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: ______________________________
Suzanne Barber

____
Adnan Aziz

ProxStor: Flexible Scalable Proximity Data Storage & Analysis

by

James Peter Giannoules, B.S.C.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 given their time and support over these past two years.

Abstract

ProxStor: Flexible Scalable Proximity Data Storage & Analysis

James Peter Giannoules, M.S.E.

The University of Texas at Austin, 2014

Supervisor: Suzanne Barber

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 (i.e. the Internet of Things). The combination provides the ability for mobile devices to detect when they enter and when they leave the proximity of a space. 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.

This report includes the motivation and requirements as well as the details on the design and implementation of ProxStor.

Table of Contents

List of Tables

List of Figures

List of Illustrations

Section 1: Introduction

1.1 Vision

Motivated by the proliferation of mobile devices and the expansion of digital fingerprints residing in fixed infrastructure [IoT reference] it should be possible to maintain a repository of location identified by these digital fingerprints. These fingerprints may include unique identifying information from a wireless access point, a Bluetooth device, Bluetooth Low Energy beacons, Near-Field Communication devices, or other emerging technologies. With these being (relatively) fixed to a location it is possible to infer the location of a mobile device based on these small observations. A database to maintain this information would potentially need to support tens of millions of locations with many times the number of fingerprints. On top of this layer would reside information on hundreds of millions of users and their devices. Each encounter with a location should be stored persistently, thus the database needs to maintain billions of such encounters. These digital fingerprints do not necessarily need to provide fine-grained location information to be useful.

1.2 ProxStor

ProxStor is an implementing of the above envisioned system. Along the journey from conceptualization to actual implementation several questions have been answered. What type of system best solves this space? How should the data be stored and queried? What is the optimal interface for these imagined mobile devices? What sort of questions can be asked of such as system?

ProxStor leverages the power of an emerging class of NoSQL databases knows as Graph Relational Databases to scale up and out to support the envisioned billions of data points 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 does not impact the feasibility assessment, but should be noted before using ProxStor in a real-world deployment.

1.3 User Stories

To aid in understanding …

These stories introduce…

These stories assume…

1.3.1 Story 1

Which of my closest friends are here (or nearby)?

1.3.2 Restaurant Food Review

A restaurant Bob only occasionally frequents has recently changed up its menu to his dismay. He opens his favorite food review application to check out what his friends who have recently dined here ordered and what they thought of their choices. Even if the friend didn't leave behind a written review Bob can contact them directly. 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.

The first way was upon entering the restaurant to check-in and therefore to determine Bob's location. This provided the application with physical context and also allowed other users of ProxStor, even independent from the users of the food review application, to know the whereabouts of Bob.

The second way was to answer the proximity aware query "which of Bob's friends has been here recently?" This information was used by the application in turn to reference its specific database of reviews to extract Bob's desired behavior. To extend the power of such a query the application can potentially request ProxStor to return which of the friends of Bob's friends have also been here recently?

1.3.2 Story 3

1.4 CONTRIBUTIONS

The primary contribution of this report is to determine the feasibility of the envisioned system as well as provide a starting point for future work in this area. The web-facing application programming interface (API) is thought through and coherent. The work completed in the design as well as internal and external interface to the system is noteworthy contributions as well.

1.5 SCOPE

This paper describes the ProxStor web service including the system design and the application programming interface and how ProxStor remains faithful to its design goal to remain flexible in API for the application designer as well as flexible in the back-end allowing deployment on top of a number of graph relational database offerings.

Cover scope of project…

What it is and what it isn't…

Scope of the report…

Cover the remaining Chapters of the paper….

Chapter 2…

Chapter 3…

Chapter 4…

Chapter 5…

Chapter 6…

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.

2.1 Requirements

To achieve the goals set forth for ProxStor the following functional and non-functional requirements must be met.

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 ("sensible" aspects of a Location)
    - 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 a unique environmental element contained within

- 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 sensors 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

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.

2.2 SPECIFICATIONS

Why ProxStor, including overarching design of flexibility and back-end agnostic design. Insert the diagram here

Simple for client – RESTful, CRUD, JSON

Java – any container

Cloud

Scalable

No-SQL store – but independent of the store

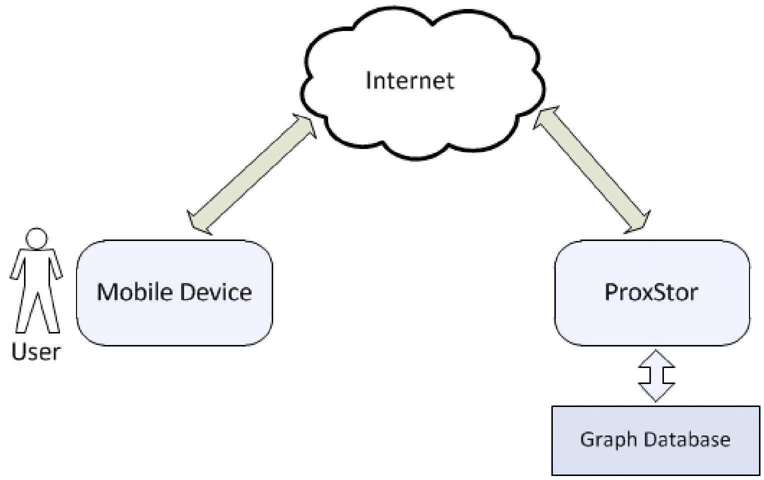

Figure 1: ProxStor configuration

Chapter 3: System Design

This chapter describes the design and implementation of ProxStor – the internals.

ProxStor components in grey:

3.1 Technology Stack

Include blueprints, jersey, other java libs, maven, netbeans, POSTman, winstone…

3.2 ProxStor Design

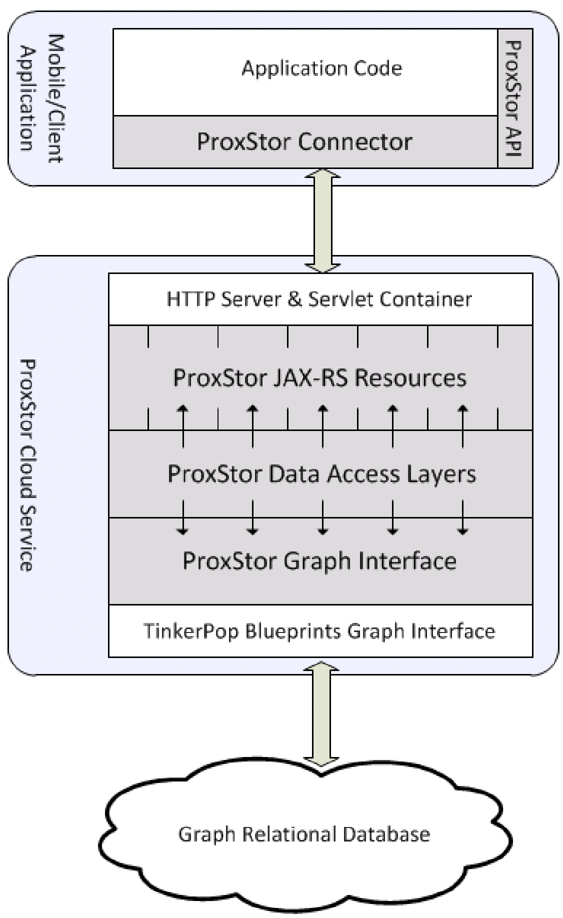

Figure 2: ProxStor Layers

3.2.1 ProxStor API

3.2.1.1 User Object

3.2.1.2 Device Object

3.2.1.3 Location Object

3.2.1.4 Sensor Object

3.2.1.5 Locality Object

3.2.1.6 Query Object

3.2.2 ProxStor Connector

3.2.3 ProxStor JAX-RS Resources

3.2.4 ProxStor Data Access Layers

3.2.5 ProxStor Graph Interface

3.6 GRAPH RELATIONAL DATABASE

Rationale for using graph relational database

Background on NOSQL?

3.6.1 Graph Model

Insert Graph Relational Database Data Models…

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 <*Connector Section*>) 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 [IBM DW]. 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) [] 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 [] or []. This paper does not further elaborate on REST or CRUD.

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 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}:8080/api
```

Figure X: Base URI

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 should be sufficient to describe the operation being attempted.

4.4 Web API

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.

4.5 Fixed Object Web Services Interfaces

The (relatively) fixed components of the Web Interface provide CRUD interfaces for lifecycle management of the following ProxStor object types:

- Users
    - *Knows* relationships between users
    - Devices *owned* by Users
- Locations
    - Locations connected by *Within* relationship
    - Locations connected by *Nearby* relationship

    - Sensors *inside* Locations
- Locality object representing a period of time User was in a Location
- Searching these fixed components
- Administration of ProxStor

Search actions are siloed to a specific object type (User, Devices, ...) 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.

4.5.1 User URI

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

/user

Figure X: 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 X: 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.

4.5.1.2 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.

4.5.1.2 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.

4.5.1.3 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.

4.5.1.4 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.

4.5.2 Knows URI

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

```
/user/{userid}/knows/strength/{s}
```

Figure X: 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.

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	/user/{userid2}	POST		Create know relationship
2	/	GET	Accept: application/json	Retrieve users who userid knows
3	/reverse	GET	Accept: application/json	Retrive users who know userid
4	/user/{userid2}	PUT		Update knows relationship
5	/user/{userid2}	DELETE		Delete knows relationship

Table X: Knows URI Methods

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.

4.5.2.1 Create Knows

To create a new knows relationship between two users the client must send an HTTP POST to the knows + 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).

4.5.2.2 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.

4.5.2.3 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.

4.5.2.4 Update Knows

To update the strength value in an established knows relationship the client must issue an HTTP PUT request to the knows + 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).

4.5.2.5 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 the strength value must be a valid value in the range 1 to 100, but the actual value is ignored in this operation. 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).

4.5.3 Device URI

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

/user/{userId}/device

Figure X: 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 X: 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.

4.5.3.1 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.

4.5.3.2 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.

4.5.3.3 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.

4.5.3.4 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.

4.5.3.5 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.

4.5.4 Location URI

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

/location

Figure X: 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 X: 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.

4.5.4.2 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.

4.5.4.2 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.

4.5.4.3 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.

4.5.4.4 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.

4.5.5 Within URI

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

/location/{locid}/within

Figure X: 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 X: 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.

4.5.5.1 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).

4.5.5.2 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.

4.5.5.3 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.

4.5.2.4 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).

4.5.2.5 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).

4.5.6 Nearby URI

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

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

Figure X: 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 X: 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.

4.5.6.1 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).

4.5.6.2 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.

4.5.6.3 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).

4.5.6.4 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).

4.5.6.5 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).

4.5.7 Sensor URI

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

/location/{locid}/sensor

Figure X: Sensor URI

Note that all sensor related operations are in the context of a specific base location + locId URI, and thus a single specific location. A sensor 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 sensor
2	/	GET	Accept: application-json	Retrieve all location's sensors
3	/{sensorid}	GET	Accept: application/json	Retrieve sensor
4	/{sensorid}	PUT	Content-Type: application/json	Update sensor
5	/{sensorid}	DELETE		Delete sensor

Table X: Sensor URI Methods

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

4.5.7.1 Create Sensor

To create a new sensor the client should prepare a proxstor.API.Sensor object containing the sensor information. Note that at this time the sensorId field is null because the system has not yet assigned an id. This Sensor object is then converted into JSON and sent via a HTTP POST request to the desired location's sensor URI with the header fields *Content-type* and *Accept* both set to *application/json*. If the sensor is successfully added ProxStor will return an HTTP status 201 (Created) with the new sensorId 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 Sensor object and can be directly used in a GET request. If the sensorId 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 Sensor object, including the sensorId, is returned to the client in JSON. This is the method used by the ProxStor Connector.

4.5.7.2 Retrieve Location's Sensors

To retrieve all sensors inside a specific location the client must sent an HTTP GET request to the base sensor URI for the owning location. Do not specify a sensorId 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 sensor 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 Sensor objects. If the location is valid, but contains no sensors, the response status will be 204 (No Content) with no contents in the body.

4.5.7.3 Retrieve Sensor

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

4.5.7.4 Update Sensor

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

4.5.7.5 Delete Sensor

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

4.5.8 Locality URI

A Locality represents the bringing together of a device and a sensor (or a user and a location). The ProxStor system collects these localities through its 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 checkin 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 X: 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 X: 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.

4.5.8.2 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.

4.5.8.2 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.

4.5.8.3 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 (up to a max of 1024).

4.5.8.4 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.

4.5.8.5 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.

4.5.9 Search URI

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

/search

Figure X: 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	/sensors	POST	Content-Type: application/json Accept: application/json	Search sensors

Table X: Search URI Methods

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

4.5.9.1 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.

4.5.10 Administration URI

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

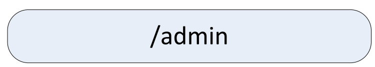

Figure X: Admin URI

#	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 X: Admin URI Methods

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

4.5.10.1 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 [].

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.

4.5.10.2 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).

4.5.10.3 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).

4.6 Dynamic Web Services Interfaces

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

- Device check-in
- User check-in
- Query
    - Fixed Format
    - Complex

4.6.1 Device Check-in URI

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

```
/checkin/device/{devid}/sensor
```

Figure X: Device Check-in URI

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

				(Sensor)
4	/{sensorid}	DELETE		Check-out (Sensor)

Table X: Device Check-in URI Methods

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

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

4.6.1.1 Device Check-in (Environmental)

When a device detects a new environmental element it should report the discovery to the ProxStor service by creating a new proxstor.api.Sensor object and filling in the known data, such as type and identifier. This Sensor object must then 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.

4.6.1.2 Device Check-out (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.Sensor object used to check-in, or optionally the complete sensor object retrieved based on the sensorId from the locality object. ProxStor will respond with an HTTP status of 204 (No Content) if the check-out was successful.

4.6.1.3 Device Check-in (Sensor)

If the client already knows the precise sensorId 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 + sensorId URI. The full URI provides ProxStor with the necessary information to associate a device with a sensor. 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.

4.6.1.4 Device Check-out (Sensor)

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 + sensorId URI. The full URI provides ProxStor with the necessary information to dissociate a device from a sensor. ProxStor will respond with an HTTP status of 204 (No Content) if the check-out was successful.

4.6.2 User Check-in URI

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

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

Figure X: 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 X: 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.

4.6.2.1 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.

4.6.2.2 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.

4.6.2.3 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).

4.6.3 Query: Fixed Format URI

All fixed-format query requests are performed relative to the query URI:

/query

Figure X: Query URI

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

Table X: Query URI Methods

4.6.3.1 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. 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).

4.6.4 Query: Complex URI

TBD

Chapter 5: Results

Software Engineering Metrics

Lessons Learned

What Worked

What Didn't Work

Chapter 6: Conclusion

Relationship to Existing Work [?]

Relationship to Prior Work [?]

Summary

Future Work

Obtaining ProxStor

Pointer to wiki…

Pointer to git repo…

Pointer to javadocs…

References

(citations not ready, complete, nor in order)

[ ] Blueprints. https://github.com/tinkerpop/blueprints/wiki

[ ] Tinkerpop. http://www.tinkerpop.com/

[ ] RESTful Web services: The basics.
http://www.ibm.com/developerworks/library/ws-restful/index.html

[ ] Jersey – RESTful Web Services in Java.
https://jersey.java.net/

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

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

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

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

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

[ ] mongoDB
http://www.mongodb.org/

[ ] Neo4j
http://neo4j.com/

[ ] Ian Robinson , Jim Webber , Emil Eifrem, Graph Databases, O'Reilly Media, Inc., 2013

[ ] ProxStor Wiki

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

[ ] ProxStor Source Code

https://github.com/jgiannoules/proxstor

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

[ ] Apache Maven

http://maven.apache.org

[ ] ArangoDB

https://www.arangodb.com/