inserted in batches, this still takes significant time because SQLite inserts are expensive. SQLite waits for the OS

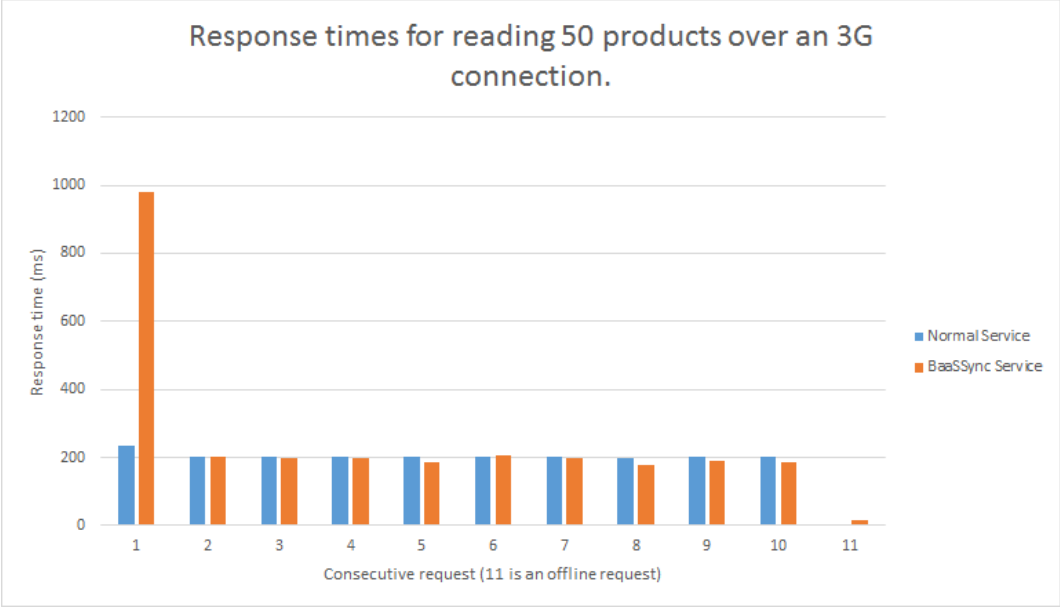

Figure 8.2: Response times on 3G for subsequent read operations.

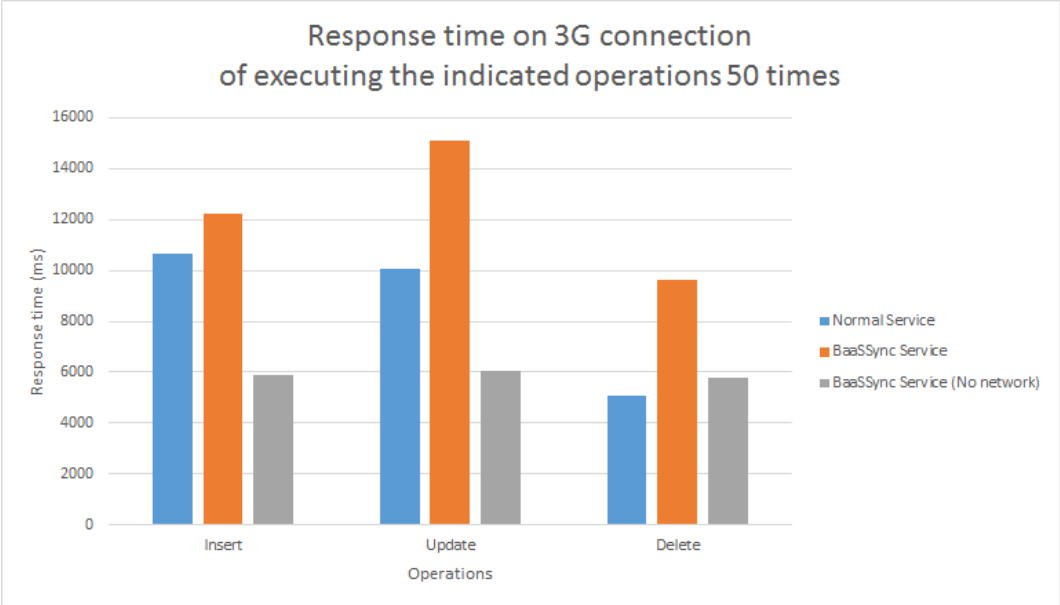

Figure 8.3: Response times on 3G for insert, update and delete operations.

to write the data to disk to ensure the transaction was Atomic, Consistent, Isolated, and Durable (ACID).

As can be seen in figure 8.2, with 3G speeds, subsequent read requests to a normal service are handled approximately just as fast as subsequent read requests to a BaaSSync enabled service. On a slower connection, subsequent read are even faster. On a WiFi connection read requests to a BaaSSync enabled service are slower than read requests to a normal service. This can be explained by the fact that sending the query to the local data store takes the most time and the network is no longer the slowest part of the operation.

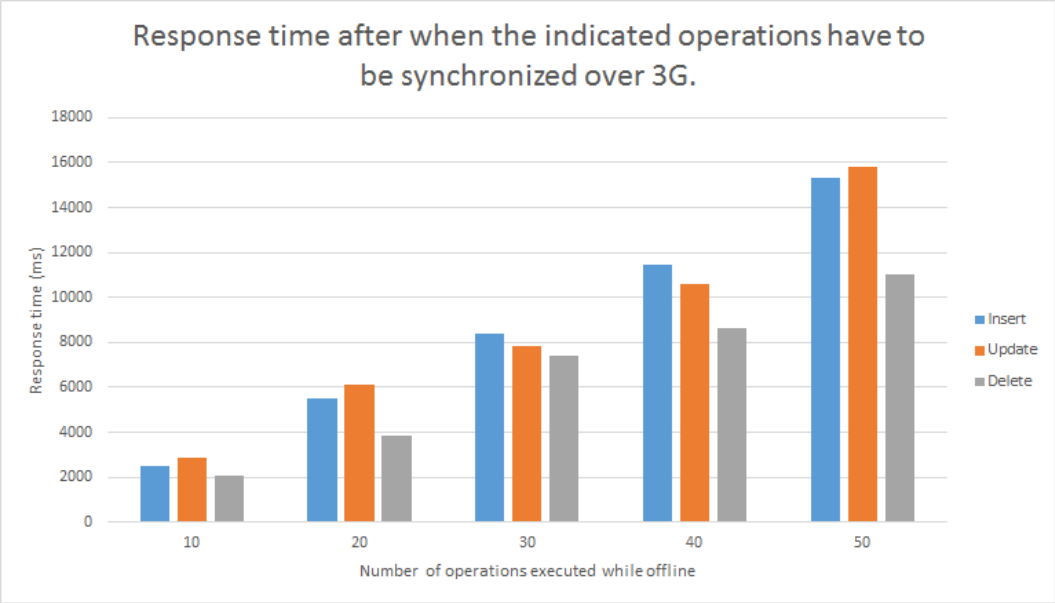

Figure 8.4: Response times on 3G for a read operation with the indicated amount of operations pending for synchronization. The first blue column show the response time for a read operation when we had 10 locally inserted items pending to be synchronized.

Operations other than read are in all situations slower on a BaaSSync enabled service as can be seen in figure 8.3. This can be explained by the fact that a BaaSSync enabled service does more work for these operations than a normal service, for example it reads from and writes to the local database. These operations again show that when we are on WiFi, the network is no longer a bottleneck, since the operations a BaaSSync enabled service without an available network are slightly slower than the operations on a normal service. Note that the operations without a network take the same amount of time independently of network type. This is expected since the network is not used for these operations.

When an application gets connected to the network after a period of disconnected operation, the first request that is executed starts the synchronization of changed items. Figure 8.4 shows that the synchronization of 50 inserted or updated items may take approximately 15 seconds to complete on a 3G connection. This synchronization takes approximately as much

time as inserting or updating 50 item while connected. The reasons this takes so much time has to do with the fact that we need to perform a request for each change. 50 insert operations performed while offline would result in 50 request being made during synchronization.

Delete operations are significantly faster on slower connections, which can be explained by the fact that these requests do not include the object in the request body and in the response object.

8.5 Bandwidth Usage

The bandwidth usage for consecutive read requests can be seen in figure 8.5 and for the other operations the results can be seen in figure 8.6 and also in appendix C.

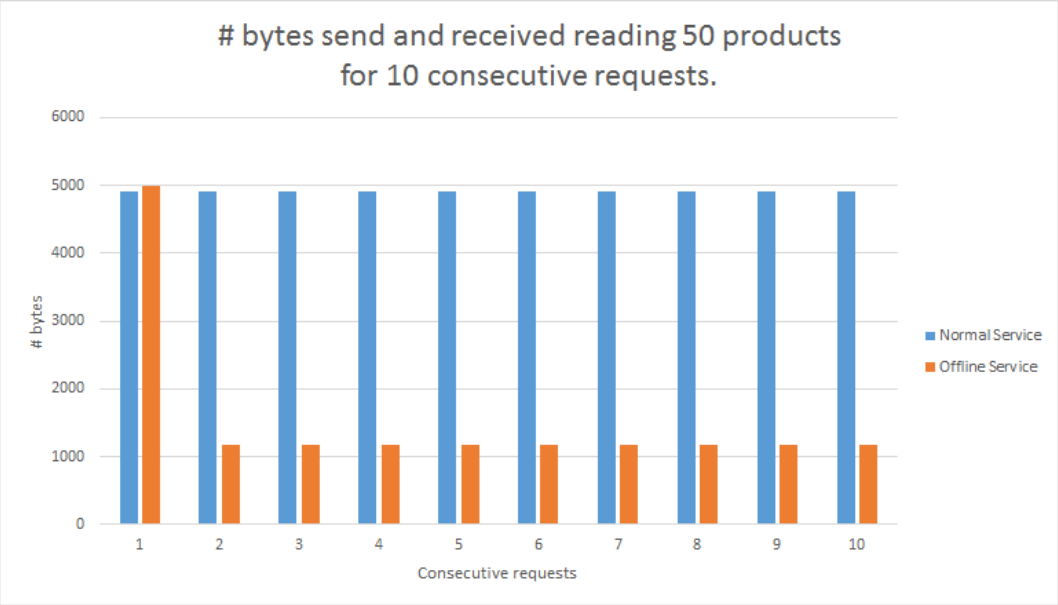

Figure 8.5: Bandwidth usage for subsequent read operations.

As can be seen in figure 8.5, the bandwidth usage drops significantly for consecutive read requests, but there is still some data send and received every time. The first time a request is executed, all data has to be sent just like the case of a normal service. The reason the bandwidth usage is a little higher is because the response of an BaaSSync enabled service includes some additional metadata including the version of the database when the response was answered.

The differences are a lot smaller for inserts and updates as can be seen in figure 8.6. This is expected because we still have to send the same data in the case of a BaaSSync enabled service as in the case of a normal service. For delete requests the server does not return content in the normal case. In the BaaSSync case we by default the ids of the deleted

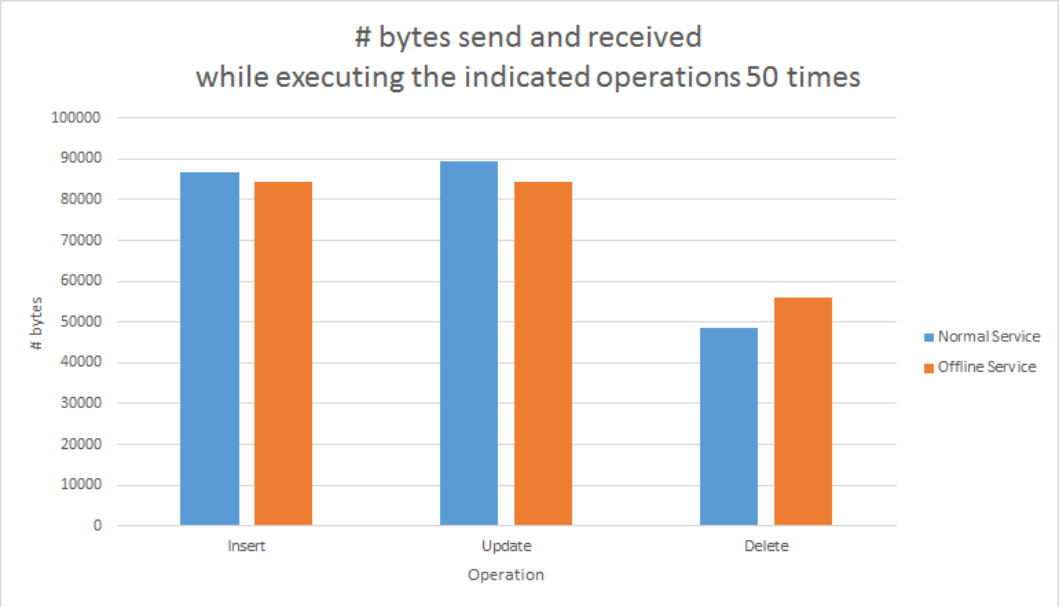

Figure 8.6: Bandwidth usage for insert, update and delete operations.

items as explained in section 6.3.1. This explains the slight increase in bandwidth usage for delete operations.

Note that there is a large difference in bandwidth usage between a single request to read 50 products and 50 different requests to insert a single item. One would expect a difference of a little more than twice the amount of data since an insert operation sends the inserted object to the service and the service returns the object back in the response. That would result in 50 items being sent and 50 items being received. The actual difference is that 50 inserts take 16 times as much bandwidth. The reason behind this is that the server is applying GZip compressing to the responses. Returning an array with 50 items with the same property names allows the server to compress the response significantly better than 50 independent requests.

Chapter 9

Offline Data (M)BaaS Offerings and Comparison

Several other Mobile Backend as a Service ((M)BaaS) solutions exist on the market and some of them offer offline data and synchronization capabilities.

In this chapter we will discuss the offline data capabilities offered by the larger players on the (M)BaaS market including Parse, Kinvey, StackMob, Google Mobile Backend Starter, and Appcelerator. These products have been chosen based on the popularity by bloggers and by the amount of questions they have on StackOverflow. Most other (M)BaaS providers have no questions at StackOverflow at all. Parse, Kinvey, and StackMob offer offline data capabilities, but only for Android, iOS and/or HTML5 with JavaScript. Google Mobile Backend Starter and Appcelerator do not offer offline data and synchronization capabilities and are therefore not further mentioned in this chapter.

9.1 Parse

Parse offers support for offline data on both Android and iOS [4], while it offers a .NET SDK without offline support. Parse data storage is based on so called "ParseObjects", containing key-value pairs. It offers support for offline scenario's, by distinguishing between saving and deleting online and offline. A developer has to opt-in by calling different methods called `saveEventually()` and `deleteEventually()`. The methods to synchronize these locally stored items later on are not documented, although according to the documentation it guarantees that the ordering of changes is preserved. In addition there is caching support. Parse does not offer conflict resolution at this moment and leaves that to the developer [1]. Parse currently offers the following functionality:

- **Offline support on iOS and Android**. Offline support is only available for iOS and Android.
- **Custom query options.** Parse supports basic query options such as filtering based on comparison, limiting the results, sorting and more.

[1] `https://parse.com/questions/resolving-conflicts`

- **Explicit offline support.** Specific methods need to be called to support changes when offline. These `saveEventually()` and `deleteEventually()` methods are different from the normal methods.
- **Caching support including different policies.** Parse can send queries only to the local cache, only to the server, or to both. There are also options to return result from both the cache and the server.
- **Results are cached per query.** Different queries are caching their results separately. Queries that do not have a result cannot be used while offline.
- **Automatic cache flushing.** Parse flushes the cache automatically when it takes up too much space. Exact limits are not specified.
- **Query does not return changes.** Changes made via `saveEventually()` are not reflected in cached results for queries when the application is offline. When the application is online, this depends on the set cache policy [2].
- **No conflict resolution.** Conflict resolution is left to the developer.

9.2 Kinvey

Kinvey supports offline data and caching on Android, iOS and HTML5 [6]. Kinvey's data APIs can be used with normal objects in the language of choice as long as they extend certain classes or implement certain interfaces. Caching can be done in several ways using the different caching policies that are offered. Offline changes are also supported. Kinvey synchronizes the data by only sending changes, most likely by keeping status flags, but that is not documented. They seem to have conflict resolution on the client, but it is only documented for the HTML5 client. Kinvey currently offers the following functionality:

- **Offline support on iOS, Android and HTML5**. Offline support is available for the three major platforms, iOS, Android and HTML5 (JavaScript).
- **Advanced query options.** Kinvey supports advanced query options such as filtering by comparison, limiting the results, sorting, applying certain functions over the results and more.
- **Caching support including different policies.** Kinvey supports a variety of caching policies including sending queries only to the local cache, first to the server, first to the cache, or to both.
- **Results are cached per query.** Different queries are caching their results separately. Queries that do not have a result cannot be used while offline.
- **Manual cache flushing.** The application can clear the cache when needed. In addition the cache is also cleared when a user logs out.
- **Query does return changes.** Although no new queries can be executed offline, changes made offline are reflected in the cached results. When the application is online, this depends on the cache policy used. New items will not be included in cached results. In this way Kinvey offers partial eventual consistency.

[2] `https://parse.com/questions/does-saving-an-object-supposed-to-update-the-local-cache-of-a-query-on-these-objects`

- **No conflict resolution on iOS and Android** There is no documentation about conflict resolution on iOS or Android.

9.3 StackMob

StackMob has support for offline data and synchronization only on iOS [7] and integrates with Core Data to provide this functionality [3]. It supports a variety of caching policies for both reading from the cache and writing to the cache. The synchronization works by using an "isDirty" flag per object. The documentation states that conflict detection and resolution occurs on the clients and not on the server. This still allows for data loss when two clients resolve a conflict for the same item locally and synchronize their changed objects to the server. StackMob also updates cached data with offline updates and can support certain queries over the cached data. The functionality that is currently offered includes:

- **Offline support on iOS only**. Offline support is only available in the iOS SDK.
- **Custom query options.** StackMob supports basic query options such as filtering based on comparison, limiting the results, sorting and more.
- **Caching support including different policies.** Similar to other (M)BaaS providers StackMob also supports various policies to handle the cache. Policies available include sending queries only to the local cache, first to the server or first to the cache.
- **Results are independently of a query.** Different queries cache their results together allowing for reuse of the data for other queries. Queries that do not have been executed before can be used while offline.
- **Manual cache flushing.** The application can clear the cache when needed.
- **Queries do return changes.** In addition to being able to reuse local data for new queries when offline, changes made offline are also reflected in the cached results. This can be modified by setting different cache policies. StackMob offers read-your-writes eventual consistency.
- **Conflict resolution and detection on the client.** Conflict resolution and detection is performed on the client. It supports custom policies are possible, it should be noted that conflict detection on the client can miss new created conflicts while the resolve step is occurring on a client. This can lead to very subtle bugs and it is unclear in the documentation how this problem is handled.

9.4 Feature Comparison

How does the BaaSSync solution proposed in this thesis compare to the above similar products that exist on the market today with respect to offered features? Table 9.1 lists the features of each product including BaaSSync. Note that because there are no backend providers offering offline data solutions on the Windows platform, we can only compare BaaSSync to products available for other platforms.

[3] `https://developer.stackmob.com/ios-sdk/offline-sync-guide`

[4] Parse flushes the cache automatically.

	iOS	Android	HTML5	Windows	Basic query options	Advanced query options	Different caching policies	Transparent cache	Cache flushing	Read-your-writes consistency	Conflict resolution on clients	Conflict resolution on server
Parse	✓	✓			✓		✓		✓[4]			
Kinvey	✓	✓	✓			✓	✓		✓	✓	✓[5][6]	
StackMob	✓				✓		✓	✓	✓	✓	✓[6]	
BaaSSync				✓		✓	✓[7]	✓	✓	✓	✓	✓

Table 9.1: Synchronization features of different (M)BaaS providers.

Compared to the similar products on the market, table 9.1 shows that the BaaSSync solution proposed in this thesis offers a combination of features and functionality that is not offered by similar products. Especially the conflict resolution is more robust and offers substantial flexibility. In addition, BaaSSync offers advanced query support locally on a transparent cache combined with read-your-writes consistency on this cache.

Although the implementation offers cache flushing, using this functionality is not as easy as it could be. You have to get to the `ICacheProvider` instance to use it. Manual synchronization is also a little harder than we would like, but this has to do with the way the library is built. To manually synchronize, one has to make a query to force it. This can be a simple lookup.

The difference between basic query options and advanced query options deals with advanced functions such as “average”, “sum” and “tolower”. Not all (M)BaaS providers offer these options in their query language.

[5]Conflict resolution is only available for the HTML5 clients.

[6]Conflict resolution can still lead to data loss, because detection happens on the client.

[7]Different cache providers can be used to implement different caching policies.

Chapter 10

Discussion and Future Work

Now that we have presented a fully working solution called "BaaSSync" and implemented it on top of Windows Azure Mobile Services, we will discuss the results that we achieved with the solution. After the discussion we will give some ideas for future work.

10.1 Discussion

In the discussion we will make a difference between the solution and the implementation of the solution on Windows Azure Mobile Services. We will start with the discussion of the solution.

Poor performance for Insert/Update/Delete.
: We have seen in section 8.4 that the insert, update, and delete operations are slower on a client with offline support. Although additional optimizations may be possible to improve the performance, these operations hit the network and the disk instead of only hitting the network and will therefore always be a little slower.

No three-way merge.
: In the current solution we do not store previous versions of objects. Therefore we always have the current version and the new version of the object when doing conflict resolution. This means we cannot do three-way merge [28] in the case of a conflict.

Bandwidth usage.
: Although BaaSSync tries to save bandwidth by only sending changes, this is done on a per request basis as discussed in section 5.5. For two different requests that have large overlap in results, the server will return two complete data sets.

No results until all conflicts are resolved.
: BaaSSync requires that a client is fully synchronized before the actual incoming request is sent to the server. This means that an application showing a list of items would show an empty list until all offline changes have been synchronized. In the case of conflict resolution on the client all conflicts also have to be resolved first.

Update operation sends the complete object.
: In section 5.1 we discussed that on the client we use status flags to determine which items need to be synchronized. We do not record the actual change that has been

made. Therefore for update operations we send the whole object to the server. In the case of a large object with a single change, this results in unnecessary data being sent. Note that the WAMS SDK does this by default too.

Order for offline changes.
Because of the status flag synchronization on the client as discussed in section 5.1, we do not preserve the order of the changes as they are applied on the client. This means the changes can be applied in a different order on the server than they happened on the client.

Now that we have discussed certain aspects of the BaaSSync solution, we will continue the discussion with the WAMS implementation. Because of the goal that it should work with the Mobile Services .NET SDK without making changes, some compromises were made in the implementation of BaaSSync.

No simple caching/offline policies.
Although the implementation supports multiple cache providers to apply different caching/offline policies, this does not work as simple and as easy as some of the solutions discussed in chapter 9. Some of these solutions offer this flexibility by passing a constant into the operation they call.

Multiple active threads.
The implementation currently supports only a single thread in the synchronization layer at a time. The current implementation has not been tested for multiple requests being active in the synchronization layer. Since the layer is free of global state, it should be possible to make this work with only minor modifications. Nevertheless, it may result in complex situations where a read request is made while at the same time data gets inserted and updated in the cache.

Manual synchronization is difficult.
Because of the way the implementation extends the WAMS .NET SDK as discussed in chapter 6, manual synchronization is a harder to do than what would be ideal. One cannot simply call `ISynchronizer.UploadChanges()`, because it requires arguments that are instantiated by the `CacheHandler` class. To manual synchronize one can request an item that does not exist, for example by specifying a random GUID. This will trigger synchronization.

Purge is difficult.
BaaSSync does not do automatic purge of the cache. Calling the `Purge()` method manually is difficult, because a reference to the cache provider is needed which is difficult to get because of the way the solution integrates into the SDK as explained in chapter 6.

Batch requests.
As we have seen in chapter 8, a request can take a significant amount of time to complete when there are lots of unsynchronized changes. This is caused by the fact that WAMS does not support batch operations. Therefore we have to send the changes to the server on at a time.

Exception for resolved conflicts.
The flexibility that is offered by BaaSSync for conflict resolution is significantly better than the offerings by similar products on the market as we have seen in section 9.4. Nevertheless the implementation on WAMS currently throws an exception to work around the response format expectations of the SDK as discussed in section 6.6.3.

10.2 Future Work

We consider the following the most important directions for future work:

Real world testing.
The BaaSSync solution has not extensively been tested on a real world application. Future work might focus on analyzing the effects of adding BaaSSync to a real world application. Although the results of the evaluation in chapter 8 has shown some effects on bandwidth and response times, a real world application might lead to more useful or different results. Such a real world test can also give us information about the effects of the increased response times on an application.

Query subsets.
The current implementation sends data twice if different request have similar responses. A request for all products with a price lower than 10 euros would result in all items being transferred even if the result set for all products with price lower than 20 euros is already stored. To solve this partially we would need a method to determine if a certain query is a subset of another query without inspecting its result set. This can be a topic of future research to see if this is possible at all.

All WAMS platforms.
Although BaaSSync has only been implemented on top of the Mobile Services .NET SDK, the code uses an extension point available on all SDKs. Future work could extend the implementation to the other WAMS SDKs.

Integrate into the SDK.
BaaSSync is implemented as an extension of the WAMS SDK as we have seen in chapter 6. Integrating the solution native into the SDK could allow for better performance, increased flexibility and it would no longer require hacks to work around assumptions made by the SDK. This could for example be implemented as an additional method `GetTableWithOfflineSupport()` on the `MobileServiceClient` class.

Chapter 11

Conclusion

Although network connections are getting faster and more reliable, there will always be situations or areas where the connection is unstable or does not work at all. In these cases it is great if an application continues to work.

With our proposed solution, BaaSSync, we have shown that for an (M)BaaS system such as Windows Azure Mobile Services it is possible to add offline support with only a minor impact on the code of the developer since it works with the existing SDK. By relaxing the consistency guarantees of the data, we are able to

1. offer a working offline data solution when the network is unavailable **G1**,
2. save bandwidth for subsequent request even if the network is available **G7**,
3. increase response times for read requests on very slow connections **G6**,
4. offer unmatched flexibility for conflict resolution **G4**,
5. answer complex queries we have never seen before, using data cached for other requests **G5**,
6. implement the solution on top of the Windows Azure Mobile Service .NET SDK **G2**.

Our evaluation shows that BaaSSync works for our test cases and it might be very interesting to see how it would work in a real world application. We hope this solution will motivate and inspire the WAMS team to make this functionality a first-class citizen in the SDK, so that the users of these WAMS applications will profit from this functionality in areas with unstable network connectivity.

Bibliography

[1] Using Core Data with iCloud Programming Notes. https://developer.apple.com/LIBRARY/ios/releasenotes/DataManagement/RN-iCloudCoreData/index.html, Jan 2013. [Accessed: 2014-02-25].

[2] Windows Azure Mobile Services. http://msdn.microsoft.com/en-us/library/windowsazure/jj554228.aspx, Jun 2013. [Accessed: 2013-12-02].

[3] Bloom filter. http://en.wikipedia.org/wiki/Bloom_filter, Jan 2014. [Accessed: 2014-01-06].

[4] Docs Overview | Parse. https://parse.com/docs/, Feb 2014. [Accessed: 2014-02-25].

[5] How to: Specify Snapshot, Download, Upload, and Bidirectional Synchronization. http://msdn.microsoft.com/en-us/library/bb726039.aspx, Mar 2014. [Accessed: 2014-03-03].

[6] Platform Picker | Kinvey. http://devcenter.kinvey.com, Feb 2014. [Accessed: 2014-02-25].

[7] Stackmob Developer Center. https://developer.stackmob.com/, Feb 2014. [Accessed: 2014-02-25].

[8] S. Agarwal, D. Starobinski, and A. Trachtenberg. On the scalability of data synchronization protocols for PDAs and mobile devices. *Network, IEEE*, 16(4):22–28, 2002.

[9] Deborah J. Armstrong. The Quarks of Object-oriented Development. *Commun. ACM*, 49(2):123–128, February 2006.

[10] Walter William Rouse Ball and Harold Scott Macdonald Coxeter. *Mathematical recreations and essays; 13th edition.* Dover Publications, 2010.

[11] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud Types for Eventual Consistency. In *Proceedings of the 26th European Conference*

on Object-Oriented Programming, ECOOP'12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] John Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed Content Delivery Across Adaptive Overlay Networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, pages 47–60, New York, NY, USA, 2002. ACM.

[13] Michael J. Carey, Nicola Onose, and Michalis Petropoulos. Data services. *Commun. ACM*, 55(6):86–97, June 2012.

[14] Mi-Young Choi, Eun-Ae Cho, Dae-Ha Park, Chang-Joo Moon, and Doo kwon Baik. A database synchronization algorithm for mobile devices. *Consumer Electronics, IEEE Transactions on*, 56(2):392–398, May 2010.

[15] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). http://tools.ietf.org/html/rfc4627, 7 2006. [Accessed: 2013-12-10].

[16] Bart De Smet. *C# 5.0 Unleashed*. Sams Publishing, 2013.

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616, Hypertext Transfer Protocol – HTTP/1.1. http://www.rfc.net/rfc2616.html, 1999. [Accessed: 2013-12-09].

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[19] E. Giguere. Mobile data management: challenges of wireless and offline data access. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 227–228, 2001.

[20] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.

[21] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[22] Kin Lane. Rise of Mobile Backend as a Service (MBaaS) API Stacks. `http://apievangelist.com/2012/06/03/rise-of-mobile-backend-as-a-service-mbaas-api-stacks/`, Jun 2012. [Accessed: 2013-11-20].

[23] P. Leach, M. Mealling, and R. Salz. RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace. http://www.ietf.org/rfc/rfc4122.txt, 2005. [Accessed: 2014-01-14].

[24] YoungSeok Lee, YounSoo Kim, and Hoon Choi. Conflict Resolution of Data Synchronization in Mobile Environment. In Antonio Laganá, MarinaL. Gavrilova, Vipin Kumar, Youngsong Mun, C.J.Kenneth Tan, and Osvaldo Gervasi, editors, *Computational Science and Its Applications - ICCSA 2004*, volume 3044 of *Lecture Notes in Computer Science*, pages 196–205. Springer Berlin Heidelberg, 2004.

[25] RichardJ. Lipton. Efficient checking of computations. In Christian Choffrut and Thomas Lengauer, editors, *STACS 90*, volume 415 of *Lecture Notes in Computer Science*, pages 207–215. Springer Berlin Heidelberg, 1990.

[26] Peter Lucas. Mobile Devices and Mobile Data-Issues of Identity and Reference. *Human-Computer Interaction*, 16(2-4):323–336, 2001.

[27] Micah Martin and Robert C Martin. *Agile principles, patterns, and practices in C#*. Pearson Education, 2006.

[28] T. Mens. A state-of-the-art survey on software merging. *Software Engineering, IEEE Transactions on*, 28(5):449–462, May 2002.

[29] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *Information Theory, IEEE Transactions on*, 49(9):2213–2218, 2003.

[30] John Osborn. Deep Inside C#: An Interview with Microsoft Chief Architect Anders Hejlsberg. `http://www.windowsdevcenter.com/pub/a/oreilly/windows/news/hejlsberg_0800.html`, August 2000. [Accessed: 2013-12-04].

[31] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[32] Open Data Protocol. ABNF for OData. `http://www.odata.org/documentation/odata-v3-documentation/abnf/`, April 2012. [Accessed: 2013-12-03].

[33] P. Seshadri and P. Garrett. SQLServer for Windows CE-a database engine for mobile and embedded platforms. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 642–644, 2000.

[34] David Starobinski, Ari Trachtenberg, and Sachin Agarwal. Efficient PDA Synchronization. *IEEE Transactions on Mobile Computing*, 2(1):40–51, January 2003.

[35] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *Internet Computing, IEEE*, 14(6):80–83, 2010.

[36] Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, January 2009.

Appendix A

Detailed synchronization algorithm of `TimestampCacheProvider`

The `TimestampCacheProvider` class implements the bi-directional synchronization algorithm discussed in section 5.5. The exact steps of the algorithm when it handles a request are described per CRUD operation below.

A.1 Create operation.

When an item gets inserted, the steps executed are the following:

1. Check if there is a connection to the network by calling `INetworkInformation.IsConnectedToTheInternet()`. If this is not the case we jump to step 3.
2. Call `ISynchronizer.UploadChanges()` and pass the new item as a parameter to synchronize the new item including all other pending changes. Store the response content and jump to step 6.
3. Call `ISynchronizer.NotifyOfUnsynchronizedChange()` to notify of unsynchronized changes.
4. Assign a new id to the item and insert it locally with status "Inserted".
5. Create the response JSON.
6. Return the response content.

A.2 Read operation.

When items are requested, the steps executed are the following:

1. Check if there is a connection to the network by calling `INetworkInformation.IsConnectedToTheInternet()`. If this is not the case we jump to step 4.
2. Call `ISynchronizer.UploadChanges()` to synchronize all pending changes.

3. Call `ISynchronizer.DownloadChanges()` with the request URI. The synchronizer retrieves the most recent version for the request from the database and adds it to the URI to download the changes since the last time the request was executed. Jump to step 5.
4. Check if the request contains "inlinecount=allpages". If so write the count property with value "-1" to the JSON response.
5. Parse the query in the URI into an expression tree.
6. Call `IStructuredStorage.GetStoredData()` and pass it the expression tree of the query.
7. Put the result of the query in the "results" property of the JSON response.
8. Return the response content.

A.3 Update operation.

When an item is modified, the steps executed are the following:

1. Check if there is a connection to the network by calling `INetworkInformation.IsConnectedToTheInternet()`. If this is not the case we jump to step 3.
2. Call `ISynchronizer.UploadChanges()` with the item that needs to be updated as a parameter, synchronize the changed item including all other pending changes. Store the response content that contains the updated item. Jump to step 5.
3. Call `ISynchronizer.NotifyOfUnsynchronizedChange()` to notify of unsynchronized changes.
4. Put the changes in the local storage by calling `IStructuredStorage.UpdateData()` and provide the object with status "Changed" if it previous was " Unmodified". The modified object is also put in the response.
5. Return the response content.

A.4 Delete operation.

When an item is deleted, the steps executed are the following:

1. Retrieve the item with the id specified in the URI from storage.
2. Check if there is a connection to the network by calling `INetworkInformation.IsConnectedToTheInternet()`. If this is not the case we jump to step 4.
3. Call `ISynchronizer.UploadChanges()` and pass the item from step 1 to delete the item on the server and to synchronize all other pending changes. Jump to step 6.
4. Call `ISynchronizer.NotifyOfUnsynchronizedChange()` to notify of unsynchronized changes.
5. If the status of the item was "Inserted", remove the item from the local storage by calling `IStructuredStorage.RemoveStoredData()`. Otherwise call `IStructuredStorage.UpdateData()` and set the status of the object to "Changed".

6. Return empty response content.

Appendix B

Response Time graphs

This chapter contains the graphs that show the effect of the offline data implementation on the response times for the different requests that can be done with the Windows Azure Mobile Services .NET SDK.

B.1 Response Times on a 56k Connection

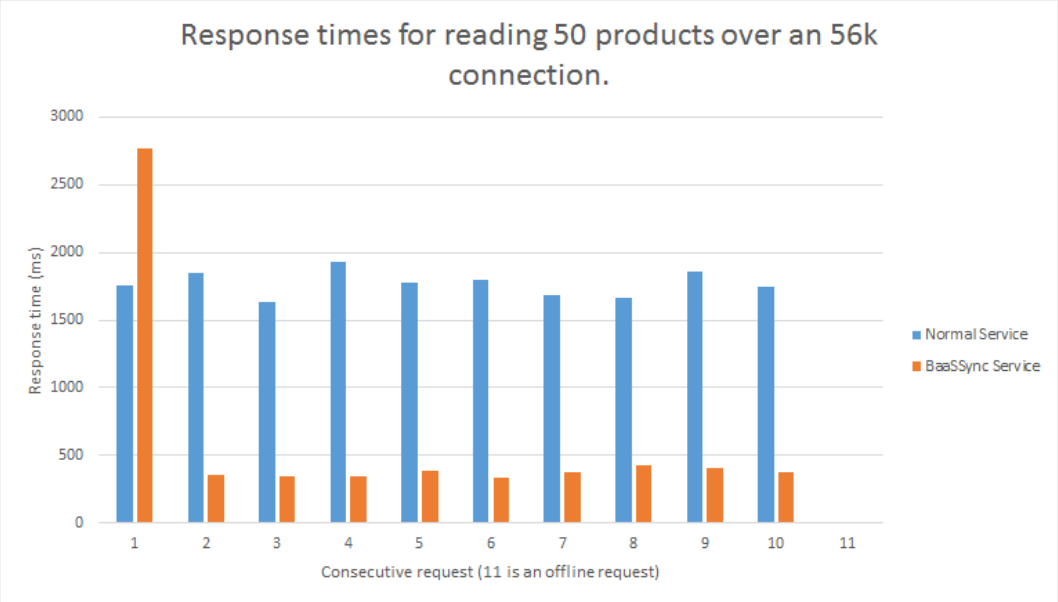

Figure B.1: Response times on 2G for subsequent read operations.

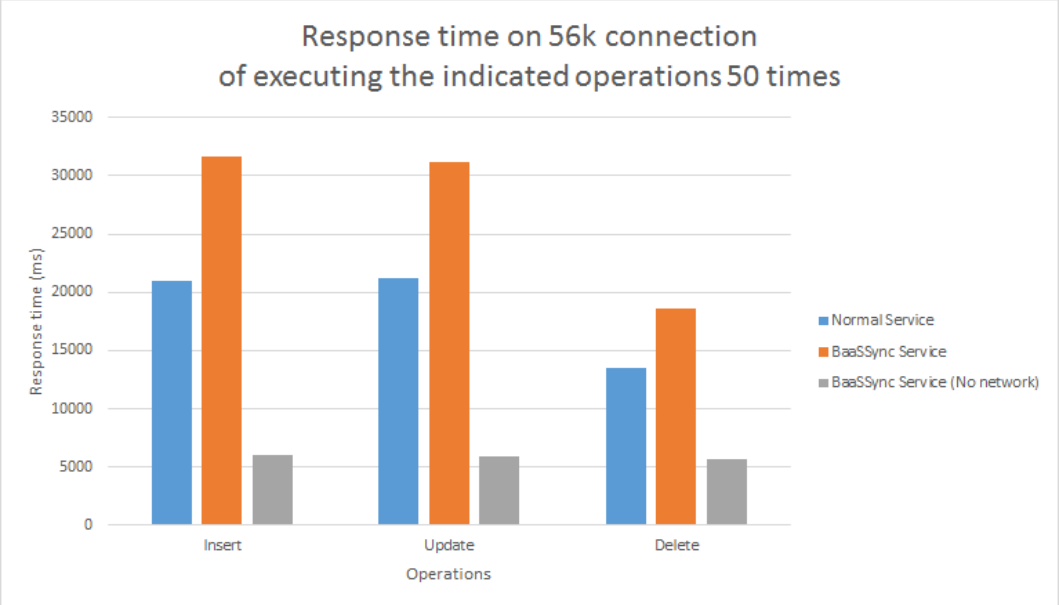

Figure B.2: Response times on 2G for insert, update and delete operations.

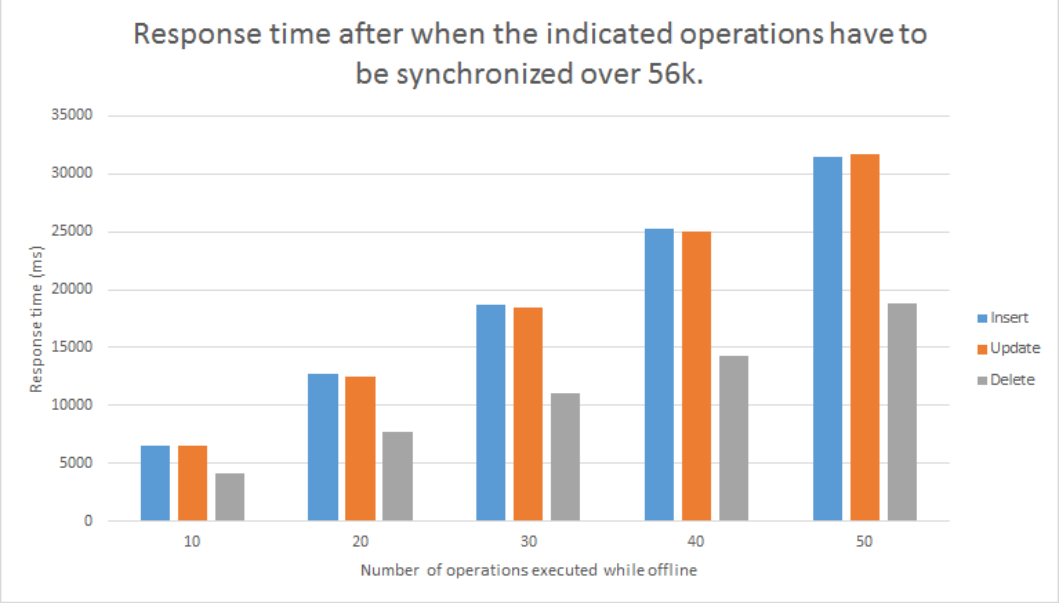

Figure B.3: Response times on 2G for a read operation with the indicated amount of operations pending for synchronization. The first blue column show the response time for a read operation when we had 10 locally inserted items pending to be synchronized.

B.2 Response Times on a 3G Connection

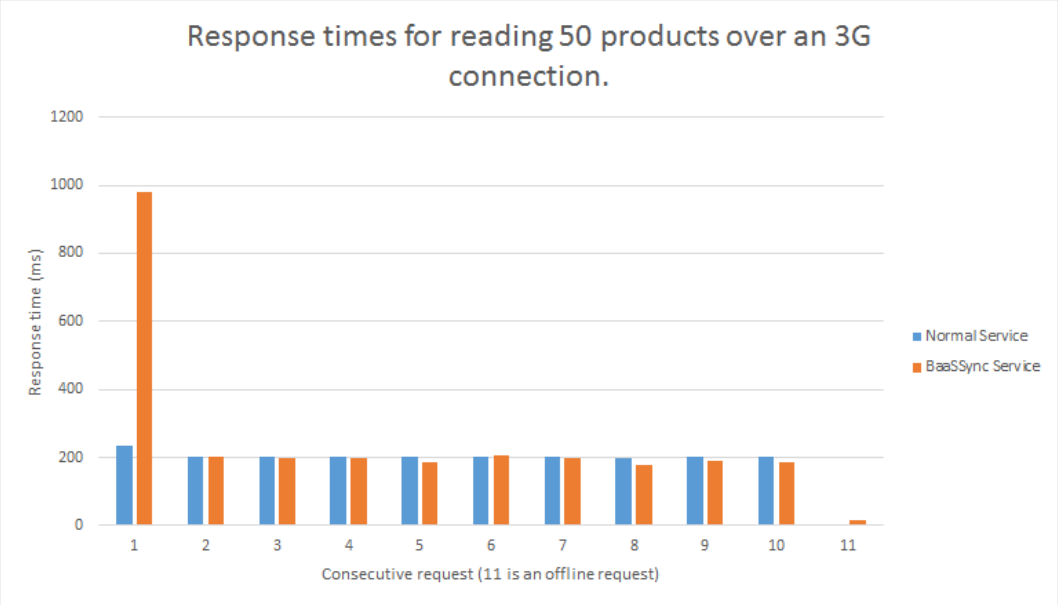

Figure B.4: Response times on 3G for subsequent read operations.

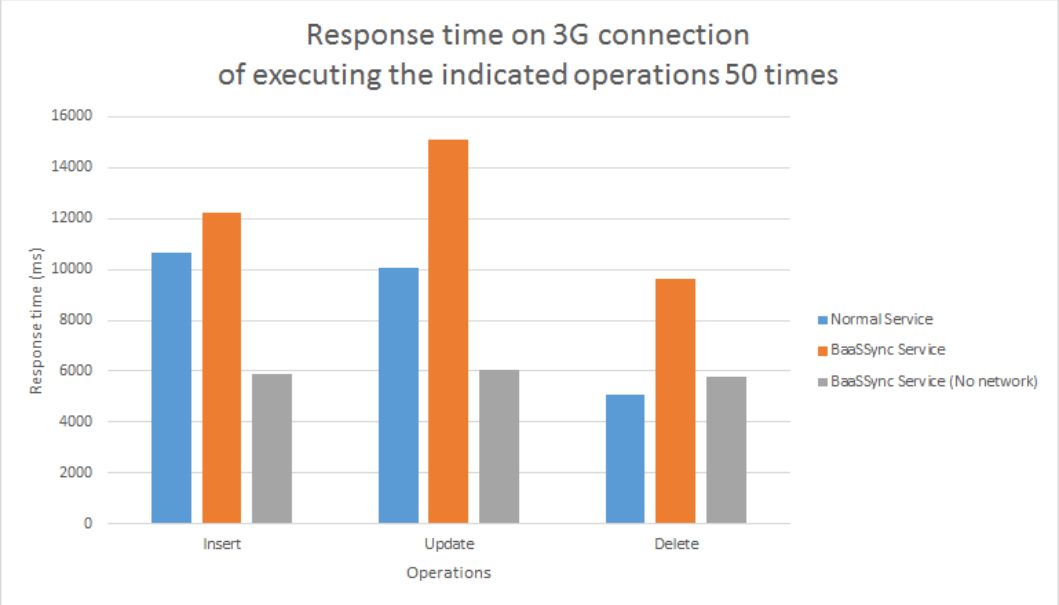

Figure B.5: Response times on 3G for insert, update and delete operations.

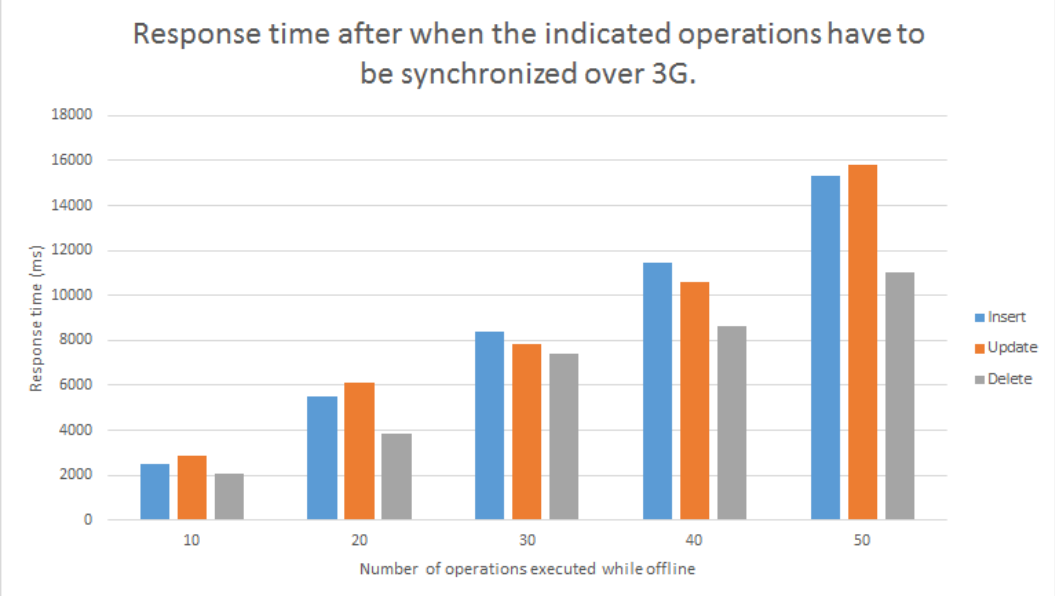

Figure B.6: Response times on 3G for a read operation with the indicated amount of operations pending for synchronization. The first blue column show the response time for a read operation when we had 10 locally inserted items pending to be synchronized.

B.3 Response Times on a WiFi Connection

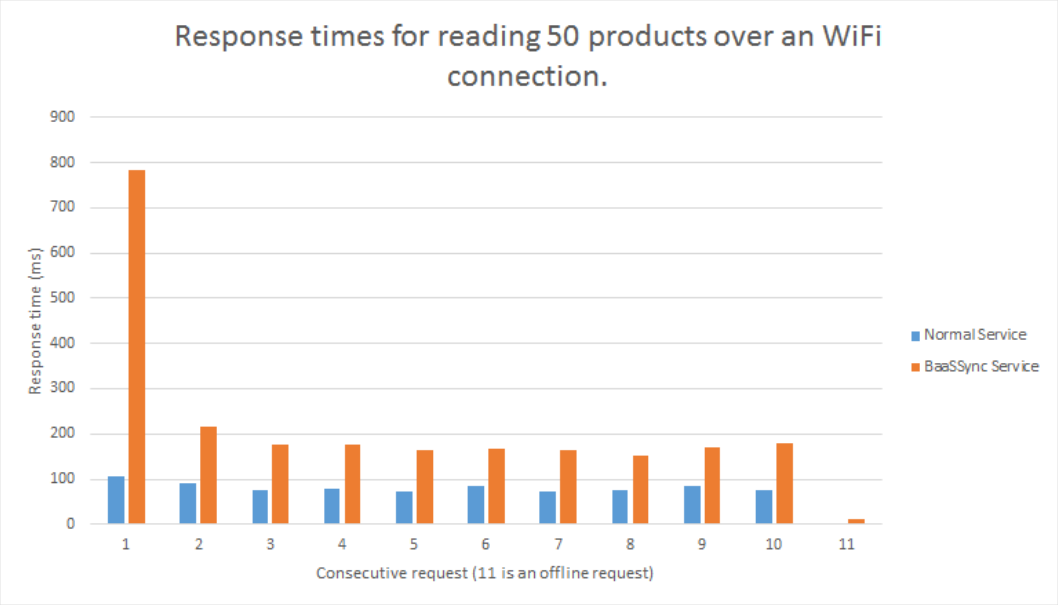

Figure B.7: Response times on WiFi for subsequent read operations.

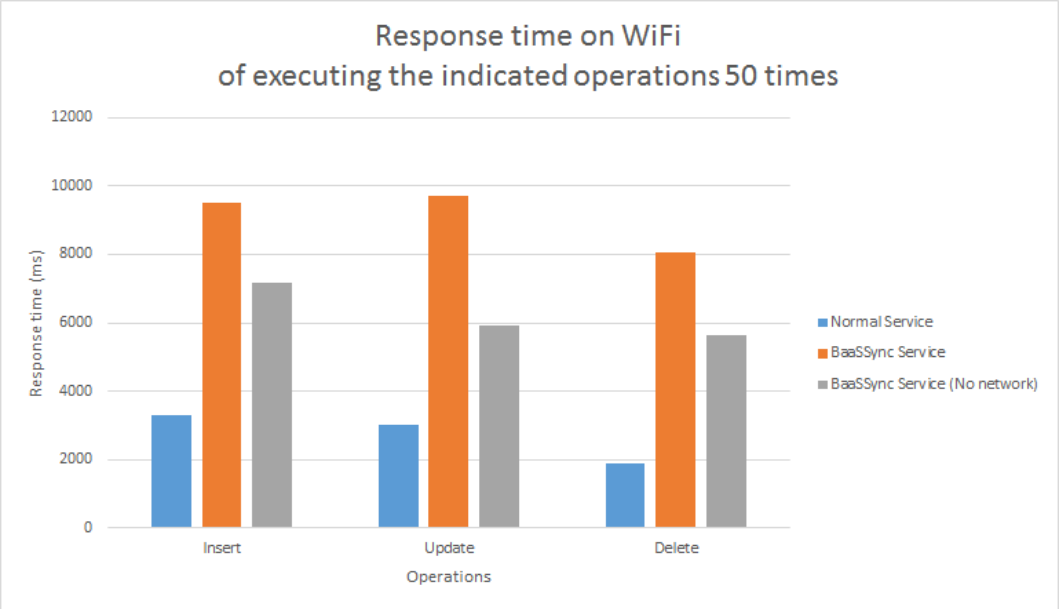

Figure B.8: Response times on WiFi for insert, update and delete operations.

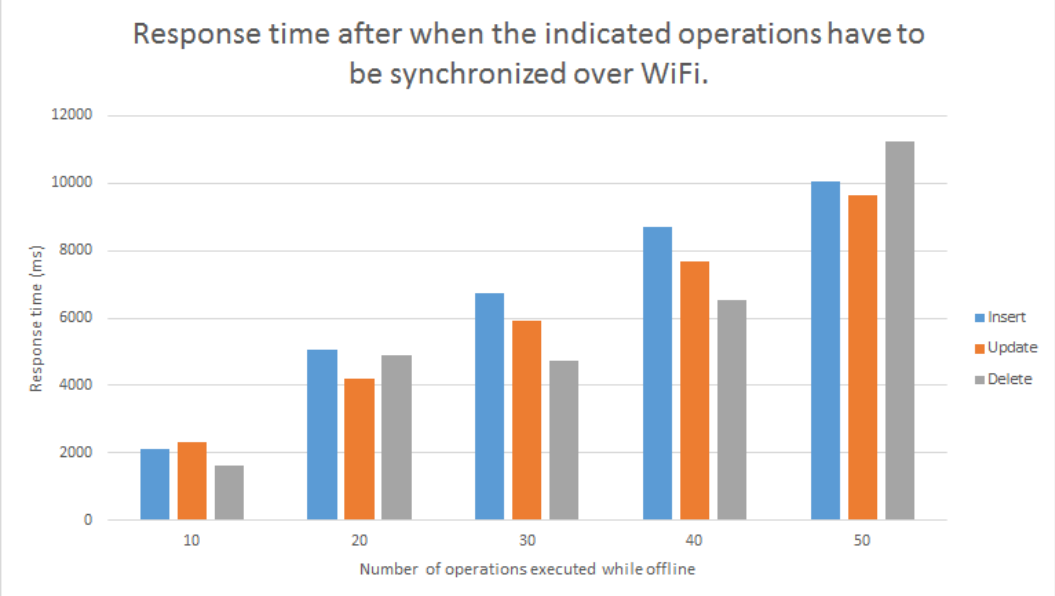

Figure B.9: Response times on WiFi for a read operation with the indicated amount of operations pending for synchronization. The first blue column show the response time for a read operation when we had 10 locally inserted items pending to be synchronized.

Appendix C

Bandwidth Usage Graphs

This chapter contains the graphs that show the effect of the offline data implementation on the bandwidth usage for the different requests that can be done with the Windows Azure Mobile Services .NET SDK.

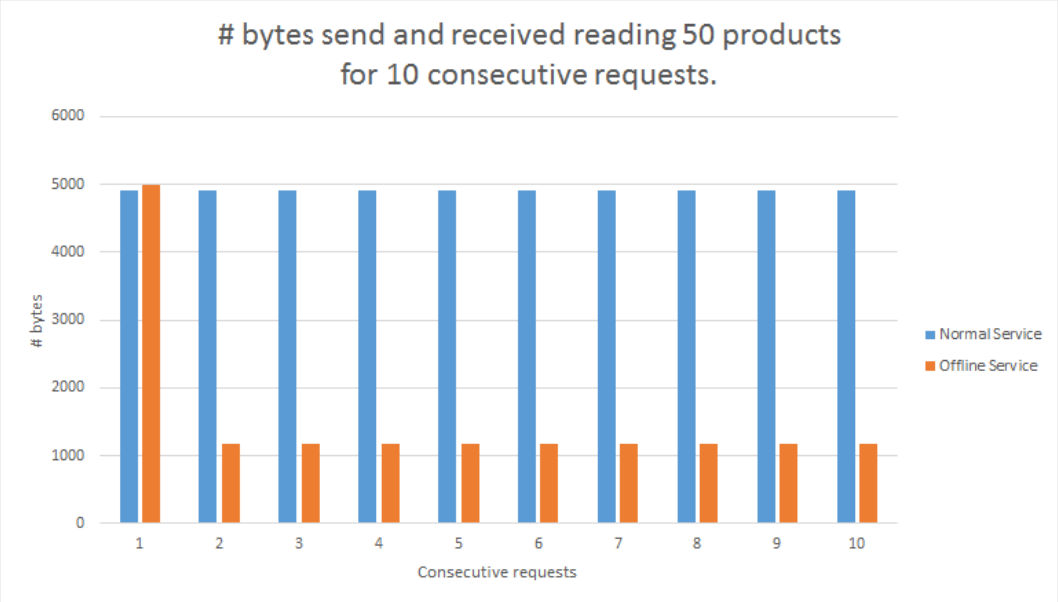

Figure C.1: Bandwidth usage for subsequent read operations.

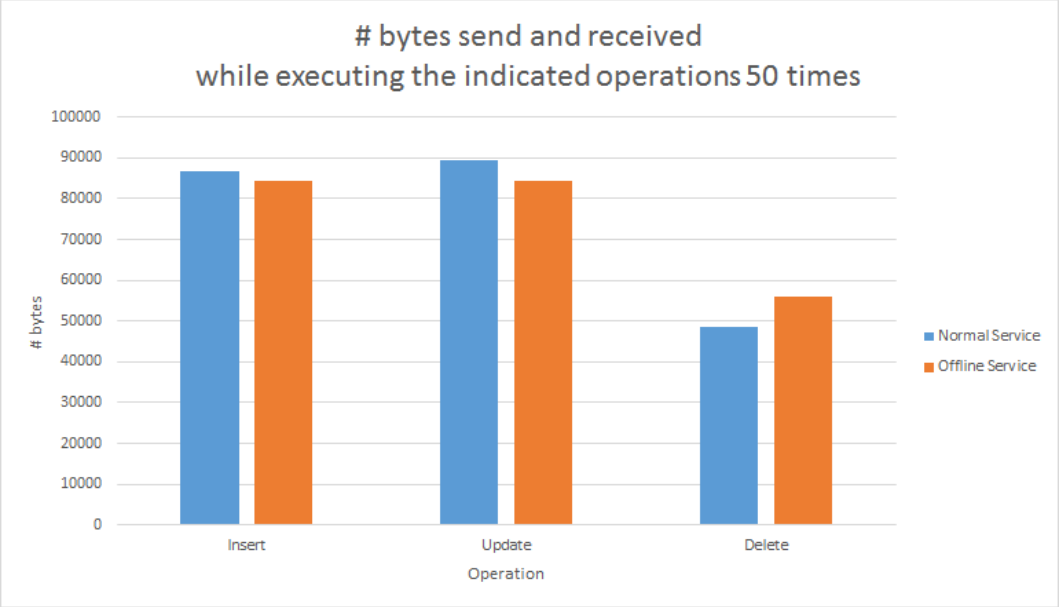

Figure C.2: Bandwidth usage for insert, update and delete operations.