FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Blockchain Based RDF Management
Student:	Remy Rojas
Supervisor:	Ing. Milan Dojčinovski
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

Recent efforts in the Semantic Web community have been primarily focused on developing technical infrastructure and technologies for efficient knowledge provisioning. Nevertheless, there is still lack of solutions for efficient tracking of changes and provenance in RDF datasets. In the past few years, the Blockchain technology has gained significant popularity as decentralized and secure public data ledger. The ultimate goal of the thesis is to investigate possible use cases and apply the blockchain technology on RDF.
Guidelines:
- Analyze and get familiar with the blockchain technology.
- Investigate possible use cases for the blockchain technology on RDF. In particular, manipulation, tracking changes and provenance in RDF.
- Design a blockchain based solution for manipulation, tracking changes and provenance in RDF.
- Validate and test the solution.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 18, 2018

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

Blockchain based RDF Management

Ing. Remy Rojas Delay

Supervisor: Ing. Milan Dojchinovski

26th June 2018

Acknowledgements

I would like to thank my supervisor, Milan Dojchinovski, for the support, faith, and meaningful discussions that led to the conception and realization of this thesis. I would also like to thank my parents and fiancée for the continuous understanding and encouragement throughout the duration of the work.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 26th June 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Rojas Delay, Remy. *Blockchain based RDF Management.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

S růstem popularity Otevřených strukturovaných dat, který je možné pozorovat například na velikosti sítě Linked Open Data (LOD), je nutné řešit problémy škálovatelnosti a řízení životního cyklu. V době vzniku této práce neexistuje žádná z metod sledování změn a původu která zároveň garantuje integritu a dostupnost dat. Tyto problémy ohrožují stabilitu systému s propojenými zdroji z různých domén sestavených z křížových referencí URI, jakou je například model Sémantického webu: RDF.jené s decentralizaci. V této práci prozkoumáme výhody a schopnosti řešení založeného na Blockchainu. Poskytneme design, implementaci, test a vyhodnocení prototypu Distributed Ledgeru který řeší operace vytvoření, čtení, úpravy, smazání (CRUD), oznámení o propojení dat, a Publish/Subscribe Observer vzor. Naše řešení poskytuje podporu pro sledování a původ verzovanych RDF tvrzení stranám, které si vzájemně nedůvěřují, za použití integrity a dostupnosti spojené s decentralizaci.

Klíčová slova Distribuovaná kniha, Sémantický Web, Blockchain, RDF, Inteligentní Smlouva, Corda

Abstract

As Structured open data sees a growth in popularity evidenced by the size of networks such as the Linked Open Data LOD cloud, aspects of its lifecycle management and scalability have yet to be adressed. At the time of writing, implementations of change tracking and provenance do not guarantee integrity and availability, and depend upon individual domain owners to be deployed and maintained. This represents a threat to the stability of a system in which data is composed of cross-domain URI references such as the Semantic Web's de-facto model: RDF. In this paper we explore the advantages and capabilities a solution based on Blockchain can provide when used as a support for RDF. We provide the design, implementation, testing, and evaluation of a Proof of Concept Distributed Ledger which addresses the use-cases of Create, Read, Update, Delete (CRUD) operations, Linked Data Notifications, and Publish/Subscribe Observer pattern. Our solution provides mutually distrusting parties a support for traceability and provenance of versioned RDF statements, leveraging integrity and availability with decentralization.

Keywords Distributed Ledger, Semantic Web, Blockchain, RDF, Smart Contract, Corda

Contents

List of Figures

List of Tables

Introduction

The efforts to develop infrastructure and technologies for efficient knowledge provisioning and querying have allowed the Semantic Web, an extension of the modern Web, to find increasing adoption. This is evidenced by the ever-growing cloud of Linked Open Data [1].

Data in the Semantic Web carries richer information by itself than other representation models or databases. The reason being meaning is not derived from subjective application-specific interpretation, but a dereferenceable globally defined context and structure. The Semantic Web is a global, structured representation of data entities with universal scope. The ultimate goal being to empower its users with the ability to describe anything and everything under one connected structural model, in a format designed for both human and machine processing.

Naturally this endeavor is not short of challenges as it represents a fundamental leap from the traditional Web concept. Unfortunately the trade-offs of such a change are not present or bear little significance in the modern document-based Web. Amongst the major ones lies the coupling of cross-domain data - the fact Semantic Web data is to be defined by resources originating from foreign domains, which an agent must dereference but has no control over. This raises questions about how will integrity and availability, properties often found rather in databases, be ensured in such a system. A proposed solution is the use of Change Tracking and Provenance as means to provide integrity. This approach takes into account the evolving nature of Web resources, and provides accountability for these changes. Several implementations exist today [1–4]. Unfortunately, they all share the problem of relying on the publisher's good will to implement and maintain transparency, in addition to expect the consumer to understand and adapt to each specific mechanism. As of today, the responsibility for stable, transparent and reliable data sources lies in each owner/maintainer of data, which is not a scalable

[1] `http://lod-cloud.net/#history`

solution. It becomes clear that the successful Semantic Web requires a built-in mechanism across all domains that provides change tracking, provenance, and guarantees availability.

Our response to this problem is not a redesign of the Semantic Web, but rather an alternative to its underlying management through the use of Blockchain.

The concept of Blockchain was born along with its most famous use-case, Crypto-currencies, and has drawn a lot of attention from the general public due to its revolutionary approach to finance. Perhaps more interestingly, it has sparked interest in the development of distributed systems capable of guaranteeing integrity and reliable availability of records. A Blockchain can be defined as a set of data *blocks*, each linked to the previous one forming a *chain*, where new blocks can only be added, and no block can be removed nor modified. A set of distributed entities called *Ledgers* keep track of the chain and agree upon which block gets added next.

The objective of this thesis is to leverage the management of the Resource Description Framework RDF, Semantic Web's de-facto data model, with Blockchain-backed technology to provide a solution whereby users consume, publish, and augment data that is enhanced with change tracking and provenance.

Chapter 1 explores both background and related works. In Section 1.1 we provide a more in depth view of Semantic Web, Blockchain, and the efforts towards bringing change tracking to RDF. Section 1.2 describes related work involving Blockchain-supported RDF. Chapter 2 describes the solution at hand, decomposed in conceptual overview, use cases, and a usage guide. Chapter 3 presents the assessment of our work, both through conceptual comparison with existing solutions in Section 3.1, experimental storage scalability in Section 3.2, and implementation testing in Section 3.3. Finally, we discuss the results achieved and future works in Chapter 3.3.

Chapter 1

Background and Related Work

1.1 Background

1.1.1 The Semantic Web

The next step in the evolution of the Web as imagined by its original creator Sir Tim Berners-Lee [5] consists of enhancing the representation of information as structured knowledge, so that interlinked documents can bear meaning to both humans and machines. The proposed mechanism to accomplish this relies on identifying and organizing entities with Universal Resource Identifiers (URI) [6]. By doing so, one can explore the meaning of an entity by dereferencing the URIs composing it, hence the term *Semantic Web* [7].

One of Semantic Web's most relevant design features is enabling applications to reason over knowledge and meaning so that the information returned can be enriched by any inference the program was able to perform. Currently, the vast majority of websites run under the form of Web 2.0 [2] applications, which in great part deliver content intended for human consumption. Alternatively, Web applications can be hard-coded to process, gather, produce, or provide information, sometimes annotated with meta-data [8] through Application Programming interfaces (API) for machine consumption. An arbitrary application in Web 2.0 might be able to retrieve the current temperature of Prague not because it knows what the concepts of Prague, Time, and Temperature correspond to, nor the way they are related to one another. It only knows the result of a specific request corresponds to the number obtained when querying a particular Web service with the parameters *Temperature*, *Prague* and *Current Time* with the correct structure (also specific to the application). In Web 2.0, data is meaningless outside the context of the application accessing it. In the Semantic Web, there exist models for the representation of knowledge in abstract structures. Concepts such as *Prague* and *Temperature*

[2] `http://www.paulgraham.com/web20.html`

can be described in existing specific representations of eg. *City*[3] and *ObservableProperty* [4] respectively, which bear properties of their own that link to other information nodes. A major part of the logic in a Semantic Web Application is embedded into the structure data has been given. The Semantic Web is designed to organize concepts in such a way independently maintained information sources across the Internet can enrich each other.

1.1.2 The Resource Description Framework

The Semantic Web Community has widely accepted the use of Resource Description Framework (RDF) [9], a directed graph model. Datasets written in RDF are composed of *Statements*, typically called *Triples* because of their 3 elementary parts: subject, predicate, and object. An RDF *Subject* can be thought of as a node, which points to an *Object* (another node) through a *Predicate*, the directed link between both. Alternatively, subjects can point to *Literals*, non-node data from which it is impossible to establish predicates - a leaf node in terms of graph representation. Common examples of literals are Text and Numbers.

On top of the RDF set of predefined predicates which provide a basic knowledge structure functionality, there exist more advanced constructs that enable higher level abstractions. RDF constitutes a base for defining new entities, but in order to model groups of similar resources and the relationship between them, the RDF Schema (RDFS) [10] recommendation was created, which is itself defined using RDF primitives. Its main purpose is to build Vocabularies: descriptions of the characteristics of other resources, such as the *Range* and *Domain* of properties (predicates). RDFS extends the capabilities of RDF by providing a hierarchical class model to express information. Subjects and Objects can be assigned a Class through the RDF *Type* predicate. Another important abstraction of RDF is the Web Ontology Language (OWL) [11]. OWL constructs allow developers to extend vocabularies based off of existing ones by inferring parts of the latter. OWL provides a way to express the subtle relationships between concepts. Eg. stating that *Human* is an *equivalentClass* of *Person*, inferring any entity whose type points to *Human* may also bear the same properties defined in the *Person* concept.

1.1.3 Linked Data

RDF in the Semantic Web is typically written following the 4 principles of Linked Data [12]:

- Use URIs as names for Things
- Use HTTP URIs so they can be looked up

[3] `https://pending.schema.org/City`
[4] `https://www.w3.org/2015/spatial/wiki/SOSA_Ontology`

- Provide useful information when dereferencing the URI using standards (eg. RDF)

- Include links to other URIs

The Linked Open Data (LOD) Cloud [5] represents the state of Linked Data currently published with an open format. In order for a dataset to be considered part of the LOD, it must obtain the 5 stars rating [6]:

⋆ Available on the Web with an open license.

⋆⋆ Use of a structured data format (machine readable).

⋆⋆⋆ This format must not be proprietary.

⋆⋆⋆⋆ This format must use open standards from W3C such as RDF or SPARQL.

⋆⋆⋆⋆⋆ It must be linked to other data in order to provide context.

Although the Semantic Web brings an evolved paradigm to describe and share knowledge at a global scale, its underlying support is still based on the stack that powers the modern Web. Consumers and Producers will still face the following problems:

- Ensuring availability, which is more critical than in Web 2.0 where a broken link to a 3rd party domain generally does not impact the integrity of the resource. In the Semantic Web, availability is critical for consumers since the description of entities depends on it. This is especially troublesome taking into account Web pages and services can go off-line without any notice, providing little to no support to handle such eventuality. Efforts addressing this issue exist. Such is the case of [13], where a database of back-links pointing at DBpedia [7] data is periodically updated.

- Bad and good actors are very difficult to distinguish solely from data. Both from the semantic and technical perspective, publishers can disturb data that is linked to their own by deliberate tampering, wrong use of vocabularies, etc. Data integrity needs to be protected from mis-use, but also account for the potential modifications rightful owners might introduce.

[5] `http://lod-cloud.net`
[6] `https://www.w3.org/DesignIssues/LinkedData.html`
[7] `https://dbpedia.org`

1.1.4 SPARQL

So far we described the publication and enrichment of Linked Data. The final use-case is consumption.

The SPARQL Protocol and RDF Query Language (SPARQL) [14] constitutes the most popular way to fetch RDF. SPARQL is a query language with similar utilities offered in NoSQL data models since RDF is comparable to key-value stores, but rather using *subject-predicate-object* triple format. SPARQL also shares some aspects of Relational Database Query languages such as *SELECT, WHERE* and *JOIN* clauses. In fact, all RDF statements can be seen as a table with 3 columns, one for each statement component (subject, predicate, object). But these similarities stop at a low level. In Relational Databases, queries require previous knowledge of the database structure, whereas we are certain all data in a SPARQL query follows the RDF statement format, requiring users to instead have background of the vocabulary structure and properties that can be expected from entities assigned to different classes.

In most SPARQL implementations, a client sends query data to a *SPARQL Endpoint* running on a server where data is stored on. This means the availability of data in SPARQL is closely tied to the server's uptime, and its capacity to respond within a reliable delay. But the more a dataset grows, the more inefficient similar queries become.

SPARQL does not keep track of changes directly either. Consumers will often rely on the data being referenced to be immutable in content or structure for the purpose of integrity, while publishers will modify existing datasets, possibly altering their semantics with every change. Ideally, changes are driven by rightful owners of the data, with a coherent, documented track of changes. Unfortunately there is no inbuilt mechanism in SPARQL to verify this. Hence the need for immutability, change tracking, and provenance in Linked Data.

1.1.5 Tracking Changes in RDF

Linked Data has many potential fields of applications. Consequently, there is no *one-size-fits-all* guideline to maintain and update RDF. Some of the currently in-use solutions include:

- **Physical Snapshots**: The technically simplest solution. Similarly to what is done on hard drives, the maintainer of a dataset periodically stores a full copy of the current data, a Snapshot. Advantages include the ease of implementation and the availability of tools to do so. The main disadvantages are its poor scaling power and granularity. Over time, maintaining versions of increasingly heavy datasets becomes unsustainable. Even worse, not all changes may be reflected by a snapshot, as short-lived statements (with a lifetime shorter than the snapshot interval) may not be recorded at all, while on the other hand creating a snapshot for every change is the worst case scenario in terms of storage

efficiency. When compared to the following alternatives, physical snapshots are the only solution that does not include an inbuilt mechanism for provenance. The *DBpedia* project [1] is an example implementation of this approach.

- **The Web Archive** [8]: an effort in the modern Web which is concerned by the ephemeral nature of web pages and respond by creating an archive for defunct sites, the *Wayback Machine* [9] . It uses Memento [15], a framework to link current and past versions of the same URI through Link headers in the Hypertext Transfer Protocol (HTTP) [2].

- **Reification** [10]: the process by which RDF statements are backed by RDF entities representing each of the components in every aforementioned statement as a resource in their own right. This allows for example to describe additional information about the statement thus opening the door to provenance. On the other hand it also implies a creation of new resources for every change in every statement. It thus has a heavy impact on storage since the records about past statements are kept through their reification. It also takes a toll in the conceptual design of the data, and makes it more difficult to query [16]. The Wikidata knowledge-base exemplifies the use of reification in [3].

- **The PROV Ontology (PROV-O)** [4]: a W3C recommended vocabulary that defines a framework which can be used to describe the aforementioned properties for data since it allows for the modeling of provenance. At its core, it describes 3 main Classes. *Entities:* The things holding value hence the reason of provenance. Entities are sub-classed into *Collections, Bundles,* and *Plan*. *Agents:* The subject that holds responsibility for the existence of an Entity. Agents are sub-classed in *Person, Organization*, and *SoftwareAgent*. *Activities:* An action perpetuated by an Agent on a certain Entity. PROV-O sub-classes feature distinct graph patterns based on the number and type of links with the other two basic Classes. The main disadvantage of this approach lies in the burden of identifying and adapting the data to fit with the model in a consistent manner, requiring additional care into its design. The W3C provides an implementation report on PROV-O where active projects using the ontology are referenced [17].

Although these models all offer a solution to the documentation of change tracking, they unfortunately share the same dependency on the publisher's good will to implement responsible change tracking and guarantee the integrity and availability of historic data records. Thus, if we wish to ensure verifiable

[8] `https://archive.org/about/`
[9] `https://archive.org/web/`
[10] `https://www.w3.org/DesignIssues/Reify.html`

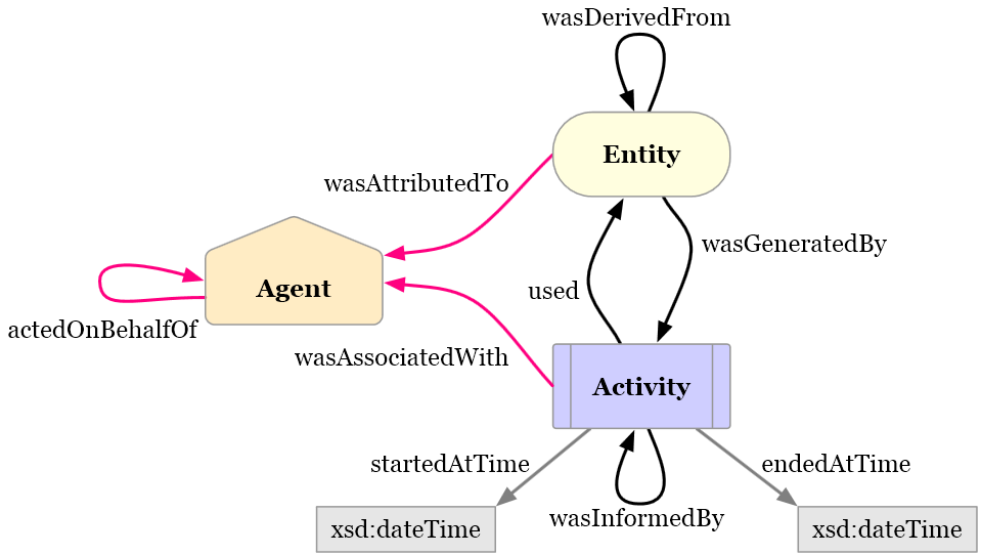


Figure 1.1: PROV-O starting point classes and properties

and immutable historic data, we need to find an alternative to individually maintained data sources. The second part of this section covers Blockchain, a technology that has recently gained a lot of attention for proposing a distributed system that guarantees immutability of asset records across a trust-less network of peers.

1.1.6 Blockchain

Blockchain is a generalized term that was first used to describe the protocol behind Bitcoin [18]. It is made of a list blocks containing data in which new blocks can be added only to the tip of the chain, and no block can be further modified or removed once added. Blockchains are decentralized meaning multiple individuals, often called Ledgers, maintain an up-to-date copy of the chain. A *Consensus Mechanism* helps the Ledgers agree over which block to add next in order to always have all of the copies in sync.

In Bitcoin and other *Cryptocurrencies*, financial transactions are stored in a block, and Ledgers publish new blocks to the network awaiting for consensus. Today the most common decentralized consensus mechanism for Blockchain is known as *Proof of Work* (PoW) [18], which gave Cryptocurrencies the possibility to overcome trust concerns, and thus centralized entities such as banks. *Proof of Work* involves a computationally expensive problem bundled with every block, usually a one-way hash function. In order to append a new block to the chain, its problem's solution (the Proof) must be submitted as well. The solution is made difficult enough so that there exist a non-negligible time between each occurrence of a peer (often called *miner*) finding the solution to

the PoW, which gives the network time to synchronize. The main incentive to develop such a system is to make it very difficult for malicious actors to successfully execute a *Double Spend* attack - where a node executes a transaction twice before the network has time to catch up, or *fork* the chain, ie. create a second competing version with illegitimate transactions. Finally, PoW requires *miners*, users willing to solve the problems presented by new blocks. Readers further interested in this topic may refer to [19].

Nevertheless, scalability of Bitcoin has started to be questioned [20]. The initial vision for Bitcoin, where independent small unrelated nodes would be in control of the PoW progressively vanished with the appearance of *Mining Pools*. These sub-networks of miners acted as a common place to dedicate computing power, effectively centralizing the decision making and thus endangering the sanity of the network. Luckily, the success of Bitcoin allowed this trend to get enough momentum so that other projects also flourished from the pioneering idea, proposing distributed alternatives to Blockchain and PoW: *Directed Acyclic Graphs* (DAGs) eg. [21], and *Proof of Stake* PoS [22] respectively. Even more so, others focused on the potential of such a distributed platform being provided to fulfill an arbitrary number of use-cases. Such is the case of Ethereum [23] and the concept of *Smart Contracts*.

1.1.7 Smart Contracts

Smart Contracts [24] can be thought as digital operations that run autonomously. As opposed to regular contracts, *Smart Contract* properties, conditions, consequences and assets can be represented in a digital format. Ethereum [23] is an application platform distributed over a network of nodes with data structured following the *Blockchain* model. Conceptually, it is a single computer, ie. Turing-complete machine, capable of executing code on a per-request basis. After writing the code blob of a contract, it is added to the chain in the same way a payment is made on Bitcoin, by adding the representative Transaction to a Block. In order to execute a Smart Contract, a new Transaction must be issued with the result of the aforementioned code after execution and a reference to it. This way, other nodes can also execute this transaction and certify its result. In other words, Transactions in Ethereum represent the execution of an aforementioned *Smart Contract*. Any parameters defined by the Contract can be defined in meta-data of the transaction.

Smart Contracts in Ethereum contain the definition for one or more transactions. The system relies on the Blockchain to keep track of executed transaction results, as well as the state of assets that were involved. Finally, the system features its own digital currency, *Ether*, which can be used to purchase *Gas* (a representation of the cost of executing lines of code). Any stateful code execution, ie. that needs to record the change of state of the Blockchain by submitting a transaction is charged a nominal amount of *Gas*. The re-

cipients are, similarly to Bitcoin's miners, individuals who offer the necessary computing resources to allow the aforementioned operations to run.

1.1.8 Corda

Corda [25] is the distributed ledger system of choice for this thesis. The underlying mechanisms powering Corda include Blockchain and a more centralized consensus mechanism based on Public Key Infrastructure PKI through Certificate Authorities [26]. Some of the major conceptual difference between Corda and other Distributed Ledgers is the way Blockchain is used, since in Corda there exist multiple linked lists of transactions. Assets are segregated throughout several chains, each of which involves only a small subset of nodes in the network, eg. those who are interested in the assets contained in a particular chain. Consensus happens at several levels. *Notary* nodes, a service provided by Corda and deployed on the network, help other Nodes approve and order the proposed changes by using consensus mechanisms [27, 28] when necessary. Once all Nodes implicated in an asset agree a proposed change is valid, the Notary service gives the final approval. Assets are represented throughout their *State* - an immutable version of the asset linked to the previous one by a transaction. Transactions are proposals submitted by a node to change the state of an asset. Before a Transaction is validated, nodes must seek the approval of specific peers pointed by the State and a the *Notary* service. Some important differences with other Distributed Ledger Solutions are:

- In Ethereum an arbitrary program can be uploaded and executed immediately once the *Smart Contract*'s transaction is approved. Corda is less dynamic. Nodes can support several simultaneous applications, commonly called *CordApps*, but they must be manually deployed on each node.

- Corda provides services to its nodes such as Notaries, a set of nodes charged of providing consensus for transaction approval, Network Map, which helps nodes discover peers across the network, and a Permissioning service, which automates the provisioning of TLS certificates.

- Application code execution happens only on the necessary nodes. This does not mean nodes are isolated. Nodes can discover other peers through a *Network Map* service, and are able to run procedures called *Flows* that may involve more than one Node to complete. This allows programmers to write protocol logic in both local and distributed scenarios.

- Data is not ubiquitous. Unlike more conventional Blockchain implementations, data is by default stored in the minimum amount of nodes

necessary. Furthermore, each change of a State can only be approved if it is proposed by a node referenced within the State. Corda abstracts this behavior under the concept of *Ownership*.

- There is no Proof of Work. This is due to the reason the commitment of the node in the network is not the same as a miner's. Instead of having a loose, identity-less, zero-cost relationship with the network as is the case of miners, Corda is designed to keep track of long-lasting identities, called *Parties*. One of the purposes of the Notary service is to offload the approval and time-stamping of transactions. Although this service follows the Interledger Crypto-Conditions specification [29], more conventional consensus resolution algorithms such as [27, 28] can be used in cases protocol compliance is at risk.

Corda provides developers with self-sufficient Nodes on top of which one can develop and deploy applications. This gives programmers a lot of flexibility over the inter-node interactions and the distribution of data protocols. In order to develop an application in Corda, one must define three main type of components:

- **States**: The representation of assets in terms of their data structure, including fields to specify which nodes are allowed to modify the data - ownership.

- **Contracts**: The set of rules for a transaction to be validated by a node is called a *Contract*. Nodes must be able to dereference Contracts in order to verify a transaction.

- **Flows**: Corda orchestrates the activities of nodes in a *CordApp* thanks to *Flows*. A flow defines a set of sequential actions to be taken by set of nodes. A *Flow* may trigger several transactions. Flows are meant to be composed into others by a *subFlow* function, and can be specified to run locally or using distributed logic, by specifying *Initiator* and *InitiatedBy* class decorators. Finally, Flows have the possibility to be called by external agents through Remote Procedure Calls RPC.

Recently, Corda has developed a feature to allow any node to follow the evolution of a state without being involved in its modification - an *Observer* pattern [11]. This implies applications can now define flows specifying the behavior of senders and recipients where the latter is not an owner of the data but can still observe its evolution.

[11] `https://www.corda.net/2017/12/observer-nodes/`

1.2 Related Work

Our main focus is to explore the use of a distributed system based on Blockchain in order to service Linked Data in hopes it will mitigate the problems of making and tracking changes which are inherited from the modern Web. More specifically, we intend to address them by using a set of concepts around Distributed Ledgers.

This section discusses efforts to use Blockchain as a support for RDF. Although still in their early stages, it will help us evaluate our own contribution later on. An interesting point of reference is given by [30], where the authors explore several levels of Blockchain adoption, ranging from a common centralized model to one relying fully on a Distributed Ledger. To do so, a number of approaches between both extremes were classified in terms of distribution across different aspects such as verification, guarantee of integrity, and cost of storage and querying. In [31], the authors benchmark previously mentioned approaches in order to support the findings. For instance, according to [30], our approach with Corda falls into the *Base case + Distributed Ledger Backend* category.

1.2.1 Temporal Streaming of Graph Data on Distributed Ledgers

In [32], Third et al. describe a system based on Ethereum Smart Contracts registered in a private instance of the Ethereum Blockchain acting as an intermediary to update a named graph RDF data store, triggered by timestamped events.

RDF data generated is pushed to a Smart Contract which generates a verification hash, along with time stamp meta-data of the event. This information is stored in the state of a new Smart Contract. Original data, along with the address of this new Contract is pushed to a data store. Each event detected is represented in RDF, and stored in a separate Smart Contract. This approach fits into the *Base case + Distributed Ledger* model described in [30], with the difference of having duplicated verification hashes.

The previously mentioned scenario responds to scenarios with highly dynamic data and the necessity of an effective storage solution with no major compromise on integrity.

A second use-case scenario where a Distributed Ledger backed RDF data store can offer a compelling solution involves sensitive *static* data whose integrity is vulnerable on otherwise classical web-servers.

1.2.2 Blockchain Enabled Privacy Audit Logs

In [33], challenges of Log Auditing in RDF are addressed by storing integrity verification data into the Bitcoin Blockchain. In this approach, meta-data

containing non-repudiation and provenance is generated after retrieval of security event log descriptions (annotated in RDF) from a centralized data store. This meta-data is then signed by a certificate authority, and an *integrity proof* is generated out of it. The digest of this integrity proof is stored on the Blockchain and written back into the data store along with the signature, and previously signed meta-data. This allows an auditor to verify whether logs of the events on the system were altered thanks to the integrity proof and its digest, by comparing it to the data present in the Blockchain.

Under both previously mentioned approaches, RDF data storage is supported by Blockchain indirectly through the use of an integrity proof or direct hash of the data. The scenario in 1.2.2 does not leverage the storage of Semantic Data on a Blockchain, in fact, one could argue that data availability is sacrificed by centralizing the storage to a data store, although this can be considered a side-effect of the use-case. In both scenarios, change tracking is supported directly by the underlying Blockchain. But whereas 1.2.1 only has provenance information coming from Ethereum accounts, 1.2.2 can retrieve more precise and trustworthy provenance thanks to the signed meta-data.

1.2.3 A Document-inspired Way for Tracking Changes of RDF Data

The previously mentioned related work can be thought as distributed solutions for storage or at least verification of integrity for RDF with an inbuilt capacity to track changes. Compared to change tracking models discussed in Section 1.1, we notice 1.2.1 authors store ever evolving data through new *Named Graphs* [34] on an ever growing data store. In 1.2.2, the meta-data about event logs is structured with the *Log to Transparency, Accountability and Privacy* L2TAP [35] ontology, which is a similar approach to reification of RDF but data logged refers to privacy events instead.

A different approach applied to RDF by means of a provenance ontology such as PROV-O [4] was developed by Peroni et al. in [36]. Moving away from the Blockchain environment, the authors tackle RDF change tracking with a solution based on differential (incremental) snapshots. Inspired by the problem of handling changes that will potentially be undone in text editors, their work extends *PROV-O* in order to support the conceptualization of this approach. The end result is a vocabulary that can be used to chain together changes on a resource, themselves expressed in the form of a *SPARQL* query composed of *INSERT* and *DELETE* clauses. Each snapshot references both the entity they are attached to and the previous snapshot by the *specializationOf* and *wasDerivedFrom* properties respectively. The following are some advantages we identified in comparison to the previously discussed approaches to change tracking:

- Previous versions of the data can be easily reproduced by inverting the

query in descending order (swapping the contents of *INSERT* and *DELETE*).

- The snapshot is invisible to the data since it is former which references the later. It bears no conceptual burden on the publisher.
- Any provenance metadata supported by *PROV-O* can be part of a snapshot.
- There is a gain in storage efficiency by not repeating overlapping data as in physical snapshotting.
- The representation of changes is atomic.

1.2.4 Summary

The contributions mentioned previously raise the question of what is the best method to support Linked Data through Distributed Ledgers. RDF can be organized in different levels of granularity - statements and named graphs, which can be stored directly or indirectly (through a hashing function) on a Blockchain. On top of this, data can be enriched by provenance information, facilitated by the model of distribution.

In previously discussed works, change tracking and provenance of data is provided either through RDF in Scenarios 1.2.2 and 1.2.3, or indirectly by the underlying Blockchain implementation (the case of Scenario 1.2.1). Although the use of Blockchain is preset, both Scenario 1.2.1 and 1.2.2 use it for the purpose of verification, assuming centralized storage of data, ensuring integrity but sacrificing availability. A hypothetical use of Blockchain as a storage Backend would solve the previously mentioned issue, but it suggests the storage consumption on each node of the network would not be scalable in the context of the global Semantic Web as they would need to store *all* RDF statements, including deleted and previous versions of records depending on the implementation. These implementations do not explore is the possibility to move away from a traditional unique-Blockchain approach - guaranteed redundancy of data across the network, limiting the applicability analysis of using other Distributed Ledger systems [21, 25]. Furthermore, most Blockchain platforms still rely on an economical incentive to publish data, as is the requirement in a *Proof of Work* based protocol. This introduces new roadblocks for the adoption of Semantic Web, should RDF be supported by these platforms at scale.

In the following chapter we describe a solution that takes the previously mentioned scenario leveraged by a different Blockchain platform which lets individual nodes store a fraction of the data available in the network based on their own preferences.

Chapter 2

Blockchain based RDF Management

This thesis addresses the challenges of implementing a distributed, efficient RDF storage model to enable change tracking, provenance, data availability and integrity. In this chapter we explore the design and implementation of such a system, which allows for more efficient manipulation of RDF through the use of Blockchain under a particular implementation. Although we focus on storage capabilities, the word *Management* refers to the spectrum of possible operations that an agent, human or software, can perform on an RDF dataset.

2.1 Overview

The high-level overview of our work consists of two main components: Backend and Frontend Interceptor.

The Frontend Interceptor acts as an access point to users who wish to publish and consume RDF. In addition to its main role, it uses the information available in the Backend to enhance and complement data submitted by issuing requests through a Web Application Programming Interface (API). The purpose of the Backend is to manage the storage of RDF Triples. By communicating with each other, multiple Backends form a distributed Ledger application where Triples are stateful, traceable, and ownable. Finally, Corda [37], the platform powering our application, requires a set of Network Services to be instantiated. These include Network Map, Notaries, and Permissioning services among the most important. Our Backends connect to these services, often running in separate hosts, in order to offload certain procedures that involve network level knowledge and authority. Figure 2.1 illustrates how these components are connected to each-other.

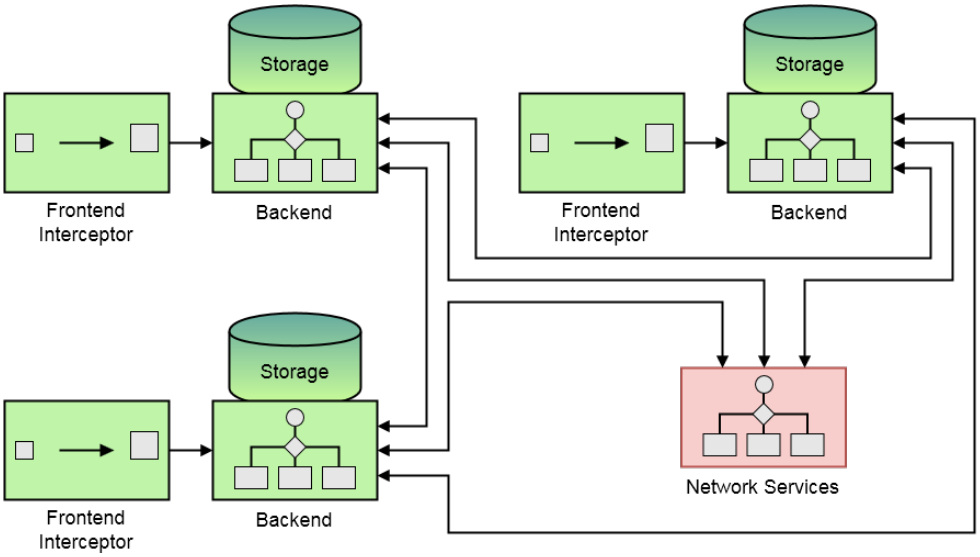


Figure 2.1: Architecture Overview

2.2 Backend

We developed a single application deployed on Corda nodes with the purpose of managing RDF triple storage, defining rules and procedures by which triples can be updated, and providing a Web API for client access.

In order to represent RDF triples in a way that features change tracking and provenance, Corda allows our us to be model data, in this case RDF Statements, in terms of *States* - versioned representations of data. The identifiable agents present on a Backend are called *Parties*. Each State (of an RDF Statement) includes additional meta-data about which Parties are interested in it. As the Triple changes, both in RDF content as in meta-data, these changes are not over-written. Instead, a new State is created and linked the previous one through a *Transaction*, similar to blocks on a Blockchain. Triples under this model are thus presented as follows:

- **Subject**: The Subject component of an RDF Triple.
- **Predicate**: The Predicate component of an RDF Triple.
- **Object**: The Object component of an RDF Triple.
- **Owner**: The Party responsible of the creation of this Triple. Along with participants, the Owner participates in the consensus agreement when an update is proposed. It is worth noting ownership changes are allowed in Corda.
- **Participants**: The set of Parties who also participate in the consensus agreement and have the power to propose updates to a State.

- **Observers**: A set of Parties to which transactions concerning this state are sent. The difference between an Observer and a Participant is the former only observes data without participating in the consensus.
- **Last Editor**: A field denoting the last Party that triggered an update on this state.
- **Linear ID**: A field uniquely identifying the Triple across all its States.

This is meant to directly relate Triples with the agents responsible for their editing, while also offering a simple way to keep track of Parties that are actively interested in the data, but can not participate in its modifications - the Observers.

2.2.1 Flows

A *Flow* [12] is the term used in Corda to denote Classes that handle requests coming from clients or other Peers. Some of them are exposed through a Remote Procedure Call (RPC) [38] interface, or concealed, only to be called by other Flows as a *subFlow* either locally or remotely. The latter case requires the Flow to be divided into two classes, decorated as *Initiating* and *InitiatedBy*.

Through Flows, we encode the protocols for creating, updating, and deleting a State - the representation of an RDF Triple alongside relevant meta-data. In the simplest scenario, a Party (an identified user of a particular Backend in the network) who owns a Triple has the latter's current and past States stored locally. A Party can propose changes to the State of this Triple only when it is part of the *Participants* list or is the Owner itself. In either case, participants and owner are required to *sign* any proposed change, which will then be approved by a Notary service and stored on each node. A State can also contain Observers. These, although referenced in the meta-data, are neither required to sign new changes, nor have the authority to propose them. Still, they receive all updates on the particular Triples they are referred in. Finally, Corda will only store a new State if it is signed by a Notary, part of the Network Services. Notaries sign the proposed change-set once all parties who are required to sign have done so. These mechanisms are illustrated in Figure 2.2, where the owner of a Triple intends to change the State of a Triple through an Update by proposing a new State with different Data. In this example the new State is first verified locally. When approved, it is sent over to any Participant Party in order to obtain its signature, signifying its approval of the changes. Once both Participants and Owner have signed, it is up to the Notary to verify signatures and forward the approved proposal to Owner and Participants, finally, the Owner or other node who initiated the change

[12] `https://docs.corda.net/key-concepts-flows.html`

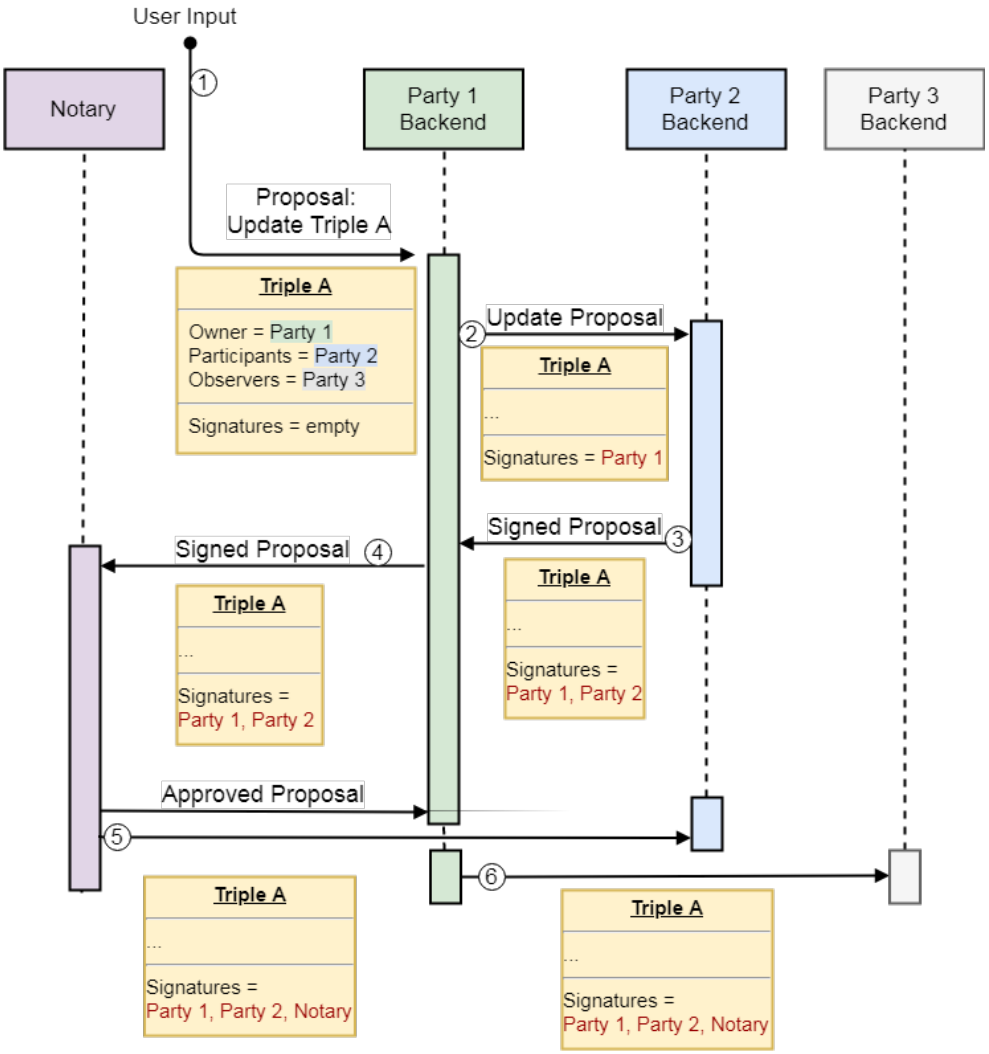


Figure 2.2: Sequence diagram for Triple State Modification

broadcasts this notary-verified state to Observers. The procedure ends with each Party storing the new State of the Triple locally.

This mechanism is used to both modify Triple RDF data, and meta-data. As a consequence, the list of Participants and Owner can only be modified by pre-existing ones. On the other hand, any Party can request to become an *Observer*. When this happens, we choose to register the new Observer to all Triples sharing the same *Subject* RDF resource. This allows Parties to implicitly receive any information concerning the observed Subject thereafter. Hence Observation is done at the Resource level rather than one specific Triple at a time, allowing clients to perform fewer requests while enjoying automatically updated local models of a Resource. Another advantage of Observers is,

unlike Participants, Transactions do not depend on their signatures. In order for a modification to be approved, the Owner and all Participants must sign it, which can lock a Triple if only one Participant is unable or unwilling to cooperate.

The Flows that enable Parties wishing to become Observers include a first phase of *Discovery*, where the requesting Party queries other Backends in order to find a copy of the resource. When successful, the meta-data returned provides the identity of Parties able to propose Transactions. Hence the second phase, *Registration*, by which a Party requests to be added on the Observers list.

The Following Classes represent the set of Flows that govern the protocols by which Triples are managed.

- **CreateTriple**: Creates a Triple based on given RDF parameters and non-mandatory Observers or Participants. The Party responsible for creating this Triple is the owner.
- **UpdateTriple**: Creates a new State based on an existing Triple with any update to the data or meta-data.
- **DeleteTriple**: Ends the Triple life-cycle by making a Blank transaction the next State of a Triple.
- **DiscoveryRequest(Initiating), HandleDiscoveryRequest**: The Initiating Flow takes data of a Triple as parameters and opens a session with every other Backend (triggering in them the InitiatedBy Flow) until a matching Triple State is found.
- **ObserverRequest(Initiating), HandleObserverRequest**: The initiating Party wishes to Observe a Resource. It thus opens a session with a known Participant or Owner of such. The InitiatedBy part of the Flow registers the requester as an Observer on all Triples with the requested resource and returns the number of new Triples to expect from this action.
- **BroadcastToObservers(Initiating), ReceiveUpdateStatus**: Used by *CreateTriple*, *UpdateTriple*, and *DeleteTriple*, handles the selective Broadcast to Observers originating from a new State where they are referenced.

2.2.2 Contracts

Corda applications rely on *Contracts* in order to approve or reject proposed operations [13]. Contracts in Corda are sets of validation checks on the proposed

[13] `https://docs.corda.net/key-concepts-contracts.html`

new State. When a Flow intends to modify a Triple, the operation must reference a Contract so that other Backends can run the validation as well. We implement 3 simple contracts for three operations:

- **Create**: An operation referencing this Contract must not originate from a previously existing State ie. have no input, and return only one Triple State.
- **Update**: One input and output pair. This represents either an update of the RDF data or the attached Party meta-data.
- **Delete**: One input, no output. The end of a Triple life-cycle.

Since Contracts are verified by each individual Backend, their design prevents them from reaching out to external facts and knowledge such as information on other Triples, as such results are likely to change during the time all Backends verify them, and may not be distributed across every Party's local storage. In some cases, one may use a Corda service called *Oracle*, which allows to gather external knowledge to some extent. We did not use such a feature. Contracts are thus restricted to read the information present by input and output States. Any other validation must be performed previously by the originating Flow.

2.2.3 RPC and Web API

A Remote Procedure Call (RPC) module which is able to call Flows and provides a Web API endpoint for external agents serves as a bridge between the Frontend and callable Flows in the Backend. The Web API accepts requests containing JSON formatted RDF Triple data, and complements it with any necessary meta-information in order to feed the underlying Flows with unambiguous parameters. Also implemented are a number of *search* utilities, which allow retrieval of the information of any Triple, stored locally or remotely in other Backends.

The following operations are provided by the Backend to the Frontend as aggregations of the underlying RPC calls, summarized in Table 2.1.

GET */network-map-cache*: Provides the list of active Parties.

GET */local-triples*: Triples stored locally, with the option of filter by RDF statement components through query parameters *s*, *p*, and *o*.

GET */triples*: A wrapper around */local-triples* which can also retrieve any Triple statement in the network.

PUT */triple*: Create a Triple by providing its JSON representation in the payload.

Table 2.1: Exposed Backend API resources and their operations

Method	**Location**	**Payload**	**Parameters**
GET	/network-map-cache	None	None
GET	/local-triples	None	s, p, o, owner
GET	/triples	None	s, p, o, owner
GET	/historical	None	s, p, o
PUT	/triple	Triple	None
POST	/triple	Triple (old), Triple(new)	None
DELETE	/triple	Triple	None
POST	/observer	Command, Subject, Party	None

POST */triple*: Update a Triple based on 2 JSON representations for the old and new version.

DELETE */triple*: Delete the specified Triple by starting a transaction with its latest State as an input and no output.

GET */historical*: Search for a Triple using URL query parameters, then return all historical States matching its *linearId*, which correspond to its past States.

POST */observer*: Initiate a Flow from the Backend to request a peer *Party* to either *add* or *remove* our own Party from the observer list in Triples with the specified *subject*.

2.3 Frontend Interceptor

The second component constitutes a Web Application which runs alongside the Corda Backend. It acts as a platform to run case specific mechanisms aimed at complementing and improving insertions, and updates of RDF Triples, with an API better designed for user interaction. Each Frontend is connected to a single Backend by a Web API. Agents can use the Frontend as a bridge to upload new RDF Triples, modify existing ones, or perform simple filtered searches in a local or global context. This allows for data aggregation based on specific elements and patterns such as the presence of a certain property in a Triple, or tested against the Backend to extract information about similar Triples. More advanced RDF management logic can be written in response to specific use cases. For instance the issuance of Linked Data Notifications [39]. This and other experimental use-cases are discussed in further sections.

The API provided by the Frontend to agents features the following operations, summarized in Table 2.2:

Table 2.2: Exposed Frontend API resources and their operations

Method	**Location**	**Payload**	**Parameters**
GET	/historical	None	s, p, o
POST	/observer	Command, Subject, Party	
GET	/triples	None	s, p, o, owner
PUT	/triple	Triple	
POST	/triple	Triple (old), Triple(new)	
DELETE	/triple	Triple	

GET */historical*: Returns, for given (*subject, object, and/or predicate*) query parameters, the historical states of the corresponding Triple.

POST */observer*: Requires an input JSON file containing a *Command* (either *add* or *remove*), the *subject* whose meta-data we wish to update, and the specific *party* which a session will be opened with. This operation manages our observer status with the Triples referencing the previously mentioned resource.

GET */triples*: Returns all active States matching the query parameters from the Backend's local storage.

PUT */triple*: By submitting a JSON representation of a Triple State in the payload, this operation calls the Creation Flow in the Backend. Additionally, the Frontend checks whether the request should be complemented with Triples for Notifications, or whether Observer meta-data should be edited.

POST */triple*: By submitting 2 Triples, one old and partial meta-data of a new one, the Frontend instructs the Backend to Update the old Triple if found with the new one.

DELETE */triple*: Specify a Triple, then make its new state blank, so no further ones can reference it, effectively deleting the Triple. Naturally, all past states are preserved as historical.

2.4 Network Services

Some provided network services are necessary for any application based on Corda to function properly [37].

- **Notaries**: Charged of approving and ordering the changes requested on all assets. They provide a signature which is necessary so that Backends store locally the new State of a Triple. A single Notary can serve multiple

Backends, and multiple Notaries can be arranged to perform Byzantine Fault Tolerance consensus [27].

- **Permissioning service**: Meant to automate the Backend's TLS certification by attributing every Party with an *x.509* identity [26] that is used to reference its participation in State meta-data.
- **Network mapping**: Publishing information about Parties on the network, specifically the mapping from Party identity to Backend IP address and port.

These services constitute useful utilities that enable the development of any application written for Corda. The main advantage is they can be deployed in a separate environment, provided they expose IP and ports reachable by the Backend. On the other hand, this implies some degree of centralization that has its own disadvantages. The presence of a single *services* node implies a Single Point of Failure vulnerability. Luckily, multiple such nodes can be deployed so that they form a distributed application in of themselves.

2.5 Use Cases

In addition to the basic Create, Read, Update, and Delete operations, use cases implemented in this thesis include automation of RDF aggregation for Observer status, as well as Linked Data Notifications [39]. Whenever a new Triple is submitted, the Interceptor executes a number of requests to the Backend in order to enrich the data before submission. They make effective use of the Backend's distributed system utilities and require no additional knowledge from the publisher.

2.5.1 Create, Read, Update, Delete (CRUD)

The system is able to perform the basic operations as would be expected from a non-distributed local storage system with a few improvements.

Create : Creation of a Triple registers the RDF data in addition to the meta-data specific to the Backend such as the owning Party and eventual observers and participants.

Read : Retrieval of Triples can occur in two contexts: local or remote. Locally stored triples can be directly read from the local storage. A Backend can do this with all Triples that reference a corresponding Party in their meta-data. Remote Triples, ie. those that do not reference a Party attached to the Backend, are retrieved by executing a queries to other Parties. This retrieves the current State of all Triples matching the query. A Party can then subscribe as an Observer in order to receive State updates.

Update : Local Triples owned by the Party in control of the Backend are updated by active user action. Triples retrieved through the Observer mechanisms are updated automatically (with no user interaction), or semi-automatically after being approved by the Backend respectively.

Delete : Similar to *Update*, with the exception there is no new State, so new potential Observers have no possibility to subscribe.

2.5.2 Change Tracking

The present solution does not rely on RDF to express change tracking. Rather, each Triple is subject to an underlying Blockchain-like mechanism. Parties mentioned in its State meta-data are the ones which store the Triple itself in their Backend. In the current implementation, a particular Backend can only track changes of a Triple starting from the State the former was first referenced in the latter's meta-data. Tracking changes is possible since each Triple has a *Linear ID*, which does not change during its life-time. Thus, the Backend can be programmed to offer an API method for retrieving past versions of a particular Triple based on its current State out of which the Linear ID can be deduced. Every transition from one State to the next is recorded by the Backend, along with a reference to the Party proposing the update by using the *lastEditor* field. Note that, from the Backend's perspective, the only distinction between current and previous States of Triples is the latter are marked as *historical*. Listing 2.1 depicts an example response to this type of request. In it, we observe although information inside the *data* field (lines 3-10 and 29-36) changes from one version of the Triple to another, identical *linearId* fields (lines 11 and 37) ensure the returned data corresponds to past versions of the same Triple.

Listing 2.1: JSON Response Payload of a historical states request

```
"states" : [ {
  "state" : {
    "data" : {
      "owner" : "C=GB,L=London,O=PartyA",
      "s" : "http://example.com/Foo",
      "p" : "http://www.w3.org/1999/02/22-rdf-syntax-ns#type/",
      "o" : "http://example.com/AnotherBar",
      "observers" : [ "O=PartyB,L=London,C=GB" ],
      "lastEditor" : "C=CZ,L=Prague,O=PartyA",
      "participants" : [ "C=CZ,L=Prague,O=PartyA" ],
      "linearId" : {
        "externalId" : null,
        "id" : "85894fc2-fed6-4381-9344-8a4cf0ccfeeb"
      }
    },
    "contract" : "org.blockchainrdf.backend.contracts.TripleContract",
    "notary" : "C=GB,L=London,O=Controller,CN=corda.notary.validating",
    "encumbrance" : null,
    "constraint" : {
      "attachmentId" : "..."
    }
  },
  "ref" : {
    "txhash" : "...",
    "index" : 0
  }
},{
  "state" : {
    "data" : {
      "owner" : "C=GB,L=London,O=PartyA",
      "s" : "http://example.com/Foo",
      "p" : "http://www.w3.org/1999/02/22-rdf-syntax-ns#type/",
      "o" : "http://example.com/AnotherBar",
      "observers" : [ ],
      "lastEditor" : "C=CZ,L=Prague,O=PartyA",
      "participants" : [ "C=CZ,L=Prague,O=PartyA" ],
      "linearId" : {
        "externalId" : null,
        "id" : "85894fc2-fed6-4381-9344-8a4cf0ccfeeb"
      }
    },
    "contract" : "org.blockchainrdf.backend.contracts.TripleContract",
    "notary" : "C=GB,L=London,O=Controller,CN=corda.notary.validating",
    "encumbrance" : null,
    "constraint" : {
      "attachmentId" : "..."
    }
  },
  "ref" : {
    "txhash" : "...",
    "index" : 0
  }
}]
```

2.5.3 Linked Data Notifications

We implement an automated procedure to issue notifications following the *Linked Data Notifications* [39] protocol. Notification Triples are appended to the input by the Interceptor whenever an update of a Triple which has observers and fulfills a certain criteria - in this case being part of a *owl:sameAs* statement [11]. The Backend is able to retrieve existing *inbox* resource owned by Observer Parties to the Interceptor, so that submitted data is complemented with the corresponding new Triples. This mechanism is illustrated in Listing A.2, extracted from the *Frontend Interceptor* component module.

1. When a user Creates a Triple, the Interceptor processes it before submission to the Backend.

2. The algorithm starts by retrieving meta-data about the resources in Triples received as input. For every observer of the *Subject* resource, it evaluates if a notification is necessary by verifying if other Backends possess a Triple with the *owl:sameAs* property and the previously mentioned *Subject* in the place of *Object*. For each of those returned, if they also feature an *inbox* property, a sample Notification is appended to the original Triple added by the user.

3. Finally, the newly created Triples are returned as a response to the user.

2.5.4 Observer Subscription Management

There are two ways a user can subscribe as an Observer to a resource, one implicit and another explicit. In the explicit way:

1. The user requests to observe a specific resource which is not in his/her possession (the user is not the owner nor a participant) by submitting a request with a JSON body containing three fields: *resource*, *otherParty* (the Party which is the owner of the resource), and *action* (either *add* or *remove*).

2. The request is processed by the User's Interceptor and Backend. The latter will contact the Backend which corresponds to the *otherParty* field using the network Map service, and request the user's Party be added to the Observers.

3. If successful, the User has now all Triples with the previously requested resource as *Subject*.

The second mechanism consists in whenever a Triple with the *owl:sameAs* OWL property [11] is submitted, the Interceptor assumes two things: the Party submitting this new Triple is the owner of the *Subject* resource, and, up

to date versions of the full definition of the *Object* resource being referenced are desired.

Upon submission, the Interceptor triggers a series of requests on the Backend. It starts by searching for the referenced *Object* resource across the network. If it finds one, it means this resource is registered in the network, and there exists a Party that can modify its contents. The application is then capable of requesting to be added as an Observer of this resource to the corresponding Party present in the meta-data of previously retrieved information. Upon doing so, all triples with the same RDF *Subject* (*Object* in the original submitted Triple) are updated so that the new Observer is added. Upon approval, the network forwards these Triple States to the originating Backend.

2.6 Build, Packaging and Execution

The source code included can be deployed on single or multiple computers, as it spawns a Java Virtual Machine (JVM) for each node. Backends and Network Services should be properly configured in such a way they have predetermined IP addresses and port. The implementation is mostly written in Kotlin [14], which is a language intended for the Java Virtual Machine. The following software resources are required to compile and run the application:

- An Internet connection to automate the download of requirements
- The Java Development Kit (JDK) version 1.7 or later

The project can be provisioned, built and deployed with Gradle [15], a popular build automation tool. To install Gradle and compile the code, one must simply execute the *gradlew* script with the *build* argument at the project root folder as shown in Listing A.3. This will download all the required dependencies and compile the source code.

The next step is to deploy Corda Nodes which run our Backend or Network Services. We use Gradle to assist us once again with the definition of tasks. Gradle tasks are passed on to the *Cordform* plugin provided by Corda, which deploys the nodes to the specified directory. One can define his or her own deployment tasks by following the example given in Listing A.4, placing them in *backend/build.gradle*, and executing the task by calling for instance *gradlew :backend:deployAB*. More information about using the command line interface can be found in Gradle's documentation [16].

The directory specified in the deployment hosts all deployable nodes. Each node configuration can be further fine-tuned in each respective directory under *node.conf* such as the IP address to allow nodes which are not in the localhost

[14] `https://kotlinlang.org`
[15] `https://gradle.org`
[16] `https://docs.gradle.org/current/userguide/command_line_interface.html`

to communicate. A script called *runnodes* is also present to automate the execution of a console for each respective node and its artifacts. This concludes the steps to deploy operational Backends. A simple user Web interface is present at the port corresponding to the *webPort* parameter for each respective node.

The Frontend can, on the other hand, be run by simply executing the corresponding *main* class, found under *org/blockchain-rdf/frontend/Server.kt*. It communicates exclusively with a specific Backend, whose IP address and port need be provided as arguments.

Both Frontend and Backend provide their own API, which accepts information serialized in JSON format. A number of dependency conflicts during implementation hindered the development of a more unified approach, hence reliance on Web APIs for the Frontend-Backend communication was preferred. Both Web APIs treat JavaScript Object Notation (JSON) payloads representing a copy of the fields which Triple States in the Backend are made of.

An example POST request to create a Triple can be found in Listing A.1, along with its response, which contains the complete description of the State as seen by the Backend.

Chapter 3

Evaluation

Our work responds to the possibility of a change tracking and provenance system for a decentralized Semantic Web by providing a tool to manage RDF based on Blockchain. In this chapter we present both qualitative and quantitative evaluations of its scalability, starting by analyzing conceptual differences with both existing centralized and distributed solutions, and then proceed to verify experimentally the achieved storage scalability. Finally, we describe the tests that drove the implementation of such a tool.

3.1 Provenance and Change Tracking

Change Tracking and Provenance are properties of data that are often undermined. In previous chapters we described some examples of their implementations for RDF in centralized applications. Such mechanisms are distinguished into those relying on RDF itself to leverage these functionalities - Reification, PROV-O ontology, and those that do not - Physical Backups, HTTP *Link* headers. Later, we described projects that implemented these features with the help of Blockchain mechanisms.

In order to compare these solutions we decided to observe the number of RDF Statements generated by each one of them. Indeed, change tracking implies the system at hand is capable of representing both present and past versions of data. Along with provenance, this makes for a meta-information rich system prone to scalability issues due to sheer amount of Statements, historical and currently used. In order to evaluate these systems, we will identify when possible concrete implementations of modern-day change tracking and provenance for RDF. Then, by observing the amount of Triples generated for each of these implementations as a function of the number of *Update* and *Create* operations, and factoring in the underlying storage solution, the resulting analysis will yield a comparison of scalability.

3.1.1 Models for Change Tracking

The following models represent meaningful work in the effort of bringing change tracking and provenance to RDF. The particular case of Reification [17] is separated into three, since the original concept found little adoption.

- *N-ary Relations (reification)*: rely on an intermediary resource x between Subject and Object so that meta-data is associated to it instead of the original statement. This model decomposes original *(s,p,o)* statements into *(s,p_s,x)*, *(x,p_v,o)*, *(p_s, subjectProperty, p)*, and *(p_v, valueProperty, p)* where p_s and p_v are generated on the spot. In other words, Predicate is extracted from the relationship, making it possible to identify different versions of the same Statement based on p_v and p_s, and attach metadata to x.

- *Singleton property (reification)*: where statements' predicate is replaced by a unique property referencing the original in order to provide a solution to differentiate duplicate statements.

- *Named Graphs (reification)*: encapsulating statements with a fourth resource element called graph, effectively extending RDF. This model differs from the other reification solutions as it is forming quadruples *(s,p,o,g)* instead, where several statements can be encapsulated in a graph g. Note that this model is used in [32], a solution whose integrity is verified with the help of a Distributed Ledger.

- *PROV-O Ontology*: a model well suited for the description of provenance. The particular implementation in [36] extends the ontology by formalizing RDF change-tracking. The details of this work involve making any *subject* resources a PROV *Entity*. From here, any change to other statements with the same Entity can be tracked by incremental snapshots, which evolve as a linked list containing two SPARQL queries. The incremental modifications on the original set of statements is represented from one snapshot to another. This approach, illustrated in Figure 3.1 bears some conceptual similarities with our own.

- *Our solution*: the resulting model, when observed on a single node, is able to perform change tracking and provenance of RDF without relying on meta-RDF statements, though it carries a footprint due to the fact this information is still stored on the node. The fact RDF is not being used to qualify these properties has its trade-offs. For instance, it is convenient for publishers in the sense they have less data to maintain and understand. On the other hand, an RDF reasoner or query engine might find it useful to express queries that specify the provenance and history of a certain Triple.

[17] `https://www.w3.org/TR/rdf-primer/#reification`

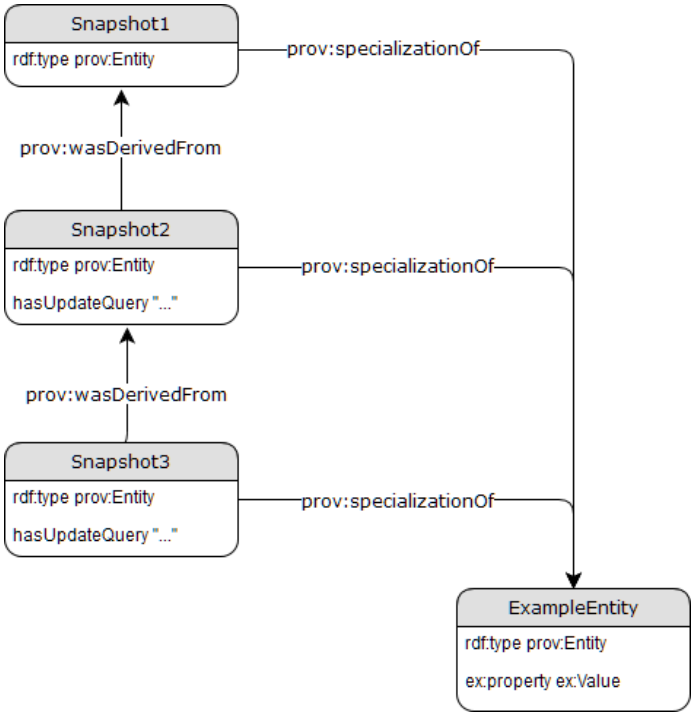


Figure 3.1: Peroni et al.'s approach to change tracking

3.1.2 Analysis

Here, we present the analysis of each model's internal structure and how adding and updating RDF Statements influences the total amount of storage needed. In order to later ease the comparison with our solution, when a concrete implementation is not specified, we propose minimal necessary properties to express change tracking and provenance. A summary of the results is presented in Figure 3.2.

- *N-ary relations (reification)*: To cope with the model described in our solution, we add hypothetical *owner* and *originatingFrom* properties to x in order to provide a minimal implementation of provenance and change tracking respectively. The resulting number of statements st for n statements with m updates and k unique properties is $st = 4m+3n+2k$ with $k < n$.

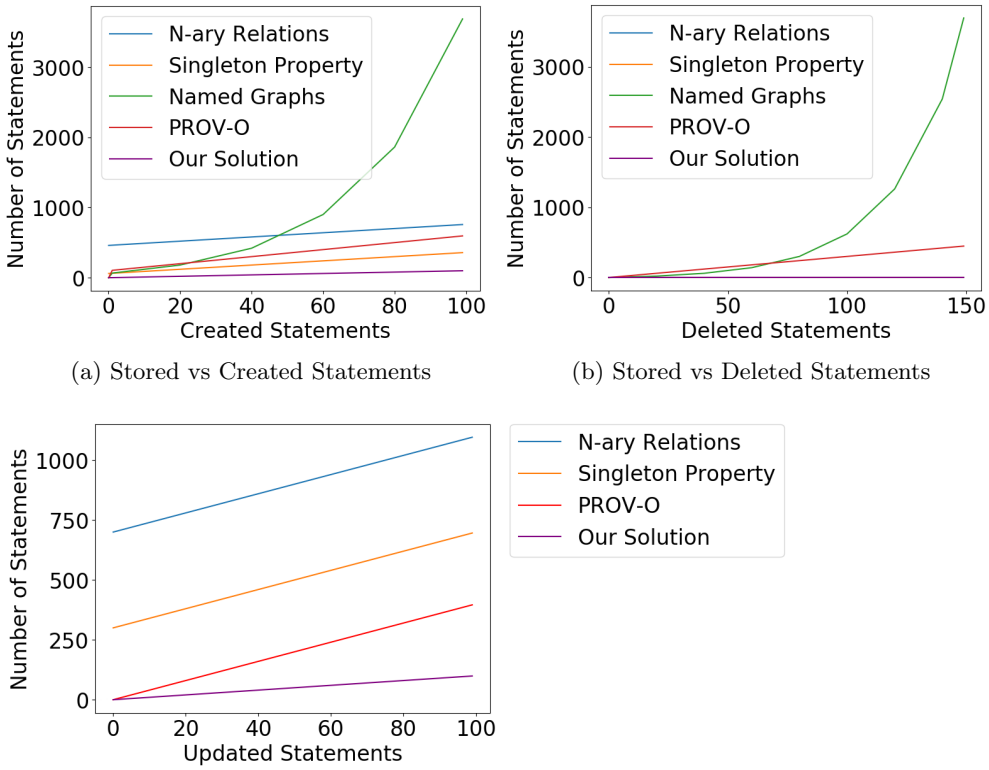
- *Singleton property (reification)*: A similar implementation under this model *(s,p,o)* Triples becomes *(s,x,o)*, where x, the unique property is linked to p by the following statement: *(x,singletonProperty,p)*. Attaching our previously defined properties for provenance and change tracking to x for n statements with m updates results in a total number of

statements $st = 4m + 3n$, since in new statements our *originatingFrom* property's range is null.

- *Named Graphs (reification)*: A viable implementation consists in segregating statements into named graphs based on the initial value of the subject resource s: *(s,p,o,g_s)*. This way, when we can offload provenance and change-tracking of any statement contained in the graph to its fourth resource g_s - *(g_s, owner, g_{owner}, g_s)*, and *(g_s, originatingFrom, g_{old}, g_s)*. Unfortunately, the consistency of the model can only be preserved if it is rebuilt for each set of modifications. For a single named graph g_s driven by subject $s \in S$, the number of statements stored is given by the sum of all statements for each version $i \in I_s$ of the graph. One can think of it as graph snapshots. Thus, we denote the number of statements introduced and removed in version i for subject s as $n_{i,s}$ and $d_{i,s}$. Note that the amount of modified Statements is irrelevant. Naturally, $n_{0,s}$ represents the initial number of statements in graph g_s. We obtain the total number of statements $st = \sum_{s \in S} \sum_{i=0}^{I_s} ((I_s - i + 1)n_{i,s} - (I_s - i)d_{i,s} + 2)$.
  Simplified $st = \sum_{s \in S} \sum_{i=1}^{I_s} (I_s - i)(n_{i,s} - d_{i,s}) + \sum_{s \in S} \sum_{i=0}^{I_s} (n_{0,s} + 2)$

- *PROV-O Ontology*: In terms of amount of statements, we obtain a pattern similar to the *Named Graphs* approach described earlier. Every new update of an Entity requires 4 new statements, but the triples themselves need not be replicated, which makes this approach comparatively compact. For every Entity $s \in S$, and every new snapshot $i \in I_s$ we obtain: $st = \sum_{s \in S} \sum_{i=0}^{I_s} (4 + \delta m_{i,s}) + \sum_{s \in S} n_s$, where $\delta m_{i,s} = n_{i,s} - d_{i,s}$ is the sum of statements added minus the ones removed from the observed Entity s for a given change i, and n_s is the initial number of statements. Note that the information contained in $\delta m_{i,s}$ can be extracted from the *hasUpdateQuery* property of each snapshot.

- *Our solution*: Each Triple has a State which evolves until the end of its life-cycle. All modifications bear the same meaning, as updating a Triple is equivalent to creating a new one, merely marking it as part of a chain - a continuation of other preexisting. The equation to find the number of Triples at any given time in the local storage of a node is $st = n+m$, n the number of new Triples, and m the number of individual modifications of a previously existing Triple. This is different from the amount of query-able Triples, which is $q = n - d$, d being the number of deleted Triples. Variable q is relatively constant. Finally, whereas in previous cases n corresponds exactly to the amount of new Triples



(a) Stored vs Created Statements

(b) Stored vs Deleted Statements

(c) Stored vs Updated Statements

Figure 3.2: Total number of statements for different operations on an RDF Dataset across proposed solutions

introduced by the user, our model breaks this rule, when seen as a whole including use-case implementations such as notification generation.

Figure 3.2 presents the theoretical results for the number of Triples generated for each of the above solutions. In the case of Creation and Deletion Operations, Named graphs suffer due to the amount of Triples needed to be reproduced after each operation. Deletion of Statements affect only PROV-O and Named Graphs, while all other solutions do not require additional Triples to track these changes. Finally, Updating a Triple involves a linear increase in the number of Generated Triples. Unfortunately it is impossible to determine how many such Triples are necessary for the Named Graph pattern, as the linear curve's angle depends on the starting number of Triples.

3.2 Storage Scalability

This section is dedicated to studying the evolution of storage used and its resulting scalability analysis. Previously in this chapter we showed that our solution is more efficient in terms of RDF Statement quantity thanks in part to the fact change tracking and provenance are offloaded to the underlying system. The goal of this experiment is to observe the evolution of real storage requirements for one node in order to support our previous claims.

3.2.1 Setup

This experiment consists of a Backend controlled by the experimenter's Party along with its local storage, and a Notary service node, which is required by the Corda implementation in order to approve incoming requests. Our interest lies in the local storage, which in Corda-v2.0 is managed by an H2 SQL database engine [18]. Its persistence, represented by a database file, can be monitored in order to measure the size of stored data.

3.2.2 Implementation

The experiment is realized by a client RPC script. Randomized strings are generated to craft Triples, which are then used to produce either the Creation of a new State, or the Update of an existing one by executing the corresponding RPC call. As means to extract results after each operation, an external system shell command reads the size of the database. In conjunction with the tracked amount and type of operations executed, we can evaluate the real impact they have on concrete storage.

Some details of the experiment might include:

- The Randomized strings used for each part of the RDF statement are always 10 characters long, ie. 10 Bytes.
- The order of operations is randomized as well. Conversely, the amount of each operation is not. We chose to run 1000 operations series over an increasing ratio of creations/updates.
- A major limitation of this experiment is the lack of a distributed computing environment at hand, which makes performance testing very difficult. Additionally, we assume the number of Operations involving a change in the meta-data of Triples are negligible compared to the amount of changes to RDF data. For this reason, the dimension of impact due to peer meta-data is not taken into consideration.

[18] `https://h2database.com/html/main.html`

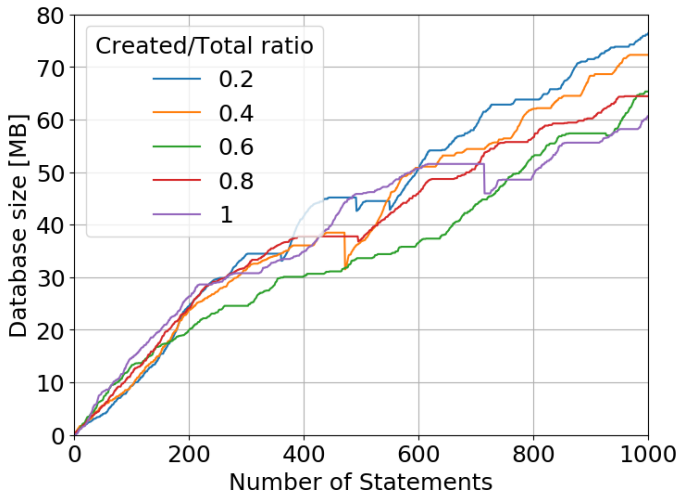


Figure 3.3: Size of the Database against the Number of Triples, classified by ratio of Create Operations, complementary to the Update ratio

The experiment's main loop is laid out in Listing A.5. Variables k and j are setup in advance. Each new file generated contains the storage data for a thousand randomly ordered but specifically numbered operations. The *create()* and *update()* calls, shown in Listings A.6 and A.7, perform the respective operation and return the database file's size. The triple updated by the *update()* function is chosen randomly as well from a list of currently active Triples depicted by the global variable named *activeTriples*.

3.2.3 Results

In Figure 3.3 we can see the size of the database change according to the execution of *Create* and *Update* operations executed in random order. The total maximum number of operation being constant, we perceive no exponential increase in the required storage for a linearly increasing number of statements. The data also suggests there is no at-scale difference in the storage consumption from any particular operation more than the other. Finally, we notice the database engine runs on-the-fly optimizations, evidenced by eventual sharp drops in the disk space value read from the system.

Table 3.1 reveals the disk usage when several runs of the algorithm are kept within the same ratio of operations for increasingly long executions. Once again, we see no signs of exponential growth, with a disturbance in the measurements caused by database optimization mechanisms.

Table 3.1: Disk usage in MB for a series of increasing amounts of operations with constant rate and randomized order

Total	**Created**	**Updated**	**Size [MB]**
100	44	56	14,67
200	100	100	24,93
300	148	152	33,51
400	200	200	37,67
500	249	251	37,21
600	290	310	49,24
700	350	350	49,73
800	385	415	45,74
900	437	463	48,18
1000	500	500	65,39

3.3 Unit Testing

In order to assess whether the execution of Flows in the Backend corresponds to the expected behavior, a number of Unit Tests were written. These tests have the advantage they do not require any infrastructure to be executed - a mock-up network with its services is emulated by the provided libraries. This allows us to recreate possible interactions between two or more nodes, as well as observing the data that is being manipulated and assessing the accuracy of our algorithms through debugging. Naturally, the development of our work was guided by the compliance with these tests. All RPC-callable Flows are covered by them. The source code for these tests can be found in the enclosed digital media at the following address: *src/bbrm/backend/src/test/kotlin/org/blockchainrdf/backend/flows/TripleTests.kt*.

- **CRUD (Create, Repeat, Update, Delete) operations**: Testing all basic management operations on Triples along with the correct replication of changes in relevant Nodes.
- **Observer commands**: Launching the set of requests to become an observer for a certain set of Triples. Then verifying the contents can be retrieved from local storage in the requesting node.
- **Discovery**: To certify that Triples are available for reading by any node regardless if they are in local storage or not.
- **Find Party by String**: To identify a Party based on its string-serialized X500 identifier.

Conclusion

We set off to deliver a system that responds to the need of change tracking and provenance in the Semantic Web, as well as guaranteeing its availability and integrity. Inspired by the Blockchain technology trend and the parallels between decentralized applications and Semantic Web, we developed, tested, and evaluated a peer-to-peer application whose Blockchain based logic delivers inbuilt change tracking and provenance, requires minimal additional knowledge from publishers, fewer RDF statements to be managed than alternative change tracking systems, and scales reasonably in terms of storage demands.

In addition to this, we explored some of the many use-case scenarios where RDF aggregations can be applied on-the-fly to published RDF for improved data quality and user experience. Such is the case of our *Observer pattern* for *owl:sameAs* links, where issuers of such links get automatic updates of entities of their interest pushed directly to their local storage, and the *Linked Data Notification pattern*, where observers of a resource can be notified to their available *ldp:inbox* of changes in a resource.

The underlying technology powering this project, Corda, constitutes a promising alternative to traditional Blockchain systems as it lets programmers define the inner workings of the communication between nodes to easily bring into reality previously unimagined protocols. Compared to traditional Blockchain implementations, the relationship users of such an application share with the network itself is more significant and long-lasting. Since no miners are required to run the network, parties enjoy a more transparent interaction with other peers. Finally, the addition of an Observer Pattern as a predefined mechanism to push changes on non-involved parties led to the implementation of a selective broadcast, wherein publishers have the freedom to manage data without depending on the consumers, effectively giving the latter a possibility to store locally an always up-to-date copy of the changes in statements and resources of their choice, ultimately contributing to the integrity and availability of all data across the network while, contrary to other solutions, not

requiring to store the entirety of data to qualify as a node.

When compared to other implementations of RDF supported by Blockchain, such as [32,33], we notice the difference in our approach which lies using a Blockchain system not as a verification support, but as a true Backend for the data we intend to decentralize. In such works, we once again come across the notion of a single chain, whose entirety must be copied in all nodes. It would only seem logical to rely on such a system for verification only, as using it as storage would become quickly expensive and a non-scalable approach. In response to this problematic, our solution spawns a separate *chain* for every statement, and proves to be a lighter solution, as not all must be replicated on every node, in some way finding a middle ground between storage demands and replication.

Future Works

We believe the decentralization of the Semantic Web is an area that deserves further attention and effort. Our solution may not be the only alternative in this direction, but compels to some extent the feasibility of such a concept. Future works might be divided into two categories:

- Backend Implementation: Some of the improvements that can be made on Backends may include the adoption of a more RDF-friendly storage in order to enable direct querying from a SPARQL endpoint, or even the implementation of an inbuilt SPARQL service capable of querying data outside the local storage and across versions of Triples.

- Frontend Interceptor: Many higher level RDF functions can be envisioned. Linked Data Notifications and Observers are just use-case examples of the potential such a system has. Further developing this area can prove relatively simple given how concise the design of communication with Backend functionalities is. We can imagine other use-cases such as Validation of RDF and dataset merging.

Bibliography

[1] Lehmann, J.; Isele, R.; Jakob, M.; et al. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, volume 6, 2015: pp. 167–195.

[2] Fielding, R. T.; Gettys, J.; Mogul, J. C.; et al. Hypertext Transfer Protocol - HTTP/1.1. *RFC*, volume 2068, 1997: pp. 1–162.

[3] Hernández, D.; Hogan, A.; Krötzsch, M. Reifying RDF: What Works Well With Wikidata? In *SSWS@ISWC*, 2015.

[4] McGuinness, D.; Lebo, T.; Sahoo, S. PROV-O: The PROV Ontology. W3C recommendation, W3C, Apr. 2013, http://www.w3.org/TR/2013/REC-prov-o-20130430/.

[5] Berners-Lee, T.; Fischetti, M. *Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor*. DIANE Publishing Company, 2001.

[6] Berners-Lee, T. Universal resource identifiers in WWW. 1994.

[7] Hitzler, P.; Krötzsch, M.; Rudolph, S. Foundations of Semantic Web Technologies. 2010.

[8] Hickson, I.; Nevile, C. M.; Brickley, D. HTML Microdata. W3C working draft, W3C, Apr. 2018, https://www.w3.org/TR/2018/WD-microdata-20180426/.

[9] Wood, D.; Cyganiak, R.; Lanthaler, M. RDF 1.1 Concepts and Abstract Syntax. W3C recommendation, W3C, Feb. 2014, http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[10] Brickley, D.; Guha, R. RDF Schema 1.1. W3C recommendation, W3C, Feb. 2014, http://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[11] Patel-Schneider, P.; Parsia, B.; Hitzler, P.; et al. OWL 2 Web Ontology Language Primer (Second Edition). W3C recommendation, W3C, Dec. 2012, http://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

[12] Bizer, C.; Heath, T.; Berners-Lee, T. Linked data-the story so far. *International journal on semantic web and information systems*, volume 5, no. 3, 2009: pp. 1–22.

[13] Dojchinovski, M.; Kontokostas, D.; Rossling, R.; et al. DBpedia Links: The Crossroad of Links for the Web of Data. In *SEMANTiCS*, 2016.

[14] Harris, S.; Seaborne, A. SPARQL 1.1 Query Language. W3C recommendation, W3C, Mar. 2013, http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[15] de Sompel, H. V.; Nelson, M. L.; Sanderson, R.; et al. Memento: Time Travel for the Web. *CoRR*, volume abs/0911.1112, 2009.

[16] Dodds, L.; Davis, I. Linked data patterns. *Online: http://patterns. dataincubator. org/book*, 2011.

[17] Zednik, S.; Huynh, T. D.; Groth, P. PROV Implementation Report. W3C note, W3C, Apr. 2013, http://www.w3.org/TR/2013/NOTE-prov-implementations-20130430/.

[18] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf.

[19] Rosenfeld, M. Analysis of bitcoin pooled mining reward systems. *arXiv preprint arXiv:1112.4980*, 2011.

[20] Croman, K.; Decker, C.; Eyal, I.; et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, Springer, 2016, pp. 106–125.

[21] Popov, S. The tangle. *cit. on*, 2016: p. 131.

[22] Bentov, I.; Gabizon, A.; Mizrahi, A. Cryptocurrencies Without Proof of Work. In *Financial Cryptography Workshops*, 2016.

[23] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, volume 151, 2014: pp. 1–32.

[24] Christidis, K.; Devetsikiotis, M. Blockchains and Smart Contracts for the Internet of Things. *IEEE Access*, volume 4, 2016: pp. 2292–2303.

[25] Brown, R. G.; Carlyle, J.; Grigg, I.; et al. Corda: An Introduction. *R3 CEV, August*, 2016.

[26] Housley, R.; Ford, W.; Polk, W.; et al. Internet X. 509 public key infrastructure certificate and CRL profile. Technical report, 1998.

[27] Bessani, A. N.; Sousa, J.; Alchieri, E. A. P. State Machine Replication for the Masses with BFT-SMART. *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014: pp. 355–362.

[28] Ongaro, D.; Ousterhout, J. K. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*, 2014.

[29] Thomas, S.; Schwartz, E. A protocol for interledger payments. *URL https://interledger. org/interledger. pdf*, 2015.

[30] Third, A.; Domingue, J. LinkChains: Exploring the space of decentralised trustworthy Linked Data. 2017.

[31] Third, A.; Quick, K.; Bachler, M.; et al. Benchmarking LinkChains: Performance of Blockchain-backed Trustworthy Linked Data.

[32] Third, A.; Tiddi, I.; Bastianelli, E.; et al. Towards the temporal streaming of graph data on distributed ledgers. In *European Semantic Web Conference*, Springer, 2017, pp. 327–332.

[33] Sutton, A.; Samavi, R. Blockchain Enabled Privacy Audit Logs. In *International Semantic Web Conference*, Springer, 2017, pp. 645–660.

[34] Carroll, J. J.; Bizer, C.; Hayes, P.; et al. Named graphs. *Web Semantics: Science, Services and Agents on the World Wide Web*, volume 3, no. 4, 2005: pp. 247–267.

[35] Samavi, R.; Consens, M. P. L2TAP+ SCIP: An audit-based privacy framework leveraging Linked Data. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012 8th International Conference on*, IEEE, 2012, pp. 719–726.

[36] Peroni, S.; Shotton, D. M.; Vitali, F. A Document-inspired Way for Tracking Changes of RDF Data. In *Proceedings of the 1st Workshop on Detection, Representation and Management of Concept Drift in Linked Open Data co-located with the 20th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2016), Bologna, Italy, November, 2016.*, *CEUR Workshop Proceedings*, volume 1799, edited by L. Hollink; S. Darányi; A. Meroño-Peñuela; E. Kontopoulos, CEUR-WS.org, 2016, pp. 26–33. Available from: `http://ceur-ws.org/Vol-1799/Drift-a-LOD2016_paper_4.pdf`

[37] Hearn, M. Corda: A distributed ledger. *Corda Technical White Paper*, 2016.

[38] Thurlow, R. RPC: Remote Procedure Call Protocol Specification Version 2. *RFC*, volume 1057, 1988: pp. 1–25.

[39] Capadisli, S.; Guy, A.; Lange, C.; et al. Linked Data Notifications: A Resource-Centric Communication Protocol. In *The Semantic Web*, edited by E. Blomqvist; D. Maynard; A. Gangemi; R. Hoekstra; P. Hitzler; O. Hartig, Cham: Springer International Publishing, 2017, ISBN 978-3-319-58068-5, pp. 537–553.

Appendix A

Code References

Listing A.1: Example payload for a PUT request/response during the creation of a Triple

```
// Request
{
  "s": "http://example.com/Foo",
  "p": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type/",
  "o": "http://example.com/Bar",
  "observers": [
    "O=PartyB,L=London,C=GB"
  ]
}

// Response
"state" : {
  "data" : {
    "owner" : "C=CZ,L=Prague,O=PartyA",
    "s" : "http://example.com/Foo",
    "p" : "http://www.w3.org/1999/02/22-rdf-syntax-ns#type/",
    "o" : "http://example.com/Bar",
    "observers" : [ "O=PartyB,L=London,C=GB" ],
    "lastEditor" : "C=CZ,L=Prague,O=PartyA",
    "participants" : [ "C=CZ,L=Prague,O=PartyA" ],
    "linearId" : {
      "externalId" : null,
      "id" : "85894fc2-fed6-4381-9344-8a4cf0ccfeeb"
    }
  },
  "contract" : "org.blockchainrdf.backend.contracts.TripleContract",
  "notary" : "C=GB,L=London,O=Controller,CN=corda.notary.validating",
  "encumbrance" : null,
  "constraint" : {
    "attachmentId" : "..."
  }
},
"ref" : {
  "txhash" : "...",
  "index" : 0
}
```

Listing A.2: Input triples being complemented with Notifications

```
1
2  input.observers?.forEach { observer ->
3    // if conditions for creating a notification are met
4    if (verifyConditions(services, observer, input)) {
5      // Get foreign inbox information from Backend's local storage
6      val inbox = findOrSubscribeToInbox(observer)
7      if (inbox != null) {
8        // Craft and add the notification
9        extraTriples.add(TripleDataClass(
10         s = inbox.o,
11         p = "<http://www.w3.org/ns/ldp#contains>",
12         o = input.s + ": "+notificationMessage,
13         owner = me,
14         participants = emptyList(),
15         observers = listOf(inbox.owner!!),
16         lastEditor = me
17       ))
18     }
19   }
20 }
21 return extraTriples.plus(input)
```

Listing A.3: Gradle commands to build the project binaries

```
1  # On Unix based
2  ./gradlew build
3  # On Windows
4  ./gradlew.bat build
```

Listing A.4: Gradle definition of a task

```
task deployAB(type: net.corda.plugins.Cordform, dependsOn: ['jar']) {
  directory "./build/nodes"
  networkMap "O=Controller,L=London,C=GB"
  node {
    name "O=Controller,L=London,C=GB"
    advertisedServices = ["corda.notary.validating"]
    p2pPort 10002
    rpcPort 10003
    cordapps = [
      "$project.group:shared:$project.version",
    ]
  }
  node {
    name "O=PartyA,L=Prague,C=CZ"
    advertisedServices = []
    p2pPort 10005
    rpcPort 10006
    webPort 10007
    cordapps = [
      "$project.group:shared:$project.version",
      "$project.group:cordapp:$project.version",
    ]
    rpcUsers = [[ user: "user1", "password": "test", "permissions": [
      "StartFlow.org.remy.backend.flows.CreateTriple",
      "StartFlow.org.remy.backend.flows.DiscoveryRequest",
      "StartFlow.org.remy.backend.flows.UpdateTriple",
      "StartFlow.org.remy.backend.flows.DeleteTriple",
      "StartFlow.org.remy.backend.flows.ManageSelfAsObserver",
      "StartFlow.org.remy.backend.flows.HandleObserverRequest"
    ]]]
  }
  node {
    name "O=PartyB,L=London,C=GB"
    ...
  }
  ...
}
```

Listing A.5: Main loop of the storage data generation and collection script

```
fun main(args: Array<String>)
{
  proxy = CordaRPCClient(NetworkHostAndPort.parse(args[0]))
    .start(username = "user1", password = "test").proxy
  val file = File("results_$j_$k.csv").printWriter().use
  { out ->
    out.write("operation, created, updated, size\n")
    activeTriples.clear()
    val initialSize = create()
    var created = 0
    var updated = 0
    for(i: Int in 0..j)
    {
      if (created<k)
      {
        if (random.nextInt(2) == 0)
        {
          created+=1
          out.write("$created,$updated,${create() - initialSize}\n")
        }
        else
        {
          updated+=1
          out.write("$created,$updated,${update() - initialSize}\n")
        }
      }
      else
      {
        updated+=1
        out.write("$created,$updated,${update() - initialSize}\n")
      }
    }
  }
}
```

Listing A.6: RPC client function for the creation of Triples

```
fun create(): Long
{
  val flow = proxy!!.startFlow
  (
    ::CreateTriple,
    genRandomString(10),
    genRandomString(10),
    genRandomString(10)
  )
  val result = flow.returnValue.getOrThrow()
  activeTriples.add
  (
    Pair(result.tx.outRef<TripleState>(0).ref,
    result.tx.outputStates.first() as TripleState)
  )
  return readSize()
}
```

Listing A.7: RPC client function for the update of Triples

```
1
2  fun update(): Long
3  {
4    val index = random.nextInt(activeTriples.size)
5    val randomTriple: Pair<StateRef, TripleState> = activeTriples[index]
6    val newTripleState = randomTriple.second.copy
7    (
8      s = genRandomString(10),
9      p = genRandomString(10),
10     o = genRandomString(10)
11   )
12   val flow = proxy!!.startFlow
13   (
14     ::UpdateTriple,
15     randomTripleUpdate.first,
16     newTripleState
17   )
18   val result = flow.returnValue.getOrThrow()
19   activeTriples.removeAt(index)
20   activeTriples.add
21   (
22     index,
23     Pair(result.tx.outRef<TripleState>(0).ref,
24     result.tx.outputStates.first() as TripleState)
25   )
26   return readSize()
27 }
```

Appendix B

Acronyms

LOD Linked Open Data

RDF Resource Description Framework

RDFS RDF Schema

OWL Web Ontology Language

SPARQL SPARQL Protocol and RDF Query Language

RPC Remote Procedure Call

Appendix C

Contents of enclosed SD Card

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `bbrm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format