The Crosscloud Project and Decentralized Web Applications

by

Martin Martinez Rivera

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author .....
Department of Electrical Engineering and Computer Science
August 18, 2015

Certified by .....
Tim Berners-Lee
Professor
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

The Crosscloud Project and Decentralized Web Applications

by

Martin Martinez Rivera

Submitted to the Department of Electrical Engineering and Computer Science
on August 18, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Crosscloud is a project spearheaded by the Decentralized Information Group, aiming to create a platform that enables developers to build decentralized Web applications in which users have total control over their data. This thesis presents the results of my work: *gojsonld*, a library for the JSON-LD linked data format written in the Go programming language, and *ldnode*, a server developed on the Node.js framework which acts as a back-end for Web applications.

gojsonld allows Crosscloud applications written in Go to use the JSON-LD format, which is based on the popular JSON format and is widely used in the design of Web application APIs. *ldnode* allows applications to create, read, and write linked data resources and implements decentralized authentication and access control.

Thesis Supervisor: Tim Berners-Lee
Title: Professor

Acknowledgments

It has almost been five years since I landed on Boston with two full suitcases and a barely understandable English. The journey was long, but like all journeys it must come to an end.

I want to thank Tim Berners-Lee, Sandro Hawke, Andrei Sambra, and Nicola Greco, as well as all other members of the Distributed Information Group, for their collaboration, ideas, and support during the completion of this thesis. To my family and friends for their company and love throughout all this years. Last but not least, to Antonio Argüelles, for without his selfless support and friendship these five years at the Institute would be nothing but the wildest dream of mine.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

Chapter 1

Introduction

Over the past two decades, Web applications have become ubiquitous and their influence has spread over a numerous professional fields and aspects of our personal life. The specifications produced by the World Wide Web Consortium for HTML[10] and HTTP[11], the original building blocks of the Web, allowed sharing and modifying static documents over a network. However, the need for more dynamic and interactive applications gave rise to various technologies, some of which are still dominant today, that allowed the World Wide Web to become not only a platform to share information, but a platform to develop applications promising interoperability and ease of use.

As the Web applications matured and became more complex, so did the data shared on them. From address books to entire vacation albums, the Web is now home to a lot of private information. Most Web applications store data in their backend systems, effectively removing control from the user. While some of the motivations to centralize user data are innocuous (ease of development and maintenance), others are less so: centralizing user's data and removing control can serve as effective ways to lock users to a particular solution and discourage them from trying other, possibly better alternatives.

Crosscloud[3] is a project started at the Decentralized Information Group, to provide an alternative solution to build Web applications that respect user's data and choices. It does so by decentralizing the Web applications built on top of the Crosscloud platform. Users are in control of their data and they can choose to store it on a machine of their own or on a dedicated provider (called from now on *pods*).

The contributions of this thesis are gojsonld[7], the implementation of a JSON-LD[15] library for the Go programming language that will allow the Go implementation of the Linked Data Platform to conform to the specification, and ldnode[18], a version of the Linked Data Platform in the Node.js framework that expands the early work done by Professor Berners-Lee and adds additional features such as decentralized authentication and access control. The rest of this thesis is divided as follows: Chapter 2 offers a more detailed explanation of Crosscloud's aims and benefits as well as of previous research and supporting technologies. Chapter 3 details the development of a JSON-LD library for the Go programming language that is part of the work done for the thesis. Chapter 4 details the implementation of ldnode, an LDP[20] server built on the Node.js framework. Finally, chapter 5 provides a short summary of the work achieved for this thesis.

Chapter 2

Background

2.1 The Semantic Web and Linked Data

The World Wide Web has given access to an enormous amount of data, ranging from scientific data to movie databases. However, not all of this data are presented in a format that allows it to be easily understood, parsed, and linked with other pieces of related data. The World Wide Web Consortium[1] is helping develop the Semantic Web [28], a stack of technologies that enables computers to do more useful work and support trusted interactions over the network. The vision of the Semantic Web is to allow people to store and publish data in a format that is searchable and linkable by machines and lets users build complex applications over this vast array of data.

One of the basic technologies that allow the Semantic Web to exist is Linked Data. Linked Data follows a similar design to that of the basic World Wide Web and a similar set of expectations [37]:

1. Use URIs as names for things

2. Use HTTP URIs so that people can look up those names.

[1] http://www.w3.org/

3. When someone looks up a URI, provide useful information, using the standards (RDF[24], SPARQL[30])

4. Include links to other URIs so that they can discover more things.

However, unlike the normal Web, Linked Data does not necessarily link to other HTML pages but it allows to link to any resource that can be represented on the Web: relationships between people and objects, song lyrics, metadata about works of art, etc. The flexibility of Linked Data as well as its openness makes it a great candidate to store data that needs to be shared among multiple computers. Crosscloud uses Linked Data to store and read user data.

RDF (Resource Description Framework) is a standard model for data interchange in the Web. RDF it is used in a variety of circumstances and it is specially suited to represent Linked Data. RDF can be serialized in multiple formats: N3[23], N-Quads[25], Turtle[32], and JSON-LD. The last of these formats is based on the popular JSON[14] format and it is one of the focuses of this thesis. Providing support for JSON-LD reduces the complexity of developing applications by providing a native and simple format that is common to both the server back-end and the browser front-end.

2.2 Crosscloud

Most of today's Web applications follow a centralized approach in which the logic and data of the application are tightly coupled and stored in machines controlled by the application provider. This style of development has many advantages, such as a decrease in the complexity in the code and a reduction of latency and waiting times. However, these advantages come at a cost that is often paid by the users.

- Application providers can lock users into their product by making it hard or impossible to migrate to competing alternatives. For example, moving away

from Facebook means losing all contacts, statuses, etc. The application provider can resort to less honorable practices. Apple has been discovered dropping text messages of users that move away from the company's mobile product line in an effort to discourage users from ditching their mobile phones [38].

- Data is locked and cannot be linked to other relevant data stored in a different website or service. Using centralized applications often results in data that is stored in proprietary or Web-unfriendly formats, reducing the usefulness of the data.

- Lack of interoperability. Users from one application cannot interact with users of other one, resulting in a number of walled gardens which no one can leave.

- Perhaps more importantly, users have no control over their data and how it is used. Many application providers have a privacy policy but it is hard for users to leverage their power and enforce said policies when vast amounts of their data are not under their control.

Crosscloud's main objective consists in providing application developers with a platform and a series of development tools that enables them to create decentralized applications. Crosscloud applications achieve decentralization by handling control of data back to the users. Some advantages that could be obtained from developing a Web application on Crosscloud are:

- **Innovation**: Reduced barriers to entry. Giving that the user controls its own data and that the data is stored in open formats, it's not hard to try alternative applications. Lowering the barriers of entry in turn encourages competition and benefits the end users.

- **Stability**: Not every change or new feature is well received by all users. Users now have to choice to stay with a particular version of the software as long as

someone (including themselves) is willing to maintain it.

- **Privacy**: While it is impossible to guarantee that user data is not being used maliciously or shared with unauthorized third-parties, users have the choice to move to other applications that share their privacy concerns.

- **Connection**: Giving user control of their data and using open standards encourages developers to provide interoperability between different application, thus reducing the *walled garden* effect observable in centralized applications.

2.3 Linked Data Platform and SoLiD

Web applications are examples of the server-client architecture style, in which multiple clients send requests to one or more servers whose job is to interpret those requests and send an appropriate response. Crosscloud applications follow the same pattern, the difference being how user's data is handled and passed to the user. The Linked Data Platform [20] is a specification of the rules and techniques that should be used to read and write *Linked Data Platform Resources*.

The Linked Data Platform specification leaves the inclusion of many features and the details of their implementation to the best judgment of the programmers following the specification. This can quickly lead to compatibility problems. For example, a programmer can decide that resources and containers will be stored as objects inside a database while another will store containers as directories and resources as files. If a user decides to change from the first implementation to the second, they will have to convert their data into the other representation.

For this reason, a separate set of conventions to build social applications has been developed under the name SoLiD, which stands for Social Linked Data Platform[29]. These conventions have three main features in mind:

1. Servers are application agnostic, enabling applications to be deployed on different servers (which might be written in different programming languages and/or runtimes) without the need for modification.

2. Works with the REST [19] protocol, which is widely used and supported by most browsers and server-side programming languages. This provides solid applications with a common language that can be composed to create more powerful applications.

3. RDF as data-model: Containers and objects are represented as RDF, which can be serialized in different formats for maximum compatibility.

2.4 Implementations

Multiple implementations of the Social Linked Data Platform are under current development. Gold[8] is an implementation written in the Go programming language, a language developed by Google that provides libraries and features that make it a good choice to develop networked and distributed applications. Gold provides support for most of the basic features detailed in the Linked Data Platform specification.

A feature lacking from gold was the lack of support for the JSON-LD format. LDP applications need to respond to requests that include a "Accept" header set to "application/ld+json". Such a header tells the server that the response should be converted to the JSON-LD format. At the time I started working on the thesis, the Go language did not have any library supporting said format. The specifics of adding support for JSON-LD are discussed in the following chapter.

2.5 Applications

A few MIT undergraduates and other developers have begun writing Crosscloud applications to demonstrate the possibilities of the new platform.:

- Warp: A LDP resource browser that allows reading and manipulating resources from a server via a Web interface developed by Andrei Sambra. It features WebID authentication, file upload, and access control management.
- Cimba: A micro-blogging application similar to Twitter and developed by Andrei Sambra. It features WebID authentication, LDP compliance, private and public posts, and multiple channels.
- webid.im: An instant messaging client written by Melvin Carvalho that provides WebID authentication, live updates, and patches via SPARQL-UPDATE.
- Schedule: An application designed to create polls and set time for schedules by using consensus written by Professor Tim Berners-Lee. It provides WebID authentication, and patches.

Chapter 3

JSON-LD support

3.1 JSON

JSON (standing for JavaScript Object Notation) was originally developed to serve as a data exchange format for the JavaScript Language. As the popularity of JavaScript increased, JSON became widely used. Nowadays most languages and frameworks include JSON support in their standard libraries or have mature third party libraries. JSON is also a lightweight format that is easy to read and translates directly or with very little effort into the native data structures of most programming languages (There exist some limitations. For example, JSON only supports floating point numbers, but numbers of other types are supported by serializing them into a string and setting the type parameter to one of the appropriate parameters[2]). For example, in Go we can decode an arbitrary JSON document in two lines of code:

```
var f interface{}
err := json.Unmarshal(b, &f)
```

The first line declares an interface (the equivalent of an Object instance in Java, for example) and the second decodes the JSON document b and saves the result in the interface f.

The above reasons makes it desirable to be able to express Linked Data in the JSON format. Fortunately, such a standard has already been developed.

3.2 JSON-LD

JSON-LD is a lightweight syntax format to serialize Linked Data in JSON. JSON-LD aims to develop a format that satisfies the following goals:

- **Simplicity**: Processing JSON-LD documents requires no extra libraries apart from the ones needed to process normal JSON documents.
- **Compatibility**: A JSON-LD document is always a valid JSON document.
- **Expresiveness**: The syntax serializes directed graphs and ensures every data model can be expressed as a JSON-LD document.
- **Terseness**: The syntax is concise, short, and readable, requiring little effort from the developer.
- **Zero edits**: Because JSON-LD is compatible with JSON and its libraries, existing REST interfaces without having to edit the files themselves. A JSON API can be converted into a JSON-LD API by adding one line to the HTTP headers[13].
- **Usable as RDF**: JSON-LD can be used as idiomatic JSON by developers without previous knowledge of RDF.However, it can be used as RDF just like other formats (N3, Turtle, etc.).

In order to support serializing Linked Data as a JSON document, JSON-LD introduces multiple changes and concepts to vanilla JSON.Most of these concepts are expressed as keywords, reserved strings that have a special meaning when a JSON

document is interpreted using the JSON-LD syntax. Some of the most important are:

- **@id**: Used to uniquely identify objects in the documents. There are two ways to specify an id. The first one is by providing a IRI (International Resource Identifier), which is similar to an URI but supports extra characters. The second is by providing a blank node identifier, which is a string starting with the characters "_:" (without quotes) and identifies nodes in the graph for which a IRI is not provided.
- **@context**: Because JSON-LD serializes graphs whose nodes and edges can be named using arbitrarily long IRIs, it is a good idea to provide short-hand names for some or all of these. The @context keyword specifies a mapping of short-name to IRI.Using a context increases the readability of JSON-LD documents.
- **@value**: Specifies the data associated with a node in the graph.
- **@language**: Specifies the language of a particular string or the default language of the document.
- **@type**: Specifies the data type of a node or typed value.
- **@graph**: A JSON-LD document can contain multiple graphs, not necessarily connected. This keyword specifies a named graph. The default graph is specified using the @default keyword. Graphs are converted into native Go objects without the need to use N3 or any other intermediate format by the serialization API described later in this document.

3.3 JSON-LD Processing API

The JSON-LD processing API [16] specifies a number of algorithms to handle JSON-LD documents. Some of this algorithms exist with the purpose of transforming the

document to a form that is either easier to process by a computer or easier to read and understand by humans. Others deal with transforming the document from and to other Linked Data formats. The following subsections detail these algorithms and their implementation in the Go programming language.

3.3.1 Expansion Algorithm

The use of a context and relative IRIs can increase the readability of the document by shortening node identifiers. However, some of the strategies used to express the document more tersely can negatively affect the performance. For example, using a keyword to shorten a set of IRIs requires the resulting identifiers to be translated into their original form every time they are accessed. Avoiding such pitfalls is the purpose of the Expansion operation.

As an example, the following document:

```
{
  "@context": {
    "t1": "http://example.com/t1",
    "t2": "http://example.com/t2",
    "term1": "http://example.com/term1",
    "term2": "http://example.com/term2",
    "term3": "http://example.com/term3",
    "term4": "http://example.com/term4",
    "term5": "http://example.com/term5"
  },
  "@id": "http://example.com/id1",
  "@type": "t1",
  "term1": "v1",
  "term2": {"@value": "v2", "@type": "t2"},
  "term3": {"@value": "v3", "@language": "en"},
  "term4": 4,
  "term5": [50, 51]
```

```
}
```

gets transformed into:

```
[{
  "@id": "http://example.com/id1",
  "@type": ["http://example.com/t1"],
  "http://example.com/term1": [{"@value": "v1"}],
  "http://example.com/term2": [{"@value": "v2",
      "@type": "http://example.com/t2"}],
  "http://example.com/term3": [{"@value": "v3", "@language": "en"}],
  "http://example.com/term4": [{"@value": 4}],
  "http://example.com/term5": [{"@value": 50}, {"@value": 51}]
}]
```

3.3.2 Compaction Algorithm

The compaction operation inverses the expansion. Given a context, the algorithm will compact the values and the IRIs that match the keys inside the context, as well as adding a *@context* key to the top level of the document. The output in the example in the previous section would be converted into the output if the same context was provided.

Implementing the compaction and expansion algorithm in Go was straightforward. There were some steps missing from the spec that caused some of the tests to fail, but fortunately other implementations ran into the same errors and provided a working solution. I have notified the JSON-LD working group in the hopes that the inconsistencies between the spec and the working implementations are fixed in future versions of the API specification[5].

There were a few issues specific to the Go language. For example, many of the steps in the algorithms require to check for null values (the keyword *nil* in Go). Since a lot of the functions can return more than one type, they were written to return

an interface, the most general object type in the language. However, when such a function is called the return value is never null, causing many of the tests to fail. The solution involved using the reflect package, which implements run-time reflection and allows the manipulation of arbitrary objects in real time.

3.3.3 Flattening Algorithm

The flattening algorithm collects all the properties of a node in a single JSON object (key-value map) and labels all blank nodes (nodes that are not identifiable by an IRI or are not a list or value). This results in a document with uniform shape that speeds processing in certain applications.

As an example, the document used an as example in the previous sections results in the flattened document below:

```
[
    {
        "@id": "http://example.com/id1",
        "@type": [
            "http://example.com/t1"
        ],
        "http://example.com/term1": [
            { "@value": "v1"  }
        ],
        "http://example.com/term2": [
            { "@type": "http://example.com/t2",
              "@value": "v2" }
        ],
        "http://example.com/term3": [
            { "@language": "en",
                "@value": "v3" }
        ],
        "http://example.com/term4": [
```

```
            { "@value": 4 }
        ],
        "http://example.com/term5": [
            { "@value": 50 },
            { "@value": 51 }
        ]
    }
]
```

3.3.4 Serialization and Deserialization

While JSON-LD documents are a good fit when sending or receiving Linked Data, they may not be the best representation to manipulate it. Instead, the processing API specifies an algorithm to convert a JSON-LD document into an RDF dataset, as well as another algorithm that executes the inverse process.

An RDF dataset is a collection of graph. Each graph consists of a set of triples. Each triple is made up of a subject, predicate, and object. In visual terms, the subject and object represent nodes in the graph while the predicate represents a directed node from subject to object.

The most simple formats to represent an RDF dataset are N3 and Turtle. N3 is a superset of Turtle with more expressive power. Another format, NTriples, is a subset of Turtle with the same expressive power. The N-Quads language is also used to serialize a graph and is the target format of the JSON-LD serialization API. N3 and Turtle share many similarities but N3 adds features such as the ability of graph to be literals and functional predicates.

The JSON-LD library includes a method to convert an RDF dataset into an N-Quads document as well as a method to do the opposite. As an example, the algorithm to convert JSON-LD documents transforms the document used in previous sections into an RDF dataset represented by the following N-Quads document (Indented lines

are part of the previous line but are displayed this way for lack of space. The lack of a graph parameter indicates that the triples will be added to the default graph):

```
<http://example.com/id1> <http://example.com/term1> "v1" .
<http://example.com/id1> <http://example.com/term2>
    "v2"^^<http://example.com/t2> .
<http://example.com/id1> <http://example.com/term3> "v3"@en .
<http://example.com/id1> <http://example.com/term4>
    "4"^^<http://www.w3.org/2001/XMLSchema#integer> .
<http://example.com/id1> <http://example.com/term5>
    "50"^^<http://www.w3.org/2001/XMLSchema#integer> .
<http://example.com/id1> <http://example.com/term5>
    "51"^^<http://www.w3.org/2001/XMLSchema#integer> .
<http://example.com/id1>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://example.com/t1> .
```

3.3.5 Integrating gojsonld library with gold

Integrating the new library with gold gave rise to a few minor issues, mostly caused by the difference in the internal representation of an RDF graph in gold and gojsonld. Meeting with Andrei Sambra (main maintainer of gold) and writing methods that converted between the two representations fixed all of the issues.

Chapter 4

ldnode

4.1 ldnode

ldnode is the implementation of the Linked Data Platoform and the Social Linked Data Platform specifications on the Node.js platform. It is designed to be compatible with the other implementations. This chapter describes in detail the features, design, and implementation of the server. It also discusses in more details the specifications (LDP and SoLiD) it follows and gives an overview of the features that still need to be implemented to achieve full compliance.

The original implementation of ldnode was written by Tim Berners-Lee. It featured handling of basic HTTP requests, patching of files, and a subscription system. I refactored the code, added support for LDP Basic Containers, used rdflib.js to handle the JSON-LD and N-Quads formats, added support for globbing requests, integrated WebID login and authentication, created a first working version of the access control subsystem, and wrote a basic test suite. Later on, Andrei Sambra and Nicola Greco joined the effort and added more tests, as well as other features and numerous bug fixes. In particular, Nicola worked on adapting the code base to use Node.js native idioms and programming constructs, such as adapting the access control subsystem to

work with Node's preferred asynchronous methods. I also worked, along all the other contributors, on reviewing pull requests, and discussing the design and engineering decisions involving the development of ldnode.

4.2 Linked Data Platform

The Linked Data Platform specification introduces a set of conventions and concepts that provides a standard and general way to handle Linked Data over existing Web standards. The Social Linked Data Platform specification further clarifies and expands on it to provide conventions on which to build social applications. The rest of this section explains some of the concepts introduced by both LDP and SoLiD and how they relate to ldnode, the implementation of these two standards on the Node.js platform.

4.2.1 Linked Data Resources

Resources are the most basic abstraction in LDP. They are HTTP resources that can be modified using HTTP verbs such as GET, PUT, and DELETE. They may be *non-RDF sources* stored in one of the many supported formats, or they can be *non-RDF sources*, such as binary or text files that hold information not presentable as linked data — e.g pictures, images, XML files, etc.

LDP resources are supported in all implementations of SoLiD and are represented as files in the server's file system. By convention, RDF sources are stored as turtle files, one of the many formats in which RDF data sets can be serialized, and are saved to disk with the file extension ".ttl". Non-RDF sources are stored in disk with their usual file extensions and in addition they may have an additional metadata file which is an RDF file holding extra information about the resource that can't be stored in the file itself. The naming convention for metadata files is the concatenation of the

file's name and a metadata extension, whose default value is ".meta". ldnode supports setting a customized value for this extension.

4.2.2 Linked Data Containers

Linked Data Containers are the main tool LDP provides to organize resources into categories that help make better sense of the data (similar to a folder or a directory). For example, Linked Data Resources containing blog posts can be grouped into a blog container, and so on. LDP provides three kind of containers: Basic, Direct, and Indirect Containers.

Basic Containers[1] provide the most basic abstraction, the *contains* relationship, represented inside the container as RDF triples whose subject is URL of the container itself, whose predicate is the *ldp:contains* (where ldp is the LDP namespace), and whose object is the URL of the contained resource. Basic Containers are implemented as directories in the file systems and the files inside that directory belong to the container. The contained triples are auto-generated every time a container receives a GET request, thus greatly simplifying the creation of resources and ensuring the triples never become stale. Containers can also have additional metadata which is stored in the same manner as in the case of a non-RDF source.

Direct[4] and Indirect[12] containers extend the concept of Containers to enable more flexible models. Direct Containers allow the subject to be something other than the container itself or the predicate to be application specific, thus allowing other containers and resources to be referenced by other containers and used to create more complex, flexible models. Indirect Containers are similar to direct containers but they allow to include external resources (such as the profile of a friend stored in their own personal server) to be included as part of the containment triples.

At this moment, gold and ldnode only support Basic Containers, as Direct and Indirect Containers are much harder to implement and Basic Containers cover most

of the needs of the applications that have been developed so far.

4.3 Implementation

4.3.1 Node.js

Node.js[22] is a server-side environment designed to build fast networking applications on JavaScript. Node.js discards the threaded, shared state concurrency of more traditional languages such as Java and C++ and the message passing model of alternatives such as Go and Erlang in favor of a single threaded model that promises to achieve high performance through the use of asynchronous operations and non-blocking I/O.

In theory, Node.js promises to deliver fast response times even in the face of high loads and to help programmers deliver code faster. In practice, Node.js design decisions translate into an environment that facilitates the fast prototyping of networking applications that do not use much computational power. However, as it will be further explained, Node.js concurrency model makes writing code that either uses a lot of computational power or executes a lot of I/O actions difficult. Said code is straightforward to write in languages with different concurrency primitives and though some Node.js libraries alleviate the problem, the solution never feels completely natural.

Perhaps the best features of Node.js are the package manager and the extensive library support. Node's package manager, named *npm*, makes it easy to create, build, and publish applications and libraries while handling package dependencies automatically. At the moment of writing, there were 167,410[36] packages available. Such a great number of libraries make for faster development times and increase modularity.

4.3.2 express

ldnode is built on top of Express [6], a Web framework for Node.js. Express is small, flexible, and does not impose a lot of constraints on the programmer. There are many libraries and middleware that can be be simply plugged in into Express to provide features not available in the core framework. Express allowed ldnode to be efficiently written and extended without much (if any) additional complexity imposed by the framework.

4.3.3 rdflib.js

There exist many libraries that allow working with RDF data across many programming languages and environments. One of them is *rdflib.js* which is published as a node module, although it works on the browser without extra dependencies. rdflib.js [26] is used to handle RDF operations in ldnode and it is one of the core building blocks of ldnode.

rdflib.js includes many features such as querying, fetching remote documents, and supporting reading and writing from and to multiple formats. The JSON-LD and N-Quads formats were not available and were integrated into rdflib as part of the work of this thesis.

Serialization is available via the serialize method, which takes a graph and outputs it as a document of the specified type. Support for JSON-LD and N-Quads already exists in Node via the packages jsonld[17] and n3 [21], and these modules allowed for support to be integrated into rdflib without writing a separate library.

There were, however, some issues integrating the modules into rdflib.js. The first one concerns the portability of the code, for rdflib it meant to be supported by mainstream browsers without additional dependencies, in this case the node runtime system and libraries. This problem was solved by a popular Node module called Browserify. Browserify allows developers to include node's modules into client-side

code (via require statements) by automatically converting the code into a second version that contains every dependency. The outputted files can grow quite large but they allowed rdflib to deploy JSON-LD support without a huge investment of time.

The second issue concerns the asynchronous nature of most of Node's modules, including *jsonld* and *n3*. The original version of *synchronize* only supported synchronous execution. Due to the limitations of Node's concurrency model, it's impossible to combine a synchronous method with an asynchronous one and have the result act in a synchronous manner. There are some libraries that claim to help organize asynchronous code in a more sane and readable manner, but none of these libraries can overcome this limitation because it is embedded too deep in Node's design and philosophy.

I settled for making the synchronize method take an extra parameter, called callback. In the case of formats other than JSON-LD and N-Quads documents, this parameter is optional and if present the callback function will be executed at the end with an error code and the serialized document as arguments instead of returning the serialized document. For JSON-LD and N-Quads documents, this parameter is mandatory. To simplify the asynchronous code needed to get this feature to work, I used node's *async* library. This library provides a set of primitives that make it easier to reason about callbacks and avoid code that is highly nested. Below is an example directly from rdflib that shows an example of how async works. The *waterfall* method takes an array of callback functions and executes one after the other, passing the result of the continuation to the next callback, and executing the error callback (the second argument to the waterfall method) if any of these functions fail.

```
asyncLib.waterfall([
  function(callback) {
    n3Parser.parse(n3String, callback);
  },
```

```
  function(triple, prefix, callback) {
    if (triple !== null) {
      n3Writer.addTriple(triple);
    }
    if (typeof callback === 'function') {
      n3Writer.end(callback);
    }
  },
  function(result, callback) {
    nquadString = result;
    nquadCallback(null, nquadString);
  },
  ], function(err, result) {
    nquadCallback(err, nquadString);
});
```

4.3.4 Server architecture

As stated before, the server is written with the help of the *Express* library, and uses its idioms and libraries. The architecture of an express application follows a simple and predictable pattern.

- An app object contains the whole application and has methods to start the server, as well as setting handler functions to process incoming requests.
- The most recent version of the Express framework allows for the creation of a router, which manages request handlers. It provides similar functionality to the one already in the app object but it allows the code to be more modular and even to be published as libraries that can then be plugged in into an existing application.
- Handlers specify the behavior of the server. A handler is a function with three

arguments: a request object, a response object, and an optional next function. The next function can be called when the function is done to pass control to the next handler. Handlers are plugged into the app or the router by specifying which HTTP method and request paths they handle. For example, the handler below handles all get requests sent to resources whose path begins with the prefix 'blogs'

```
router.get("/blogs/*", function(req, res, next) {
  //Handler logic here
  next();
});
```

4.3.5 Handlers

This sections provides a more thorough description of the Social Linked Data Platform specification of the HTTP verbs and some of the challenges that arose during its implementation on *ldnode*.

HEAD

The HEAD method returns basic information about a resource without serving the resource itself. This information includes the standard HTTP headers as well as two link headers that point to the locations of the metadata and access control files [9]. An example response looks like the following:

```
HTTP/1.1 200 OK
....
Link: <https://example.org/data/.acl>; rel="acl", <https://example.org/
   data/.meta>; rel="describedby"
```

GET

The GET method implements all the behavior of the HEAD method but in addition is required to serve the content of the resource or container.

In the case of Non-source RDF resources, the content will be sent without further modifications, only ensuring that the *Content-Type* header of the response matches the MIME type of the resource. For RDF resources, the content will be converted to the appropriate format, which can be specified by the user by setting the standard *Content-Type* header in the request. If the header is not sent, the value will default to "text/turtle" which represents Turtle files, and since RDF resources are already stored in Turtle the server sends the resource without further processing.

Containers require to be handled different. There are two main tasks that the server executes when handling a container. First, it reads the directory on which the container is stored and gets a list of files. This list of files is used to automatically generate a list of contain triples. Second, the server uses that same list to add metadata about the resources in accordance to the SoLiD spec. The server adds information about the type (file or directory), size, and modified time of the resources and then looks for the presence of metadata files and adds more information about the type of the resource if it is found.

An additional feature implemented by the GET handler is support for *glob* requests. Support for glob requests is not specified in the LDP specification but was added to support applications that require getting many resources at once. Glob requests are of the form '/abc*' where the presence of the star character tells the server to look for all resources whose names start with 'abc' and might have any other character following 'abc'. This example query will match a resource named 'abcd.ttl' but it won't match a resource named 'babc.ttl'. When the server receives a glob requests it finds the matching RDF resources, checks if the client sending the request is allowed to read the resources that are globbed, and if so adds them to the

response. The result is the union of the RDF graphs of all matching resources.

POST

Creation of new containers and resources happens primarily through the POST method. A new resource or container is created by sending a POST request to the parent container. The server differentiates the two types of requests by looking at the Link header. If the Link header contains the value `<http://www.w3.org/ns/ldp#BasicContainer>` it will create a container. Otherwise it will create a resource, though resource creation should be explicitly stated by adding the value `<http://www.w3.org/ns/ldp#Resource>` to the link header.

The user can also specify the name of the new resource or container via the *Slug* header. The server will try to use that name but if the name is already in use by another resource the server will create a new name based on a unique identifier. After that, the server proceeds to create the corresponding file system object (file or directory) and write the content of the container to disk, which is sent in the body of the requests. In the case of containers and RDF source resources which are sent in a format other than Turtle, the handler will look at the converted text, which was generated by a previous handler, and use said text instead. Since containers are represented as directories, the contents of the request are written to the metadata file instead.

Finally, the server sends the response and if the resource or container was created correctly, it sets the Location header to its URL. This step is necessary because the client is not guaranteed that the name will be deterministic.

The implementation of this handler was straightforward but an optimization was implemented to speed up the conversion of data represented in JSON-LD or N-Quads data. This functionality was already implemented by rdflib in the *parse* method. However the parse method reads the input and inserts the corresponding triples into

a graph. If rdflib were being used, this graph would have to be immediately serialized back into Turtle, which is an operation already being done inside the parse function. Avoiding doing this work twice helps the server better handle high traffic loads.

PUT

An alternate way to create resources exists through the use of the PUT method. This method is preferred to create non-source RDF resources, which may have a different MIME type, as the POST method can only handle the creation of containers and source resources. By default, the PUT method will overwrite the resource if the corresponding file already exists.

PATCH

The PATCH method allows a user to modify part of a RDF resource or container without having to overwrite the whole file. A client communicates queries to the server through the use of a SPARQL-UPDATE request. SPARQL allows to query an RDF database in a similar way to which SQL allows clients to query the contents of a traditional relational database. SPARQL-UPDATE is an extension of the original SPARQL language that allows data to be manipulated. An example of PATCH with SPARQL-UPDATE is shown below, where a client updates the title of a container:

```
PREFIX dc:  <http://purl.org/dc/elements/1.1/>
PREFIX dcmitype: <http://purl.org/dc/dcmitype/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

DELETE DATA { <> <http://purl.org/dc/terms/title> "Basic␣container" };
INSERT DATA { <> <http://purl.org/dc/terms/title> "My␣data␣container" }
```

rdflib.js already implements support for SPARQL-UPDATE, which makes the implementation of the PATCH method in ldnode much simple, and mostly focused on ensuring the file system correctly reflects the changes done by the request.

DELETE

The *delete* handler contains the logic to remove objects and containers. This handler makes sure any metadata and access control files are deleted along with the resource.

4.3.6 WebID

LDP servers can be accessed by a variety of agents, which can include users, organizations, groups, third party applications, etc. Agents' identities are stored and verified through the WebID-TLS protocol[34]. A WebID is an HTTP URL that links to a profile document. This profile document must be available in the Turtle format and contains triples describing the identity and properties of the agent or person. Some of these triples might include personal information as well as URL links to the WebIDs of other people (friends, colleagues, etc.). These links can be used by clients reading the WebID to estimate the trustworthiness of the agent, thus creating a web of trust that can not only be used for authentication but for other features such as the rankings of posts on a social Web application.

ldnode uses this profile to provide secure authentication without the need for usernames or passwords. The behavior of the authentication protocol is outlined by the WebID-TLS protocol[35]. WebId-TLS provides secure cryptography support through the popular RSA protocol[27] and clients and servers communicate through the TLS protocol[31]. Both protocols are widely used and supported by all popular browsers and network tools such as curl.

An outline of WebID-TLS is presented below:

1. The client creates a public and a private key. The public key can be shared with others but the client must ensure the security of the private key or else all other parts of the protocol can be compromised by an attacker.

2. The client sends the public key in a POST request to a server capable of being

an identity provider. This identity provider will create a valid X509 Certificate, and send it to the client, who will find the matching private key and save the certificate along with it.

3. After this step, the client's WebID contain triples that describe and verity the public key.

4. When the client tries to authenticate to a server, it will connect to it via a protocol with TLS support.

5. If the server requires authentication for the resource being requested, it will initiate TLS. TLS protocol cannot be explained fully within the scope of this document. The core concept of the protocol is that the client and server will try to exchange a message using their own pair of private and public keys and the other parties key. The token can only be successfully exchanged if both parties have the correct keys, which proves their trust to each other.

ldnode implements authentication through the *node-webid* module. The module had been abandoned and did not work originally, but the bug fix was simple. The module had been written using CoffeeScript, a programming language that compiles into native JavaScript, but Nicola Greco (a Crosscloud project collaborator) rewrote the module into proper JavaScript. The login system uses this module to complete the authentication process. If the authentication is successful, the WebID URL of the client will be saved into a session object and a session id will be sent to the client as a cookie. This cookie allows the client to talk to the server without re-authentication for the validity of the cookie.

4.3.7 Access Control

One of the most important tasks *ldnode* is assigned with is to protect its resources from being access and/or modified by unauthorized agents. Thanks to WebID-TLS,

ldnode can authenticate agents connecting to it, and then use that information when applying access control policies using the Web Access Control protocol [33]. Users can describe access policies to a resource or container by writing to the access control file. Inside this file there is an RDF graph stored in turtle that uses a dedicated vocabulary to describe the policies. A container's access control file can also specify policies for the files contained by it, but it is not required to do so.

There are four access control modes: *Read*, *Write*, *Append*, and *Control*. Read mode allow selected agents to access a resource through the use of the GET method. Append and Write modes are used together to implement access control for the PATCH, PUT, and POST methods. Write mode is also used to allow agents to delete resources. Finally, Control mode allows selected agents to apply all the previous operations in addition of modifying the access control file itself. This mode should only be accessible to the owners of the resource and acts similarly to the root user in Unix-like operating systems.

Web Access Control also allows agents to delegate access to other agents and to give access control policies to groups, which are access through a separate URL that points to a document containing triples that describe all users belonging to the group.

Web Access Control policies contain triples whose subject is the policy to be implemented (example of policies can be: *owner*, *public*, *restricted*, *append-only*, etc). The predicates and objects describe the properties and values of the policy. The most common properties are:

- accessTo: the resources this policy applies to.
- agent/agentClass: the agent or class of agents this policy applies to.
- mode: the list of modes this policy describes.
- origin: an optional property, which if enabled will ensure only requests whose origin header matches the value of the property will be accepted. Useful to

control a request by the name of the client making it.

- defaultForNew: optional property which takes a container by value and if enabled will allow this policy to be applied to resources inside the container which do not have an access control file of their own.

A sample access file is presented below. This file allows anyone to read a the card document but only allow the owner to write.

```
@prefix acl: <http://www.w3.org/ns/auth/acl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#owner>
    a acl:Authorization ;
    acl:accessTo <card.ttl> ;
    acl:agent <https://example.org/user#id> ;
    acl:mode acl:Read, acl:Write .

<#read>
    a acl:Authorization ;
    acl:accessTo <card.ttl> ;
    acl:agentClass foaf:Agent ;
    acl:mode acl:Read .
```

The access control implementation was adopted from the one originally written by Andrei Sambra for *gold*. An overview of the algorithm is presented below:

1. The caller passes a mode and a request object.
2. The path parameter is initialized to the path in disk of the request.
3. The algorithm looks for an access control file. If it does not find it, it skips to step six.

4. The access control file is read and the algorithm looks for a policy that allows the file to be accessed. It does so by first looking at control policies (since if such a policy exists it overrides policies of other modes), and then looking for policies with the specified mode.

5. If a policy is found, the algorithm returns a success value.

6. Otherwise, the algorithm will change the path variable to point to the resource one step up in the file system hierarchy, which is the path of the parent directory.

7. If no policy is found, then the algorithm returns a failure value. The caller is then in charge of rejecting the request with a 403 error code.

There were multiple issues implementing this algorithm. Access control is written as a synchronous algorithm, and uses a lot of blocking operations. Since the Node.js runtime is single threaded, the access control module could easily become a bottleneck during high load. Another issue steaming from the previous one is that when the algorithm checks for group permission it needs to download the group file from the given URL, but if the URL is that of the ldnode server, the server will deadlock since the access control module is waiting for the group file to be downloaded and the file cannot be downloaded until the access control terminates. This issue was fixed by checking the group file's URL and reading the file directly from disk if the URL points to the ldnode server.

4.4 Missing features and future work

There are some features still needed in ldnode for it to be considered a thorough implementation of the LDP and SoLiD specifications. The most important of those features are listed below:

- Live updates: Modern Web applications often require information to be updated constantly. For example, social media applications must immediately notify the user when another user interacts with them. Due to the limitations of HTTP, the client can't receive automatic notifications unless it periodically asks the server or avoids closing the connection, a solution that does not scale well. Websockets provide a way to do efficient two-way communication and is used in the SoLiD specification to implement live updates.

- Login and sign-up API: the SoLiD specification describes a REST API which users can use to obtain information about users as well as creating new ones.

- Account creation: Once a user has signed up, SoLiD delegates the server with the task of creating a workspace (a default set of resources, containers and access control files that make starting an account easier) and a preferences document that stores options and can be used by applications to store their settings.

- WebID-RSA: A different authentication mechanism similar to WebID-TLS. The public key is stored in the user's profile and the client and server exchange a token.

4.5 Testing

ldnode contains a number of tests that are tasked with ensuring the correctness of the implementation and preventing changes in the code from introducing unintended bugs. An original testing suite was originally written by Professor Berners-Lee using curl and other Unix commands, but those tests along with many others have been since reworked using standard Node.js tools. Some of these tools include *mocha* (an extensive and easy to use testing framework), *chai* (an assertion library), and *supertest* (a library that simplifies the testing of HTTP APIs).

These tests have been integrated with Travis[1], a continuous integration service that runs the tests every time a contributor sends a patch to the main git repository. This automates the testing process, provides immediate feedback, and prevents incorrect code from being introduced into the repository. More tests need to be implemented in order to have full coverage of all features and edge cases, but at this time the testing infrastructure works seamlessly and has become an integral part of the development of ldnode.

4.6 Analysis

The Node.js framework comes with many tools and libraries that made it simple to start working on ldnode. For example, npm and its libraries saved much time that would have been spent writing utility libraries. The simplicity of the Express framework provided the right amount of abstraction and utilities to get started and did not impose many constraints on the application unlike other frameworks such as Ruby on Rails that make it harder to implement applications that deviate from those assumptions. However, Node.js was also the cause of many of the difficulties that were encountered while implementing ldnode.

- Lack of compile type guarantees: Node.js uses the V8 JavaScript engine to interpret and run applications. The engine itself provides a great deal of optimization that speed up the execution of the code, but the lack of compilation and a decent type system tricks programmers into a false sense of security on which basis they assume the code runs correctly. While extensive testing remains a good practice regardless of programming language, entire classes of bugs exist in Node.js applications that would otherwise be simply impossible in statically typed languages and languages that prohibit null references.

[1] http://travis-ci.org

- Concurrency model: Node.js promises a way to easily write non-blocking concurrent programs. However, Node.js fails in a fundamental way because the platform delegates all the complexity of writing concurrent applications to the programmer instead of hiding said complexity behind the interpreter and the scheduler. Instead, the programmer must manually instruct the program on how to split the work using asynchronous function calls and callbacks.

- Callbacks: The two most important issues arising from the use of callbacks are linked with each other. First, callbacks break the flow of control and therefore functions that use asynchronous calls do not compose. In other words, said functions cannot be combined with other functions nor with the most basic language constructs like if statements and for loops. For example, consider the following functions that returns a true value if an user should be allowed to read a file and returns false otherwise:

```
function allow(file, callback) {
  fs.readFile(file, function(err, data) {
    if(err) {
      callback(err, false);
    } else {
      callback(err, true, data);
    }
  })
}
```

This function cannot be used to check the return value by simply using

```
if(allow(file)) {
  //logic
} else {
  //logic
}
```

which renders most of the language useless and inverts the flow of control to result in a style on which the return value must be checked during the execution of the callback function. The natural consequence of this decision creates programs that are heavily nested and hard to read and reason about. There are multiple libraries that claim to fix the problem but all of them are mere band aids to a problem that should not even exist in the first place. Instead, if Node.js fulfilled its promise of making concurrent programming easy and accessible, it would do so by allowing the programmer to keep writing in a sequential style and handling the management of events and I/O under the hood instead of forcing the programmer to become their own scheduler.

Chapter 5

Conclusions

Over the past twenty-five years, the World Wide Web has democratized the access to information to an extent never seen before in human history. Much of this accomplishment rests on the commitment to open standards, interoperability, and vendor-neutrality. In recent times, the openness of the Web has been threatened by the rise of popular centralized services as well as by privacy and surveillance concerns.

The goal of Crosscloud is to provide an open environment on which to build Web applications that return control and privacy to their users. The work presented on this thesis helped transform Crosscloud's vision into a reality by adding support for new data formats and programming environments in order to fulfill Crosscloud's goals of interoperability, ease of use, and openness. I hope my work becomes a step on the path to the project's success and wide adoption and that the applications built on Crosscloud offer users practical solutions without sacrificing their privacy and personal freedoms.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Basic [container]. http://www.w3.org/TR/ldp/#ldpbc. Accessed: 2015-07-30.

[2] Built-in datatypes. http://www.w3.org/TR/xmlschema-2/#built-in-datatypes. Accessed: 2015-08-15.

[3] Crosscloud. http://crosscloud.org. Accessed: 2015-07-30.

[4] Direct [container]. http://www.w3.org/TR/ldp/#ldpdc. Accessed: 2015-07-30.

[5] Duplicate and shared list nodes in rdf to json-ld serialization. https://lists.w3.org/Archives/Public/public-linked-json/2014Nov/0022.html. Accessed: 2015-08-15.

[6] Express. http://expressjs.com. Accessed: 2015-07-30.

[7] gojsonld. https://github.com/linkeddata/gojsonld. Accessed: 2015-07-30.

[8] gold. https://github.com/linkeddata/gold. Accessed: 2015-07-30.

[9] The HEAD method. https://github.com/linkeddata/SoLiD#the-head-method. Accessed: 2015-07-30.

[10] HTML, The Web's Core Language. http://www.w3.org/html/. Accessed: 2015-07-30.

[11] Hypertext Transfer Protocol – HTTP/1.1. https://tools.ietf.org/html/rfc2616. Accessed: 2015-07-30.

[12] Indirect [container]. http://www.w3.org/TR/ldp/#ldpic. Accessed: 2015-07-30.

[13] Interpreting json as json-ld. http://www.w3.org/TR/json-ld/#interpreting-json-as-json-ld. Accessed: 2015-08-15.

[14] JSON. http://www.json.org. Accessed: 2014-12-11.

[15] JSON-LD 1.0. http://www.w3.org/TR/json-ld/. Accessed: 2015-07-30.

[16] JSON-LD processing API. http://www.json.org. Accessed: 2014-12-11.

[17] jsonld. https://www.npmjs.com/package/jsonld. Accessed: 2015-07-30.

[18] ldnode. https://github.com/linkeddata/ldnode. Accessed: 2015-07-30.

[19] Learn REST: A Tutorial. http://rest.elkstein.org/. Accessed: 2015-07-30.

[20] Linked Data Platform. http://www.w3.org/TR/ldp/. Accessed: 2014-12-10.

[21] n3. https://www.npmjs.com/package/n3. Accessed: 2015-07-30.

[22] Node.js. https://nodejs.org/. Accessed: 2015-07-30.

[23] Notation3 (N3): A readable RDF syntax. http://www.w3.org/TeamSubmission/n3/. Accessed: 2015-07-30.

[24] RDF. http://www.w3.org/RDF. Accessed: 2014-12-10.

[25] RDF 1.1 N-Quads. http://www.w3.org/TR/n-quads/. Accessed: 2015-07-30.

[26] rdflib. https://www.npmjs.com/package/rdflib. Accessed: 2015-07-30.

[27] RSA Cryptography Specifications Version 2.1. https://tools.ietf.org/html/rfc3447. Accessed: 2015-07-30.

[28] Semantic Web. http://www.w3.org/standards/semanticweb/. Accessed: 2014-12-10.

[29] Social Linked Data Platform. https://github.com/linkeddata/SoLiD#quick-intro. Accessed: 2015-07-21.

[30] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/. Accessed: 2015-07-30.

[31] The TLS Protocol Version 2.1. https://tools.ietf.org/html/rfc5246. Accessed: 2015-07-30.

[32] Turtle - Terse RDF Triple Language. http://www.w3.org/TeamSubmission/turtle/. Accessed: 2015-07-30.

[33] WebAccessControl. http://www.w3.org/wiki/WebAccessControl. Accessed: 2015-07-30.

[34] WebID. http://www.w3.org/wiki/WebID. Accessed: 2015-07-30.

[35] WebID TLS. http://www.w3.org/2005/Incubator/webid/spec/tls/. Accessed: 2015-07-30.

[36] What is npm? https://docs.npmjs.com/getting-started/what-is-npm. Accessed: 2015-07-30.

[37] Tim Berners-Lee. Linked Data. http://www.w3.org/DesignIssues/LinkedData.html. Accessed: 2014-12-10.

[38] Tom Huddleston, Jr. Apple sued over vanishing text messages. http://fortune.com/2014/11/11/apple-lawsuit-android-text-messages/. Accessed: 2015-07-30.