JAVASCRIPT PERSISTENCE OBJECTS WITH ADAPTIVE SYNCHRONIZATION TIMING

Supervised by: Dr. Shehan Perera
Prepared by: T.A.M.P Fernando (138212T)

M.Sc. in Computer Science

Department of Computer Science and Engineering
University of Moratuwa
Sri Lanka

February 2016

JAVASCRIPT PERSISTENCE OBJECTS WITH ADAPTIVE SYNCHRONIZATION TIMING

Supervised by: Dr. Shehan Perera

Prepared by: T.A.M.P Fernando (138212T)

Thesis submitted in partial fulfillment of the requirements for the Degree of MSc in Computer Science specializing in Software Architecture

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

December 2016

DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another per-son except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: …… Date: …….

Name: T.A.M.P Fernando

The above candidate has carried out the research for the Masters thesis under my supervision.

Signature of the supervisor: ……. Date: ……..

Name: Dr. Shehan Perera

ABSTRACT

Traditionally web applications limited to web server generated, dynamic HTML content, based on requests sent by web browser clients. With the advancements of web standards such as HTML5, AJAX and powerful languages such as ECMA5 JavaScript, web browser clients became application platforms. This allowed rich interactions from the web applications, running inside the browser, to directly communicate with RESTful Web Services. Persisting JavaScript Objects from browser to RESTful Web Services became a widely accepted pattern in synchronizing web application state. This works often well for static objects, but when it comes to frequently changing JavaScript Objects, issues such as response time and application performance remain a very real concern today. Lengthy response times for persisting objects may cause lower satisfaction and poor productivity among users. The aim of this research is to come up with a framework to reduce, unexpected response times when persisting frequently changing JavaScript Objects with RESTful Web Services increasing the overall user satisfaction.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my profound gratitude to my advisor, Dr. Shehan Perera, for his invaluable support throughout the research project by providing relevant knowledge, supervision and useful suggestions. His expertise and continuous guidance enabled me to complete my work successfully. Further I would like to thank Dr. Chandana Gamage for providing valuable resources and advice and insight of the applicability of this research in its initial phases.

I am also grateful for the support and advice given by Dr. MalakaWalpola, by encouraging continuing this research till the end. Further I would like to thank all my colleagues for their help on finding relevant research material, sharing knowledge and experience and for their encouragement.

I am deeply grateful to my parents for their love and support throughout my life. I also wish to thank my loving wife, who supported me throughout my work. Finally, I wish to express my gratitude to all my colleagues at 99XTechnology, for the support given me to manage my MSc research work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBRIVATIONS

API	Application Programming Interface
AJAX	Asynchronous JavaScript and XML
BOM	Browser Object Model
CSS	Cascading Style Sheets
DOM	Document Object Model
ECMA	European Computer Manufacturers Association
HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
JSON	JavaScript Object Notation
REST	Representational State Transfer
SQL	Structured Query Language
WORA	Write Once Run Anywhere
WORE	Write Once Run Everywhere

CHAPTER 1

INTRODUCTION

Web application development has changed drastically over the years with introduction of different set of technologies. It has evolved from simple content delivery web applications to highly interactive web applications which requires specialized engineering skills. This required utilizing software engineering practices to keep the complexities to minimal, reusing code and practices from libraries and frameworks to aid the development process.

1.1 Evolution of Web and Mobile Application Paradigms

Over the past few years World Wide Web has grown immensely with boundless technology innovations. Initially it all started with static web pages that only had the simply formatted content and links to other pages. With the rise of user expectations, serving more dynamic content was needed, which lead to the innovation of web server dynamically generated pages, giving birth to real web application engineering paradigm. Overtime dynamic web applications began to get popular but had its inherited inefficiencies of limited user interactions, delayed loading web pages and delayed response for user actions. Although server performance is improved by caching dynamic data [1] [2] it partially solved the issue. This was due to the limitation of having rich user interactions, where most of the user actions required to interact with the web server over the internet with inherent latencies. With the advancement of web browsers and web technologies, modern web applications started to use the web browser as an application platform where significant part of the application logic is executed inside the browser adding dynamism using JavaScript, solving some of the inherent inefficiencies and providing comparable user experience with Desktop Applications.

With the introduction of fast and efficient JavaScript engines, web browser containers began to be available on smart phone devices, which can run Apps developed using web technologies like HTML, JS and CSS. This has given birth to a new paradigm called Hybrid Application development [3].

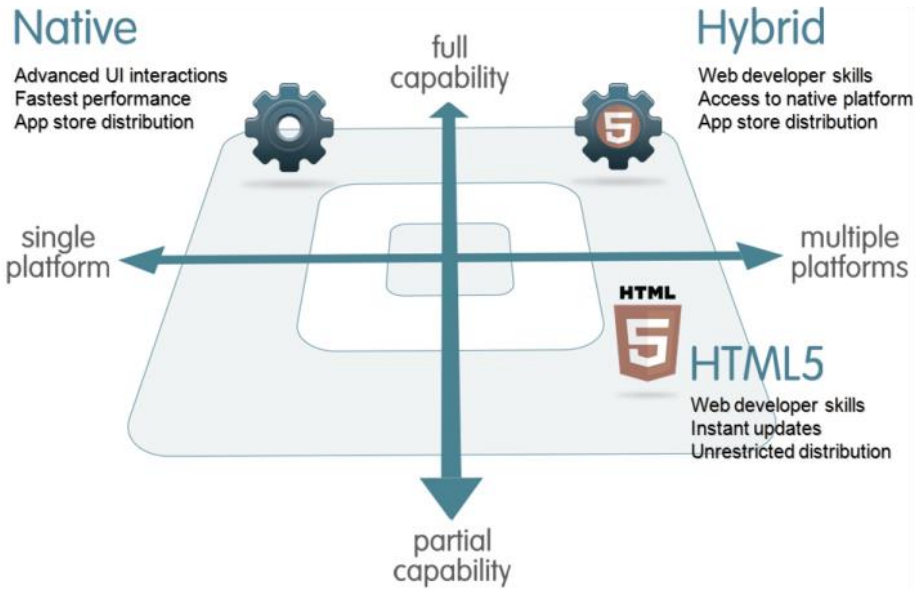

Figure 1: Native, HTML5 and Hybrid Landscape

Source: http://svitla.com/wp-content/uploads/2013/02/Native_html5_hybrid.png

All together these innovations revolutionized the Web Application development landscape, to build web applications that runs on range of devices in different forms.

1.2 Web Browser as an Application Platform

Over the past few years, Web browsers had inherent short comings. These were the major challenges that delayed the development of rich internet applications. According to the technical report, produced by Sun Microsystems [4], list of shortcomings of the past web browsers, related technologies and findings of this research on how it is being addressed in recent browser enhancements in summary is given below.

1.2.1 Related to performance

Until recently when high performance JavaScript virtual machines were build, native functionality provided by Web Browsers were pretty slow especially when it comes to computationally intensive applications. The limitation of being single threaded increased

the problem. Most of these challenges are now being addressed by the introduction of high performing JavaScript engines such as Google's V8 and the implementation of multithread support for JavaScript included in HTML5 specification.

1.2.2 Related to user interactions

Until very recently Web Browsers had a very rigid user interaction model with very primitive set of features. They lacked the desktop style rich user interactions, such as drag and drop, printing & etc. Today almost all the Web Browsers supports these as very basic set of features with smooth rendering.

1.2.3 Related to network and security model

Most of the issues related to security [5] are originated from the document based security model that is driven by the "One size fits all" browser security model. Decisions on security were mainly originated from the origin of web server but not by the specific need or functionality of the web application since these applications run on sandboxed environment with limited access to host machine that runs the web browser. Now most of these challenges are getting revolutionized since the browsers provide more capabilities to use host machine resources (E.g. File access) deploying different access control mechanisms.

Apart from these in the past web browsers had the inherent limitation of request initiation from the browser client that required inefficient communication for real-time applications. Now with modern web browsers, these restrictions are getting out of the way which provides capability for the web server to notify the browser client effectively utilizing the network.

1.2.4 Related to compatibility and interoperability

Over the past few years, Web browser compatibility was a major challenge. Although Web Applications are written, "Write once, run anywhere" (WORA), or sometimes write once, run everywhere (WORE) it didn't work that smoothly in different browsers. Most of the time, rich web applications failed to operate in certain browsers due to the differences of

Browser Object Model (BOM), Document Object Model (DOM) and rendered in unintended compositions and colors due to CSS rendering differences. This has given birth to libraries such as jQuery to be really popular to provide a uniform interface in manipulating. With the introduction of HTML5 and CSS3 it has shown a light of hope in having unified interfaces implemented by different browser venders.

1.2.5 Related to development and testing

The transition of changing the mind set of developers from statically typed languages to dynamically typed languages has always being a challenge. Web application development has more specifically made it perhaps rather confusing in applying software development methodologies due to the fundamental style differences like asynchronous, event driven using higher order functions (E.g. in JavaScript). Still this is a challenge and developers needs to sharpen their skills in these areas.

1.2.6 Related to deployment

One of the main strength of Web applications are given by the inherent ability of instant deployment. This has revolutionized the software deployment landscape allowing up to date software to be available instantly to the users with minimal distribution overhead. More recently with the introduction of HTML5 app cache, web application distribution has gone an extra mile in supporting the instant distribution of the updated web application to available offline in web browser clients.

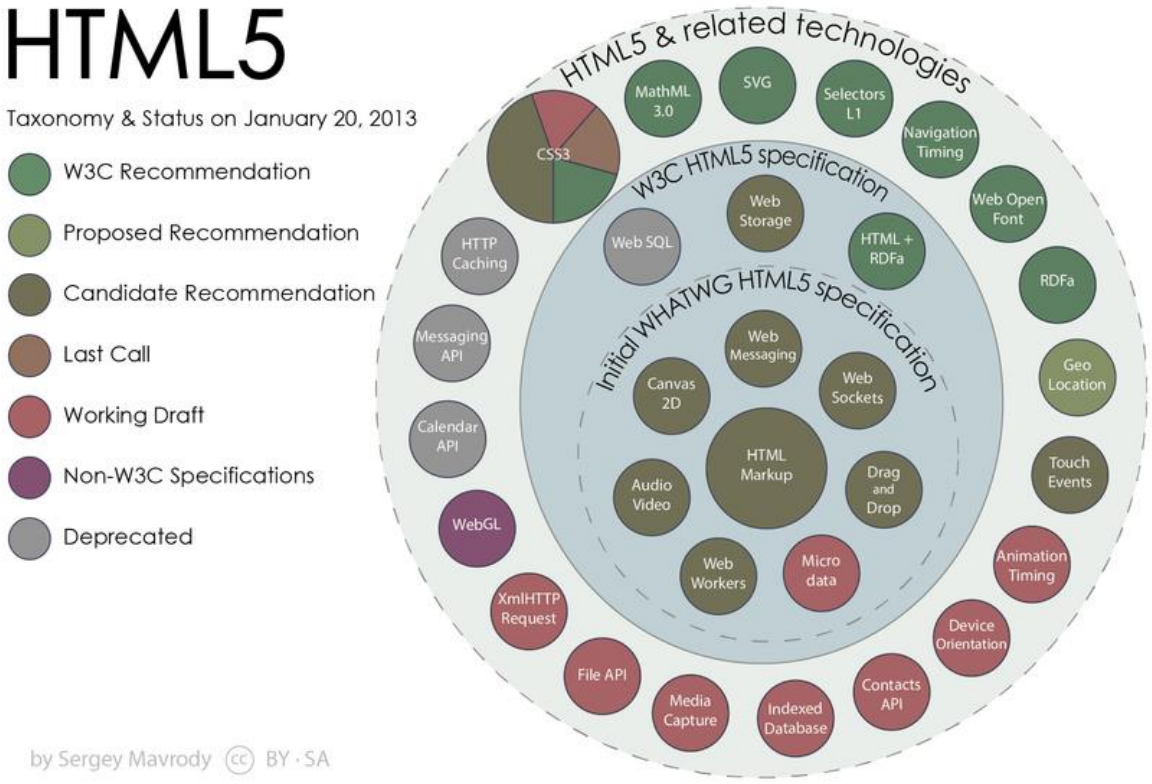

Figure 2: HTML5 APIs and related technologies

Source: [6] http://upload.wikimedia.org/wikipedia/commons/f/f7/HTML5-APIs-and-related-technologies-by-Sergey-Mavrody.png

As shown in the above figure, the implementation of HTML5, CSS3 web standards in modern web browsers changed the capabilities of browser environments to reduce the above mentioned shortcomings and to transform from traditional web page viewing tool to a real application platform [7].

1.3 Data driven web applications

With the advancement of web browsers, shift from server side to client side functionality became more apparent. This made the possibility for web applications to implement data handling inside the browser thus providing well needed features to support as an application platform. Client side data handling inside web browsers involved in modifying

data in the memory mainly in the form of JavaScript Objects as well as persisting them in client side storage by serializing the data or directly storing the object depends on the storage of choice.

Table 1: Comparison of HTML5 client-side storage mechanisms.

Client-side storage	**Standardization**	**Main features**	**Supported data types**	**Storage space**
Web Storage	W3C, Candidate Recommendation, December 2011	simple key-value pair data, no duplicate values for a key	String (including string serialized JSON)	5 MB per origin (recommendation, can be increased)
Web SQL Database	W3C, Working Group Note, November 2010	Relational data-base (SQLite), users a variant of SQL, supports transactions and callbacks, includes synchronous and asynchronous APIs	many(SQL data types)	5 MB per origin (recommendation, can be increased)

IndexedDB	W3C, Working Draft, May 2012	indexed and hierarchical key-value pair data, duplicate values for a key, indexes, supports transactions and callbacks, includes synchronous and asynchronous APIs	Many (e.g., JSON, array, string, and date)	50 MB per origin (Firefox, can be increased)

Source: Table 5 [8]

These web applications are capable of using the web browser storage or in memory storage, as well as distant storage to synchronize data for long term persistency. Following figure provides a basic overview of Web and Hybrid Mobile applications data access model.

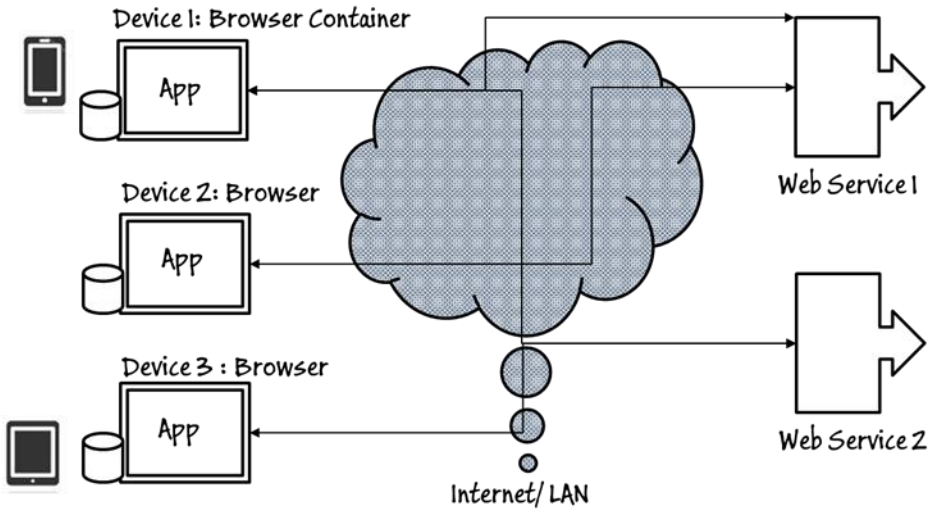

Figure 3: Web and Hybrid Mobile Applications

1.4 Research Problem

When developing JavaScript Heavy web applications or mobile hybrid applications that runs on web browsers or web containers with client side data handling, it requires to persist JavaScript Objects to RESTful web services for long term persistency and sharing. Out of these, majority of the applications depend on user interactions to persist JavaScript Objects with RESTful web services. If the user interaction frequency becomes high, it could lead to higher frequency of JavaScript Object persistence requests to the RESTful web services which could overall increase the response time for persistence. On the other hand, queuing these requests increases staleness of modified Objects. However, this creates challenges for web developers to additionally include adaptive timing to improve persistence response time which impacts the user satisfaction in terms of overall application performance. In addition, JavaScript Object Size, user interaction frequency and platform differences, network performance is difficult to predict.

This research is mainly focused on developing an adaptive framework that will facilitate web developers in building JavaScript heavy applications which will improve JavaScript Object persisting performance for frequently modified JavaScript Objects in between web browser client and RESTful web services, with minimal staleness and increased user satisfaction.

CHAPTER 2

LITERATURE REVIEW

This chapter contains the detailed information about related research that builds the foundation for the proposing methodology to address the given problem.

2.1 Storage in Web Applications Context

Web applications are capable of storing, data on the web servers for long term persistence and on client-side storage for temporal persistence. Conventional interaction model with server, has the following disadvantage [8].

1. Need of network connection availability for data persistence
2. Increased network traffic
3. Increase server load

These leads to the reduced performance. The conventional client-server interaction model is shown in the following figure.

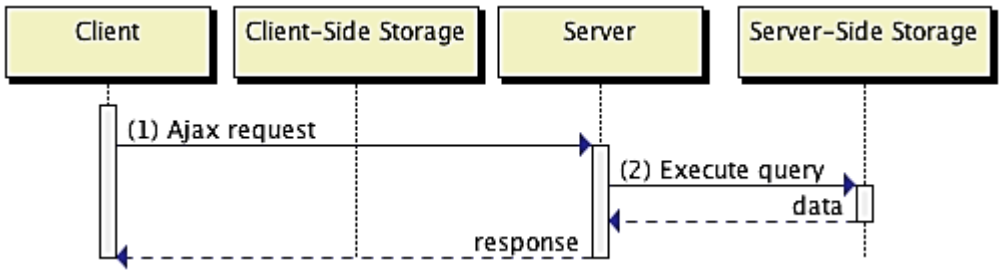

Figure 4: Client-server Interaction Model

Source: Figure 1 [8]

Usage of Client-side storage can be used to minimize the above disadvantages illustrated using the following interaction model.

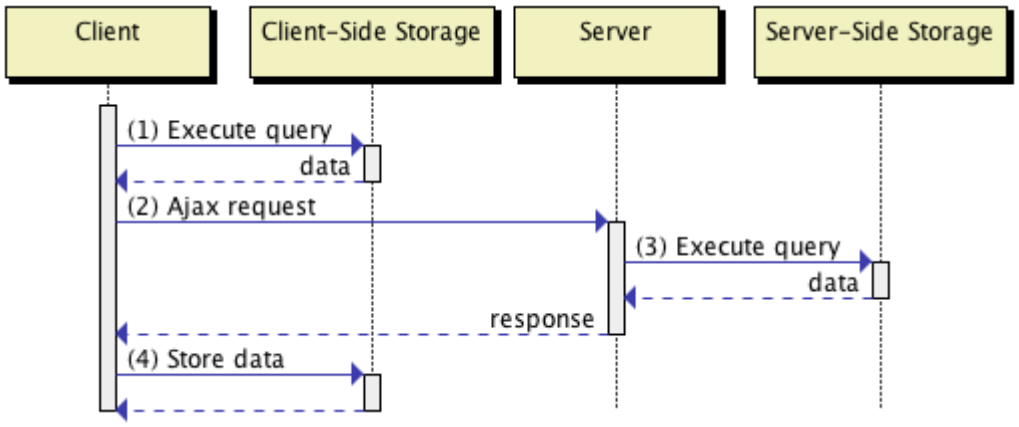

Figure 5: Client-server Interaction Model Utilizing Client-side Storage

Source: Figure 2 [8]

However this introduces another challenge of data staleness, where client side is not having the most up-to-date data.

2.2 Abstracting Storage in Web Applications

2.2.1 Separating Web Applications from User Data Storage with BSTORE

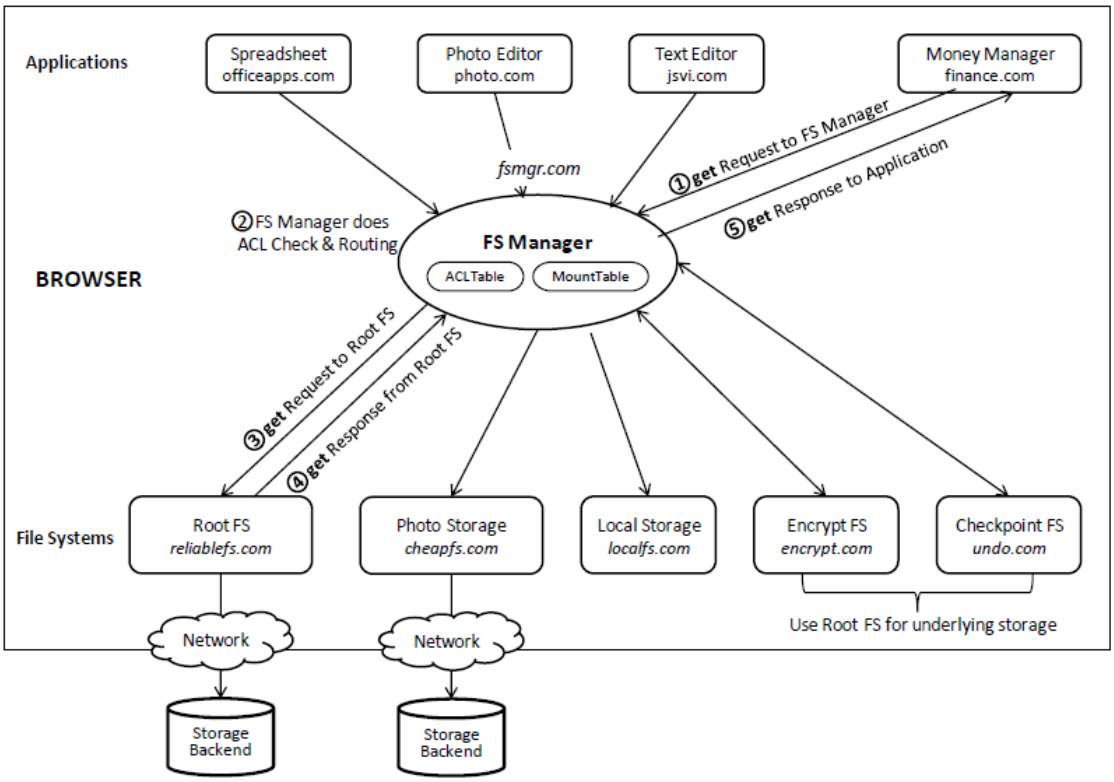

Figure 6: BSTORE Architecture

Source: Figure 1 [9]

BSTORE is a framework that allows web developers to separate the application logic from its data storage providing a unified API to access data. It is designs to achieve independence between applications and storage provides allowing almost any application avoiding the need for reserving any special functionality for user.

Above figure shows an overview of the BSTORE architecture where component in the browser corresponds to a separate window whose web page contains JavaScript code that

communicates with BSTORE. All requests sends from each web page to the BSTORE file system are mediated by the FS manager.

BSTORE client-side components are implemented using JavaScript targeting FireFox and Google Chrome web browsers. The specialty of BSTORE is that, each component runs on separate browser window loaded from a separate domain which also becomes a drawback when it comes to commercial applications where BSTORE to work requires multiple windows to be open at the same time. Communication between each of these browser windows are done via postMessage. BSTORE communicates with the server storage system through AJAX and authenticated using, plain user credentials.

2.3 Utilizing Client-side Storage for Web Application Performance

2.3.1 Sync Kit

Sync Kit [10] is a client/server toolkit for improving the performance of data intensive websites. Sync Kit offers a mechanism to offload some of the data from web server to (Google) Gears [11] on the client side, requiring the installation of the Gears plug-in in order to work. In future Gears could be, however, replaced with a W3C-specified client-side storage mechanism to get rid of the dependencies. On the server side, Sync Kit requires a specially designed web server that handles Sync Kit requests. In their case, Sync Kit was implemented as a collection of scripts for the Django10 web framework. The focus of Sync Kit is to use Gears to store (cache) segment of web pages called web page templates on the client side. These templates include a JavaScript library and data endpoint definitions to access dynamic contents residing on the server. Data endpoints, on the other hand, cache database objects to Gears and keep the cached data consistent with the backend database. When a browser loads a web page for the first time, the template is returned as a response from the Sync Kit aware server and stored in the client-side storage. Then, new data is requested via template's data endpoints. Finally, the retrieved data is added to the template, cached on the client-side storage, and the result is displayed to the user.

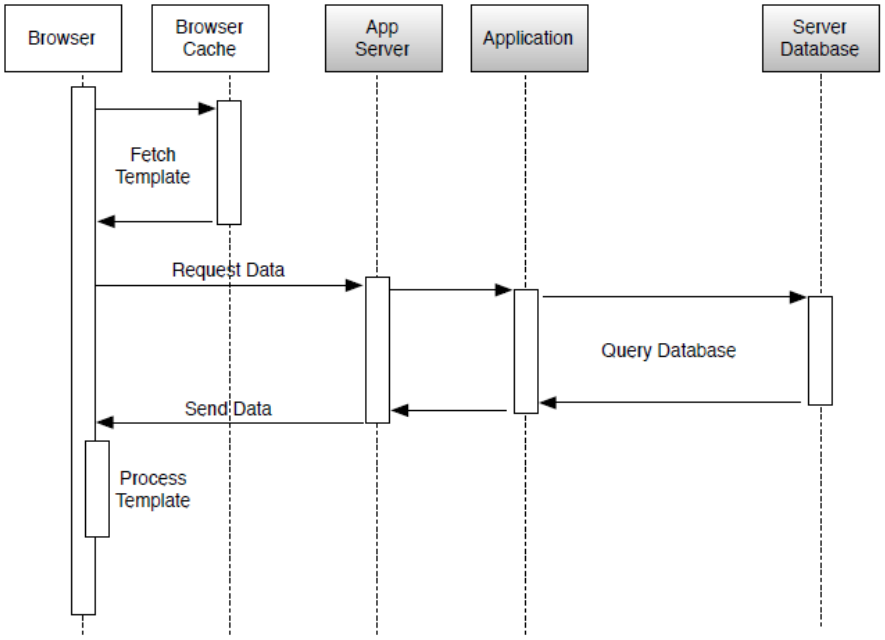

Figure 7: Template Caching: Template Rendering is on Client

Source: (b) [10]

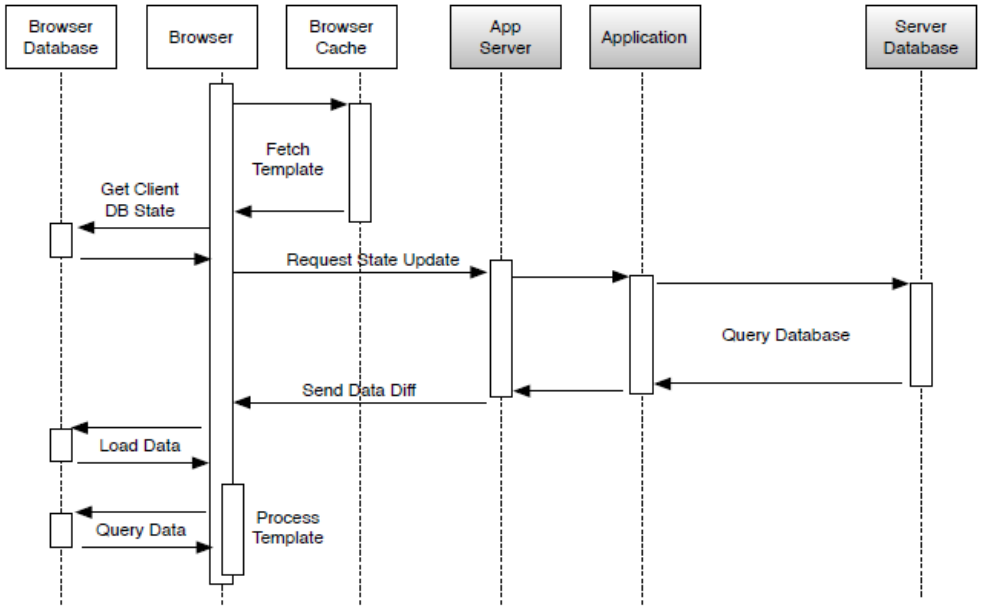

Figure 8: Sync Kit: Template Rendering and Database Accesses

Source: (C) [10]

According to the research [10], their solution given by Sync Kit reduces server load by a factor of four and data transfer up to 5% compared to the traditional approach, when cache hit rates are high.

One of the main challenges of Sync Kit is that it requires Sync Kit aware web server implementation in server side that limits its usage in practical web application developments where more of these applications are built on top of third party APIs and services. Apart from that, it restricts the client-side implementation to be based on Sync Kit provided client-side template mechanism where better mechanisms are coming out more frequently suiting different JavaScript libraries and framework stacks.

2.3.2 Silo

Silo [18] is a framework that allows to create fast-loading web applications. It uses JavaScript and Client-side storage based on current web standards allowing the approach to be working on modern web browsers. On the server side, a Siloaware web server is required. The main purpose of Silo is to use Web Storage to cache website's JavaScript and CSS chunks (2 kB in size) on the client side so that it does not needed to be requested from the server in each page load.

When a browser requests for a specific web page, the server returns a modified version of the web page containing a small JavaScript file with a list of required chunk ids and logic to fetch missing chuck ids. Then the loaded JavaScript file logic checks the client-side storage for available chunks and informs the server of the missing chunks using their ids where the server replies with the raw data for the missing chunks. Finally, the web page is reconstructed into its original form on the client using the JavaScript logic. Apart from that, for the latter HTTP request, the round trip can be completely eliminated by utilizing HTTP cookies in the initial HTTP request for sending the already available chunk ids. Silo was evaluated with multiple real-world websites, such as CNN, Twitter, and Wikipedia. Based

on their experiments it has proven that Silo can reduce web page load times by 20-80% for web pages with large amounts of JavaScript and CSS.

With the latest HTML5 standards, almost all the functionality provided by Silo can be achieved using App Cache and Client-side storages but stored separately in the browser. But the challenge addressed by providing a unified way of loading web page content (Both content and data in forms of chunks) provide several other challenges when it comes to practical web application development, making things really hard to test and debug.

2.4 Client-Server Data Synchronization and Persistence

2.4.1 Data Synchronization Patterns in Mobile Application Design

Many mobile applications are data centric [12] . This is same with modern web applications. This research [12] describes common concerns related to data synchronization as a collection of patterns, grouped by the problems they address. These patterns are based on examining open source applications, inspecting platforms and frameworks of mobile systems and examining experiences developing mobile applications taking input from different experience developers.

The paper states that it is not possible to find one-size-fits-all solution for data synchronization which provides the need of configuring a data synchronization framework to suit different domains and environments. This becomes really obvious when considering indirect factors like power usage, especially when it comes to mobile devices, and efficient synchronization mechanism [13] needs to be selected based on various criteria.

As described in the paper these collection of patterns can be further classified in to the following 3 collections of patterns.

1) Data Synchronization Mechanism Patterns
2) Data Storage and Availability Patterns

3) Data Transfer Patterns

2.4.1.1 Data Synchronization Mechanism Patterns

These collection of patterns addresses the question: “when should an application synchronize data between a device and a remote system (such as a cloud server)?” [12]. These patterns can be further classified into two types based on the mechanism that the data transfer occurs.

- Asynchronous Data Synchronization:
  Where data synchronization happens asynchronously without blocking the user interface.
- Synchronous Data Synchronization:
  Manage a data synchronization event synchronously; blocking the user interface while it occurs.

2.4.1.2 Data Storage and Availability Patterns

These collection of patterns address the questions: “how much data should be stored?” and “how much data should be available without further transfer of data?” [12]. These considerations often depend on the limitations of mobile platforms which is similarly applicable for web browser platforms. Based on the amount of data stored locally, these collections of patterns can be further classified in to following types.

- Partial Storage:
  Synchronize and store data only as needed to optimize network bandwidth and storage space usage.
- Complete Storage:
  Synchronize and store data before it is needed so the application has better response or loading time.

2.4.1.3 Data Transfer Patterns

These collection of patterns addresses address the problem of transfer quantity in set reconciliation: “how can we synchronize between sets of data such that the amount of data transmitted is minimized?” [12]. Selection of the transfer quantity needs to be carefully selected based on the application domain to optimize the network bandwidth. Further classification of data transfer patterns are as follows.

- Full transfer:

  On a synchronization event, the entire dataset is transferred between the mobile device and the remote system.
- Timestamp Transfer:

  On a synchronization event, only the parts of the dataset changed since the last synchronization are transferred between the mobile device and the remote system using a last-changed timestamp.
- Mathematical Transfer:

On a synchronization event, only the parts of the dataset changed since the last synchronization are transferred between the mobile device and the remote system using a mathematical method.

2.4.2 Efficient Synchronization of Replicated Data in Distributed Systems (nSync)

This research present nsync [14], a tool for synchronizing large replicated data sets in distributed systems. Although these concerns are not directly applicable for web applications, the strategy of optimizing the synchronization plans provides a sound approach in optimally synchronizing batches of replicated data when multiple servers are present [15]. When it comes to data synchronization, nsync approach computes nearly optimal synchronization plans before actual synchronization happens to minimize the amount of data needs to be transferred using hierarchy of gossip algorithms that take the network topology into account.

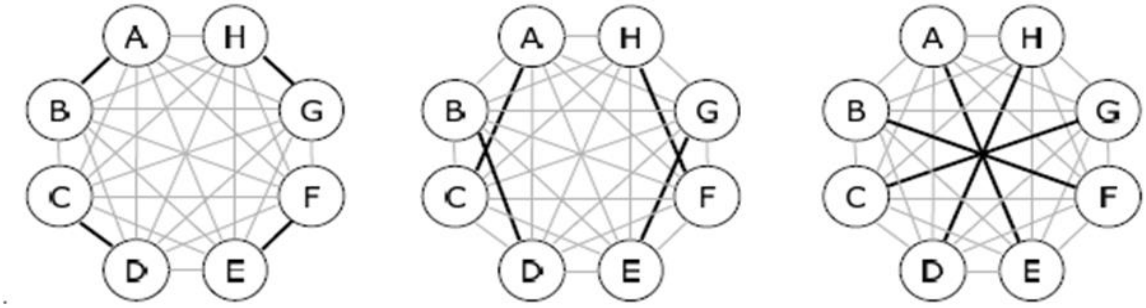

Figure 9: Three rounds of n2n syncs can synchronize eight nodes

Source: Fig. 1 [14]

The above figure illustrates the capability of nsync when it comes to synchronize multiple nodes. In native approach without nsync, each single connection would trigger an independent disk access, provoking many disk head movements and therefore resulting in a slow data transfer rate. To avoid this nsync approach uses only node to node (n2n) syncs where each node participates in at most one n2n sync at a time.

2.4.3 Automated Object Persistence for JavaScript

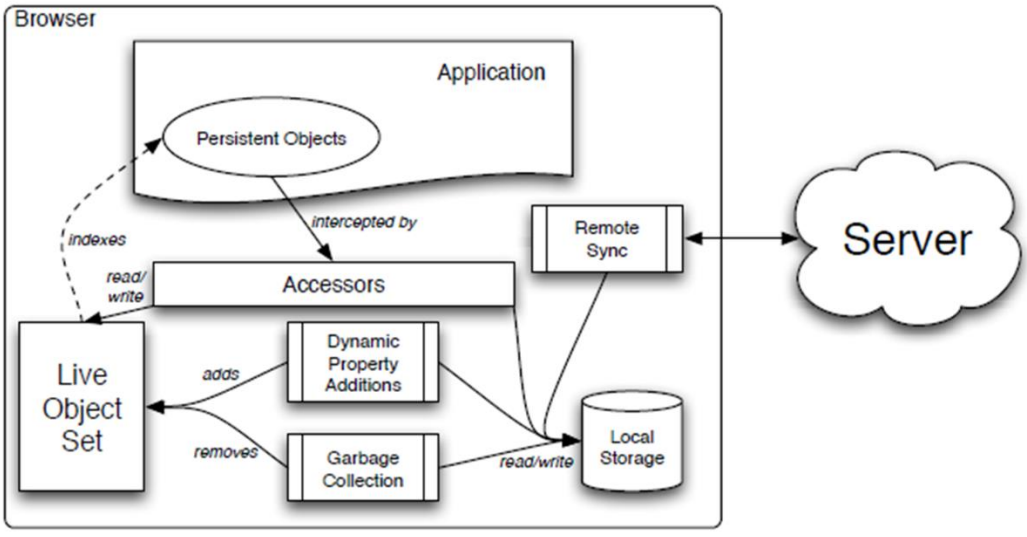

Figure 10: Architecture Diagram of Automated Object Persistence Mechanism

Source: Figure 1 [16]

Automated persistence of JavaScript Objects [16] provides a transparent mechanism on persisting structured objects. Still with automated persistence developers needs to determine how and when to save data using the heap-allocated JavaScript objects to local browser storage balancing several tradeoffs. If the changes are saved quite often, it causes degrade of performance and the single threaded nature of JavaScript (At the time being research was conducted) causes to freeze the browser when saving single large batch of changes. To deal with this problem, they have investigated the user of persistence by reachability [17] for JavaScript. This approach divides the JavaScript object heap into transient and persistent sub-graphs allowing the framework to manage persistence-related tasks periodically. This way the framework manages between transient and persistence objects not immediately persisting the objects each time it gets updated.

Secondly to synchronize data between client-storage to server, they have investigated and developed an application framework for automated persistence at an individual object granularity allowing the browser to send updates of the data for individual objects which maps directly to the object model created by the application developer. When there is a conflict happens, the framework triggers a conflict resolution callback where the developer needs to write a semantically relevant resolution strategy.

Automated Object Persistence framework uses special functions called accessors in JavaScript to identify whether an object attribute is updated. These accessors are called transparently when an object property is read or written. In this framework, when a write operation happened triggering the accessor, it records that the object as mutated and stores the written value so it can be returned by the read accessor. Since JavaScript is dynamic language, a scheduled process runs to identify whether new properties are added to the JavaScript object, carefully selecting the schedule intervals to avoid overhead.

As shown in the Architecture Diagram in Figure 11, accessors act as a read barrier to alert the framework when a persistent object has mutated. These Mutations including both accessors and the dynamic property addition maintenance task, causes objects to be added to the live object set that is constantly updated by the application and have their state stored in local storage. This is to assure that the objects are stored safely in the browser if the browser crashed before persisting the objects. Garbage collection removes dead objects from local storage. Finally, mutations are sent to a server during remote synchronization.

This paper provides a sound approach, in providing semi-transparent object persistence mechanisms using a framework. But there are shortcoming when it comes to latest web browser technology advancements. Some of the key decision made based on single thread nature of JavaScript is no longer valid since now JavaScript supports multiple threads.

2.5 Response time Impact for Web Application User Experience

In modern web applications user satisfaction is heavily depend on application response times. With advancements of web browser performance, network speeds, one might not expect to have to deal with performance issues in web applications such as slower response time. But these issues remain a very real concern today.

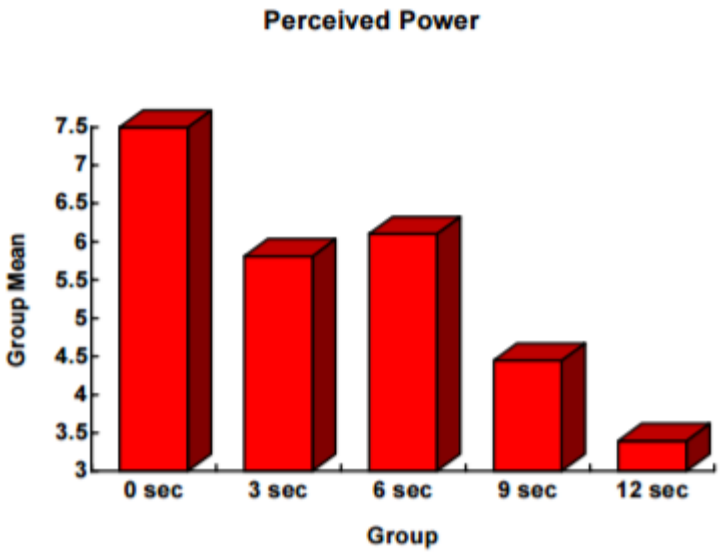

Figure 11: Perceived Power Construct

Source: Figure 4 [18]

The research [18] indicates that system response time has a related affect for user satisfaction. According to the study above figure shows that that indeed satisfaction does decrease as response time increases. It also showed that for web applications, it appears to be a level of intolerance in the 12-second response range.

CHAPTER 3

METHODOLOGY

The JavaScript Object Persistence framework consists of multiple components. The persistence algorithm in the framework intervenes the JavaScript Object persisting action and adjust the persistence timing for high frequency actions by adding a modification window, optimizing for reduced average persistence time. After the modification window is calculated over a time period, it is stored with the metadata of overall operation for future prediction of web application operation.

3.1 JavaScript Persistence Objects

In web applications object representation can take multiple formats. Most popular form of JavaScript objects created with JavaScript Object Literal Notation. These Object consists of attributes and collections in form of primitives and arrays. Example JavaScript Object is shown in the following illustration.

```
{
  "todos": [
    {
      "title": "Item 1",
      "completed": false
    },
    {
      "title": "Item 2",
      "completed": false
    },
    {
      "title": "Item 3",
      "completed": false
    },
    {
      "title": "Item 4",
      "completed": false
    }
  ],
  "request_id": "8ae873a4-5634-75d6-3bf8-ed93bf4572d1"
}
```

Figure 12: JavaScript Object Illustrated

Apart from these JavaScript Object Notation, JavaScript Engines in web browsers also allows to upload binary objects which could be directly sent to web services.

3.2 Object Persistence Web Services

Object Persistence Web Services are set of web services which allows to store Objects sent over HTTP Protocol with a payload of JSON or Blob objects. These services internally handle the persistence complexities of the objects and also provides other capabilities such as updating, deleting objects.

3.3 Factors and their ranges

JavaScript Object Persistence performance is impacted by various parameters such as

- Object persistence frequency
- Object size
- Network performance (Bandwidth and Latency)
- Platform (Browser/Operating System)
- Web Application specific parameters
- Performance of persistence Web Service

If we take a particular feature of a web application that needs to persists a JavaScript Object, it has a size, and user interaction frequency to persist the object within a quantifiable range which allows to model a characteristics of a feature. Optionally other factors also help to classify the user such as Authenticated User Identity, Platform identifiable data (Browser Cookies) which could be used for further increase the predictability of user behavior with respect to persist frequency. Apart from that most of these factors impact persistence response time, in different levels and it is quite difficult to identify ranges for some of these factors, for example Web Application Specific parameters or Performance of persistence Web Service, where it will be different based on implementations and also changes over time. Therefor an adaptive method is needed to

predict the modification window for a given feature to optimize the performance for Object Persistence for high persistence frequencies.

3.4 JavaScript Object Persistence Models

3.4.1 Persistence Model: Direct Persistence

In modern web application, the data model change as a result of user DOM interactions, is automatically reflected in JavaScript Object through MVVM frameworks such as Angular.js. If we model the interaction with the object, it is a series of modifications that happens with time. To store the state of the object in the remote server it requires to trigger a server data synchronization Ajax request that takes a round trip time (RTT) to the sever to persist the data remotely and send back the acknowledgement. Following figure illustrates a model on how these steps taken place.

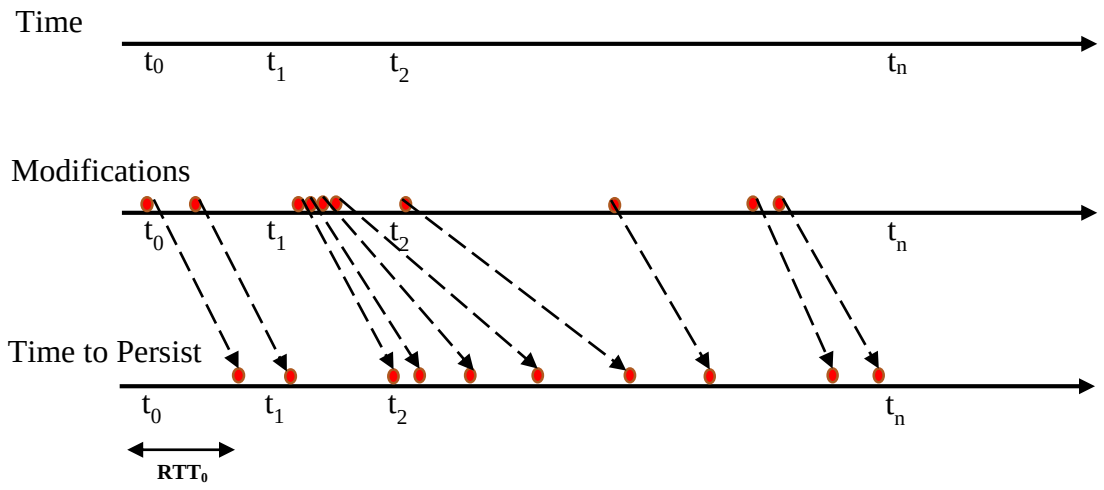

Figure 13: JavaScript Persistence Illustrated

RTT- Round Trip Time (Client-Server-Client)

W- Modification Window

In the direct persistence model shown above, it immediately persists the Object, when the object identifies a change in its attributes.

3.4.2 Persistence Model: Dynamic Modification Window Persistence

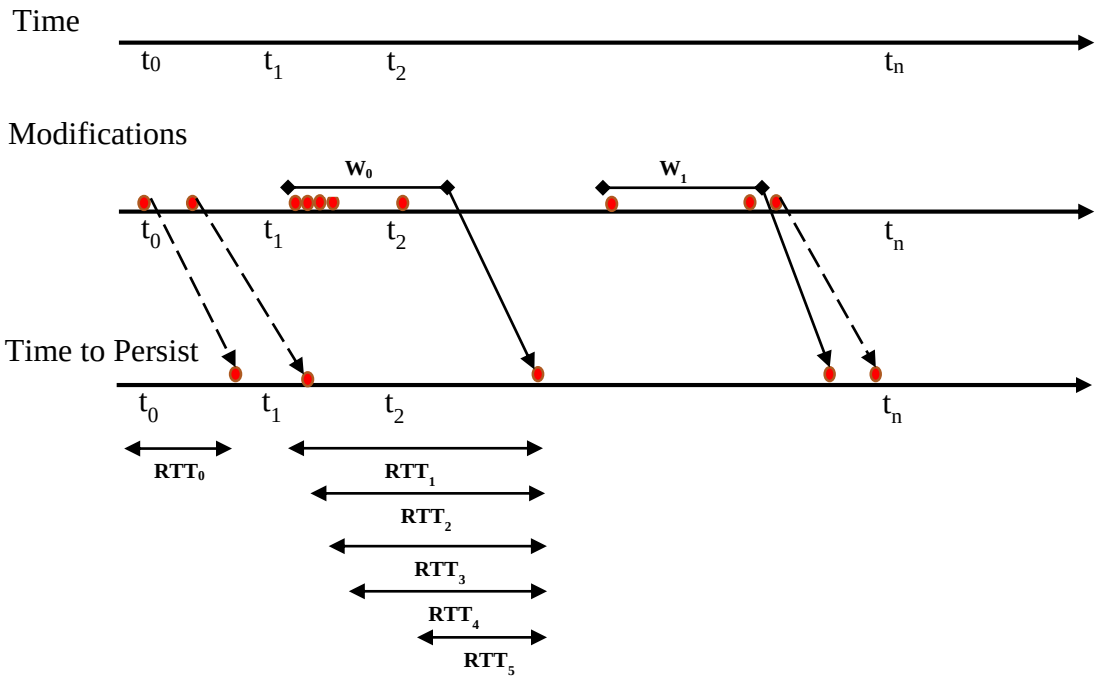

Figure 14: Dynamic Modification Window Persistence

This forms the following equation for round trip time Average (WRTT (m)) taking modification window into account, to persist an object after **m** requests.

$$\text{Window RTT Average [WRTT (m)]} = \frac{\sum_{x=1}^{x=m} RT(x) + W(x) - ST(x)}{\sum_{x=1}^{x=m} x}$$

$$\text{Request RTT for request } \mathbf{m}\text{, [RTT (m)]} = RT(x) - ST(x)$$

Where:

ST(x) = Start Request Time, RT(x) = Response Received Time, W(x) = Modification Window for a request **x**.

3.4.2.1 Predicting Next Request Round Trip Time

To calculate the Persistence Window for Dynamic Modification Window Persistence, Polynomial Regression is being used to predict the next request round trip time. Fitting higher order polynomial regression with model with persistence data sets and calculating the coefficients as shown below,

$$y = 0x\text{^}10 + 0x\text{^}9 + 0x\text{^}8 + 0x\text{^}7 + 0x\text{^}6 + 0x\text{^}5 + 0x\text{^}4 + -0.12x\text{^}3 + -5.73x\text{^}2 + 765.62x + 943.16$$

It has been found that a polynomial regression with order 3 is sufficient to predict the next request round trip time. After retrieval of each response, each round trip times are logged and use to predict the next value using polynomial regression.

3.4.2.2 Calculating Dynamic Persistence Window

W_{size} is calculated as follows after **m** requests

If $RTT(m+1) - RTT(m)) > \Delta RTT_{threshold,}$ Then

$W(m) = W(previous) + \Delta W_{increment}$

Else

$W(m) = MAX((W(previous) - \Delta W_{decrement}), 0)$

End

Where:

Round Trip Time Threshold = $\Delta RTT_{threshold,}$ Predicted Next Request Round Trip Time = RTT(m+1), Modification Window Threshold = $\Delta W_{threshold}$, Modification Window Increment = $\Delta W_{increment,}$ Modification Window Decrement = $\Delta W_{decrement}$

Other matrices calculated

Minimum Round Trip Time = RTT_{min}, Maximum Round Trip Time = RTT_{max}

3.4.3 Persistence Mode: Predictive Modification Window Persistence

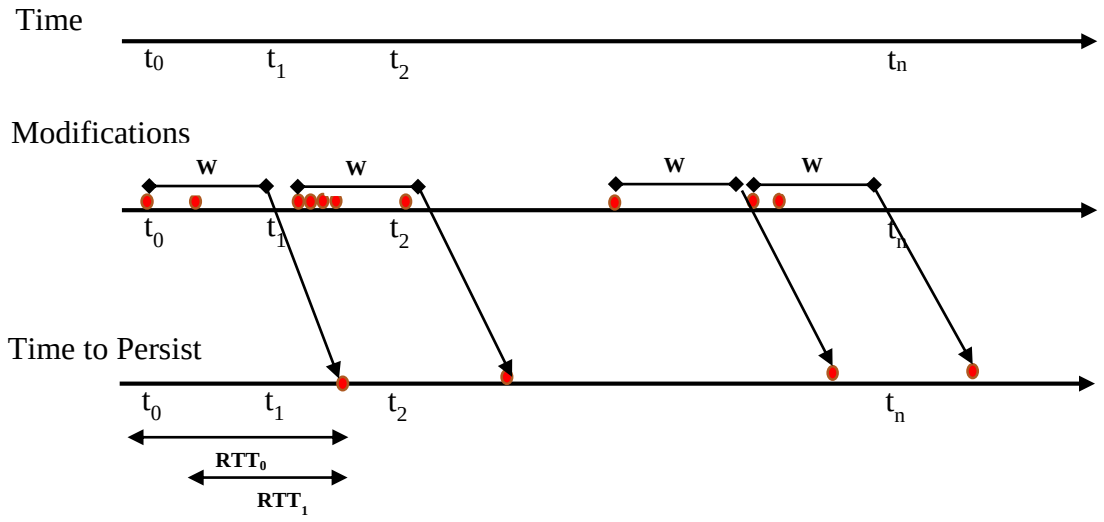

Figure 15: Predictive Modification Window Persistence

Using the pre-computed modification window from dynamic modification window persistence model, this model initializes the modification window at the beginning of the persistence request. To predict the Initial Persistence Window Size $W_{initial\ size}$, further research needs to be carried out by using the data from dynamic modification window persistence model experiments.

3.5 Experiment Setup

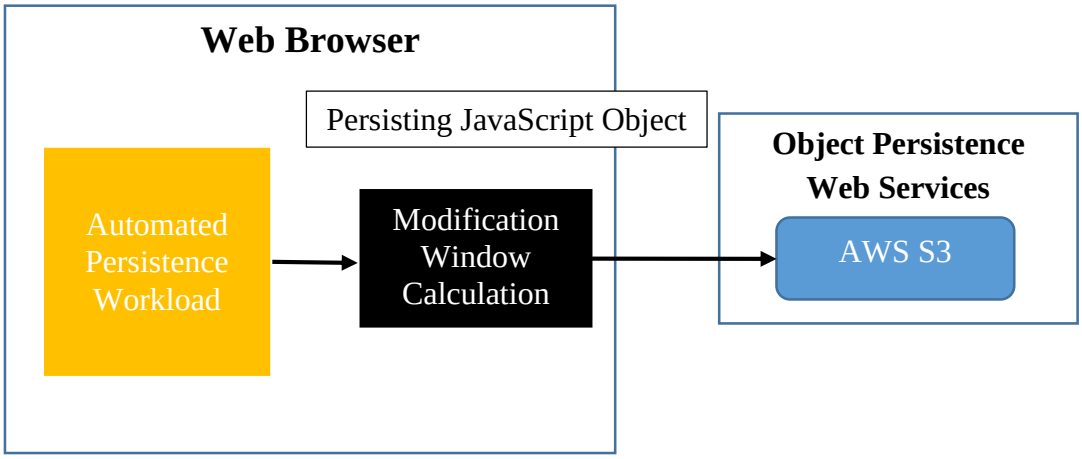

Figure 16: Experiment Setup

3.5.1 Automated JavaScript Object Persistence Experiment

Persistence workload is created with JavaScript objects considering JavaScript object size, persistence frequency and network in to consideration.

3.5.2 Predicting Next Request Round Trip Time

To predict Next Request round trip time, following function is used where the order of polynomial regression algorithm is specified.

```
predictNextRTT = function(req) {
    if (REQ.log.length < 3) {
        return req.requestRTT;
    };
    var result, reqLogs = utils.clone(REQ.log);
    reqLogs.push([reqLogs.length, null]);
    result = regression('polynomial', reqLogs, 3);
    return Math.abs(Math.round(_.last(_.last(result.points))));
},
```

Figure 17: Predicting Next Request Round Trip Time

3.5.3 Changing Persistence Frequency

JavaScript Object persistence frequencies are simulated using a JavaScript interval function and using QUnit framework.

```
asyncTest('Experiment with 1KB and 20 requests/second with 100 iterations', function() {
    var iterations = 100,
        uploadLogs;

    var refreshIntervalId = setInterval(function() {
        iterations--;
        reqwin.adaptiveWindow(EXPERIMENT.upload).then(function(logs) {
            uploadLogs = logs;
        })
        if (iterations === 0) {
            clearInterval(refreshIntervalId);
        }
    }, 50);

    expect(1);
    _.delay(function() {
        var logs = formatLogs(uploadLogs);
        ok(true, JSON.stringify(logs));
        start();
    }, 20000);
});
```

Figure 18: Simulating JavaScript persistence frequency

3.5.4 Changing Latency, Network Bandwidth

Throughout the experiment, Latency is approximately kept same using the same geographical distance between the client and the persistence service. Also many iterations are carried out to reduce the error factor when comparing results. Network bandwidth is simulated using Google Chrome Developer Tools as shown below.

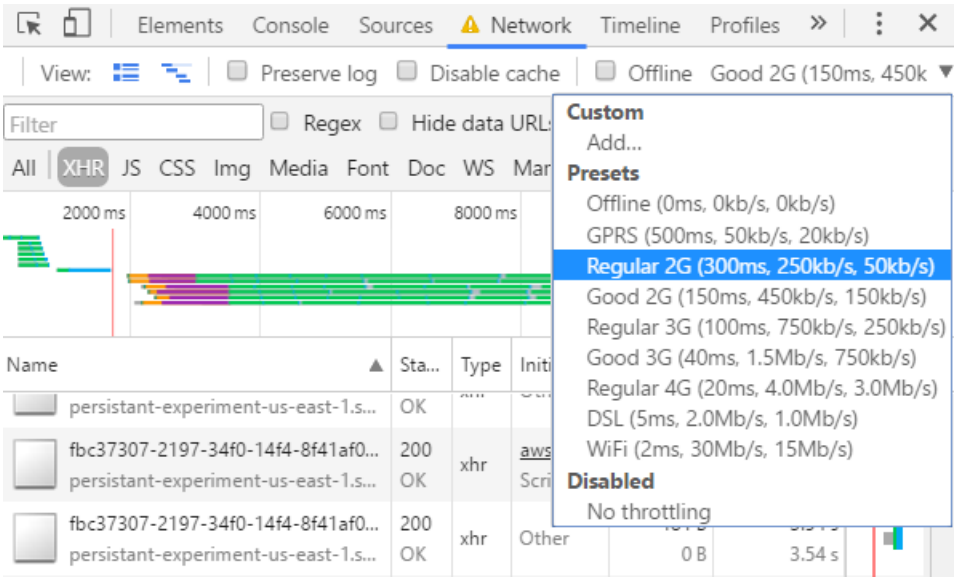

Figure 19: Simulating Network Bandwidth

3.5.5 Changing Persistence Modification Window Algorithm Parameters

The persistence algorithm works asynchronously adding persistence window for a provided “Save Callback” function so that, the actual persistence logic is written outside the algorithm itself. This helps to inject various persistence mechanisms reusing the algorithm implemented. For instance, saving on Amazon S3 Persistence Service is written as follows.

```
(function(W, $) {
    W.AP = W.AP || {};
    AWS.config.update(W.AP.AWS_CREDENTIALS);
    var recordsLog = {};

    AP.persist = {
        upload: function(obj) {
            var deferred = Q.defer(),
                s3 = new AWS.S3(),
                params = {
                    Bucket: 'persistant-experiment-us-east-1',
                    Key: obj.request_id,
                    StorageClass: 'REDUCED_REDUNDANCY',
                    Body: JSON.stringify(obj)
                };
            recordsLog[obj.request_id] = W.AP.utils.clone(obj);
            s3.upload(params, function(err, res) {
                deferred.resolve(recordsLog[res.key]);
            });
            return deferred.promise;
        }
    };
})(window, jQuery);
```

Figure 20: Persistence logic in Amazon S3

In the persistence algorithm, there are several parameters we need to adjust to optimize the algorithm for varying scenarios. During this experiments the effect of these parameters are further analyzed and predicting it for various scenarios requires further research using an analytical approach for real web applications, collecting the data of its performance results. Following are the parameters that needs to be modified to optimize the algorithm for specific usage.

1. Initial Window Size
2. Window Increment and Decrement
3. Round Trip Time Threshold

3.5.6 Implementing Persistence Window Algorithm

```
adaptiveWindow = function(saveCallback) {
    var deferred = Q.defer(),
        currentRequestAt = utils.timestamp(),
        waitingWindow;
    records.lastRequestedAt = records.lastRequestedAt || utils.timestamp();
    waitingWindow = currentRequestAt - records.lastRequestedAt
    REQ.total++;

    records.window.push({
        request_at: currentRequestAt
    });

    var executeRequest = function() {
        var requestId = utils.guid();
        records.lastRequestedAt = utils.timestamp();
        records.saving[requestId] = utils.clone(records.window);
        records.saving[requestId].request_at = records.lastRequestedAt;
        records.window.length = 0;
        saveCallback(requestId, records.lastRequestedAt).then(function(obj) {
            W.count++;
            var requestRTT = (utils.timestamp() - records.saving[obj.request_id].request_at);
            W.time = W.time + requestRTT;
            records.saving[obj.request_id].forEach(function(record) {
                var result = {
                    requestRTT: requestRTT,
                    serveTime: utils.timestamp() - record.request_at + W.size
                };
                updateVariables(result);
                adjustRequestWindow(result);
            });
            deferred.resolve(LOGS);
        });
    };

    if (!W.size || waitingWindow >= W.size) {
        executeRequest();
    } else if (!records.pendingDelayedExecutions) {
        records.pendingDelayedExecutions = true;
        _.delay(function() {
            records.pendingDelayedExecutions = false;
            executeRequest();
        }, (W.size - waitingWindow));
    }

    return deferred.promise;
};
```

Figure 21: Persistence Window Buffer and Persistence Callback

Persistence Window Calculation Algorithm has 3 steps. In the first step in Figure 21, algorithm buffers further requests, if they are within the Window Size. If the Window Size is 0, then all the persistence requests are being sent to the server. To send particular versions of the object change to the server, the persistence callback is called and upon its success. In the second step algorithm re-calculates its statistics as in Figure 22 which is used to dynamically calculate the persistence window during the next step.

```
updateVariables = function(record) {
    REQ.served++;
    REQ.time = REQ.time + record.requestRTT;
    REQ.average = Math.round(REQ.time / (REQ.served || 1));
    RTT.average = Math.round(W.time / (W.count || 1));
    RTT.min = (record.requestRTT > RTT.min) && RTT.min ? RTT.min : record.requestRTT;
    RTT.max = (record.requestRTT > RTT.max) ? record.requestRTT : RTT.max;

    var log = utils.clone({
        RTT: RTT,
        W: W,
        REQ: REQ
    });
    log.REQ.windowRTT = record.windowRTT;
    log.REQ.requestRTT = record.requestRTT;
    LOGS.push(log);
},
```

Figure 22: Adjusting Algorithm Statistics

During the third step, it adjusts its Window size accordingly using Persistence Mode: Predictive Modification Window Persistence algorithm as shown in Figure 23.

```
adjustRequestWindow = function(req) {
    if (!W.disabled) {
        if (RTT.nextRTT - req.requestRTT > RTT.threshold) {
            W.size = W.size + W.increment;
        } else {
            W.size = W.size - W.decrement;
            W.size = W.size > 0 ? W.size : 0;
        }
    }
},
```

Figure 23: Adjusting persistence window

3.5.7 Executing Workload with Predictive Modification Window Persistence

The experiment is carried out, adding the Predictive Modification Window Persistence algorithm, in between the Automated Object Persistence Workload and Amazon S3 Web service. Using the automated interval script, workload frequency is simulated. Object size and persistence algorithm window parameters are modified accordingly. Apart from that to simulate the network bandwidth Google chrome developer tools are used to specify different bandwidth categories. After simulation completes, a delayed function shows experiment results and captured for further analysis.

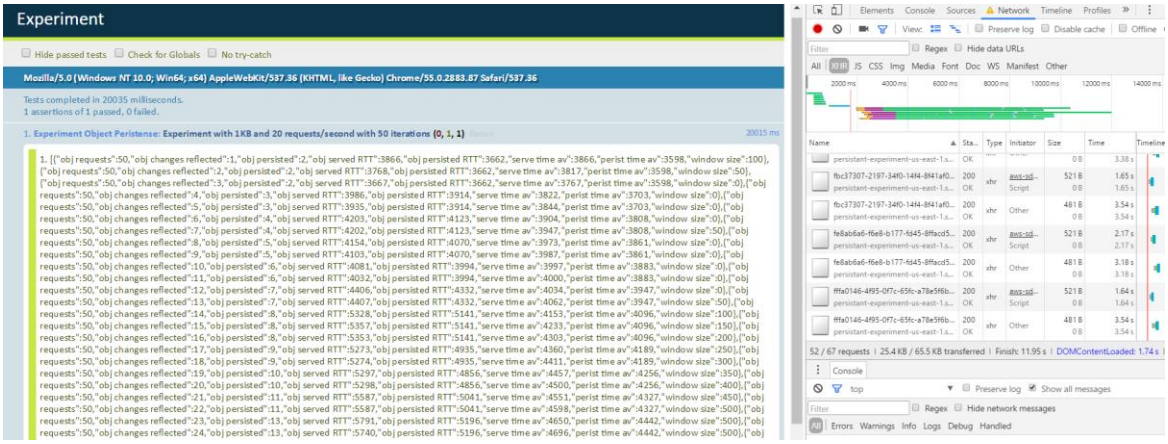

Figure 24: Displaying results using QUnit Framework and Google Chrome inspector

Also the results of the experiments automatically added to an Excel Sheet for further analysis.

CHAPTER 4

Results & Conclusion

4.1 Directly persisting in high frequencies of object modifications

When the object is frequently modified and needs to be sent to the persistence service for long term storage, for high frequent modifications, it increases the round trip time due to the network congestion and also persistence service throttling and performance limitations. Following graph shows how persistence round trip time changes for a 1Kb object with increasing frequency on different network speeds and bandwidth simulations.

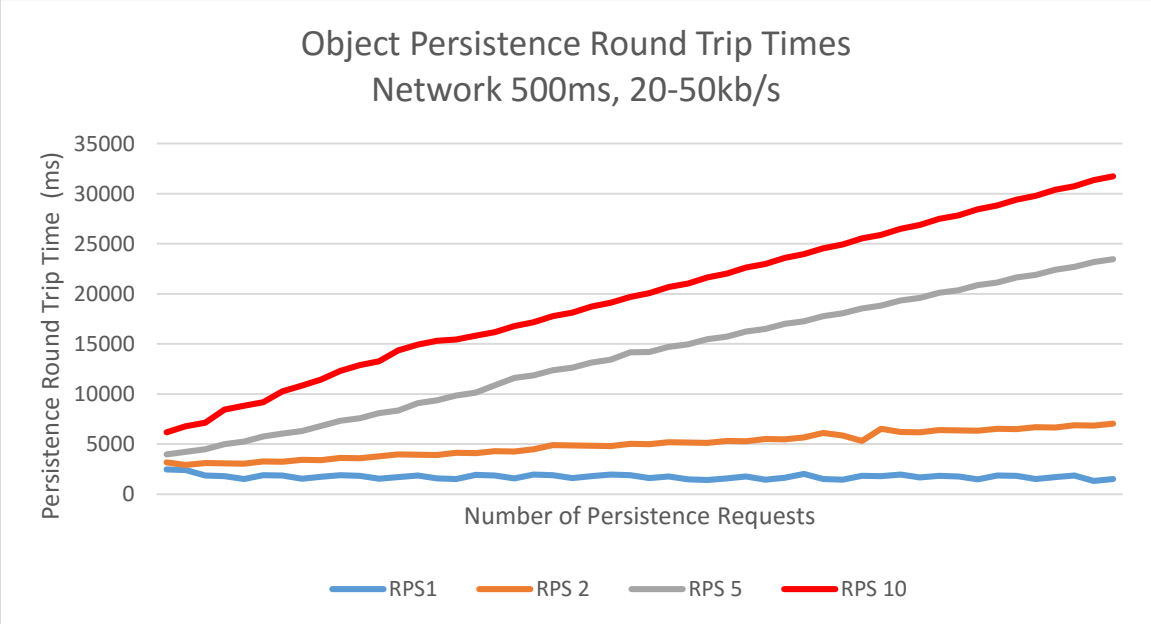

Figure 25: Round Trip Time & Frequency 10 req/s in Network 500ms, 20-50kb/s

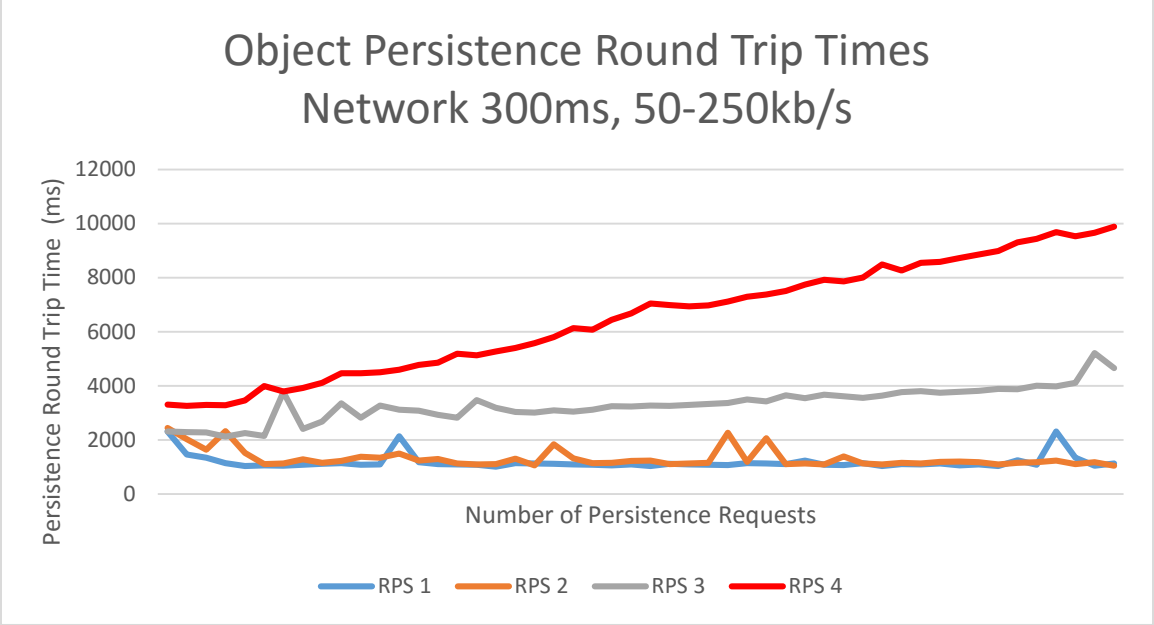

Figure 26: Round Trip Time & Frequency 10 req/s in Network 300ms, 50-250kb/s

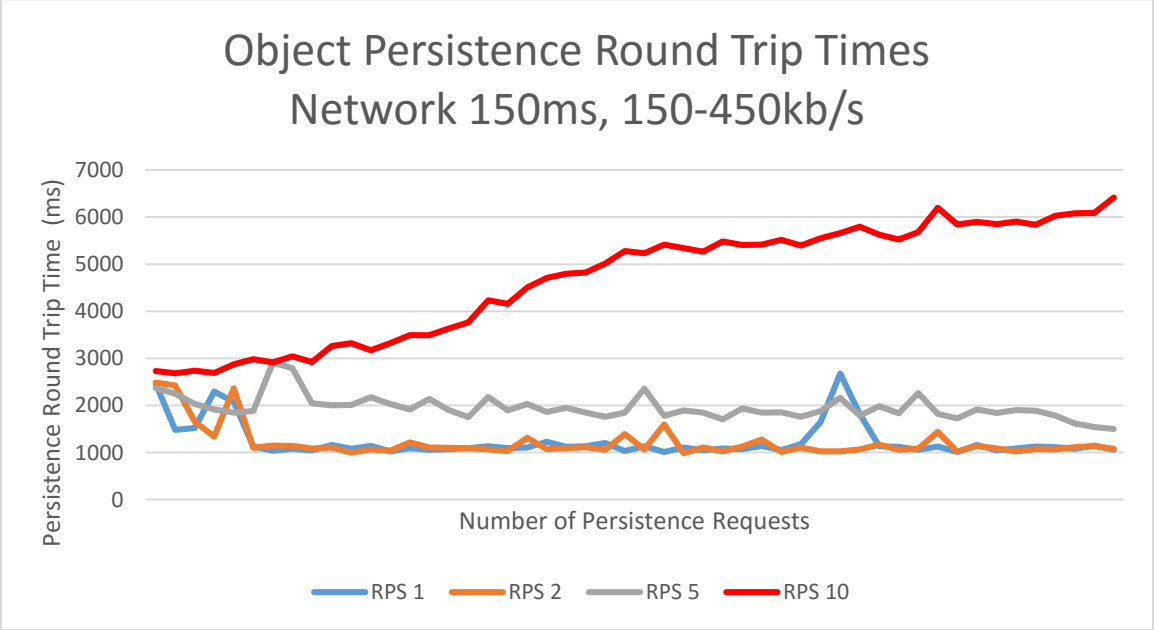

Figure 27: Round Trip Time & Frequency 10 req/s in Network 150ms, 150-450kb/s

According to the diagrams above, increase in frequency affects the round trip time of persistence, inverse proportionally for higher frequencies. The network speed affects the round trip time inverse proportionally.

4.2 Persisting objects with fixed persistence window

Since the most deviation on persistence timing occurred with the Network setup of 300ms, 50-250kb/s, which is shown in Figure 25 same, experiment is carried out with different set of fixed window values and the results are shown below.

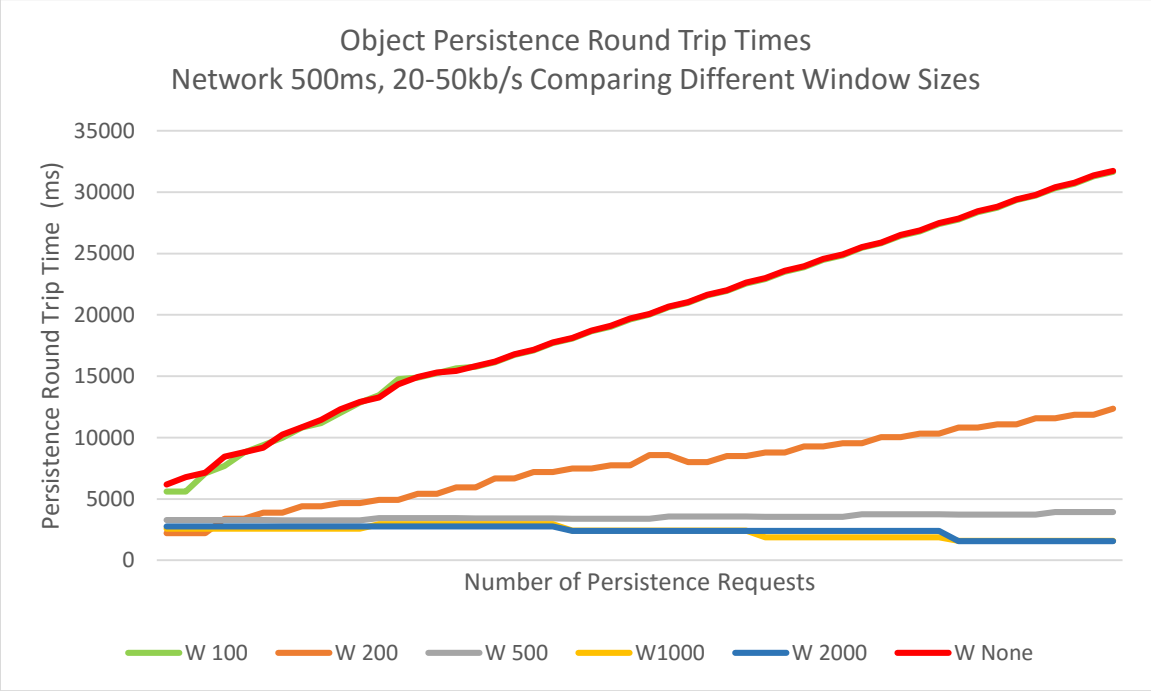

Figure 28: Round Trip Time & Frequency 10req/s in Network 500ms, 20-50kb/s & Persistence Window 0-2000ms

By adding a Persistence Window, it reduces the congestion traffic of sending multiple changes of the object to the persistence storage service at once hence clearly reducing the Round Trip Times near to low frequency workloads. However, it is observed that adding a Persistence Window size close to the Request Frequency & much higher Persistence Window size closer to the Round Trip Time, doesn't reduce the Request Round Trip Time.

In fact, according to the Figure 28, best Request Round Trip Time, for the least Persistence Window Size happens for the Persistence Window Size of 1000ms.

For further analysis of number of requests made to the Persistence Service compared to the number of requests included in the Persistence Window, following diagram is plotted with varying Window Sizes and Number of Persistence Requests sent to the Persistence Service.

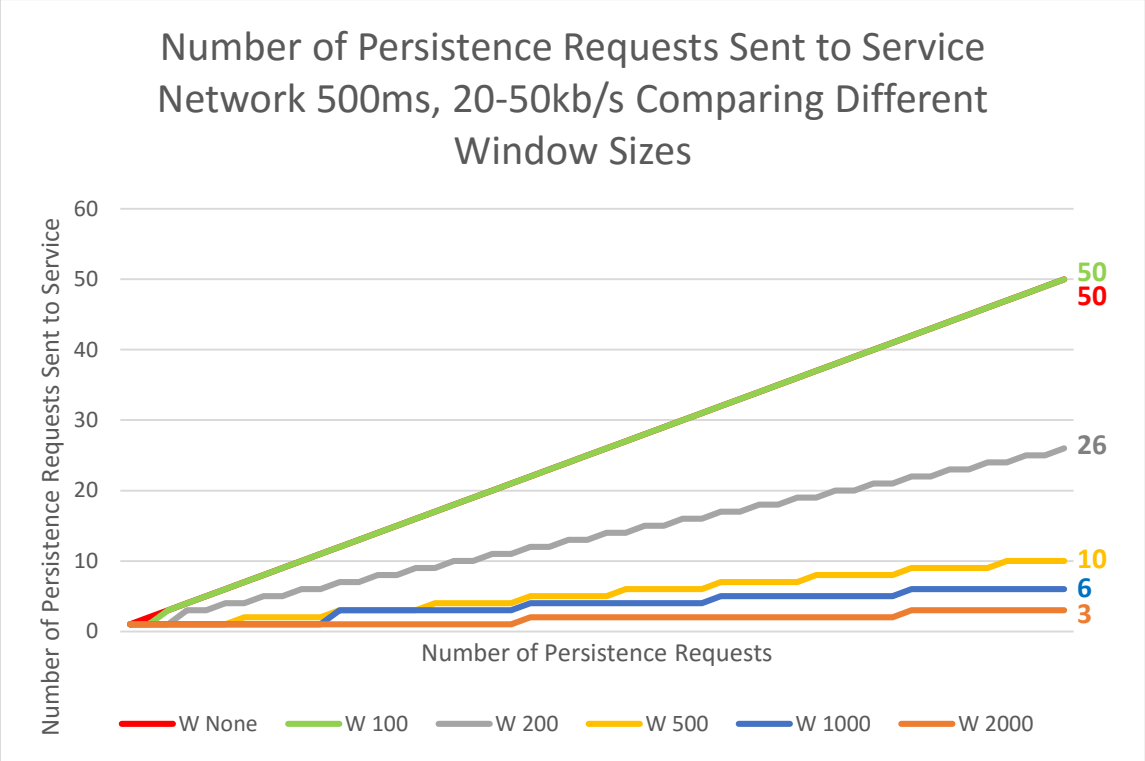

Figure 29: Number of Persistence Requests Sent to Service & Frequency 10req/s in Network 500ms, 20-50kb/s & Persistence Window 0-2000ms

According to Figure 29, it is evident that with the increase in Persistence Window Size many object change requests are not sent to the Persistence Service and in fact for Persistence window of 2000, it drops to 3 where 47 of Object Changes are made. If we consider a parameter called Total Persistence Time of an Object which is equant to Request Round Trip Time + Waiting Time in the Persistence Window, increasing Persistence

Window, increases the Total Persistence Time of an Object as shown in the following diagram.

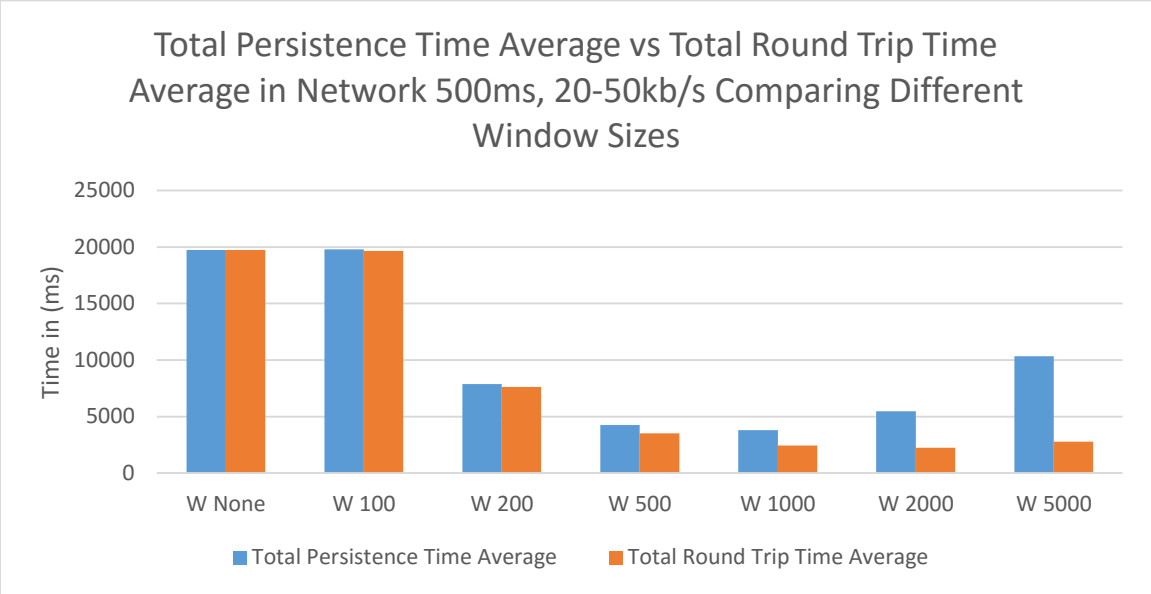

Figure 30: Total Persistence Time Average vs Total Round Trip Time

According to the above diagram optimal Total Object Persistence Time Average and Total Round Trip Time Average is achieved at the Persistence Window Size of 1000. Therefore, it requires to select suitable size for the Persistence Window to frequently Persist the Objects to the Persistence Service but not too quick to Congest the Network or Service itself which will greatly slow down individual round trip times and Total Persistence Time Average.

4.3 Persisting objects with dynamic persistence window

Based on the fixed size persistence window experiment carried, similar experiment conducted with dynamic persistence window calculation with Network setup of 300ms, 50-250kb/s and persistence frequency of 10 requests per second which had the highest deviation. For the experiment maximum persistence window is set to 1000ms since it was found as the optimum value.

After carrying out the experiment with dynamic persistence window calculation, it was observed that, if the request frequency is high compared to the Persistence Round Trip Time, too many requests are being sent initially (Before Persistence Window Size is calculated after receiving multiple responses) which will influence an increase of Persistence Round Trip Times of all the requests sent overall not getting the optimal results we found in fixed persistence window.

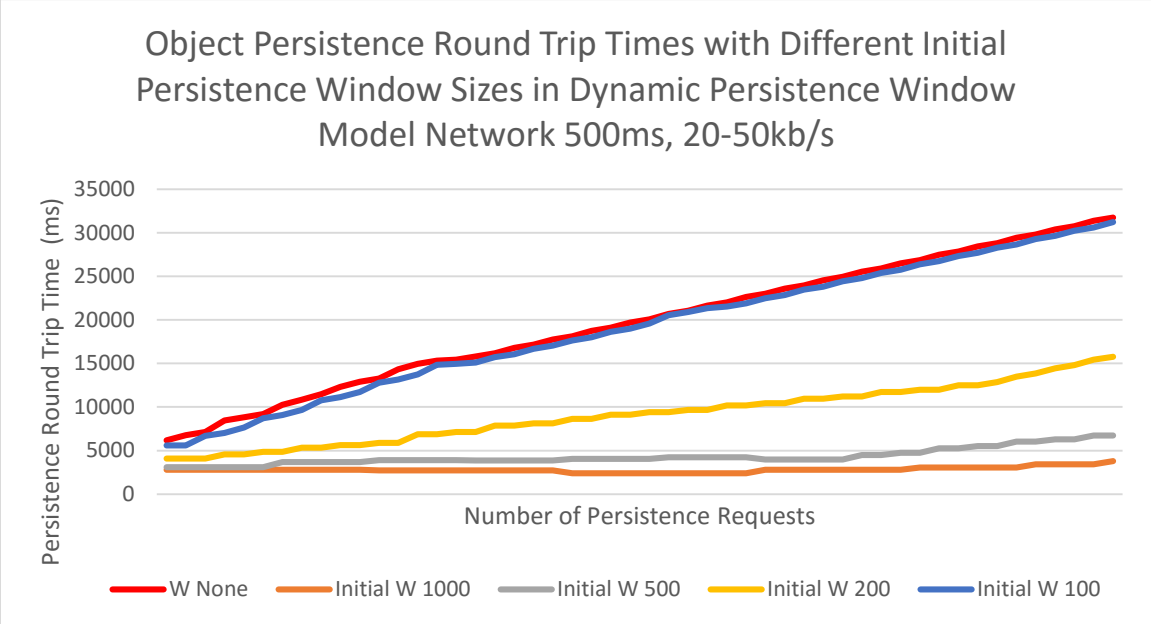

Figure 31: Object Persistence Round Trip Times with Different Initial Persistence Window Sizes in Dynamic Persistence Window

When comparing Fix Persistence with Dynamic Persistence Window Round trip Times, both provides better Round Trip Times in compared to Without Having Any Persistence Window. Also the Round Trip Time values are much closer for the respective initial Persistence Timing. Overall in this experiment, Persistence Window has shown better Round Trip Times for respective fixed and initial Persistence Windows. In this experiment the persistence workload is simulated with a fixed high frequency and according to the Figure 32 shown below, Dynamic Persistence Window Model, starts to reduce the Total Persistence Time of an Object when Persistence Window Size increases. This is because,

when a large Fix Persistence Window Size is Applied, all the Objects need to wait Maximum of Window Size before actual Persistence Round Trip happens.

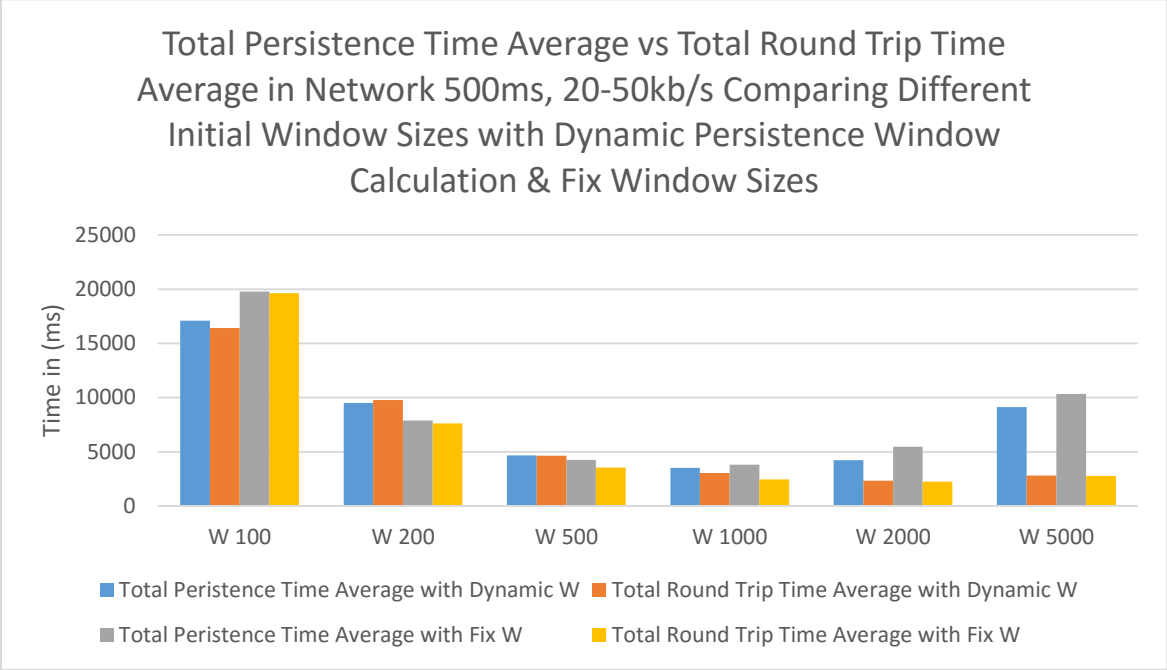

Figure 32: Total Persistence Time Average vs Total Round Trip Time Average in Fix Window & Dynamic Window Persistence Models

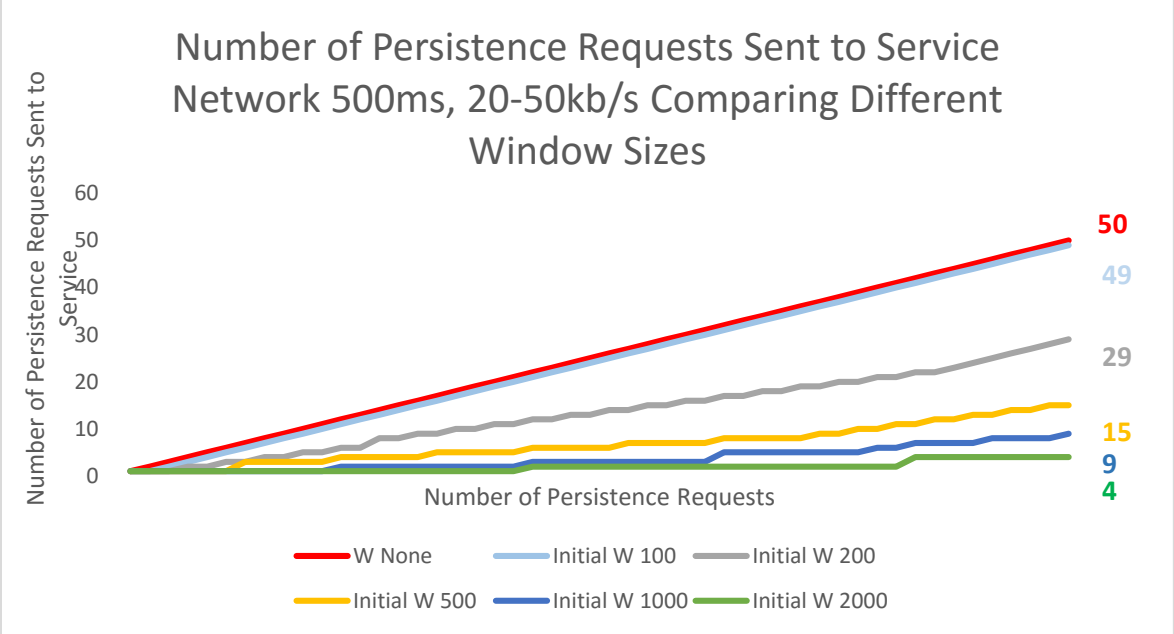

Figure 33: Number of Persistence Requests Sent to Service & Frequency 10req/s in Network 500ms, 20-50kb/s & Initial Persistence Window 0-2000ms

When comparing Figure 29 and Figure 33, where in Dynamic Persistence Window Model, more requests has been made compared to the Fix Window. Overall for Dynamically varying Object Persistence Frequencies like user interactions with a web application that needs to persist JavaScript Objects, Dynamic Persistence Model better suits, since it can adapt and change the Persistence Window Size not keeping Object stalled for too long if the User reduces his interacting speed with the application.

4.4 Future Work

Initial persistence window calculation needs to be carried out using an analytical approach, using the experimental data specially for applications such as Games, where high frequent burst object modifications happens.

References

[1] A. Lyengar and J. Challenger, "Improving Web Server Performance by Caching Dynamic Data," in *USENIX Symposium on Internet Technologies and Systems*, Monterey, California, USA, 1997.

[2] C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston and A. Tomasic, "Scalable Query Result Caching for Web Applications," in *VLDB `08*, Auckland, New Zealand, 2008.

[3] N. P. Huy and D. vanThanh, "Evaluation of mobile app paradigms," in *Mobile Computing & Multimedia*, Bali, Indonesia, 2012.

[4] C. Anderson and M. Wolff, "The Web is Dead – Long Live the Internet," *Wired Magazine,* pp. 118-127 & 164-166, 2010.

[5] W. West and S. Pulimood, "Analysis of Privacy and Security in HTML5 Web," *Journal of Computing Sciences in Colleges,* vol. 27, pp. 80-87, 2012.

[6] S. Mavrody, Sergey's HTML5 & CSS3 Quick Reference. 2nd Edition, Belisso Corp., 2012.

[7] M. Anttonen, A. Salminen, T. Mikkonen and A. Taivalsaari, "Transforming the Web into a Real Application Platform: new technologies, emerging trends and missing pieces," *ACM Symposium on Applied Computing,* pp. 800-807, 2011.

[8] M. Laine, "Client-Side Storage in Web Applications," 2012.

[9] R. Chandra, P. Gupta and N. Zeldovich, "Separating Web Applications from User Data Storage with BSTORE," in *USENIX conference on Web application development*, 2010.

[10] A. M. D. K. S. M. E. Benson, "Sync Kit: A Persistent Client-Side Database Caching Toolkit for Data Intensive Websites," in *International World Wide Web Conference*, Raleigh, North Carolina, USA, 2010.

[11] Google, "Google Gears," 31 May 2007. [Online]. Available: https://code.google.com/p/gears/. [Accessed 5 March 2014].

[12] Z. Mccormick and D. Schmidt, "Data Synchronization Patterns in Mobile Application Design," in *Pattern Languages of Programs (PLoP)*, Tucson, Arizona, USA, 2012.

[13] H. Shen, M. Kumar, S. Das and Z. Wang, "Energy-Efficient Data Caching and Prefetching for Mobile Devices Based on Utility," *Mobile Networks and Applications*, pp. 475-486, 2005.

[14] T. Schütt, F.Schintke and A. Reinefeld, "Efficient Synchronization of Replicated Data in Distributed Systems," *Springer-Verlag*, pp. 274-283, 2003.

[15] S. Agarwal, D. Starobinski and A. Trachtenberg, "On the scalability of data synchronization protocols for PDAs and mobile devices," *Network, IEEE*, vol. 16, no. 4, pp. 22-28, 2002.

[16] B. Cannon and E. Wohlstadter, "Automated Object Persistence for JavaScript," in *International World Wide Web Conference*, Raleigh, North Carolina, USA, 2010.

[17] A. Hosking and J. Chen, "Mostly-copying reachability-based orthogonal persistence," *Object-oriented programming, systems, languages, and applications*, pp. 382-398, 1999.

[18] J. A. Hoxmeier and C. DiCesare, "System Response Time and User Satisfaction: An Experimental Study of Browser-based Applications," in *Americas Conference on Information Systems*, California, 2000.

[19] K. Driscoll, B. Hall, H. Sivencrona and P. Zumsteg, "Byzantine Fault Tolerance, from Theory to Reality," *Springer-Verlag,* pp. 235-248, 2003.

[20] M. Malekpour, "A Byzantine-Fault Tolerant Self-Stabilizing Protocol for Distributed Clock Synchronization Systems," *Springer-Verlag,* pp. 411-427, 2006.

[21] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter and J. J. Wylie, "Lazy Verification in Fault-Tolerant Distributed Storage Systems," *Lazy Verification in Fault-Tolerant Distributed Storage Systems,* pp. 179-190, 2005.

[22] D. Bermbach and J. Kuhlenkamp, "2.3 Consistency in Distributed Storage Systems an Overview of Models, Metrics and Measurement Approaches," *Springer,* vol. 7853, pp. 175-189, 2013.