The Report Committee for James Peter Giannoules
Certifies that this is the approved version of the following report:

ProxStor: Flexible Scalable Proximity Data Storage & Analysis

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Adnan Aziz

David Chimitt

ProxStor: Flexible Scalable Proximity Data Storage & Analysis

by

James Peter Giannoules, B.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2014

Dedication

For

Breanna, Peter, Robert, and Lucas

Acknowledgements

I would like to acknowledge the support of my professors at the University of Texas who have graciously given their time and guidance over these past two years. Professor Adnan Aziz has been particularly influential for me, including, but not limited to, supervising this software engineering project and accompanying report.

Abstract

ProxStor: Flexible Scalable Proximity Data Storage & Analysis

James Peter Giannoules, M.S.E.

The University of Texas at Austin, 2014

Supervisor: Adnan Aziz

ProxStor is a cloud-based human proximity storage and query informational system taking advantage of both the near ubiquity of mobile devices and the growing digital infrastructure in our everyday physical world, commonly referred to as *the Internet of Things* (IoT). The combination provides the opportunity for mobile devices to identify when entering and leaving the proximity of a space based upon this unique identifying infrastructure information. ProxStor provides a low-overhead interface for storing these proximity events while additionally offering search and query capabilities to enable a richer class of location aware applications.

ProxStor scales up to store and manage more than one billion objects, while enabling future horizontal scaling to expand to multiple systems working together supporting even more objects. A single seamless web interface is presented to clients system.. More than 18 popular graph database systems are supported behind ProxStor.

Performance benchmarks while running on Neo4j and OrientDB graph database systems are compared to determine feasibility of the design.

Table of Contents

List of Tables

List of Figures

Section 1: Introduction

1.1 Vision

Motivated by the proliferation of mobile devices and the expansion of digital fingerprints residing in fixed infrastructure [1] it is should now possible to maintain a repository of digital fingerprints and the locations associated with them. These fingerprints may include unique identifying information from a WiFi access point, a Bluetooth device, Bluetooth Low Energy beacons [2], Near-Field Communication devices, or any future technology. Because these fingerprints are typically fixed in place within a location it is possible to create a software service which can infer the location of a mobile device based upon the device observations of these fingerprints. The process of reporting is engineered as a low-overhead operation and thus enabled for even the class of low-power IoT devices of the future.

While certain high-power mobile devices, such as today's smart phones, are capable of determining their location, this envisioned system has two advantages. The primary is that these fingerprints may provide location information in environments where traditional systems such as Global Positioning System (GPS) cannot operate due to an obstructed view of the sky. The second advantage is that by sending these observations into a service, a whole wealth of data mining opportunities open up. For example, the system can determine where your closest acquaintances currently are locations, or provide a listing of the places and people who are within walking distance from you at this moment. History can be maintained and important data such as the typical days and time that certain individuals frequent a business can be provided. This is the power of using the data of not only the one single device, but all participating devices.

A secondary benefit is achieved from maintaining the association between a fingerprint and a location in one central location as opposed to delegating the responsibility to each device.

The database to maintain such information potentially needs to support tens of millions of locations with each location containing dozens of fingerprints. Additionally the database must support information on hundreds of millions of users and their devices. Each user's encounter with a location should be stored persistently, thus a single system and associated database instance need to maintain and manage billions of records.

1.2 ProxStor

ProxStor is an instantiation of the above envisioned system. It answer the question of how such as system should be built, explores how the system should be used, and reveals how the system will perform.

ProxStor is a service, not an end-user application. Applications and services make use of ProxStor to enrich their user experience.

ProxStor uses a client-server model where the clients are mobile devices detecting and reporting these fingerprints. The knowledge of the association between fingerprint and location is maintained in ProxStor's web service. The device only needs to send ProxStor the breadcrumb detected - whether ProxStor knows of the breadcrumb a priori is not the client's concern. This model enables very low-power devices to participate by simply adopting a model of "send and forget".

Inherent to ProxStor is the association between a device and its user. If a device detects a breadcrumb then ProxStor places the user of the device in the location associated with the breadcrumb.

The server portion of ProxStor is a web application managing both a database repository and servicing the client queries. ProxStor defines the database schema and provides well defined interfaces for adding, updating, removing, and retrieving information. ProxStor defines *how* to access the database through a well-defined interface, but it does not define the conditions for *when* to do this other than the requisite that something must exist before it can be accessed. The expectation is that any service building upon ProxStor addresses this dilemma in a manner appropriate for its users. One possibility is a form of crowdsourcing. Every time an unknown breadcrumb is found the user of a high-power interactive device is prompted to fill in the necessary information.

ProxStor leverages the power of an emerging class of NoSQL databases known as Graph Relational Databases [3] specifically to support billions of data objects while still rapidly satisfying user queries. The data models used for various objects in ProxStor are designed to enable this evaluation, but themselves are not as data rich as a real-world application would require. This should not impact the feasibility assessment, but should be noted before using ProxStor in any real-world deployment.

Lastly, being a demonstration of feasibility, ProxStor does not address security concerns that naturally arise when storing and processing people's movements. The need to address this topic is highlighted at the conclusion of the report in the Future Work section.

1.3 User Stories

To demonstrate how ProxStor can enable a richer class of location aware applications the following user stories are presented. These stories introduce Bob, a fictitious user of various imagined mobile applications. These stories assume that a running instance of ProxStor is available on the Internet and furthermore that the database behind ProxStor has been populated with necessary data to satisfy the particular story including records defining the user Bob, his associated device(s), and his relationship to other users of the system. The scenarios also assume populated data on locations, their relationship between each other, and the breadcrumbs contained within.

The functionality of ProxStor described in each story exists today.

1.3.1 Story 1 - Friend Locator

On a whim Bob decides to head into a busy downtown district to spend his evening with his friends. Bob isn't entirely certain where his friends are currently – they have the habit of roaming about. He opens his "friend locator" application to view which of his friends might be nearby. As Bob has hoped, the application displays a nearby close friend who is within walking distance. Using an on-screen walking map Bob heads off in the friend's direction while the application automatically messages his friend notifying that Bob is on his way.

This imagined "friend locator" application used ProxStor in three important ways:

1. Upon launching the application ProxStor was used to "check in" and therefore to determine Bob's current location. Bob's location is also now recorded within ProxStor for current and future queries.
2. The application sent a query asking *which of Bob's closest friends are here (or nearby)?* The query may have been limited to only a radius the application interpreted as 'walking distance'. A second query could have been used to extend the radius beyond walking distance. The application then applied its logic for best fit, taking into account the degree of friendship and the distance Bob specified he was willing to travel.

3. Once the target friend was identified, ProxStor provided the “friend locator” application with his/her current location, which combined with Bob’s current location, provided the starting and ending points used to generate the on-screen walking directions map.

1.3.2 Story 2 - Restaurant Food Review

A restaurant Bob only occasionally frequents has recently changed its menu, to his dismay. He opens his favorite “food review” application to check what his friends who have recently dined here ordered, and what they thought of the food. The nice thing about this application is that even if his friends did not leave behind a written review Bob has the option of contacting them directly to ask his questions. Bob is pleased to see several of his friends have been here just in the past month. Since Bob knows which of these friends have compatible palates he is better able to make his dining selection.

This imagined food review application used ProxStor in two important ways:

1. Upon entering the restaurant ProxStor was used to “check in” and therefore to determine and store Bob’s location. This provided the “food review” application with the necessary physical context. Note that this “check in” also allows other users of ProxStor, even independent from the users of the food review application, to know the whereabouts of Bob.
2. ProxStor answered the proximity aware query *which of Bob’s friends have been here recently?* This information was used in turn by the imagined application to reference its food review specific database to extract relevant reviews. To extend the power of such a query the application could have further requested ProxStor to return friends of

Bob's friends who have also dined here recently. Such a potential enhancement operates under the assumption that even people one further degree separated from Bob will still have similar food tastes.

1.3.2 Story 3 - Event Planning

Bob is planning a week long business trip to Austin and he knows he will have some free time in the evenings. He opens his favorite "event planning" application and informs it that he will be heading to Austin. The application then shows him the top activities his friends in the Austin area typically are up to, broken down by day of the week. With a tap Bob extends the scope of the displayed information to include his friend's closest friends as well. Using the application's recommendations as a guide Bob schedules activities for several of his evenings in Austin.

This imagined event planning application used ProxStor in two important ways:

1. ProxStor answered the query *where were my friends on the evening of a specific day of the week?* ProxStor already knows who Bob's friends are, and further filters results to provide only the listing of locations for the requested time period. Note that in this example the query is repeated for each day of the week because the range of interest is limited to evenings. Each day breaks the continuity and disallows one large date range query.
2. The "event planning" application also used to answer whether the query result location(s) were within an acceptable radius of where Bob planned to be (e.g. downtown). This acceptable area was either defined by Bob or calculated by the application.

1.4 Contributions

The primary contributions contained in this report are:

- A complete coherent client application programming interface (API) for implementing the envisioned system is provided. Consideration is given to both low- and high-powered devices.
- A working example of the envisioned system, ProxStor, is documented. This includes documenting the system design features, both macro and micro in scale. In many cases the rationale for decisions is also provided.
- The feasibility of the envisioned system, based on experimentation with ProxStor, is provided.

It is hoped that the work contained herein can be a starting point for future work in area of spatiotemporal database data and processing.

1.5 Outline

This remainder of this report describes ProxStor's system design, its application programming interface. The rationale for key design points is provided.

Chapter 2 of the report contains the requirements and specifications necessary to successful implement the envisioned system. This includes functional and non-functional requirements as well as high-level design specifications.

Chapter 3 explores the internal design of the ProxStor implementation including all the technologies brought together to build a successful system. The chapter also describes the overall system design, and the user-exposed object types, and the ProxStor connector used to enable Java client application connectivity. The web interface is explored including how the system accesses the back-end database.

Finally, the chapter concludes with a discussion of the rationale for electing to use a graph relational database including the data model for some common usage scenarios.

Chapter 4 documents all of the possible HTTP response statuses and how the HTTP methods map into the RESTful web API. Two example exchanges over the web API are provided. This chapter highlights the consistent and simple ProxStor API, which is a key contribution of this report.

Chapter 5 includes the results of building and testing the prototype. This includes performance benchmarking across a variety of operations and system configurations. Software engineering metrics and lessons learned are also included.

Chapter 6 summarizes what was learned from this exercise as well as outlining planned future work.

Chapter 2: Requirements and Specifications

This chapter covers both what ProxStor should do and how it should do it. These are presented as both functional and non-functional requirements as well as design specifications. The implementation internals are described in Chapter 3.

2.1 Requirements

2.1.1 Functional Requirements

The following functional requirements must be met to enable the necessary basic operations of ProxStor:

- Data Representation: the system shall represent the following data types and provide the capability to create, retrieve, update, and delete each. Additionally each object shall be uniquely addressable:
    - Users
    - Locations
    - Location environmental items (“breadcrumbs”)
    - Devices
    - Sensors within devices
- Data Relationship: the system shall provide the capability to create and dissolve the following associations:
    - A user *uses* a particular device
    - A user *knows* another user
    - A devices *contains* sensor(s)
    - A location is *contained within* another location
    - A location is *within* a specified distance of another location

    - A location is *identified* by unique environmental elements which it *contains*
- Data Query: the system shall provide the capability to query based on the following relationships between objects:
    - Retrieve users known by specified user
    - Retrieve users who know specified user (the reverse direction of the above)
    - Retrieve devices owned by specified user
    - Retrieve locations within specified location
    - Retrieve locations which contain specified location (the inverse of above)
    - Retrieve locations within specified distance of another location
    - Retrieve all environmentals within specified location
- Administration: the system shall expose basic administrative operations:
    - Instantiate a new database
    - Connect to existing database
    - Report usage (administrative) statistics
- Persistence: All data objects remain persistently within the system until a delete operation is performed, regardless of the age or inactivity of the object. A full history of a user's movements is retained. No pruning of older data shall be performed.
- Current Location: the system shall support a concept of each user's current location, implying a temporal processing aspect to the system.

- Environmental-based Movements: the system shall support the concept of accepting client reported check-in and check-out operations based on a device sensing an environmental:
    - A check-in indicates that a device has just detected the environmental and thus the associated user's current location shall be updated accordingly.
    - A check-out indicates that a device has just ceased detection of the environmental and thus the associated user's current location shall become unknown.
- Manual Movements: the system shall support the user manually specifying his or her current location.
- Data Processing: The user's current location shall be used for a class of queries
    - Retrieve user's current location
    - Retrieve those users in the user's current location
    - Retrieve
- Data Set: the system shall handle arbitrarily large data sets with no pre-defined upper limit. Data sets of billions of entries shall be possible.
- Concurrency: the system shall gracefully and coherently handle multiple concurrent client accesses. If more than one client attempts access to the same database object the system must ensure ordering and atomicity of operations.
- Data Exchange Format: the system shall exchange information in a standard data format easily understood by clients.

- Data Exchange Interface: the system shall exchange data over a standard network protocol interface to ease implementation of support by client systems.

2.1.2 Non-Functional Requirements

The following are the non-functional requirements goals for ProxStor. Striving for these goals is done as a best effort.

- Scalability: The system should be designed to easily scale both vertically and horizontally to support increased user load.
- Extensibility: The system should permit extending capabilities with minimal impact to the existing code. These extensions should be possible in a manner such that the external interface remains coherent.
- Interoperability: The system should support a wide range of both client devices and of data center deployments. This applies to both the machine running ProxStor as well as the persistent storage solution. The ideal system hardware platform is x86.
- Availability: Because of the nature of check-in and check-out events the system should be available continuously with no downtime.
- Capacity: The system should permit billions of records accessed by millions of users concurrently.
- Speed: The client devices accessing ProxStor will vary widely in their computing power, battery life, and network connectivity. The system should accept, process, and respond to requests as quickly as possible. Note that efficiency is not a requirement.

- Privacy: This is non-requirement. Privacy is not a concern of the ProxStor project. It is acknowledged that a real world deployment of a ProxStor-like system would require stringent privacy measures.

2.2 SPECIFICATIONS

The above requirements combined with the current state of mainstream computing lead to the following specifications.

- The system should be accessible over the Internet to provide ubiquitous client accessibility.
- The web presence component of the system should be flexible enough in nature to be deployed in one of the many web container architectures in use today.
- The web presence component should exchange data with clients using JavaScript Object Notation (JSON).
- The web presence components should reduce potential downtime by providing redundancy.
- The web presence component should be based on HTTP.
- The persistent data store should be inaccessible to the clients.
- The web presence should support a multitude of persistent data store solutions.
- A client should only need to comprehend basic HTTP communications and basic object [de]serialization.

The following is a high-level diagram of the specified system.

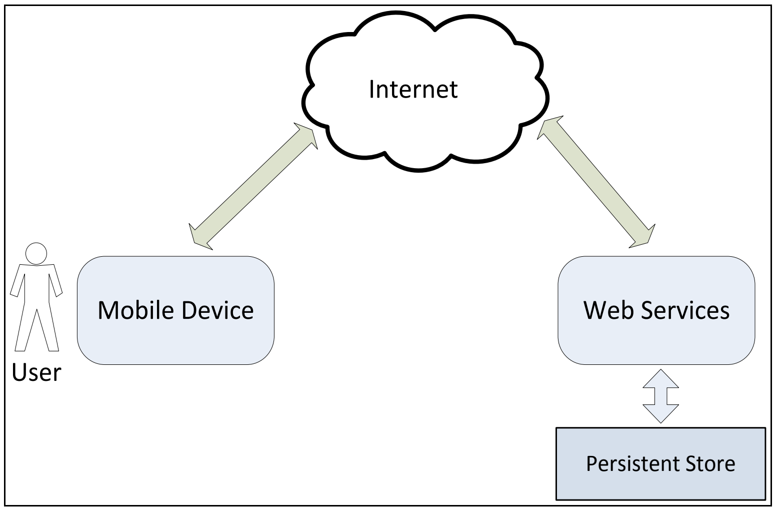


Figure 1: Specified System

Chapter 3: System Design

This chapter describes the design and implementation of ProxStor.

3.1 Technology Stack

The implementation of ProxStor leveraged several technologies and built upon existing freely available software. Principle among these software offerings are:

- Jersey – provides a framework for development of RESTful web services providing support for the JAX-RS API. See [4] and [5].
- Tinkerpop Blueprints – the TinkerPop [6] project provides a common interface for developing applications on top of graph databases known as Blueprints. Currently the Blueprints project has enabled back-end support for more than 18 popular graph database products. The project website describes it as "analogous to the JDBC, but for graph databases". See [7].
- Gson – Java library used to convert objects into their JSON representation and to convert strings back into Java objects.
- Postman REST Client – provides an easy to use graphical interface to test and refine a REST interface, including the ability to save and restore saved commands and adapt to different networking environments. See [8].
- Winstone Servlet Container – provides a fast minimalist servlet container for running ProxStor. See [9].

All ProxStor code was written in the Java programming language. The NetBeans IDE was used as the development environment. Maven [10] was tasked with managing project dependencies. Git was used for version control. GitHub hosted the code repository and provided a convenient wiki used to continually document ProxStor as it was developed.

3.2 PROXSTOR DESIGN

The ProxStor project consists of two primary components:

1. The ProxStor Connector and API for client application consumption
2. The ProxStor Cloud Service serving as the web interface as well as providing back-end database access

Figure 2 shows the components of ProxStor and their relative positions within the stack. The ProxStor components are shaded grey.

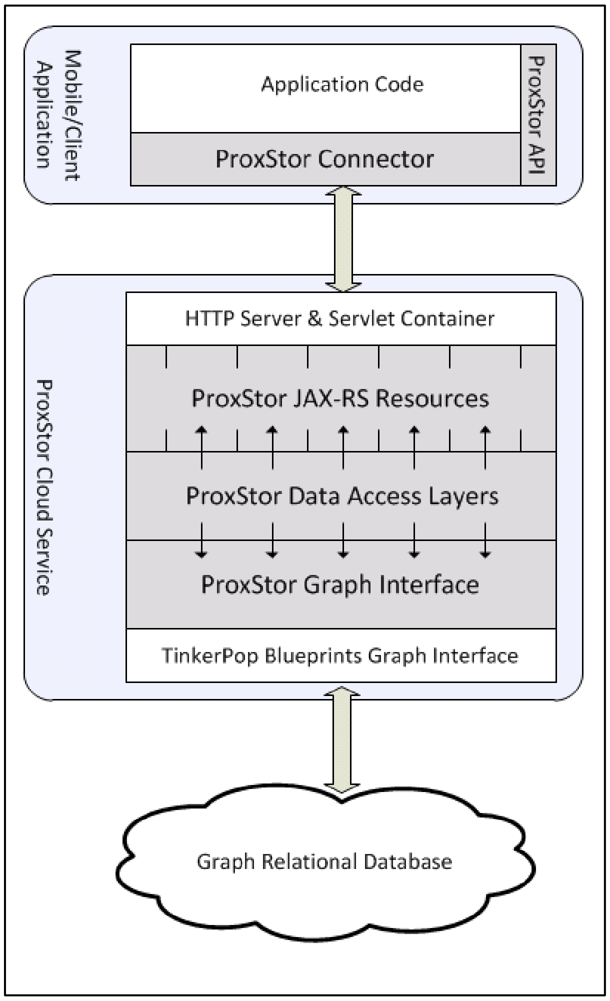

Mobile/Client Application
Application Code
ProxStor Connector
ProxStor API
HTTP Server & Servlet Container
ProxStor JAX-RS Resources
ProxStor Data Access Layers
ProxStor Graph Interface
TinkerPop Blueprints Graph Interface
ProxStor Cloud Service
Graph Relational Database

Figure 2: ProxStor Components

Additionally a ProxStor Testing project was developed to simulate client access, including data loading and querying. The testing project is not covered in detail in this report.

3.3 ProxStor Cloud Component

ProxStor's cloud component is implemented as a Java servlet. This component provides both the RESTful HTTP interface as well it coordinates access to the back-end database and enforces the desired data model.

3.3.1 ProxStor JAX-RS Resources

ProxStor provides the web interface as a series of Java API for RESTful Web Services (JAX-RS) classes. A package exists for each resource exposed and within the package several resource handling classes may be present. The multiple classes are used to separate base resource locator from sub-resource locators. All JAX-RS resource locators are implemented on the Jersey framework. Each handler accepts a request from the network, makes a call into the appropriate data access layer, and once complete return a meaningful HTTP status, potentially with data. All data coming into and out of the resource locators must be de-serialized and serialized using JSON. All objects defined in the API are annotated with `@XmlRootElement` to facilitate Jersey's conversion to/from JSON.

3.3.2 ProxStor Data Access Layers

Each resource type has a unique Data Access Object (DAO) which provides the interface for the resources to access the Graph database. Each DAO exposes functionality which matches the capabilities exposed by their partner JAX-RS resource class. All DAOs interact with an instance of ProxStor Graph.

The DAOs serve to limit the accessibility of a given resource by exposing only the minimal necessary functionality as well as to maintain all related code in a single class. All DAOs throw exceptions in the event a resource's request cannot be processed. The exceptions in ProxStor are expressive enough for the caller to know which aspect of the request was problematic.

3.3.3 ProxStor Graph Interface

ProxStor has a `Graph` object for all the DAO objects to call into for access in to the database. The ProxStor `Graph` object converts all these calls into TinkerPop Blueprints calls. The Blueprints project allows common access to any Blueprints-enabled Graph database. Thus the `Graph` object allows ProxStor to be built upon many popular NoSQL graph database such as Neo4j [11], OrientDB [12], MongoDB [13], ArangoDB [14], or any of a multitude of other choices. Beyond the flexibility offered to exchange back-end databases, the `Graph` object is an ideal point in the design to tightly control access. It currently provides just enough functionally to enable each of the DAOs. Although ProxStor `Graph` is designed independent of the actual database in use, it does need to know whether transactional operations are support. This is determined by checking whether `Graph` is an instance of `KeyIndexableGraph`. If it is then the `Graph` object takes care of issuing the commit after each transaction.

3.4 ProxStor Client Components

The client component is divided into two pieces. The first is the class `ProxStorConnector` which provides a Java application native communication to ProxStor. The second is known simply as the ProxStor API and it includes the class definitions for all the JSON objects passed between client and server.

3.4.1 ProxStor Connector

The ProxStor Connector package provides a Java client with:

- Simplified API for communicating with ProxStor
- Class definitions for objects which are exchanged between client and server.
- JUnit testing suite to validate the ProxStor connector when powered by an actual running ProxStor instance.

3.4.1.1 Simple Connector API

The `ProxStorConnector` class enables a client to:

- Connect to a running ProxStor instance
- Manage the lifecycle of all exposed object types
- Perform device or user check-ins and check-out

All the client invocations of these methods involve handling of the ProxStor object types. The client does not need to be fluent in network communication design. Additionally, the connector returns simple Boolean status for operations. The client need not perform exception handling.

Please refer to the `ProxStorConnector.java` source file and the respective JavaDoc for more details.

3.4.1.2 Class Definition API

The ProxStor Connector API package defines the following object types:

- **`User`** – Uniquely identifying a user of the ProxStor system.
- **`Device`** – Uniquely identifying a user's device. ProxStor enforces the relationship that a device must be used by one and only one user.

- **`Location`** and **`LocationType`** – A location in the physical world as well as the type of location. Locations can be related to each other in either a nesting fashion or by defining the distance in between.
- **`Environmental`** and **`EnvironmentalType`** – A sense-able environmental element within a location. When a device detects a environmental the system can infer location.
- **`Locality`** – Created whenever a user is within location. A Locality persistently records the user, device, environmental, location, and time information on each check in.

The design point is for each of the above classes to be a simple object easily converted to/from JSON. When receiving objects as JSON ProxStor cannot assume that any specific fields have been populated, thus validation must be performed on such data.

Please refer to the `proxstor.api` source package and the respective JavaDoc for more details. The internals of the classes, for example how exactly a User is defined, is not critical to evaluate the feasibility of ProxStor. The above classes have purposefully been designed to provide a basic level of expressiveness. Real deployments would want to capture much more information.

3.4.1.3 JUnit Test Suite

The `ConnectionTesterSuite` JUnit Suite class runs a series of JUnit tests to validate, both positively and negatively, the following connector interface categories: device, environmental, locality, location, location nearby, location within, user, and user knows. The JUnit tests must be configured to connect to a running ProxStor instance. It is recommended to run this suite anytime a modification is made to the `ProxStorConnector`.

3.5 Graph Relational Database

From the beginning of ProxStor development it was understood that all the data being collected and managed needed to be stored in a Database Management System (DBMS). Traditionally Relational DBMS (RDBMS) do not scale well to the types of dataset sizes envisioned in this system [3]. This limitation is a primary motivation for the NoSQL movement of databases. NoSQL databases are available in many different types of data structure models such as column, key/value, document, and graph. After evaluation of the available types it was decided to build ProxStor on top of graph databases. As the name implies the graph model is built around data with relationships which can be expressed in graphs, which is a natural fit for the data used in ProxStor. The number of relationships, types, attributes, and direction of edges do not need be defined in a rigid schema. Graph databases are very flexible and forgiving and promise to perform well even at the scale of billions of nodes and relationships [3]

3.5.1 Graph Models

To demonstrate the flexibility of the graph model a few relationships from the ProxStor data model will be examined. While the models are faithful to the internal implementation of ProxStor, certain details have been omitted for ease of discussion. Note that the figures below are not a formal modeling language such as Unified Modeling Language (UML).

3.5.1.1 User Knows User

See the following figure for the model of how the *knows* relationship is established between users. The full definition of the `User` class is not essential to this discussion, so the user is left mostly undefined here.

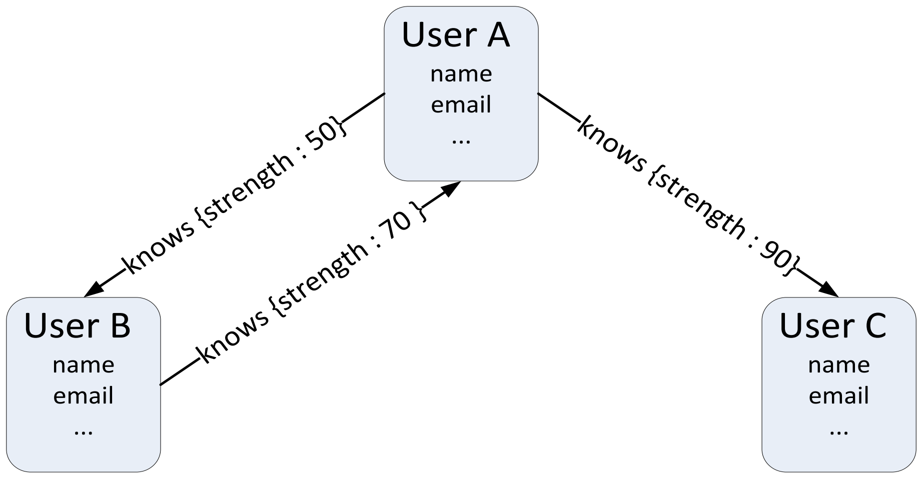


Figure 3: User 'Knows' Relationship

Note that the vertices in the graph (users) are connected to each other with directed edges (relationships). Here the relationship is *knows* and contains the attribute *strength*. The attribute is a critical component of this relationship because it allows ProxStor to perform intelligent queries from the database extracting only the necessary data. For example, if a query is being processed searching for all the users that user A *knows* with *strength* greater than or equal to 70, ProxStor will perform the following actions:

- Retrieve the graph vertex representing user A
- Query the graph relational database for all outgoing edges labeled *knows* with attribute *strength* greater than or equal to 50
- Follow each matching edge to its vertex and add to the matching set

Note that all edges in the database are directed and thus a natural asymmetric relationship forms for *knows*. User B claims to know user A with a higher *strength* than is reciprocated by user A. User C has not set a *knows* relationship back to user A and thus any query processed by ProxStor for users that user C *knows* will not include user A.

3.5.1.2 User Uses Device

Here we see how a user is related to a device which they use.

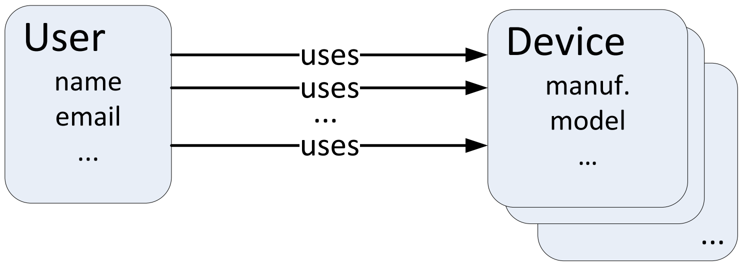


Figure 4: User ‘Uses’ Relationship

An edge from the user vertex labeled *uses* is directed to the device vertex. The edge has no attributes in this case. Although the edge is directed (it must be by definition) the *uses* edge can be followed back from the device to the user. This is critical in ProxStor as many operations are performed with only the device context. For example, when a device senses an environmental and performs a check-in operation one of the actions ProxStor must take the reporting device identifier, retrieve the device vertex, follow the *uses* edge backwards to the user, and create the appropriate locality.

As with the *knows* relationship above between users, there is no limit on the number of *uses* edges that can exists from user to device.

3.5.1.3 Location Contains Environmental

The key piece of data linking a device’s sensing of an environmental fingerprint to a location is the environmental vertex, which is connected via the *contains* relationship.

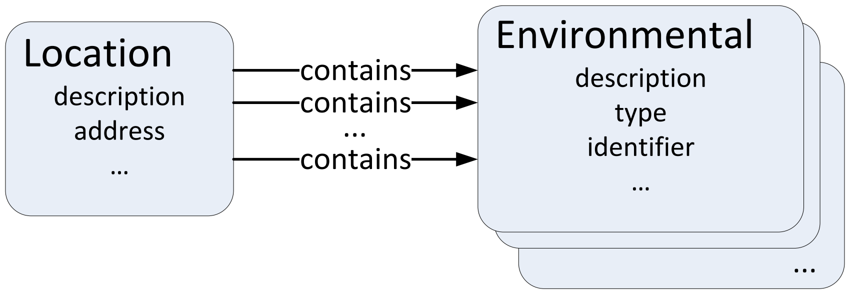


Figure 5: Location ‘Contains’ Relationship

An edge labeled *contains* is directed from the location to the environmetal. The utility of this is very similar to the uses relationship above. When a device detects an environmental fingerprint ProxStor must determine which, if any, location the device is within. The first step is to find a matching environmental for the type and identifier reported by the device. Once that is complete the *contains* edge is followed backwards to the containing location, and combined with the process described in the section above ProxStor now associates a user as being in a location.

3.5.1.4 Location Nearby/Within Location

In order for ProxStor to know which locations are within one another as well as the distance between locations the following data model is maintained.

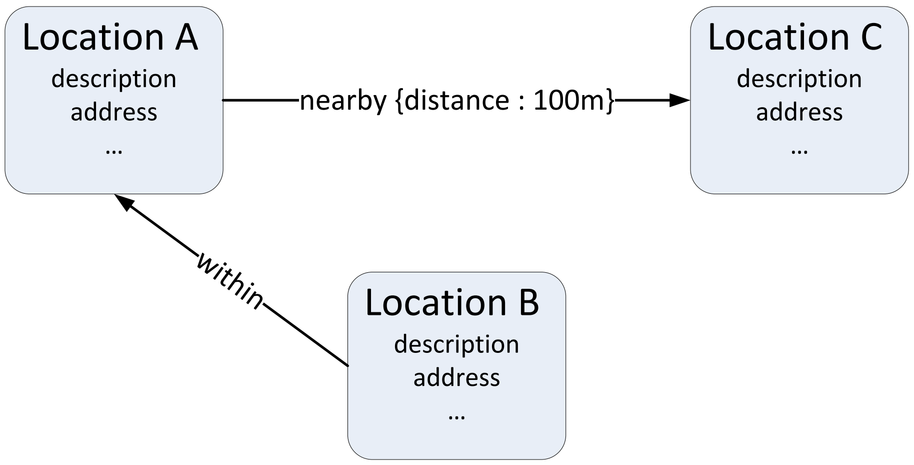


Figure 6: Location ‘Within’ and ‘Nearby’ Relationship

The *nearby* relationship contains the *distance* attribute, which is used in distance-bound queries to reduce the number of edges traversed to only this inside the acceptable range. Although the edge is directed all ProxStor use of the edge is traversed without consideration of the direction.

ProxStor will also use the latitude and longitude information for each location (not shown in figure) to calculate a bounding box approximating the range involved in any distance-based query. This allows filtering of results to only those locations or localities whose coordinates fall within the rectangular region.

The *within* relationship has no attribute and the direction is important. In the figure above location B is within location A, not the other way around.

3.5.1.5 User Currently/Previously at Locality

Localities are generated whenever a user and a location come together within proximity of each other. A user can have only a single active locality at a time (otherwise he/she would be in two places at one). A user is related to his/her current locality through the *currently_at* relationship.

Once the user is no longer present in a location the locality becomes related to the user through a *previously_at* edge. There is no limit on the number of *previously_at* relationships a user may have to localities. See the following data model.

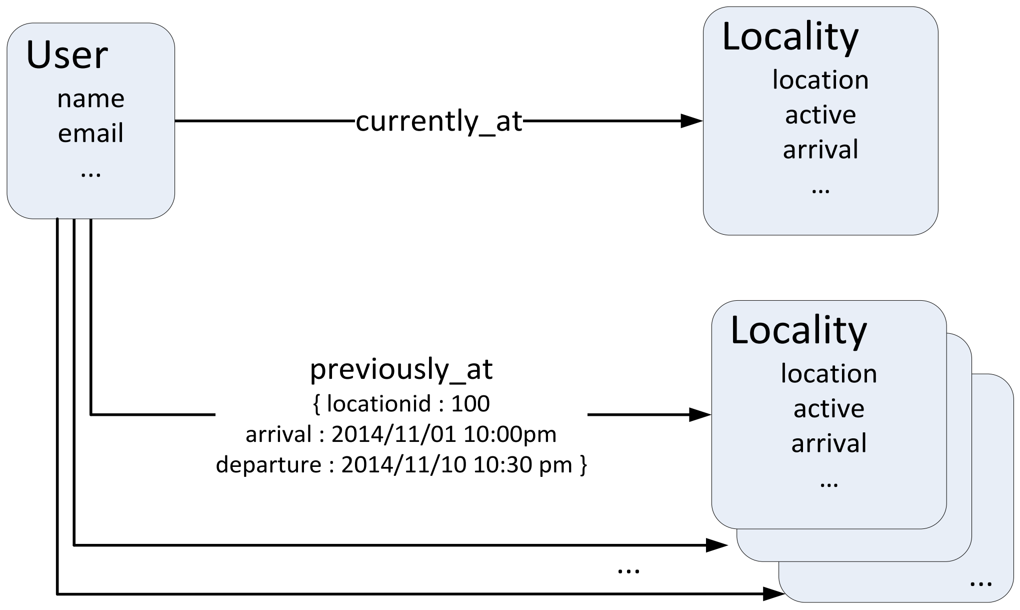


Figure 7: User 'Currently At' and 'Previously At' Relationships

The *currently_at* relationship has no attributes and simply tells ProxStor by its existence that a user is indeed checked-in somewhere.

The *previously_at* relationship to older locality instances carries three important attributes. The first is *locationid*, the location identifier of the location associated with the locality. This enables rapid filtering of a user's check-in history for a certain location. The other two attributes are the arrival and departure times associated with that locality, known as *arrival* and *departure* respectively. When ProxStor is processing a query for locality instances within a certain timeframe the *previously_at* edges from the user are filtered by a check for date range overlap.

Chapter 4: REST API

ProxStor's services are exposed to the world as a web service using meaningful URIs and returning consistent HTTP responses. Applications consuming the Java ProxStor Connector (see Section 3.4) do not need to comprehend the below details as the connector provides a more simplified interface. For those directly accessing the web interface, for example implementing their own connector, this section provides a guide to the interface.

ProxStor's web interface is modeled as a RESTful interface. REST stands for Representational State Transfer and is a dominant design model for web services [15]. In brief, a REST service is:

- Stateless
- Uses standard HTTP methods
- URIs are used in a directory-like structure to access resources
- Uses JSON to transfer objects

Through this RESTful interface ProxStor exposes CRUD (Create, Retrieve, Update, and Delete) [16] operations for all static object types. Note that this matches well with the function requirements for data creation, update, persistence, and deletion.

The reader who is unfamiliar with REST is encouraged to read [15] or [4].

4.1 HTTP METHODS

ProxStor uses the following standard HTTP methods for all interactions:

- GET
- POST
- PUT
- DELETE

4.1.1 GET

For all retrieval operations the GET method is used. Examples include retrieval of a user's current location or to perform a static object search.

4.1.2 POST

For all create operations the POST method is used. Examples include addition of a new user into the system or a new location check-in.

4.1.3 PUT

For all update operations the PUT method is used. Examples include updates to the profile for a location or to modify the knows (friendship) relationship between two users.

4.1.4 DELETE

For all delete operations the DELETE method used. Examples include removing the within relationship between two locations or removing a user from the database.

In addition to the traditional deletion of an object from the persistent data store, DELETE also is used in ProxStor to:

- Check-out from a location
- Shutdown the running database instance

The following sections clarify the usage of all exposed URIs.

4.2 HTTP Responses

ProxStor strives to implement consistent HTTP responses [17] in all situations simplifying client (and client library) development. The below sections document the various HTTP responses and under what circumstances they are returned.

4.2.1 OK (200)

HTTP status 200 (OK) is returned when the requested operation completed successfully without error. If the request was for the retrieval of information the body of the response will contain JSON representation of such data.

4.2.2 Created (201)

HTTP status 201 (Created) is returned when a request to create new content inside ProxStor completes successfully. In the header of the 201 response the *Location* field will indicate the URI of the new content. As fitting the specific request the HTTP response body may also include a JSON representation of the newly created content.

4.2.3 No Content (204)

HTTP status 204 (No Content) is returned when a request to perform an operation is successfully completed and the context of the request does not involve the transfer of data. In ProxStor status 204 is returned for successful updating and deleting of objects as the status code is all the information a client requires.

4.2.4 Forbidden (403)

HTTP status 403 (Forbidden) is currently only returned in situation, when the administrator attempts to create a new graph database instance while one already exists.

4.2.5 Not Found (404)

HTTP status 404 (Not Found) is returned in both the case of a malformed URI and an invalid request. In the bad URI case the status 404 is returned by the servlet container, not ProxStor per se.

ProxStor return status 404 if the client request references an unknown or nonexistent object. For example, if the request was to retrieve a user object with a non-existent userId.

4.2.6 Server Error (500)

HTTP status 500 (Server Error) typically indicates an unhandled exception within ProxStor which reached up to the servlet container; however ProxStor does intentionally return 500 in two cases.

The first is if an URISyntaxException is thrown while preparing the *Location* header field in a 201 (Created) response.

The second is if an attempted instantiation of a new back-end Graph instance fails.

4.2.7 Service Unavailable (503)

HTTP status 503 (Service Unavailable) is returned when an attempt is made to retrieve the database instance information, but no running database instance exists.

4.3 URIs

A uniform resource identifier (URI) is a string used to uniquely identify the name of a resource and uniform resource locators (URL) used in HTTP requests (i.e. a web address) are in fact URIs. All communications with ProxStor is initiated with an HTTP request to *some* URI. Thus it is necessary to review the URI layout to actually comprehend the ProxStor web interface.

All URIs in ProxStor are relative to the base URI, which is actually a factor of the servlet container in use. For example, when running ProxStor within Winstone the default base URI becomes:

```
http://{host}:{port}/api
```

Figure 8: Base URI

In the URI the {host} and {port} segments must be replaced by the system hostname and listening TCP port respectively. The exact base URI will depend on the specifics of your ProxStor deployment. The following sections list all exposed URIs and describes their use. The key concept to grasp is that all the URIs documented herein are actually relative (appended) to the system base URI.

When designing the web interface care was taken to ensure all ProxStor URIs were meaningful and expressive. Combining the HTTP request type and the URI is sufficient to describe the operation being attempted.

4.4 WEB API

Based on the requirements and specifications from Chapter 2, as well as the system design from Chapter 3, the RESTful web API was derived. The following section provides some example uses of the ProxStor Web API. The complete API documentation is in Appendix A.

4.4.1 Example – Device Check-in

In this example a user enters a new location carrying a device connected to the Internet. This device has been previously associated with the user, so ProxStor makes the assumption that the user is wherever the device happens to be.

This device has onboard sensors which detect the presence of a Bluetooth Low Energy (BLE) Universally Unique Identifier (UUID). This is how the mobile device notifies ProxStor of this observation and how the device ultimately learns its location.

First the device creates a new `proxstor.api.Environmental` object, sets the type field to `proxstor.api.EnvironmentalType.BLE_UUID`, and the identifier to the sensed UUID value. Note that additional fields exist within this class, but at this time only above information are known to the device.

Next the device submits this *partial* `Environmental` object to ProxStor by issuing an HTTP POST request containing the JavaScript Object Notation (JSON) representation of the `Environmental` to the following URI:

```
/checkin/device/{devid}/environmental
```

Figure 9: Device Check-in URI

Where {devid} is replaced by the actual device identifier assigned to the mobile device by ProxStor.

ProxStor will internally reference the BLE UUID to find the associated `Environmental`, follow the `Environmental` back to the containing `Location`, and create a new `Locality` associated the `Device` owner with this `Location`. The response from ProxStor will contain this new `Locality` in JSON format which the client will de-serialize. This new object now provides the client with the following new information:

- Identifier for this `Locality`
- Identifier for the `Location`
- Identifier for the `Environmental`
- Date & Time of check-in

The identifier for the current Location is an object identifier reference specific to the back-end database instance. The value is not meaningful to the device or to the user. To turn this into a `Location` object the client must issue another request to ProxStor.

4.4.2 Example – Location Retrieval

To retrieve the Location object the client issues an HTTP GET request to the following URI:

```
/location/{locid}
```

Figure 10: Location Retrieval URI

Where {locid} is replaced by the location identifier from the `Locality` returned from the check-in operation.

ProxStor will internally retrieve the `Location` associated with the `locid` and returns the structure in JSON format. Again the client must de-serialize the object to have a functioning `Location` accessible memory. The client now has the following new information on the user's current location:

- Description
- Address
- Type (business, residence, etc.)
- GPS coordinates

Chapter 5: Results

Although ProxStor remains a proof of concept without fully optimized critical code paths, benchmarking is performed to confirm feasibility of design. The ease of ProxStor system deployment as well and ease of client application development is qualitatively assessed. The system's runtime performance is evaluated quantitatively to ascertain the feasibility of the current design.

Software engineering metrics on the ProxStor codebase are provided as well as lessons learned while developing the system.

The software and hardware used in this section is documented below.

Item	Detail
ProxStor	v0.1
Operating System	Linux Mint running kernel 3.13.0-24-generic
Java SE Runtime Environment	v1.7.0_67
Winstone Servlet Container	v0.9.10
TinkerPop Blueprints	v2.6.0
OrientDB	Community v1.7.9
Neo4j	Community v2.1.5
NetBeans	v8.0.1

Table 1: Software Tested

Item	Detail
Hardware platform	Dell PowerEdge R710
Processor	2 x six-core Intel Xeon X5670 @ 1.93GHz
System Memory	96 GiB DDR 1066MHz
Storage Controller	Dell PERC H700
Hard Disk	8 x Toshiba PX02SSU020 200GB SSD

Table 2: Hardware Tested

5.1 USAGE ASSESSMENT

5.1.1 System Deployment

The system deployment is simple and straightforward. ProxStor is distributed as a Web Application Archive (WAR), which is a JAR file containing all necessary libraries and support files for the web application. After copying this single WAR file onto the target system, the administrator directs the chosen Java servlet container to this archive.

All testing performed in this report used the Winstone servlet container invoked with the following command:

```
$ java -jar winstone.jar --warfile=proxstor-webapp.war
```

Note that the specific deployment steps will vary with the chosen container.

5.1.2 Graph Database Connection

Experimentation in moving ProxStor between different graph database systems validates the design decision to remain flexible through the use of the Blueprints interface. The testing in this report involves database connections that are established and tested on OrientDB, Neo4j, and TinkerGraph graph implementations. Only the connection information differs.

Because the system natively supports the use of any graph database which itself implements the Blueprints interface the administrator simply uses the Administrator URI to either connect to a running graph database server or create a new instance.

5.1.3 Client Application

The ProxStor Connector and API are validated by enabling the rapid development of the benchmarking applications used in this report. The client simply creates an instance of `ProxStorConnector` exposing all necessary ProxStor-related functionality. This simple interface frees up development time to focus on other aspects such as mock data collection, thread management, and result data collection and reporting. The ProxStor connection API and object [de]serialization work as designed.

5.2 PERFORMANCE METRICS

The `proxstor.testing` package holds all the tests used to gather the below performance data. For more insight into how the testing was run (including more testing not shown here) please refer to the source.

5.2.1 Static Testing

The first class of benchmarking is performed to gauge the impact of the growth in size of the back-end graph database has on the latency of common requests. The category is referred to as static because the database is loaded with data to reach the targeted graph database node size before the identified requests are performed. No concurrent use of ProxStor occurs during the benchmark. The goal is to confirm that ProxStor's performance is not adversely affected by a large back-end database as well as to determine if the performance results will be consistent across two different database implementations.

The following requests are tested:

- Check into random location
- Check out of random location
- Query current location of random user
- Query all user's within a randomly selected location

The queries are performed on the same database as it grows in size, with each growth factor of 10 recorded. Each individual test is performed 1,000 times with the average response time in milliseconds (ms) captured. For all tests care is taken not to benefit from a cached response from the system.

The testing is performed on OrientDB and repeated on Neo4j. Due to the time required to load the database will one billion records the Neo4j testing did not move past the 100 million node count. One billion records is the growth point where a single ProxStor database should be sharded and the work distributed across multiple nodes.

The results of static testing on OrientDB are in the below table.

Request Type	**Database Size (k-Nodes)**					
	10	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Check-in	56	56	66	67	69	74
Check-out	5	5	5	5	5	7
Current Location	8	7	7	8	8	10
All Within Location	6	6	7	7	8	10

Table 3: Static Testing Response Time (ms) – OrientDB

The results of static testing on Neo4j are in the below table.

Request Type	**Database Size (k-Nodes)**				
	10	**100**	**1,000**	**10,000**	**100,000**
Check-in	53	53	56	57	59
Check-out	3	3	3	3	3
Current Location	7	6	6	7	7
All Within Location	6	6	6	7	7

Table 4: Static Testing Response Time (ms) – Neo4j

Both tables show similar degradation in response time as the database size increased, although in all benchmarks the Neo4j backed testing produced lower response times.

- The *check-in* operation is the longest running operation, which is a reflection that the operation must perform two lookup operations as well as two database insertion operations.
- The *current location* and *all within location* operations involve database look-up operations, but no updates. This explains their lower response time.
- The *check-out* operation involves both a lookup and a modification to an existing edge between vertices. The operation involves no data marshalling, potentially explaining the very low response time.

As database size grew by a factor of ten thousand the response time to a user check-in increased 23% and 11% for OrientDB and Neo4j respectively. In the most extreme example OrientDB check-in response time increased by 33% by the final database size of one billion nodes, which represents a growth factor of one hundred thousand in data set size. This meets performance expectations.

To better characterize the impact database access, and its associated persistent storage commits, have on the response time a subset of tests is now performed on a TinkerGraph database instance. TinkerGraph is an in-memory reference property graph implementation [18] which avoids any access to storage. Because the overhead of updates to main memory are an order of magnitude faster than mass storage updates, the goal is to reveal the actual latency inherent in the non-disk related actions - network connections, data marshalling, and the ProxStor service computations. Note that TinkerGraph instances are not stable with a database size greater than a million nodes.

Request Type	**Database Size (k-Nodes)**		
	10	**100**	**1,000**
Check-in	51	51	51
Check-out	3	4	4
Current Location	6	6	6
All Within Location	6	6	6

Table 5: Static Testing Response Time (ms) – TinkerGraph

These results strongly indicate that the majority of the request response time in both OrientDB and Neo4j deployments is spent outside of database access. For example, the relatively expensive *check-in* operation continues to require more than 50 milliseconds even when there are no disk accesses required.

To further investigate this phenomenon ProxStor is instrumented to record the time taken inside each JAX-RS request handler, taking care to track all possible code paths. To minimize the impact of this data collection the processing times are stored in an in-memory data structure and only retrieved once all testing is complete. The goal is to identify how much time is spent *within* ProxStor's code.

Below is the table of results for a ten thousand node database. All times are in milliseconds (ms).

Request Type	**10-k Node Database Size**		
	OrientDB	**Neo4j**	**TinkerGraph**
Check-in	8.89	5.46	0.15
Check-out	3.28	0.44	0.43
Current Location	0.42	0.18	0.07
All Within Location	0.87	0.40	0.08

Table 6: ProxStor Internal Processing Times

These results show that the majority of the request response time is not contained within ProxStor itself. This reveals that the majority of the time is spent in the network handlers, HTTP listener, JAX-RS processing, and data marshalling.

As expected, the results show that the database implementations with physical manifestations take significantly more time than an in-memory version. The data confirms that Neo4j is faster than OrientDB for all request types. This performance advantage for Neo4j is consistent with the advantage seen when comparing Table 3 against Table 4 above. Recall that ProxStor is executing the same code path regardless of the underlying database.

5.2.2 Data Evolution

To explore how ProxStor performance ages as data evolves over time an increasing load is applied to a system starting from a fresh initialization. Initially there are 10 synthetic clients creating content and performing queries. Every 10x growth in the size of the database causes a corresponding 2x increase in the client load. The aggregate average latency of the responses is recorded throughout each step. This test approximates the real world observation that as the number users of a system grows so does the size of the working dataset. All data below is collected while ProxStor is connected to an OrientDB instance.

Database Size	**Client Count**	**IOPS**	**Average Response Time (ms)**
100,000	100	1024	98
1,000,000	200	727	275
10,000,000	400	630	635
100,000,000	800	570	1403

Table 7: ProxStor Dynamic Testing Results

At 100 k-nodes × 100 clients ProxStor is able to maintain an average response time of 98ms, exceeding the non-functional requirement expectation for a speed. At 1,000 k-nodes × 200 clients ProxStor is meeting expectations for response time with an average of 275ms. At 10,000 k-nodes × 400 clients the user is likely to perceive a lag in response time, which does not meet performance expectations. Finally, at 100,000 k-nodes × 800 the response time is approaching 1.5 seconds, which is unacceptable.

Based on these results alone a recommendation that a single ProxStor instance should not exceed 400 concurrent users to maintain acceptable performance is reasonable. However, due to the identification of a source of high latency outside of ProxStor's processing (see above section) instrumentation is added to the final state of the data evolution database to track time spent within ProxStor's request handler. The results of the instrumentation show that under the high load of 800 concurrent clients ProxStor's request handlers take an average of 245 milliseconds to complete. This is an improvement of more than 1.1 seconds, which is encouraging. If the total processing time outside of the handler can be reduced the overall performance of ProxStor under load will improve dramatically.

Identification of the source of high response time, as well as correction, is called out in the future work section of Chapter 6.

5.2.3 Benchmarking

ApacheBench [18] is used to benchmark ProxStor's performance in the situation when multiple requests are outstanding (i.e. concurrency). ApacheBench is a popular HTTP server benchmarking tool from the Apache software foundation included in the standard Apache HTTP web server source distribution [20]. ApacheBench is chosen here because it is recognized and respected as best in class in the field of web application performance benchmarking. Because it is written in C these benchmarks can further be used to highlight the Java threading and network socket layer overhead contained in the results from Section 5.2.2.

To fully stress ProxStor the ApacheBench for Multi URL [21] modification is used which allows different URLs to be accessed concurrently. For this testing modifications are made to ProxStor to work within the operational constraints of ApacheBench. For example, because ApaceBench has no DELETE request support, ProxStor will temporarily interpret a certain GET request to the check-in URI as a request to check-out of a location. In all cases of temporary modification care is taken to ensure all internal operations remain identical to the unmodified version, thus preserving the value of these benchmarking results.

ApacheBench is run against ProxStor instances connected to OrientDB, Neo4j, and TinkerGraph. For each of these configurations the database size is set to one million nodes. Benchmarking is performed at 10, 100, and 250 concurrent request levels. For each request type (see below), ten thousand individual requests are performed in a single run, with the average response time calculated across ten runs.

The following requests are tested:

- Check into random location
- Check out of random location
- Query current location of random user
- Query all user’s within a randomly selected location

The tables below contain the average (mean) response time for the indicated request type across all runs against the specific database at the specified concurrency level.

The results for OrientDB are presented below:

Request Type	**Concurrent Request Count**		
	10	**100**	**250**
Check-in	3	18	47
Check-out	3	18	46
Current Location	2	12	34
All Within Location	2	13	34

Table 8: ApacheBench Response Time (ms) – OrientDB

The results for Neo4j are presented below:

Request Type	**Concurrent Request Count**		
	10	**100**	**250**
Check-in	3	17	40
Check-out	3	16	38
Current Location	2	12	26
All Within Location	2	13	31

Table 9: ApacheBench Response Time (ms) – Neo4j

The results for TinkerGraph are presented below:

Request Type	**Concurrent Request Count**		
	10	**100**	**250**
Check-in	18	170	516
Check-out	4	36	91
Current Location	2	11	28
All Within Location	2	11	27

Table 10: ApacheBench Response Time (ms) – TinkerGraph

Across all three databases the response time rises as the number of concurrent requests increases, which is expected. Both OrientDB and Neo4j installations gracefully handle the increasing concurrency with a worst case response time increasing by a factor of 17 for a corresponding concurrency load increase factor of 25.

At the starting concurrency level of 10, TinkerGraph's response time is six times longer than the other two databases revealing that this in-memory implementation is not optimized for multiple outstanding operations, even at a minimal concurrency level. The TinkerGraph installation further reveals poor scaling with response time increasing by a factor of 29 for a concurrency increase factor of 25. Both these observations are consistent with TinkerGraph's stated goal to be a reference implementation to aid development and unit testing [18], not performance.

The ApacheBench performance results for OrientDB show response time increasing by a factor of approximately 2.6 as concurrency increases from 100 to 250. Table 7 in Section 5.2.2 shows response time increasing by a factor of approximately 2.8 as client count increases from 100 to 200. Because these increases are roughly equivalent it is concluded that response time scaling factors are independent of the specific client implementation. These results further show that the Java-based testing client used in Section 5.2.2 introduces a response time overhead factor of greater than five times that incurred by ApacheBench supporting the observation that a majority of time is spent within Java code outside ProxStor, such as networking and data marshalling layers.

5.3 Software Engineering Metrics

ProxStor itself was written entirely in the Java programming language. In addition to the project code some small test scripts were developed and Postman configuration was maintained. Below are the source lines of code (SLOC) for each category. SLOC determined by the loc-calculator [22] tool.

Type	SLOC
Java	12,635
Scripts	41
Other	1,250

Table 11: ProxStor SLOC by Type

The Java SLOC is further broken down by project below.

Project	SLOC
proxstor-connection	3258
proxstor-testing	2950
proxstor-webapp	6427

Table 12: ProxStor SLOC by Java Package

The git revision control system [23] was used to manage the changes throughout ProxStor development. The following statistics were gathered from the repository holding all components of ProxStor, accompanying scripts, Postman configuration, and this report.

Type	Value
Commits	312
Lines Inserted	26,829
Lines Deleted	10,912

Table 13: Git Repository Statistics

Chapter 6: Conclusion

This report has described ProxStor, an implementation of one potential form of the motivating envisioned system. The realized system confirms that such systems are feasible and can scale well to high number of connected devices. When developing such a system the designer must:

- Avoid lock-in to any one single NoSQL database solution
- Participate in the cloud computing paradigm shift by abstracting the system into a service
- Enable a wide range of client devices, from the high-powered mobile phone down to low-powered embedded sensors
- Consider closely the performance consequences of any potential web container technologies and accompanying libraries

It is recognized that ProxStor would be most effective (and more prolific) if deployed as a service running on mobile devices – not simply a standalone application. Future work should strive to enable this goal otherwise they risk the system being irrelevant.

6.1 Lessons Learned

6.1.1 What Worked

Below is a listing of the things which worked well when developing ProxStor, broken down by category.

The following software packages were essential to the success of this project: Jersey, Netbeans, Maven, Blueprints, Postman, and Winstone. Without the hard work already invested in these projects ProxStor would never have been achieved.

The following design decisions were also essential: servlet-based design, JAX-RS, separation of concerns into DAOs, use of the graph relational database model, and abstracting away from specific database implementation with the use of blueprints. With the selection of each of these points the realization of the system became more stable, more extensible, more standards based, and most importantly, more concrete.

6.1.1.1 Design Patterns

Development of ProxStor utilizes some well-known design patterns, which are critical to realizing the successful system.

The Singleton pattern is used in both the Data Access Object (DAO) layers and in the ProxStor connection package. Concurrency issues are side-stepped by enforcing serialization through the use of static methods in these layers.

The Factory pattern is used to create and configure instances of the back-end graph database without exposing instantiation logic for all possible supported Blueprints enabled databases.

The Proxy pattern is used to provide a simplified interface to client. The complexity of the underlying data model and objects is not exposed. The ProxStor connector wraps the already simplified web API in an even simpler interface.

6.5.2 What Didn’t Work

While developing ProxStor there were some things which didn’t work too well. These were: the Blueprints documentation, the TinkerPop projects other than Blueprints, and both the Tomcat and Glassfish servlet containers.

The Blueprints documentation left much to be desired on how to actually the library in anything other than the trivial cases. This forced the reader to consult the JavaDoc, which were sparse, but good naming conventions allowed forward progress.

The TinkerPop projects beyond Blueprints, such as Gremlin and Furnace, are even more poorly documented. The stated goals for these projects excite, but the trial and error required to use them in ProxStor was beyond the scope of this report.

The Tomcat [24] and Glassfish [25] servlet containers proved to be more unstable and slower to launch than alternatives. Although support for launching and debugging servlets from with the IDE is integrate with these two container frequent crashes or hangs drove ProxStor development to Winstone.

6.2 Relationship to Existing Work / Relationship to Prior Work

A General Framework for Geo-Social Query Processing [26] provides a framework for data management algorithms relating to querying Geo-Social Networks (GeoSNs). The approach is algorithmic in nature whereas ProxStor approaches the problem from a software engineering perspective.

RAML [27] is a modeling language for describing RESTful APIs. While one of the goals for ProxStor was to create a coherent and succinct API, the creation of a broader purpose API *description* language was not.

Research exists in the area of database research and optimization, but ProxStor remains agnostic to the details of the back-end graph database implementation. This project specifically does not head down the path of tuning behavior for a specific database solution.

The ProxStor project remains orthogonal to all the above.

6.3 Future Work

Along the journey from conceptualization to actual implementation several questions have been answered, yet despite this success much future work remains in this area. This work falls into four categories: code improvements, enhanced testing, maintainability, and well as general conception level concerns.

6.3.1 Code Improvements

The first identified code improvement is to optimize resource intensive code paths related to executing multi-dimensional queries. The TinkerPop Gremlin [28] and Furnace [29] projects have been identified to help in this area.

The second is code enhancement to support complex queries with a more expressive language. This should permit clients to ask more specific questions and actually consume fewer resources within ProxStor. Such an addition includes extensions to the client web API as well as the language processing and execution components.

The third relates to determining the distance between Locations for the purposes of queries. Today ProxStor relies upon the database to be populated purposefully populated with such data, which is unrealistic in true deployments. The system should be extended to either calculate these distances programmatically as a Location is added, or perhaps as a batch processing in an offline mode.

Lastly, as identified in Chapter 5, performance optimizations opportunities exist somewhere in the request processing both before and after the ProxStor request handlers are involved. These source of these performance opportunities are not well understood, but identification and correction should be completed.

6.3.2 Enhanced Testing

Scale out both horizontally and vertically must be performed to understand how ProxStor scales. Of particular concern is whether the current design properly scales horizontally. With the variety of approaches taken by the generally available popular Graph Database offerings it is believed the answer will depend not only on ProxStor's design, but also on the specifics of the database deployment. Care will have to be taken to properly shard the database to maintain optimal performance. For example, a user and his/her proximity history should reside within a single database server instance.

Real world mobile application development and testing should also be performed. This applies to both low- and high-powered devices. The assumptions regarding low-powered devices ease of use should be confirmed and a real-world example of a mobile application performing check-in and check-out operations should be developed to identify unknown hurdles.

6.3.3 Maintainability

How does ProxStor's API documentation remain synchronized with the current revision of the web interface? Appendix A of this report was generated manually by a combination of referencing both ProxStor's source code and wiki [30] and was likely successful only because of this author's familiarity with the interface. Effort should be invested in a system to maintain and generate API documentation from the source code. A system such as Swagger [31] may be a good fit.

The wiki is also a susceptible to drifting out of date.

6.3.4 General Concerns

The ProxStor approach does not currently address how to relocate digital fingerprints. For example, if a Bluetooth Low Energy beacon associated with location A is moved to Location B, how can ProxStor become aware of this? What should ProxStor do with all the previous localities and location references associated with the particular environmental? One possibility for *detection* of the relocation is to solicit the assistance of the high-powered device users. Each check-in could be challenged by the user if he/she believes the calculated location is incorrect. The application would prompt the user to enter the correct information and when enough users have reported the error then ProxStor would be switched over to the new location. Which layer performs this ‘enough’ check, mobile application or ProxStor, has not been identified.

Secondly, ProxStor has no awareness of moving environmental fingerprints. How would a user check into a moving city bus? One half of this works today – the association of environmental(s) with a location for the bus. The query for the user’s current location breaks down however. This is an area open to future work.

Also, perhaps most concerning, is that there is no security model built into ProxStor. Do users want all movements tracked? At the very least only those users who you consider a friend should be allowed to access your movements, but perhaps some locations are more private than others. Models exist to address at least some of these concerns, but clearly opportunity exists to go deeper into this area.

6.5 Obtaining ProxStor

All ProxStor source code is hosted on the Github service. The source includes the ProxStor web service, ProxStor Connector, test clients, Postman configuration, and this report. The code is available at:

https://github.com/jgiannoules/proxstor

The ProxStor Wiki documents many of the details in this report, but also provides some sample queries and a useful API cheat sheet. The Wiki is locations at:

https://github.com/jgiannoules/proxstor/wiki

The ProxStor Connector, as well as all internal classes, is documented using Javadoc. The HTML Javadoc for ProxStor is available at:

https://jgiannoules.github.com/proxstor/apidocs/

Appendix A

The complete ProxStor Web API is documented in the following sections. For each URI the following is described:

- Supported HTTP requests
- Operations performed within ProxStor
- Success and Failure HTTP response status codes

The Web Services interface can roughly be broke into fixed-object and dynamic categories. In either case all URIs are relative to the same base.

A.1 Fixed Components Web Services Interfaces

The Web Interface provides CRUD interfaces for lifecycle management of the following ProxStor object types. These are considered *fixed objects* due to their relative static nature.

- Users
    - *Knows* relationships between users
    - Devices *owned* by Users
- Locations
    - Locations connected by *Within* relationship
    - Locations connected by *Nearby* relationship
    - Environmetals *inside* Locations
- Locality object representing a period of time User was in a Location
- Searching the above components
- Administration of ProxStor

Search actions are siloed to a specific object type (User, Devices, etc.) while fixed component queries are formed with URI constructs. The static data returned has context assumed from the clients query. For example, querying ProxStor for all users a user with a specified *userId* knows with a certain strength *minStrength* will return a list of User JSON object without supporting metadata. The client must maintain the context of whose friends list is represented.

A.1.1 User URI

All operations related to manipulation of user objects within the database happen relative to the user URI:

Figure 11: User URI

#	URI	Method	HTTP Header	Description
1	/	POST	Content-Type: application/json Accept: application/json	Create user
2	/{userid}	GET	Accept: application/json	Retrieve user
3	/{userid}	PUT	Content-Type: application/json	Update user
4	/{userid}	DELETE		Delete user

Table 14: User URI Methods

The URI /{userid} is referred to as the base user + userId URI for convenience. The {userid} notation is meant to signify that the numeric database-specific user id is to be inserted in place of the {userid} string.

A.1.1.2 Create User

To create a new user the client should prepare a `proxstor.API.User` object containing the user information. Note that at this time the userId field is null because the system has not yet assigned an id. This User object is then converted into JSON and sent via a HTTP POST request to the user URI with the header fields *Content-type* and *Accept* both set to *application/json*. If the user is successfully added ProxStor will return an HTTP status 201 (Created) with the new userId in two locations. The client is free to choose whichever extraction method it wishes.

The first location is with the Location field of the response header. This Location contains the full URI to the User object and can be directly used in a GET request. If the userId alone is needed the client must process the field to retain only the id portion after the final forward slash.

The second location is in the body of the response. The full User object, including the userId, is returned to the client in JSON. This is the location used by the ProxStor Connector.

A.1.1.2 Retrieve User

To retrieve a user from the database the client must send a GET HTTP request to the base user + userId URI path with the header field *Accept* set to *application/json*. If the specified userId is valid ProxStor will respond with an HTTP status of 200 (OK) with the JSON representation of the User object in the response body.

A.1.1.3 Update User

To update a user the requestor must send the JSON representation of the updated User object in an HTTP PUT request to the base user + userId URI. Note that the userId in the URI and the userId in the JSON representation of User must be identical in addition to being a valid user id in the database. If the user update was successful ProxStor will respond with an HTTP status of 204 (No Content) with no content in the response body.

A.1.1.4 Delete User

To delete a user from the database the client must send an HTTP DELETE request to the base user + userId URI. No special header fields need be specified. ProxStor will respond with an HTTP status of 204 (No Content) if the deletion was successful.

A.1.2 Knows URI

All operations relating to the *knows* relationship between users are performed relative to the knows URI:

```
/user/{userid}/knows
```

Figure 12: Knows URI

Note that all knows operations are in the context of a specific base user + userId URI, and thus the context of a user. A knows relationship does not exist without at least specifying the user who is asserting the level to which they know someone else and the URI structure is representing this constraint.

In the following table the strength value, {s}, is to degree to which the knows relationship is established. The supported values are 1 through 100, with higher values representing stronger relationships. It is envisioned that an application building upon ProxStor will categorize the value ranges into terms understood to the user. For example, friendship could be anything greater than 50. For the purposes of this report keep in mind that the strength value is a required component of the URI.

#	URI	Method	HTTP Header	Description
1	/strength/{s}/user/{userid2}	POST		Create know relationship
2	/strength/{s}	GET	Accept: application/json	Retrieve users who userid knows
3	/strength/{s}/reverse	GET	Accept: application/json	Retrive users who know userid
4	/strength/{s}//user/{userid2}	PUT		Update knows relationship
5	/user/{userid2}	DELETE		Delete knows relationship

Table 15: Knows URI Methods

The URI /strength/{s} is referred to as the strength URI for convenience. The {s} notation is meant to signify that the numeric value for the strength of this knows relationship is to be inserted in place of the {s} string.

The URI /user/{userid2} is referred to as the userId2 URI for convenience. The {userid2} notation is meant to signify that the numeric database-specific user id is to be inserted in place of the {userid2} string.

A.1.2.1 Create Knows

To create a new knows relationship between two users the client must send an HTTP POST to the knows + strength + userId2 URI. The URI encodes all the information ProxStor needs to establish the relationship, therefore no JSON representation is sent by the client. If both the userId and userId2 are valid users (and not the same value), the strength value is in the valid range (1-100), and a knows relationship does not already exist, then ProxStor will respond with an HTTP status 201 (Created).

A.1.2.2 Retrieve Knows

To retrieve all the users whom a specific user knows with at least a minimum strength the client must send an HTTP GET to the knows URI with the HTTP header field *Accept* set to *application/json*. ProxStor will confirm the validity of the user id in the URI and find all the users who the user knows with at least strength from the URI. If the user id is valid ProxStor will respond with HTTP status 200 (OK) with the body of the response containing a JSON representation of a list of `proxstor.api.User` objects.

A.1.2.3 Retrieve Knows Reverse

To retrieve the users who know a specific user with at least a minimum strength the client appends /reverse to the knows URI. Note that this retrieval is the opposite direction of that in section 4.3.2.2. In other words, these are the users who know the specified user id with minimum strength s – not users who user id knows. The remainder of the interface is identical to 4.3.2.2.

A.1.2.4 Update Knows

To update the strength value in an established knows relationship the client must issue an HTTP PUT request to the knows + strength + userId2 URI with the updated strength value encoded in the URI. If a knows relationship already exists from userid to userid2 and the strength value is valid, then ProxStor will update the relationship and respond with HTTP status 204 (No Content).

A.1.2.5 Delete Knows

To remove the knows relationship between two users the client must send an HTTP DELETE request to the knows + userId2 URI. Note that no strength value is specified. If a knows relationship exists between userid and userid2, then ProxStor will delete this relationship and respond with HTTP status 204 (No Content).

If either of the user ids was invalid or the knows relationship was not already established ProxStor will respond with HTTP status 404 (Not Found).

A.1.3 Device URI

All operations related to manipulation of device objects within the database happen relative to the device URI:

```
/user/{userId}/device
```

Figure 13: Device URI

Note that all device-related operations are in the context of a specific base user + userId URI, and thus a single specific user. A device does not exist in ProxStor without being associated with a user and so the URI naturally expresses this.

#	URI	Method	HTTP Header	Description
1	/	POST	Content-Type: application/json Accept: application/json	Create device
2	/	GET	Accept: application-json	Retrieve all user's devices
3	/{devid}	GET	Accept: application/json	Retrieve device
4	/{devid}	PUT	Content-Type: application/json	Update device
5	/{devid}	DELETE		Delete device

Table 16: Device URI Methods

The URI /{devid} is referred to as the base device + devId URI for convenience. The {devid} notation is meant to signify that the numeric database-specific device id is to be inserted in place of the {devid} string.

A.1.3.1 Create Device

To create a new device the client should prepare a `proxstor.API.Device` object containing the device information. Note that at this time the devId field is null because the system has not yet assigned an id. This Device object is then converted into JSON and sent via a HTTP POST request to the desired user's device URI with the header fields *Content-type* and *Accept* both set to *application/json*. If the device is successfully added ProxStor will return an HTTP status 201 (Created) with the new devId in two locations. The client is free to choose whichever extraction method it wishes.

The first location is with the Location field of the response header. This Location contains the full URI to the Device object and can be directly used in a GET request. If the devId alone is needed the client must process the field to retain only the id portion after the final forward slash.

The second location is in the body of the response. The full Device object, including the devId, is returned to the client in JSON. This is the location used by the ProxStor Connector.

A.1.3.2 Retrieve User's Devices

To retrieve all devices owned by a specific user the client must sent an HTTP GET request to the base device URI for the owning user. Do not specify a devId specific portion to the URI. The request header field *Accept* should be set to *application/json*. If the user specified in the URI is valid and owns at least one device ProxStor will respond with an HTTP status of 200 (OK) and the body of the response will contain a JSON representation of a list of Device objects. If the user is valid, but owns no devices, the response status will be 204 (No Content) with no contents in the body.

A.1.3.3 Retrieve Device

To retrieve a single device from the database the client must send an HTTP GET request to the base device + devId URI path for the owning user with the header field *Accept* set to *application/json*. If the specified devId is valid and owned by the userId in the URI ProxStor will respond with an HTTP status of 200 (OK) with the JSON of the of the Device object in the response body.

A.1.3.4 Update Device

To update a device the requestor must send the JSON representation of the updated Device object in an HTTP PUT request to the owning user's base device + devId URI. Note that the devId in the URI and the devId in the JSON representation of Device must be identical in addition to being a valid device id in the database and be owned by the userId in the URI. If the device update was successful ProxStor will respond with an HTTP status of 204 (No Content) with no content in the response body.

A.1.3.5 Delete Device

To delete a device from the database the client must send an HTTP DELETE request to the owning user's base device + devId URI. No special header fields need be specified. ProxStor will respond with an HTTP status of 204 (No Content) if the deletion was successful.

A.1.4 Location URI

All operations related to manipulation of location objects within the database happen relative to the location URI:

Figure 14: Location URI

#	URI	Method	HTTP Header	Description
1	/	POST	Content-Type: application/json Accept: application/json	Create location
2	/{locid}	GET	Accept: application/json	Retrieve location
3	/{locid}	PUT	Content-Type: application/json	Update location
4	/{locid}	DELETE		Delete location

Table 17: Location URI Methods

The URI /{locid} is referred to as the base location + locId URI for convenience. The {locid} notation is meant to signify that the numeric database-specific location id is to be inserted in place of the {locid} string.

A.1.4.2 Create Location

To create a new location the client should prepare a `proxstor.API.Location` object containing the location information. Note that at this time the locId field is null because the system has not yet assigned an id. This Location object is then converted into JSON and sent via a HTTP POST request to the location URI with the header fields *Content-type* and *Accept* both set to *application/json*. If the location is successfully added ProxStor will return an HTTP status 201 (Created) with the new locId available in two locations. The client is free to choose whichever extraction method it wishes.

The first location is with the Location field of the response header. This Location contains the full URI to the Location object and can be directly used in a GET request. If the locId alone is needed the client must process the field to retain only the id portion after the final forward slash.

The second location is in the body of the response. The full Location object, including the locId, is returned to the client in JSON. This is the method used by the ProxStor Connector.

A.1.4.2 Retrieve Location

To retrieve a location from the database the client must send a GET HTTP request to the base location + locId URI path with the header field *Accept* set to *application/json*. If the specified locId is valid ProxStor will respond with an HTTP status of 200 (OK) with the JSON representation of the Location object in the response body.

A.1.4.3 Update Location

To update a location the requestor must send the JSON representation of the updated Location object in an HTTP PUT request to the base location + locId URI. Note that the locId in the URI and the locId in the JSON representation of Location must be identical in addition to being a valid location id in the database. If the location update was successful ProxStor will respond with an HTTP status of 204 (No Content) with no content in the response body.

A.1.4.4 Delete Location

To delete a location from the database the client must send an HTTP DELETE request to the base location + locId URI. No special header fields need be specified. ProxStor will respond with an HTTP status of 204 (No Content) if the deletion was successful.

A.1.5 Within URI

All operations relating to the *within* relationship between location are performed relative to the within URI:

```
/location/{locid}/within
```

Figure 15: Within URI

Note that all within operations are in the context of a specific base location + locId URI, and thus the context of a location. A within relationship does not exist without at least specifying the location which is within another location.

#	URI	Method	HTTP Header	Description
1	/{locid2}	POST		Create within relationship
2	/	GET	Accept: application/json	Get locations within
3	/reverse	GET	Accept: application/json	Get location containing
4	/{locid2}	GET		Test location within
5	/{locid2}	DELETE		Delete within relationship

Table 18: Within URI Methods

The URI /{locid2} is referred to as the locId2 URI for convenience. The {locid2} notation is meant to signify that the numeric database-specific location id is to be inserted in place of the {locid2} string.

A.1.5.1 Create Within

To create a new within relationship between two locations the client must send an HTTP POST to the within + locId2 URI. The URI encodes all the information ProxStor needs to establish the relationship, therefore no JSON representation is sent by the client. If both the location identifiers are valid (and not the same value) and a within relationship does not already exist, then ProxStor will respond with an HTTP status 201 (Created).

A.1.5.2 Retrieve Within

To retrieve all the locations within a specific location the client must send an HTTP GET to the within URI with the HTTP header field *Accept* set to *application/json*. ProxStor will confirm the validity of the location id in the URI and find all the locations within the specified location. If the locsation id is valid ProxStor will respond with HTTP status 200 (OK) with the body of the response containing a JSON representation of a list of `proxstor.api.Location` objects.

A.1.5.3 Retrieve Within Reverse

To retrieve the locations which contains a specific location the client appends /reverse to the within URI. Note that this retrieval is the opposite direction of that in section 4.3.5.2. The remainder of the interface is identical to 4.3.5.2.

A.1.2.4 Test Within

To test whether a location is within another location the client may issue an HTTP GET request to the within + locId2 URI. If a within relationship exists between locid and locid2 then ProxStor will respond with HTTP status 204 (No Content).

A.1.2.5 Delete Within

To remove the within relationship between two locations the client must send an HTTP DELETE request to the within + locId URI. If a within relationship exists between locid and locid2, then ProxStor will delete this relationship and respond with HTTP status 204 (No Content).

If either of the locations ids was invalid or the within relationship was not already established ProxStor will respond with HTTP status 404 (Not Found).

A.1.6 Nearby URI

All operations relating to the *nearby* relationship between locations are performed relative to the nearby URI:

```
/location/{locid}/nearby/distance/{d}
```

Figure 16: Nearby URI

Note that all nearby operations are in the context of a specific base location + locId URI, and thus the context of a location. A nearby relationship does not exist without at least specifying the location who is asserting the distance to which they are nearby some other location. The URI structure is representing this relationship.

The distance value, {d}, is to distance in meters between locations.

#	URI	Method	HTTP Header	Description
1	/{locid2}	POST		Create nearby relationship
2	/	GET	Accept: application/json	Retrieve locations within distance
3	/{locid2}	GET		Tests location distance
4	/{locid2}	PUT		Update nearby relationship
5	/{locid2}	DELETE		Delete nearby relationship

Table 19: Nearby URI Methods

The URI /{locid2} is referred to as the locId2 URI for convenience. The {locid2} notation is meant to signify that the numeric database-specific location id is to be inserted in place of the {locid2} string.

A.1.6.1 Create Nearby

To create a new nearby relationship between two users the client must send an HTTP POST to the nearby + locId2 URI. The URI encodes all the information ProxStor needs to establish the relationship, therefore no JSON representation is sent by the client. If both the locid and locid2 are valid locations (and not the same value) and a nearby relationship does not already exist, then ProxStor will respond with an HTTP status 201 (Created).

A.1.6.2 Retrieve Nearby

To retrieve all the locations within a specified distance of a specific location the client must send an HTTP GET to the nearby URI with the HTTP header field *Accept* set to *application/json*. ProxStor will confirm the validity of the location id in the URI and find all the locations nearby within the distance encoded in the URI. If the location id is valid ProxStor will respond with HTTP status 200 (OK) with the body of the response containing a JSON representation of a list of `proxstor.api.Location` objects.

A.1.6.3 Test Nearby

To test whether a location is nearby within a specific distance to another location the client may issue an HTTP GET request to the nearby + locId2 URI. If a nearby relationship exists between locid and locid2 and the distance is less than or equal to d then ProxStor will respond with HTTP status 204 (No Content).

A.1.6.4 Update Nearby

To update the distance value in an established nearby relationship the client must issue an HTTP PUT request to the nearby + userId2 URI with the updated distance value encoded in the URI. If a nearby relationship already exists from locid to locid2 then ProxStor will update the relationship and respond with HTTP status 204 (No Content).

A.1.6.5 Delete Nearby

To remove the nearby relationship between two locations the client must send an HTTP DELETE request to the nearby + locId2 URI. Note that distance must be included in the URI, but the value is ignored in this operation. If a nearby relationship exists between locid and locid2, then ProxStor will delete this relationship and respond with HTTP status 204 (No Content).

If either of the location ids was invalid or the nearby relationship was not already established ProxStor will respond with HTTP status 404 (Not Found).

A.1.7 Environmental URI

All operations related to manipulation of environmental objects within the database happen relative to the environmental URI:

```
/location/{locid}/environmental
```

Figure 17: Environmental URI

Note that all environmental related operations are in the context of a specific base location + locId URI, and thus a single specific location. An environmental does not exist in ProxStor without being associated with a location and so the URI naturally expresses this.

#	URI	Method	HTTP Header	Description
1	/	POST	Content-Type: application/json Accept: application/json	Create environmental
2	/	GET	Accept: application-json	Retrieve all location's environmentals
3	/{environmentalid}	GET	Accept: application/json	Retrieve environmental
4	/{environmentalid}	PUT	Content-Type: application/json	Update environmental
5	/{environmentalid}	DELETE		Delete environmental

Table 20: Environmental URI Methods

The URI /{environmentalid} is referred to as the base environmental + environmentalId URI for convenience. The {environmentalid} notation is meant to signify that the numeric database-specific environmental id is to be inserted in place of the {environmentalid} string.

A.1.7.1 Create Environmental

To create a new environmental the client should prepare a `proxstor.API.Environmental` object containing the environmental information. Note that at this time the environmentalId field is null because the system has not yet assigned an id. This Environmental object is then converted into JSON and sent via a

HTTP POST request to the desired location's environmental URI with the header fields *Content-type* and *Accept* both set to *application/json*. If the environmental is successfully added ProxStor will return an HTTP status 201 (Created) with the new environmentalId in two locations. The client is free to choose whichever extraction method it wishes.

The first location is with the Location field of the response header. This Location contains the full URI to the Environmental object and can be directly used in a GET request. If the environmentalId alone is needed the client must process the field to retain only the id portion after the final forward slash.

The second location is in the body of the response. The full Environmental object, including the environmentalId, is returned to the client in JSON. This is the method used by the ProxStor Connector.

A.1.7.2 Retrieve Location's Environmentals

To retrieve all environmentals inside a specific location the client must sent an HTTP GET request to the base environmental URI for the owning location. Do not specify a environmentalId specific portion to the URI. The request header field *Accept* should be set to *application/json*. If the location specified in the URI is valid and contains at least one environmental ProxStor will respond with an HTTP status of 200 (OK) and the body of the response will contain a JSON representation of a list of Environmental objects. If the location is valid, but contains no environmentals, the response status will be 204 (No Content) with no contents in the body.

A.1.7.3 Retrieve Environmental

To retrieve a single environmental from the database the client must send an HTTP GET request to the base environmental + environmentalId URI path for the owning location with the header field *Accept* set to *application/json*. If the specified environmentalId is valid and contained within the locId in the URI ProxStor will respond with an HTTP status of 200 (OK) with the JSON representation of the Environmental object in the response body.

A.1.7.4 Update Environmental

To update an environmental the requestor must send the JSON representation of the updated Environmental object in an HTTP PUT request to the owning location's base environmental + environmentalId URI. Note that the environmentalId in the URI and the environmentalId in the JSON representation of Environmental must be identical in addition to being a valid environmental id in the database and be inside the location specified in the URI. If the environmental update was successful ProxStor will respond with an HTTP status of 204 (No Content) with no content in the response body.

A.1.7.5 Delete Environmental

To delete an environmental from the database the client must send an HTTP DELETE request to the owning location's base environmental + environmentalId URI. No special header fields need be specified. ProxStor will respond with an HTTP status of 204 (No Content) if the deletion was successful.

A.1.8 Locality URI

A Locality represents the bringing together of a device and an environmental (or a user and a location). The ProxStor system collects these localities through the system lifetime. These are used to answer questions about a user's current and historic locations.

Manipulating a Locality is not very useful by itself (that's what check-in is for). These operations are provided mainly for development and testing purposes.

All operations related to manipulation of Locality objects within the database happen relative to the locality URI:

Figure 18: Locality URI

#	URI	Method	HTTP Header	Description
1	/	POST	Content-Type: application/json Accept: application/json	Create locality
2	/{localityid}	GET	Accept: application/json	Retrieve locality
3	/user/{userid}	GET	Accept: application/json	Retrieve localities for user
4	/{localityid}	PUT	Content-Type: application/json	Update locality
5	/{localityid}	DELETE		Delete locality

Table 21: Locality URI Methods

The URI /{localityid} is referred to as the base locality + localityId URI for convenience. The {localityId} notation is meant to signify that the numeric database-specific locality id is to be inserted in place of the {localityid} string.

A.1.8.2 Create Locality

To create a new locality the client should prepare a `proxstor.API.Locality` object containing the locality information. Note that at this time the localityId field is null because the system has not yet assigned an id. This Locality object is then converted into JSON and sent via a HTTP POST request to the locality URI with the header fields *Content-type* and *Accept* both set to *application/json*.

If the locality is successfully added ProxStor will return an HTTP status 201 (Created) with the new localityId in two locations. The client is free to choose whichever extraction method it wishes.

The first location is with the Location field of the response header. This Location contains the full URI to the Locality object and can be directly used in a GET request. If the localityId alone is needed the client must process the field to retain only the id portion after the final forward slash.

The second location is in the body of the response. The full Locality object, including the localityId, is returned to the client in JSON.

A.1.8.2 Retrieve Locality

To retrieve a locality from the database the client must send a GET HTTP request to the base locality + localityId URI path with the header field *Accept* set to *application/json*. If the specified localityId is valid ProxStor will respond with an HTTP status of 200 (OK) with the JSON representation of the Locality object in the response body.

A.1.8.3 Retrieve User's Localities

To retrieve previous localities for a specified user the client must send an HTTP GET request to the base locality + user + userId URI with header field *Accept* set to *application/json*. If the specified userId is valid ProxStor will respond with HTTP status 200 (OK) and the body containing the JSON list representation of the previous proximity objects associated with the specified user.

A.1.8.4 Update Locality

To update a locality the requestor must send the JSON representation of the updated Locality object in an HTTP PUT request to the base locality + locd URI. Note that the localityId in the URI and the localityId in the JSON representation of Locality must be identical in addition to being a valid locality id in the database. If the locality update was successful ProxStor will respond with an HTTP status of 204 (No Content) with no content in the response body.

A.1.8.5 Delete Locality

To delete a locality from the database the client must send an HTTP DELETE request to the base locality + localityId URI. No special header fields need be specified. ProxStor will respond with an HTTP status of 204 (No Content) if the deletion was successful.

A.1.9 Search URI

All operations related to the searching are relative to the search URI:

Figure 19: Search URI

#	URI	Method	HTTP Header	Description
1	/users	POST	Content-Type: application/json Accept: application/json	Search users
2	/devices	POST	Content-Type: application/json Accept: application/json	Search devices
3	/locations	POST	Content-Type: application/json Accept: application/json	Search locations
4	/environmentals	POST	Content-Type: application/json Accept: application/json	Search environmentals

Table 22: Search URI Methods

The URI for the respective object type is referred to as the object search URI for convenience.

A.1.9.1 Submitting Search

All four URIs for searching are used very similarly. To return search results the client determines which object type to search through and sends an HTTP POST request to appropriate object search URI. The request header fields *Content-type* and *Accept* must both set to *application/json.* The body of the request shall contain a *partially* specified JSON representation of the corresponding object type. For example, to search through users the client might send a partial `proxstor.api.User` with only the email address specified. This causes ProxStor to find all matching users – in this case the single user with the specified email address.

If ProxStor find one or more matches to the search then it responds with HTTP status 200 (OK) and the JSON list representation of the appropriate object types is contained within the body.

If no matches are found ProxStor responds with HTTP status 204 (No Content).

Note that using wildcards or regular expressions inside the fields of the objects is not currently supported.

A.1.10 Administration URI

All operations related to the administration of ProxStor are relative to the admin URI:

/admin

Figure 20: Admin URI

#	URI	Method	HTTP Header	Description
1	/graph	POST	Content-Type: multipart/form-data	Create/connect to database instance
2	/graph	GET		Retrieve database instance
3	/graph	DELETE		Shutdown running database instance

Table 23: Admin URI Methods

The URI /graph is referred to as the base graph admin URI for convenience.

A.1.10.1 Create Database Instance

To instruct ProxStor to create or connect to a backend database instance the administrator must send an HTTP POST to the admin + graph URI containing a multipart/form-data encoded in the URL. These form data elements will be converted into a Map<String, String> and passed to GraphFactory. This allows the administrator to connect ProxStor to any Graph instance supported by Blueprints, whether it's a new instance of a database or a reconnection to an existing one. For more information on GraphFactory see the Blueprints documentation.

If the Graph instance is successfully created ProxStor will return an HTTP response of 200 (OK).

If a Graph instance is already running ProxStor will return an HTTP status of 403 (Forbidden).

If a Graph instance cannot be created from the form parameters provided an HTTP status of 500 (Internal Server Error) will be returned.

A.1.10.2 Retrieve Database Instance

To retrieve information on the running database instance the administrator must send an HTTP GET to the admin + graph URI.

If a running database instance exists ProxStor will return an HTTP status of 200 (OK) and the body of the response will contain the plain text status.

If no running database instance exists ProxStor will return an HTTP status of 503 (Service Temporarily Unavailable).

A.1.10.3 Shutdown Database Instance

To stop (shutdown) a running database instance the administrator must sent an HTTP DELETE request to the admin + graph URI.

If a running database instance exists ProxStor will stop that running (including committing all transactions to disk) and return HTTP status 200 (OK).

If a running database instance does not exist ProxStor will return HTTP status 404 (Not Found).

A.2 Dynamic Components Web Services Interfaces

The dynamic components of the web interface provide the following operations:

- Device check-in
- User check-in
- Query

These *dynamic* components are contrasted against the *fixed* components by both the frequency of their access and update, but also because of the increased complex of the API interface.

A.2.1 Device Check-in URI

All Device check-in actions are relative to the URI:

```
/checkin/device/{devid}/environmental
```

Figure 21: Device Check-in URI

#	URI	Method	HTTP Header	Description
1	/	POST	Content-Type: application/json Accept: application/json	Check-in (Partial Environmental)
2	/	DELETE	Accept: application/json	Check-out (Partial Environmental)
3	/{environmentalid}	POST		Check-in
4	/{environmentalid}	DELETE		Check-out

Table 24: Device Check-in URI Methods

Here a device (devid) reports detecting a environmental artifact. The device is used by a User, and the environmental is in a Location. ProxStor will create a Locality instance associated with the User referencing the Location.

The URI /{environmentalid} is referred to as the environmentalId URI for convenience. The {environmentalid} notation is meant to signify that the numeric database-specific environmental id is to be inserted in place of the {environmentalid} string.

A.2.1.1 Device Check-in (Partial Environmental)

When a device detects a new environmental element it should report the discovery to the ProxStor service by creating a new `proxstor.api.Environmental` object and filling in the known data, such as type and identifier. This Environmental object must then be converted into JSON and sent via a HTTP POST request to the device check-in URI with the header fields *Content-type* and *Accept* both set to *application/json*. If a locality is successfully created from the request ProxStor will return an HTTP status 201 (Created) with the new Locality available in the body of the response as well as indicated in the *Location* field in the header.

A.2.1.2 Device Check-out (Partial Environmental)

After a device successfully checks into a location using the above interface it must also monitor the environmental and notify ProxStor when the device no longer senses it. To report this check-out (no longer sensing environmental) the client must send an HTTP DELETE request to the base device check-in URI with the header field *Content-type* set to *application/json*. The body of the request should contain the partial `proxstor.api.Environmental` object used to check-in, or optionally the complete Environmental object retrieved based on the environmenalId from the Locality object. ProxStor will respond with an HTTP status of 204 (No Content) if the check-out was successful.

A.2.1.3 Device Check-in

If the client already knows the precise environmetnalId corresponding to the environmental being sensed it may use a more optimized non-JSON POSTing interface. The client sends an HTTP POST request to the device check-in + environmentalId URI. The full URI provides ProxStor with the necessary information to associate a device with a environmental. If a locality is successfully created from the request ProxStor will return an HTTP status 201 (Created) with the new Locality available in the body of the response as well as indicated in the *Location* field in the header.

A.2.1.4 Device Check-out

The same non-JSON POSTing interface can be used to check out of a location as well. The client sends an HTTP DELETE request to the device check-in + environmentalId URI. The full URI provides ProxStor with the necessary information to dissociate a device from an environmental. ProxStor will respond with an HTTP status of 204 (No Content) if the check-out was successful.

A.2.2 User Check-in URI

All User related check-in are relative to the URI:

```
/checkin/user/{userid}
```

Figure 22: User Check-in URI

#	URI	Method	HTTP Header	Description
1	/location/{locid}	POST	Accept: application/json	Check-in (Manual)
2	/location/{locid}	DELETE		Check-out (Manual)
3	/	GET	Accept: application/json	Retrieve current locality

Table 25: User Check-in URI Methods

Here a User (userid) reports being in Location {locid}. The request is taken literally. ProxStor will create a Locality instance associated with the User referencing the Location.

A.2.2.1 User Check-in

If a user wishes to manually check into a location this may be achieved by sending an HTTP POST request to the user check-in + location URI. The request header field *Accept* should be set to *application/json*. If a locality is successfully created from the request ProxStor will return an HTTP status 201 (Created) with the new manually created Locality available in the body of the response as well as indicated in the *Location* field in the header.

A.2.2.2 User Check-out

If a user wishes to manually check out of a location this may be achieved similar to the manual check-in process. The client sends an HTTP DELETE request to the user check-in + location URI, but this time there are no requirements on the request header. ProxStor will respond with an HTTP status of 204 (No Content) if the check-out was successful.

A.2.2.3 Retrieve User Locality

To retrieve a specified user's current locality the client must send an HTTP GET request to the base user check-in URI with the request header field *Accept* set to *application/json*. If the user from the URI has a currently active locality ProxStor will respond with an HTTP status of 200 (OK) containing the JSON representation of the Locality in the body. If the user is not currently in any active locality the response will be 204 (No Content).

A.2.3 Query

All query requests are performed relative to the query URI:

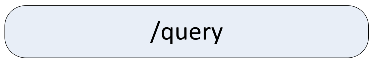


Figure 23: Query URI

#	URI	Method	HTTP Header	Description
1	/	POST	Content-Type : application/json Accept: application/json	Submit Query

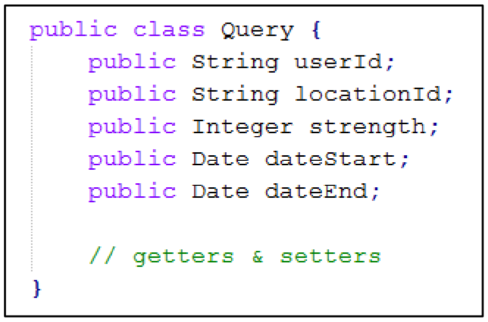
Table 26: Query URI Methods

A.2.3.1 Submit Query

To submit a fixed format query to ProxStor the client must first prepare a JSON representation of `proxstor.api.Query` containing the appropriate defined fields (see next section). The client then sends an HTTP POST request to the query URI with the header fields *Content-type* and *Accept* both set to *application/json*. If the query is accepted then ProxStor will respond with an HTTP status of 200 (OK). The body of the response will contain the JSON list representation of the matching Localities. If the query was valid, but returned no results ProxStor will return an HTTP status of 204 (No Content).

A.2.3.2 Building Query Requests

ProxStor has a single Query URI which accepts a single JSON object representing the class `proxstor.api.Query`.

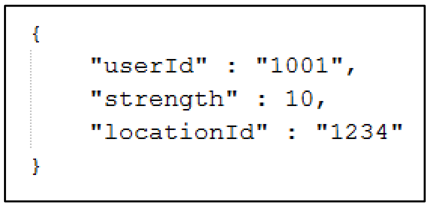


```
public class Query {
    public String userId;
    public String locationId;
    public Integer strength;
    public Date dateStart;
    public Date dateEnd;

    // getters & setters
}
```

Figure 24: Query Class

To build a Query for ProxStor consumption one or more of the fields must be non-null. If ProxStor is able it will return all the Locality instances which match the criteria. For example, to ask ProxStor to return the individuals who are currently at the coffee shop the following JSON representation of Query can be used.



```
{
    "userId" : "1001",
    "strength" : 10,
    "locationId" : "1234"
}
```

Figure 25: Example Query JSON

In this example the *userId* is the User for which the Query is based. It is this user's friends that are the subject of the query. The *strength* is the minimum degree of *knows* needed in this Query. The *locationId* is the coffee shop. So, this Query will return a list of Locality instances representing all of User 1001's friends (that is *knows* ≥ 10) who happen to currently be in the coffee shop right now.

This single Query class is used to ask ProxStor about *who* is *somewhere* as well as *where* was *someone*. Both styles can be constrained by a defined timeframe. If only the start of the timeframe is defined then ProxStor assumes the timeframe runs from *start* up to *now*. Below is a summary of the various query types which ProxStor understands. For all types *userId* must be defined, so it is left off the table for space reasons. Note that the interpretation of *userId* does vary.

userId	**locationId**	**strength**	**dateStart**	**dateEnd**	**Description**
Y	Y	N	N	N	Return *userId*'s current location
Y	N	N	Y	Y/N	Return *userId*'s locations within dateStart to dateEnd timeframe
Y	N	Y	N	N	Return current location of Users who *userId*'s knows with at least *strength*
Y	N	Y	Y	Y/N	Return locations of Users within dateStart to dateEnd timeframe who *userId*'s knows with at least *strength*
Y	Y	Y	N	N	Return Users in *locationId* who *userId*'s knows with at least *strength*
Y	Y	Y	Y	Y/N	Return locations of Users in *locationId* within dateStart to dateEnd who *userId*'s knows with at least *strength*

Table 27: Query Examples

To issue a Query about yourself use your own *userId*. To issue a form of the above Query types, but for all Users regardless of whether *userId* knows them, set *strength* to 0.

References

[1] Atzori, Luigi, Antonio Iera, and Giacomo Morabito. "The internet of things: A survey." *Computer networks* 54.15 (2010): 2787-2805.

[2] Get Started with Bluetooth Low Energy. http://www.jaredwolff.com/blog/get-started-with-bluetooth-low-energy/.

[3] Robinson, Ian, James Webber, and Emil Eifrem. *Graph Databases*. Sebastopol, Calif.: O'Reilly Media, 2013.

[4] Jersey. https://jersey.java.net/.

[5] Java API for RESTful Services (JAX-RS). https://jax-rs-spec.java.net/.

[6] TinkerPop. http://www.tinkerpop.com/.

[7] Tinkerpop/blueprints. https://github.com/tinkerpop/blueprints/wiki/.

[8] Postman REST Client. https://twitter.com/postmanclient/.

[9] Winstone Servlet Container. http://winstone.sourceforge.net/.

[10] Maven. http://maven.apache.org/.

[11] Neo4j. http://neo4j.com/.

[12] OrientDB. http://www.orientechnologies.com/orientdb/.

[13] MongoDB. http://www.mongodb.org/.

[14] ArangoDB. https://www.arangodb.com/.

[15] RESTful Web Services: The Basics. http://www.ibm.com/developerworks/library/ws-restful/index.html.

[16] An Overview of the Emerging Graph Landscape (Oct 2013). http://www.slideshare.net/emileifrem/an-overview-of-the-emerging-graph-landscape-oct-2013.

[17] HTTP Status Codes. http://www.restapitutorial.com/httpstatuscodes.html.

[18] TinkerGraph. https://github.com/tinkerpop/blueprints/wiki/TinkerGraph.

[19] Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.4/programs/ab.html.

[20] Apache HTTP Server Project. http://httpd.apache.org/.

[21] ApacheBench for Multi URL. https://code.google.com/p/apachebench-for-multi-url/.

[22] Loc-calculator. https://code.google.com/p/loc-calculator/.

[23] Git. http://git-scm.com/.

[24] Tomcat. http://tomcat.apache.org/.

[25] Glassfish. https://glassfish.java.net/.

[26] Armenatzoglou, Nikos, Stavros Papadopoulos, and Dimitris Papadias. "A general framework for geo-social query processing." Proceedings of the VLDB Endowment 6, no. 10 (2013): 913-924.

[27] RAML. http://raml.org/index.html.

[28] Tinkerpop/gremlin. https://github.com/tinkerpop/gremlin/wiki.

[29] Tinkerpop/furnace. https://github.com/tinkerpop/furnace/wiki.

[30] Proxstor Wiki. http://github.com/jgiannoules/proxstor/wiki.

[31] Swagger. https://helloreverb.com/developers/swagger.