

# Comparing file-based and query-based access techniques for decentralized social networks

Lukas Vanhoucke

Supervisor: Prof. dr. ir. Ruben Verborgh  
Counsellor: Joachim Van Herwegen

Master's dissertation submitted in order to obtain the academic degree of  
Master of Science in Computer Science Engineering

Department of Electronics and Information Systems  
Chair: Prof. dr. ir. Koen De Bosschere  
Faculty of Engineering and Architecture  
Academic year 2017-2018





# Comparing file-based and query-based access techniques for decentralized social networks

Lukas Vanhoucke

Supervisor: Prof. dr. ir. Ruben Verborgh  
Counsellor: Joachim Van Herwegen

Master's dissertation submitted in order to obtain the academic degree of  
Master of Science in Computer Science Engineering

Department of Electronics and Information Systems  
Chair: Prof. dr. ir. Koen De Bosschere  
Faculty of Engineering and Architecture  
Academic year 2017-2018



© Ghent University

The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In the case of any other use, the copyright terms have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master dissertation. (May 30, 2018)

# Acknowledgements

During the past year, I've spent numerous hours familiarizing myself with the wonderful ecosystem of Linked Data. This would never have went this smooth without the help of Ruben Verborgh and Joachim Van Herwegen, who pointed me in the right direction whenever I was unsure. Ruben, you were the one to inspire me into working on this subject and continued to do so throughout the year. Your insights into decentralized social media were invaluable. Joachim, you provided me with lots of technical knowledge about Linked Data and were always ready to evaluate my progress. You pointed out many irregularities while going over my thesis with much care. I especially liked that you were not afraid to mention your style preference, even when it is just something minor like disliking the color of syntax highlighting in code listings. Both Ruben and Joachim were very quick to respond to mails, which I appreciate a lot. A big thanks to the both of you.

The subject of the thesis was not set in stone. At the start of the first semester, I was planning to collaborate closely with the Open Weblides team. I am thankful to Esther De Loof for introducing me to this platform and advising me to look into certain annotation systems. I would also like to thank the whole Open Weblides team for developing this product. I have used it plenty of times to test my annotations plugin. However, as the year progressed, I started focusing more on access techniques, and less on the actual implementation of an annotation system for your platform. I hope the results of this thesis may still be meaningful to you.

Finally, thanks to the people of W3C and especially the ones of the Social Web Working Group and the people that collaborated to the Solid project. Although I never had the need to contact any of them directly, their documentation and discussions helped me out a lot.

# Abstract

## Comparing file-based and query-based access techniques for decentralized social networks

Social media applications of today have been heavily criticized for not fully disclosing what their users' data is used for. As it is stored on their servers, they often claim ownership over all data. Each platform has its own centralized database, or information silo. This vendor lock-in makes it almost impossible to transfer your data from one platform to another.

In this master's dissertation, a potential decentralized solution to these problems is explored, specifically in the context of decentralized annotations. Data is stored as Linked Data in the user's own personal online datastore (pod) and can be accessed by applications in multiple ways.

Here, file-based access techniques are compared against query-based access techniques. When file-based access is used, information is stored in files, and only whole files can be collected. Querying happens client-side. Query-based access requires information to be stored in a single graph on a SPARQL endpoint, and responds to requests by querying the data server-side. The advantages and limitations of both techniques are first explored for decentralized annotations. Afterwards, it is checked whether the results are still valid for social applications other than annotations.

Multiple test setups that implement decentralized annotations, along with an actual annotation plugin, were used to compare the two access techniques. For file-based access, a Solid server is used, while query-based access required a RESTful server on top of a SPARQL endpoint to be written from scratch.

With query-based access, complex intersections can be performed in a single query. This for instance allows more advanced ACL to be used. It also outperforms file-based access when collecting many annotations, but this difference can sometimes be neglected through a good caching design. File-based access techniques are able to reuse existing file-based technologies such as directory watchers and Git. It also comes with an inherent directory structure, it is easier to manually manipulate data, and it is easily reusable in contexts other than annotations.

The annotation plugin shows that the investigated techniques have the potential to replace present-day centralized social applications.

**Keywords:** Decentralized – Social media – Linked Data – Solid – Annotations

# **Extended abstract**

# Comparing file-based and query-based access techniques for decentralized social networks

Lukas Vanhoucke  
Supervisor: Ruben Verborgh  
Counsellor: Joachim Van Herwegen

**Abstract**—Social media applications of today have been heavily criticized for not fully disclosing what their users' data is used for. As it is stored on their servers, they often claim ownership over all data. Each platform has its own centralized database, or information silo. This vendor lock-in makes it almost impossible to transfer your data from one platform to another. In this master's dissertation, a potential decentralized solution to these problems is explored, specifically in the context of decentralized annotations. Data is stored as Linked Data in the user's own personal online datastore (pod) and can be accessed by applications in multiple ways. Here, file-based access techniques are compared against query-based access techniques. When file-based access is used, information is stored in files, and only whole files can be collected. Querying happens client-side. Query-based access requires information to be stored in a single graph on a SPARQL endpoint, and responds to requests by querying the data server-side. The advantages and limitations of both techniques are first explored for decentralized annotations. Afterwards, it is checked whether the results are still valid for social applications other than annotations. Multiple test setups that implement decentralized annotations, along with an actual annotation plugin, were used to compare the two access techniques. For file-based access, a Solid server is used, while query-based access required a RESTful server on top of a SPARQL endpoint to be written from scratch. With query-based access, complex intersections can be performed in a single query. This for instance allows more advanced ACL to be used. It also outperforms file-based access when collecting many annotations, but this difference can sometimes be neglected through a good caching design. File-based access techniques are able to reuse existing file-based technologies such as directory watchers and Git. It also comes with an inherent directory structure, it is easier to manually manipulate data, and it is easily reusable in contexts other than annotations. The annotation plugin shows that the investigated techniques have the potential to replace present-day centralized social applications.

**Keywords:** Decentralized, social media, Linked Data, Solid, Annotations

## I. INTRODUCTION

The size of social networks such as Facebook, Twitter and Instagram blew up in the past decade. For instance, Facebook has over two billion active users. This amount of data is an invaluable asset to sell to third parties, e.g. for market predictions and personalized ads.

An additional problem are the growing number of *information silos*. The information that is stored on one traditional social network, cannot easily be transferred to another social network. This information includes messages, friends, photos, personal details, and more. This problem is illustrated in Figure 1. The main cause of this development is the fact that

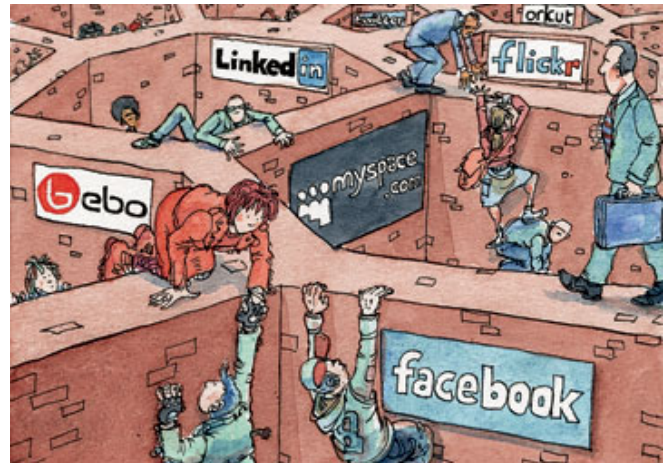


Figure 1: Drawing which illustrates the silo problem. People are stuck within particular social networking platforms. They would like to benefit from other social networking platforms and share the data from one platform with people on a different platform. (Source: David Simonds, [www.economist.com](http://www.economist.com), 2008)

it is simply much easier to manage, publish and search large amounts of similarly structured content using a centralized platform [1]. A side-effect of this is that people tend to stay with the same social network since changing is too hard. This is called vendor lock-in and is advantageous for the companies. The information that is stored in the silo, can be used by the silo owners for whatever they want, as long as it corresponds with the agreed policies [2]. This is usually stated in the end user license agreement, which is unfortunately seldom read. With fewer than 1% of the servers serving over 99% of the content, the current phenomenon of information silos goes against the original intention of the Web to become a decentralized platform [1].

## II. LITERATURE REVIEW

The problems stated in the introduction can be avoided by using decentralized technologies.

### A. Linked Data

Linked Data is an example of a decentralized technology. It uses the Resource Description Framework (RDF) to express meaning by encoding triples of subject, predicate and object



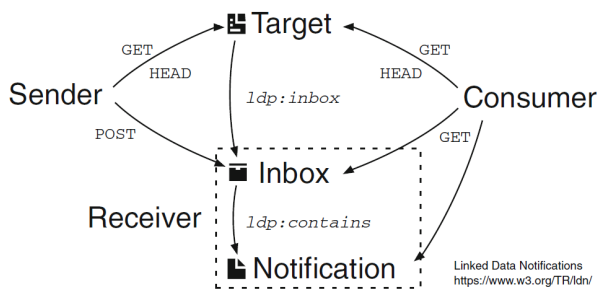


Figure 2: Graphical representation of the Linked Data Notifications protocol. (Source: Capadislis et al. [4])

as Universal Resource Identifiers (URIs). The main advantage of expressing nodes and their relationships as URIs is to make it easier for computers to draw unambiguous conclusions. Ontologies bundle the concepts and relationships concerning a specific topic. Just like regular databases, RDF datasets can be queried using a specialized query language: SPARQL.

### B. Decentralized social network technologies

Preferably, the application logic of a social network should be fully decoupled from the data to avoid a vendor lock-in. Interoperability can only be maximized if well-defined protocols and standards are developed. At this point, social platforms become views over your data instead of fully contained applications [3].

In order to have a decentralized social network with the same capabilities as traditional social networks, some additional services may be required. For instance, the Linked Data Notifications (LDN) protocol [4] allows notifications to be represented as Linked Data so that they can be shared and reused across applications. It uses an inbox to store a collection of received notifications, which are retrievable resources that return RDF data. Because these notifications are reusable and accessible by all applications, there needs to be a way for consumers to autonomously find this inbox. Therefore, the inbox should be announced by using a `ldp:inbox` triple. The protocol is shown in Figure 2.

### C. Linked Data Platform

The W3C specification for Linked Data Platform (LDP) [5] describes HTTP as a means to implement CRUD operations (Create, Read, Update, and Delete) on Linked Data Platform resources. All LDP resources (including containers) have their own URI. In this thesis, Solid will be used extensively. This is a set of conventions and tools to build decentralized social applications implementing LDP and based on Linked Data Principles.

## III. PROBLEM DESCRIPTION

The centralization of current-day’s social networks has two large consequences: data cannot be reused across applications,

and the user loses ownership over the data. These problems can be solved by making use of Linked Data technologies.

Although Linked Data uses the same syntax (i.e. RDF), triples can be accessed in multiple ways. A first option is to store triples in multiple files. Accessing a triple then requires collecting the whole file in which the triple resides, and querying the file client-side. This is a file-based access technique. At the time of writing, Solid uses this access technique by default and will therefore be used as this technique’s reference implementation. A second option is to store all triples in the same graph on a SPARQL endpoint, and use server-side queries to collect the relevant triples. Custom middleware will be inserted between the SPARQL endpoint and the data-requesting user to allow the middleware to generate queries at the user’s request. For instance, when a user wants to request all information about his friends, he would send a GET request to the RESTful `/friends` endpoint. This type of access will be referred to as graph- or query-based access.

The goal of this work is to find a comprehensive answer to the following research questions:

- Q<sub>1</sub>: What are the advantages and limitations of using graph-based techniques over file-based techniques in the context of decentralized collection of annotations?
- Q<sub>2</sub>: How does the number of annotations influence their collection time when using graph-based and file-based access techniques?
- Q<sub>3</sub>: How do these access techniques compare on user-friendliness and difficulty of implementation?
- Q<sub>4</sub>: How are these techniques reusable in social application contexts other than annotations?

Using the gathered knowledge, following hypotheses can then either be confirmed or falsified:

- H<sub>1</sub>: In the context of annotations, query-based access has faster retrieval of annotations
- H<sub>2</sub>: File-based access techniques can reuse existing filesystem-based techniques to make certain tasks easier, such as watching for updates using directory watchers or changing data using common text editors.
- H<sub>3</sub>: File-based access techniques can easily be reused in other social contexts, while this is harder for query-based access techniques.
- H<sub>4</sub>: The differences between file-based and query-based access techniques are still present when they are used for decentralized social applications other than annotations.

To test these hypotheses, we implemented decentralized annotations using both access techniques, as they are considered to be a social application.

## IV. ANALYSIS

### A. Decentralized Annotations

Most websites that implement some sort of annotation system, store annotations on their own servers. Here, they have to be stored in the user’s datapod instead. To allow interoperability between different applications, the annotations are stored as Linked Data using the Web Annotation Data

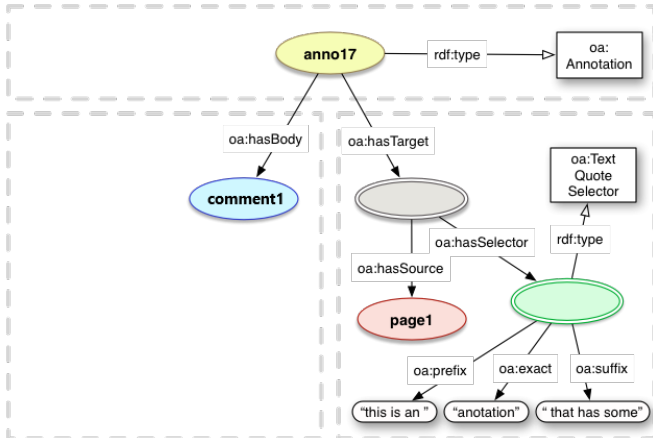


Figure 3: Basic annotation structure using the TextQuoteSelector. Here, the word “annotation” would be highlighted. (Source: [6])

Model. Annotations consist of a body and a target that it relates to.

When an annotation is created, it is stored in a data pod of the user’s choice. If the user’s datapod uses a hierarchical structure, the annotations should be saved in an *annotations directory*, which is usually announced by using the `oa:annotationService` predicate. Here, `oa` denotes the Web Annotation Ontology. This could for example be added to the user’s WebID, or to the website by embedding it as RDFa.

To “upload” an annotation, the website needs to be notified that an annotation for the website was created. This can be accomplished by sending a Linked Data Notification to website’s `ldp:inbox`, which should be announced by the website (again, preferably through the use of RDFa). Upon receiving the notification, a link is added to the website’s local list of annotations. To display the annotations, the website exposes this list and lets the users collect the annotations by themselves.

### B. Data manipulation

Both the file-based and query-based setup expose data in a RESTful way, although they are fundamentally different in the way data is manipulated. First, manually manipulating data is easier when data is stored as files since common text editors can be used. Querying the data usually happens client-side, after collecting the files. In the case of annotations, each annotation is represented in a single file.

In contrast, query-based techniques use SPARQL queries to retrieve certain predefined fields. Since only field values are returned, less information is available compared to file-based techniques. Requesting a query-based resource (i.e. query-based access) requires a specific query for each request, which is generated by the server. This means that the server code has to be aware of what the files look like in order to expose an interface to the data. In other words, the server needs application knowledge in order to write meaningful queries.

### C. Detecting updates

The Linked Data Notifications specification (LDN) [4] benefits from being able to process new or updated resources as they enter the server’s inbox. Detecting this can be accomplished in 3 ways. First, the state of the inbox can periodically be polled. Depending on the polling rate, processing updates might happen with a significant delay. Alternatively, the server can inspect all incoming packets and see whether they are being watched by the user. This is possible for both access techniques. The final option is to attach a file or directory watcher to the resource you want to track. This is only possible for file-based servers. This option offers more flexibility to the end user.

### D. Directories

A user’s datapod can contain large amounts of data. To avoid losing track of what data belongs together, some kind of structure is necessary. Servers that implement their Linked Data as files can use the operating system’s file system for organization. Here, query-based servers are at a disadvantage, since they initially do not have any inherent structure. All data is stored in the same graph as an ocean of triples. However, an artificial directory structure can be implemented by using the LDP standards.

### E. Complex intersections

When a file-based server is used, each annotation is stored in a separate file. But what if complex intersections have to be performed? Assume for example that someone wishes to collect all annotations for which the creator’s age is over fifty years. In such cases, all files have to be collected in order to do perform the intersection. When all information is stored in a single graph on a SPARQL endpoint, executing intersecting queries becomes a lot easier, as all data can be queried at once, and server-side.

### F. Access control

Solid uses WebAccessControl (WAC) to allow or deny access to resources. This uses an ontology with simple rules to determine whether users or groups, identified by their WebID, can read a resource, write or append to it, or a combination of the three. The granularity of WAC is one file. Sometimes, it can be useful to have an even finer-grained ACL, which is only possible with query-based access.

First of all, WAC can be implemented within a single graph on a SPARQL endpoint that stores all relevant data. For hierarchical ACL to work, a directory structure needs to be present. Otherwise, a single root ACL controls access to all resources in the graph. Therefore, using a SPARQL endpoint has no disadvantages concerning ACL compared to using Solid. However, having all data in a single graph allows more complex queries, which can be used to implement more complex ACL rules as well.

For this thesis, an experimental extension to WebAccessControl was developed to demonstrate these two types of additional ACL rules. A graph consists of many triples, each

of which is considered a field. In some cases, it can be useful to determine which fields can be accessed by the user or application. For instance, a phonebook application should only be able to see the names and phone numbers of your friends, and a birthday reminder app should only see the names and the birthdays of your friends.

Entire results should also be filterable. For instance, instead of sending all annotations, those that refer to an admin-only page are filtered out of the results for normal users. This is horizontal filtering, whereas in the previous case vertical filtering was used.

Things start to get more difficult when complex structures – such as annotations – are used. In this case, the server needs to know the structure, and how the possible fields are related to this structure. This is a large drawback, since it cannot easily be extended to other applications. We solved this problem by allowing the ACL to give hints towards generating a meaningful query. Unfortunately, this option is tightly coupled to the application and how the server generates the queries and is not comprehensible for other servers that generate the queries in a different way. It would be more effective when an annotation’s structure is set in stone and included in the graph, so that applications know exactly what the annotation looks like. SHACL [7] could help with this.

### G. Caching

Caching is the act of keeping copies in order to speed up the system. A drawback of decentralization is that resources have to be collected from all around the globe, which takes a long time and also stresses the network. Instead, copies from multiple sources could be distributed. Doing so is possible for both file-based and query-based access, since both servers are implemented as RESTful endpoints. The corresponding ACL files have to be included in the cache, otherwise plenty of requests still have to be sent. However, this implies that the data owners allow their ACL files to be public, which is definitely not always the case!

### H. Versioning

To keep track of changes in files, file-based datapods can use existing technologies such as Git. These technologies cannot be used in combination with query-based access techniques, since they’re simply not available for SPARQL endpoints. Instead, custom middleware could be written to keep track of changes to the graph, and to recreate the content at any point in time. This is however very complex and is still an active topic of research.

## V. IMPLEMENTATION OF AN ANNOTATION PLUGIN

In order to objectively compare file-based and query-based access, we developed multiple tools. Most of these tools manipulate decentralized annotations. The end result is an annotation plugins for file-based access and another for query-based access.

The plugins require the website to have a `oa:AnnotationService` and a `ldp:inbox` field. The utilized architecture is shown in Figure 4.

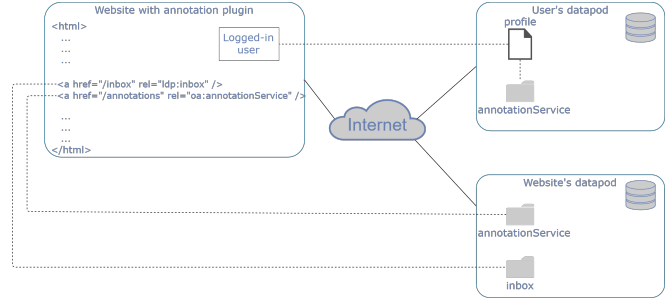


Figure 4: Architectural diagram showing the connections between the annotation plugin and the datapods. Note that the website, the user’s datapod and the website’s datapod may reside on different servers.

### A. Storing annotations

After selecting some text to highlight, an annotation graph is created. The graph is then posted to the user’s annotation directory, which is announced in his profile as `oa:AnnotationService`. Upon success, the inbox of the website is notified about this new annotation using the LDN protocol, so that it can add the annotation URL to its list of annotations. However, when query-based access is used, only the link to the user’s annotation service needs to be added to this list, as all annotations are collected through this service.

### B. Loading annotations

There are two options to load annotations. The first option is to let the client collect a list of annotations. Therefore, the client will first request the list of annotations to the server’s `annotationService`. All annotations on the list are then collected. File-based access requires a request to be sent for each annotation, while only a request per datapod is necessary when query-based access is used. The second option is let the server collect its own list of annotations. Doing so, the server will need to look at its local list of annotations, and then request all foreign annotations. The biggest advantage over client-side collection is the fact that the server can cache annotations of other users.

## VI. EVALUATION

### A. Performance

The performance of both access techniques are compared by measuring the timing differences between manipulating annotations on file-based and query-based servers. All tests are executed on a laptop with 8 GB RAM and an Intel® Core™ i7-7500U CPU @ 2.70GHz and a 50 Mbps internet connection. The annotations are stored on a remote server, for which a DigitalOcean droplet with a single vCPU, 2 GB RAM is used.

A storage test was performed for 1, 10, 100 and 500 annotations. It was attempted to do this test for 1000 annotations, but the Solid server could not handle this many requests. The results of this test can be found in Figure 5.

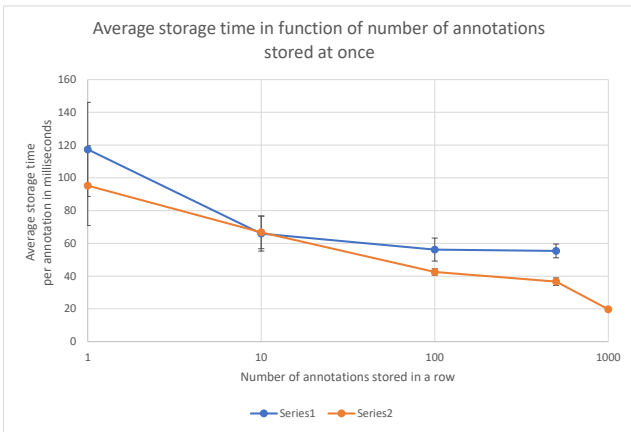


Figure 5: Results of the storage test visualized using a graph with corresponding standard deviations over 5 runs. The x-axis uses log-scale and the y-axis uses linear-scale.

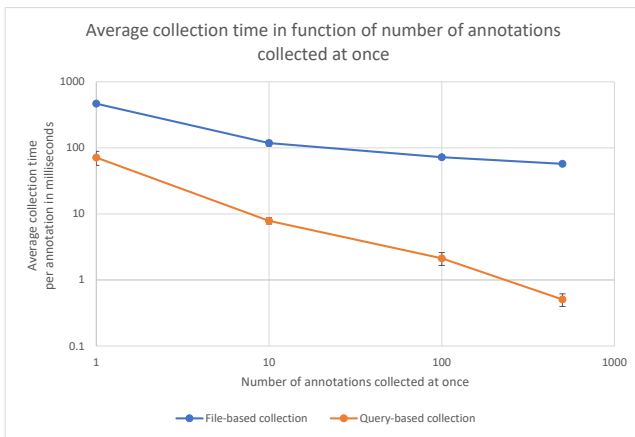


Figure 6: Results of the loading test visualized using a graph with corresponding standard deviations over 5 runs. Both axes use log-scale.

Clearly, storing annotations is slightly faster when using query-based techniques. However, the storage times for both access techniques are within the same order of magnitude. This is because each annotation is sent as a single request, and these communication delays are bottlenecking the system.

After having stored these annotations, they can now be loaded. In this test, all annotations for all websites are collected at once. The results are visualized in Figure 6. There is now a large difference between file-based and query-based retrieval. This time, the communication delay is less of a bottleneck for query-based access, since all annotations can be retrieved using a single request.

### B. Comparing the access techniques in the context of annotations

File-based implementations such as Solid are able to deal with all kinds of RDF documents due to its simplicity of only

manipulating at file-level granularity. Meanwhile, in order to implement query-based access to new types of data, one or multiple new access endpoints have to be programmed since each request requires a new custom SPARQL query. This confirms hypothesis  $H_3$ .

Detecting updates is easier when using file-based access, since you can simply attach a listener to the directory you would like to watch. Similarly, versioning is easy by using versioning systems such as Git, and raw graph data can be edited with your standard text editor, or any file-based API. These are all examples of technologies that already exist and can be reused in the context of file-based access, but not when using query-based access, confirming hypothesis  $H_2$ .

Another important aspect of a good access technique in the context of annotations is retrieval performance. When loading annotations, users prefer to have them show up as fast as possible. File-based access has user experience flaws when a large amount of annotations have to be collected. It is possible to reduce this delay by implementing application-specific caching mechanisms, such as caching all annotations for a webpage in the same file and returning this file upon a request to the `AnnotationService`. This is however not something Solid supports by default. Therefore, hypothesis  $H_1$  can be confirmed, although file-based access can negate performance differences in some scenarios. First, if all annotations are cached within the `AnnotationService`'s datapod, query-based access seems to be faster. However, file-based access endpoints can be adapted to return all annotations on a list with just a single request. This requires modifying server-side code, and goes against the rules of the LDP standard. An even better solution is to cache all annotations for a webpage in the same file, and then return this file when the `AnnotationService` is contacted. This can be done in a single request and is fully LDP-compliant. A second scenario is the following. When a small number of annotations per user are stored, and the annotations' ACL states that they are not publicly viewable by all users, and the ACL itself is not public either, the annotations cannot be cached! In this case, the performance is determined by communication delays, and thus the number of users. A last scenario occurs when each user stores a large amount of annotations with the same ACL rules as previous scenario. In this case, query-based access easily outperforms file-based access. However, note that the users' datapods can still implement good caching mechanisms, in which case the performance differences will again become negligible.

### C. Decentralized social applications in general

During this dissertation, it was assumed that decentralized annotations are representative for all decentralized social applications, as stated by hypothesis  $H_4$ . Now that most differences have been examined, it becomes clear that annotations were indeed representative.

The discussed differences are still there when looking at social applications other than annotations, confirming hypothesis  $H_4$ . For instance, assume a social network application

where images are shared between people. Conceptually, this is still similar to annotations: adding an annotation to a specific website becomes posting an image and a description to the application’s website. The image and its metadata are still stored on your own server, while the website keeps track of all URIs.

## VII. CONCLUSION

The Cambridge Analytica scandal has recently sparked conversation about what social networks are allowed to do with your data. This company used the data of about 87 million Facebook profiles to manipulate elections. It shows a clear need to regulate personal data. With the arrival of the General Data Protection Regulation (GDPR), which was enforced on 25 May 2018, some of the privacy issues concerning traditional centralized social networks have been resolved. In the context of this thesis, data portability is the most important right given to the users. This allows people to request data in a machine readable format which can be used in other contexts, such as a competing social media website. However, only the data that was provided by the data subject can be retrieved, not the data that was additionally generated by the platform. Also, there is no incentive for the companies to represent this data as something that is easily reusable by a competitor. Decentralization is therefore still the preferred solution.

During this thesis, both file-based and query-based access was explored by using a decentralized annotations set-up. We proved that annotations are easily decentralizable. While query-based access outperforms file-based access when collecting many annotations, this difference can sometimes be neglected through a good caching design. File-based access techniques are able to reuse existing file-based technologies such as Git and directory watchers. It also makes manually manipulating data easier, and implementing a directory structure is straightforward since it is inherent to a file-system. File-based access relies on a simple API (e.g. the LDP specification) which allows easy reusability in other contexts. Meanwhile, the back-end to handle query-based access currently requires a new specific implementation for each endpoint. As there is already a file-based LDP implementation, Solid, it is easier for a developer to set up a decentralized application. To the best of our knowledge, there are no existing datapod technologies that allow query-based access. While there are many advantages to using file-based access, there is one feature that is exclusive to query-based access: performing complex intersections in a single query. This may, depending on the application, be a good reason to use query-based access over file-based access. In fact, through its better performance, query-based access can still compete with file-based access, especially when it is supported by easy-to-use user interfaces that abstract away the differences for end users.

Some of the topics that were handled during this thesis can still require some additional research. First, while advanced caching may solve some performance issues that occur with file-based access, it lacks an actual implementation. Furthermore, when a datapod’s ACL is not publicly visible, it is not

always possible to cache the datapod’s contents. This is a problem for both file-based and query-based access techniques. Future work can explore ways to increase performance without having to share the datapod’s ACL. Also, the advanced ACL for query-based access to annotations currently has the flaw of being application dependent. We proposed to use SHACL to solve this problem, but this was not implemented. Future research may explore other options, or implement SHACL in order to prove that it solved this problem. Finally, one of the biggest flaws of query-based access is the fact that each new endpoint requires some new endpoint-specific code. There may be ways to simplify this task, potentially even automating it.

This thesis proves that decentralization of social applications is perfectly feasible. In the future, a utopian internet in which everyone has their own datapod and where applications are reduced to interfaces of our data [3] may become reality. Nevertheless, some bridges still need to be crossed, like developing feasible business models, as decentralization does not come for free. Applications may charge the users to use their interface and there will be a need for good service providers who will host your data for a small fee. However, once most people grasp the potential of a decentralized social ecosystem and decide to switch, the application developers will follow. In the end, the competition for the best interfaces – now based on service quality instead of data ownership – will result in something that is essentially not very different from what we have now, but without many of its disadvantages.

## REFERENCES

- [1] S. Capadisli, A. Guy, R. Verborgh, C. Lange, S. Auer, and T. Berners-Lee, “Decentralised authoring, annotations and notifications for a read-write web with dokiel,” in *International Conference on Web Engineering*. Springer, 2017, pp. 469–481.
- [2] C.-m. A. Yeung, I. Liccardi, K. Lu, O. Seneviratne, and T. Berners-Lee, “Decentralization: The future of online social networking,” in *W3C Workshop on the Future of Social Networking Position Papers*, vol. 2, 2009, pp. 2–7.
- [3] R. Verborgh, “Paradigm shifts for the decentralized web,” 2017, <https://ruben.verborgh.org/blog/2017/12/20/paradigm-shifts-for-the-decentralized-web/>.
- [4] S. Capadisli, A. Guy, C. Lange, S. Auer, A. Sambra, and T. Berners-Lee, “Linked data notifications: a resource-centric communication protocol,” in *European Semantic Web Conference*. Springer, 2017, pp. 537–553.
- [5] J. Arwe, A. Malhotra, and S. Speicher, “Linked data platform 1.0,” W3C, W3C Recommendation, Feb. 2015, <http://www.w3.org/TR/2015/REC-ldp-20150226/>.
- [6] B. Young, P. Ciccarese, and R. Sanderson, “Web annotation vocabulary,” W3C, W3C Recommendation, Feb. 2017, <https://www.w3.org/TR/2017/REC-annotation-vocab-20170223/>.
- [7] D. Kontokostas and H. Knublauch, “Shapes constraint language (SHACL),” W3C, W3C Recommendation, Jul. 2017, <https://www.w3.org/TR/2017/REC-shacl-20170720/>.

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>  | <b>1</b>  |
| 1.1      | Social networks . . . . .                                      | 1         |
| 1.2      | Criticism of traditional social networking platforms . . . . . | 3         |
| 1.3      | Open Webslides . . . . .                                       | 4         |
| 1.4      | Structure of the thesis . . . . .                              | 5         |
| <b>2</b> | <b>Related literature</b>                                      | <b>6</b>  |
| 2.1      | Decentralized social networking platforms . . . . .            | 6         |
| 2.1.1    | Trusted personal server . . . . .                              | 7         |
| 2.1.2    | Peer-to-peer architecture . . . . .                            | 8         |
| 2.1.3    | Blockchain . . . . .   | 9         |
| 2.1.4    | Solving the silo problem . . . . .                             | 10        |
| 2.2      | Linked Data . . . . .  | 10        |
| 2.2.1    | Semantic Web . . . . .   | 10        |
| 2.2.2    | Web of Data . . . . .  | 11        |
| 2.2.3    | Publishing Linked Data . . . . .                               | 14        |
| 2.2.4    | RDF serialization formats . . . . .                            | 16        |
| 2.2.5    | Querying RDF: SPARQL . . . . .                                 | 18        |
| 2.3      | Semantic social networks . . . . .                             | 20        |
| 2.3.1    | Designing a distributed semantic social network . . . . .      | 20        |
| 2.3.2    | Web ID . . . . .   | 20        |
| 2.3.3    | Storage . . . . .  | 21        |
| 2.3.4    | Services . . . . .   | 21        |
| 2.4      | Linked Data Platform . . . . .                                 | 23        |
| 2.4.1    | Solid . . . . .  | 24        |
| 2.5      | Annotation systems . . . . .                                   | 25        |
| 2.5.1    | W3C Web Annotation Data Model . . . . .                        | 25        |
| 2.5.2    | Hypothesis . . . . .   | 27        |
| 2.5.3    | dokieli . . . . .  | 27        |
| <b>3</b> | <b>Problem description</b>                                     | <b>29</b> |
| 3.1      | Accessing Linked Data . . . . .                                | 29        |
| 3.2      | Goal of the thesis . . . . .                                   | 30        |
| <b>4</b> | <b>Analysis</b>  | <b>31</b> |

|          |   |           |
|----------|---|-----------|
| 4.1      | Decentralized Annotations . . . . .   | 31        |
| 4.1.1    | Annotations as Linked Data . . . . .  | 31        |
| 4.1.2    | Writing and reading annotations . . . . .   | 33        |
| 4.2      | Data manipulation . . . . .   | 33        |
| 4.2.1    | Retrieval of data . . . . .   | 34        |
| 4.2.2    | Manually manipulating data . . . . .  | 36        |
| 4.2.3    | Hosting public resources . . . . .  | 37        |
| 4.2.4    | Non-RDF data . . . . .  | 37        |
| 4.2.5    | Current Web . . . . .   | 38        |
| 4.3      | Detecting updates . . . . .   | 38        |
| 4.4      | Directories . . . . .   | 38        |
| 4.4.1    | Hierarchic structure . . . . .  | 38        |
| 4.4.2    | Directories in file-based and query-based servers . . . . .                               | 39        |
| 4.5      | Intersections . . . . .   | 40        |
| 4.5.1    | File-based intersections . . . . .  | 40        |
| 4.5.2    | Intersections over a single graph . . . . .   | 40        |
| 4.6      | Access control . . . . .  | 41        |
| 4.6.1    | File-based ACL . . . . .  | 41        |
| 4.6.2    | Graph-based ACL . . . . .   | 42        |
| 4.6.3    | Development of an ACL extension . . . . .   | 42        |
| 4.6.4    | Adding advanced ACL to the SPARQL server . . . . .  | 45        |
| 4.6.5    | Adapting the ACL extension for annotations . . . . .                                      | 47        |
| 4.7      | Caching . . . . .   | 49        |
| 4.8      | Versioning . . . . .  | 49        |
| <b>5</b> | <b>Implementation</b>   | <b>50</b> |
| 5.1      | Test setup . . . . .  | 50        |
| 5.1.1    | Setup with file-based access . . . . .  | 50        |
| 5.1.2    | Setup with query-based access . . . . .   | 51        |
| 5.1.3    | Generator . . . . .   | 52        |
| 5.2      | Implementation of an inbox listener for file-based access . . . . .                       | 53        |
| 5.3      | Implementation of a directory structure for the single-graph query-based server . . . . . | 53        |
| 5.4      | Demonstrative setup for intersections . . . . .   | 54        |
| 5.5      | Annotation Plugin . . . . .   | 56        |
| 5.5.1    | Inspiration . . . . .   | 57        |
| 5.5.2    | User interface . . . . .  | 57        |
| 5.5.3    | Highlighter . . . . .   | 57        |
| 5.5.4    | Comments . . . . .  | 58        |
| 5.5.5    | Loading annotations . . . . .   | 58        |
| 5.5.6    | Query-based annotation plugin . . . . .   | 60        |
| 5.5.7    | Possible extensions . . . . .   | 61        |
| <b>6</b> | <b>Evaluation</b>   | <b>62</b> |

|          |   |           |
|----------|---|-----------|
| 6.1      | Performance . . . . .   | 62        |
| 6.1.1    | Storing annotations . . . . .   | 62        |
| 6.1.2    | Loading annotations . . . . .   | 64        |
| 6.1.3    | Loading specific annotations . . . . .                                  | 64        |
| 6.1.4    | Conclusion . . . . .  | 66        |
| 6.2      | Annotation plugin for Open Weblides . . . . .                           | 66        |
| 6.3      | Comparing the access techniques in the context of annotations . . . . . | 66        |
| 6.3.1    | Difficulty of implementation . . . . .                                  | 67        |
| 6.3.2    | Retrieval performance . . . . .   | 67        |
| 6.3.3    | State of research . . . . .   | 67        |
| 6.3.4    | Implementing update detection . . . . .                                 | 67        |
| 6.3.5    | Weighing the pros and cons . . . . .                                    | 68        |
| 6.4      | Decentralized social applications in general . . . . .                  | 68        |
| <b>7</b> | <b>Conclusions</b>  | <b>69</b> |
| 7.1      | SWOT analysis for decentralization using Linked Data . . . . .          | 69        |
| 7.1.1    | Strengths . . . . .   | 69        |
| 7.1.2    | Weaknesses . . . . .  | 69        |
| 7.1.3    | Opportunities . . . . .   | 69        |
| 7.1.4    | Threats . . . . .   | 69        |
| 7.2      | Relevance . . . . .   | 70        |
| 7.3      | Conclusion . . . . .  | 70        |
|          | <b>Bibliography</b>   | <b>72</b> |
|          | <b>Appendix A Using advanced ACL for annotations</b>                    | <b>77</b> |
|          | <b>Appendix B JavaScript code for routing and handling requests</b>     | <b>81</b> |



# List of Figures

|      |   |    |
|------|---|----|
| 1.1  | Monthly active users on social media . . . . .                  | 2  |
| 1.2  | Facebook user growth . . . . .                                  | 2  |
| 1.3  | Social network silos . . . . .                                  | 4  |
| 1.4  | Open Weblides example . . . . .                                 | 5  |
| 1.5  | Open Weblides co-creation . . . . .                             | 5  |
| 2.1  | Framework of decentralized social network . . . . .             | 7  |
| 2.2  | Components of Safebook . . . . .                                | 8  |
| 2.3  | Decentralized privacy using blockchain technology . . . . .     | 9  |
| 2.4  | RDF triple . . . . .  | 11 |
| 2.5  | Linked Open Vocabularies . . . . .                              | 12 |
| 2.6  | BBC Linked Data . . . . .                                       | 13 |
| 2.7  | Growth of LOD cloud . . . . .                                   | 14 |
| 2.8  | 303 redirects . . . . .   | 15 |
| 2.9  | Architecture of a distributed semantic social network . . . . . | 20 |
| 2.10 | Linked Data Notifications . . . . .                             | 23 |
| 2.11 | Solid architecture . . . . .                                    | 24 |
| 2.12 | Solid pod architecture . . . . .                                | 24 |
| 2.13 | Annotation Model . . . . .                                      | 26 |
| 2.14 | Architecture of dokieli . . . . .                               | 27 |
| 4.1  | Basic annotation structure . . . . .                            | 32 |
| 4.2  | Solid editing interface . . . . .                               | 37 |
| 5.1  | Annotation generator . . . . .                                  | 52 |
| 5.2  | Annotation plugin architecture . . . . .                        | 56 |
| 5.3  | GUI of the annotation plugin. . . . .                           | 57 |
| 5.4  | Posting an annotation . . . . .                                 | 58 |
| 5.5  | Loading annotations . . . . .                                   | 59 |
| 6.1  | Test: Storing annotations . . . . .                             | 63 |
| 6.2  | Test: Loading annotations . . . . .                             | 65 |
| 6.3  | Test: Loading specific annotations . . . . .                    | 65 |
| 6.4  | Prefix-exact-suffix . . . . .                                   | 66 |

# List of Tables

- 2.1 SPARQL query results on example data . . . . . 19
  
- 6.1 Time required to store a number of annotations using the file-based and query-based test setups. . . . . 63
- 6.2 Time required to collect a number of annotations using the file-based and query-based test setups. . . . . 64
- 6.3 Time required to collect a specific number of annotations out of a total of 200 annotations using the file-based and query-based test setups. . . . . 64

# List of abbreviations

**ACL** Access Control List

**API** Application programming interface

**CRUD** Create, Read, Update, Delete

**DSSN** Decentralized Semantic Social Network

**HTML** Hypertext Markup Language

**HTTP** Hypertext Transfer Protocol

**JSON** JavaScript Object Notation

**LDN** Linked Data Notifications

**LDP** Linked Data Platform

**pod** Personal on-line datastore

**REST** Representational State Transfer

**RDF** Resource Description Framework

**RDFa** RDF in attributes

**SHACL** Shapes Constraint Language

**SPARQL** (*Recursive acronym for*) SPARQL Protocol and RDF Query Language

**UI** User Interface

**URI** Uniform Resource Identifier

**URL** Uniform Resource Locator

**W3C** World Wide Web Consortium

**WAC** Web Access Control

# List of namespaces

**as:** <http://www.w3.org/ns/activitystreams#> (Activity streams)

**acl:** <http://www.w3.org/ns/auth/acl#> (Web Access Control)

**dc: or terms:** <http://purl.org/dc/terms/> (Metadata terms)

**foaf:** <http://xmlns.com/foaf/0.1/> (Social relations)

**ldp:** <http://www.w3.org/ns/ldp#> (Linked Data Platform)

**oa:** <http://www.w3.org/ns/oa#> (The Web Annotation Data Model)

**rdf:** <http://www.w3.org/1999/02/22-rdf-syntax-ns#> (RDF namespace)

**rdfs:** <http://www.w3.org/2000/01/rdf-schema#> (Core data modeling vocabulary)

# Chapter 1

## Introduction

### 1.1 Social networks

Nowadays, most people live their lives alongside an online image of themselves. Platforms such as Facebook, Instagram and Twitter are some of today's most popular social networks. All three of these platforms allow their users to create an account, also known as Internet persona, and become friends with other users, follow products and celebrities, share photos, talk to each other, and a lot more. Thanks to the Internet, we are more connected than ever.

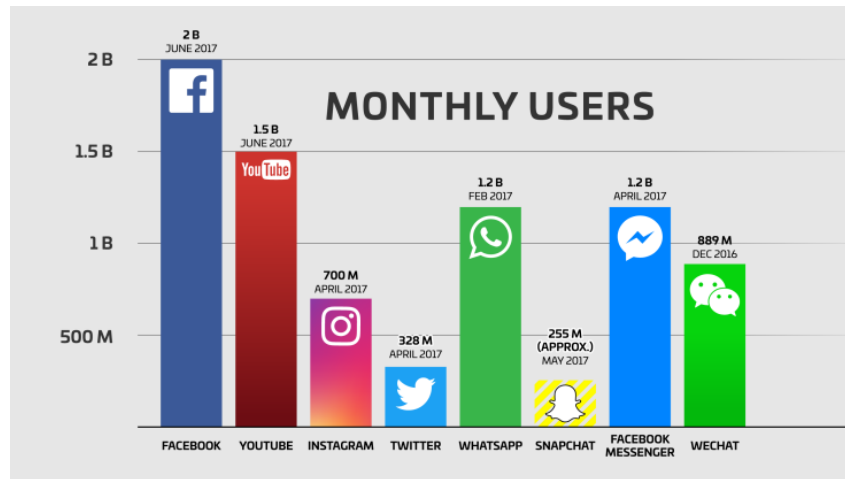
But why do we need these online social networks? Humans have lived thousands of years without them, building up their own offline social network by writing down addresses, visiting and sending 'snail' mail<sup>1</sup>. This last term already suggests the most important reason: speed of communication. It allows people to meet other people all around the world in a matter of seconds: friends, family, people that share your interests, bloggers, etc. Often, long lost friends have once again found each other through social media. It is not just a recreational concept: professionals can, for instance, use LinkedIn as a means to improve their career chances. All popular platforms are user friendly, allowing less tech savvy people to enjoy all of these benefits. And the best part? Participation is usually free!

But social networks have a much broader meaning. According to Scott [74], the idea of social networks in sociology includes face-to-face relationships, political associations and connections, economic transactions among business enterprises, and geopolitical relations among nation states and international agencies. In fact, any online platform where you cooperate with other people is considered part of your social network. For instance, Google Docs is an alternative to Microsoft Word, allowing people to write on the same document simultaneously. Doodle enables a group to schedule the best possible date for a meeting, and online forums are social platforms to bring together people with similar interest. Harris [40] states that there are hundreds of different types of social media platforms.

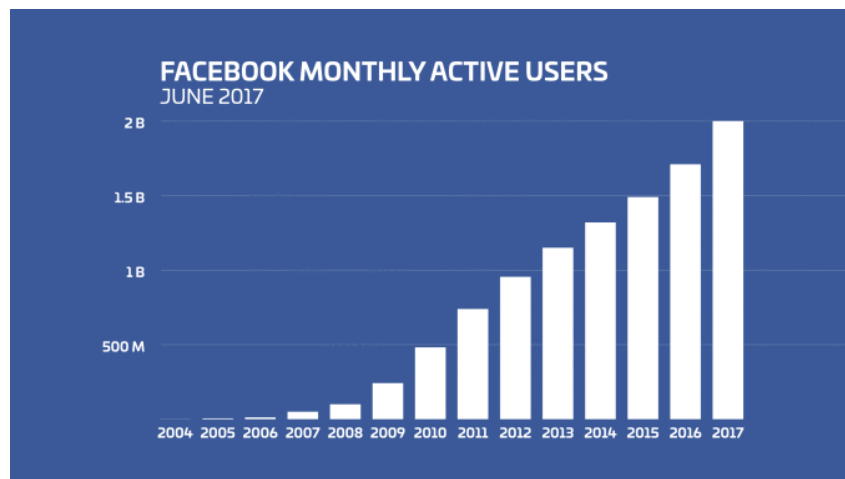
Clearly, social networks offer many advantages. But is it really this perfect innovation it appears to be? There is one core observation that suggests an answer to this question. The platforms are usually free, which means that the hosting and development costs should be covered not by the users, but by someone else. One option could be that social networks are sponsored by the government, but this is not the case. In fact, according to Shirky [76], a government should maintain Internet freedom as a general goal, not as a tool for achieving immediate policy aims in specific countries. Therefore, most traditional social network platforms are government independent. Another option is offering an improved experience through a subscription model. In a similar model, the social networks are giving users the ability to purchase virtual goods. This is commonly known as microtransactions. For instance, Facebook offered a service to send virtual gifts to friends, along with a personalized message, for a small fee. Also, many on-site games, often called Social Network Games, or SNGs (e.g. CityVille, Texas Hold'em Poker, Farmville, etc.), offer in-game advantages or cosmetic upgrades as microtransactions [53]. Facebook and Twitter started out with

---

<sup>1</sup>Snail mail is a retronym for mail, distinguishing postal mail from electronic mail.



**Figure 1.1:** Chart showing the monthly users on some of current day's most popular social network platforms. (Source: <https://techcrunch.com/2017/06/27/facebook-2-billion-users/>)



**Figure 1.2:** Facebook monthly active users per year since 2004. (Source: <https://techcrunch.com/2017/06/27/facebook-2-billion-users/>)

funding from venture capitalists [25], raising several million dollars each. These investors expect that those sites will become profitable in the future. The way this is most commonly done, is through advertising. Companies can promote their products by buying advertisements (ads). Because many social network sites have millions of daily users, their ads can reach many potential customers. More importantly, ads can be targeted towards certain profiles, improving the efficiency of their marketing campaign. This is made possible by analyzing the one thing users give back to the social networking platforms: data. According to Casteleyn et al. [29], profile pages of people are not always a correct illustration of themselves, but are often shaped to how they want to be perceived by others. For this reason, data could be seen as a crystal ball for future consumer intentions. Besides using data for advertising, data is commonly sold to third parties. The size of these social networks become increasingly bigger. In 2008, Facebook surpassed the user count of MySpace, which was 109 million visitors per month [33]. On the 27th of June, 2017, Mark Zuckerberg notified<sup>2</sup> the world that Facebook officially reached more than two billion monthly active users. The number of monthly users on some of the most popular social network platforms is shown in Figure 1.1, and the growth of Facebook is shown in Figure 1.2. Having this much data is an invaluable asset to sell to third parties for market predictions. This is made clear by Wolf [87]: “Data is just as likely to be as big a piece of the business equation as advertising. Facebook, LinkedIn, MySpace and Twitter

<sup>2</sup><https://www.facebook.com/zuck/posts/10103831654565331>

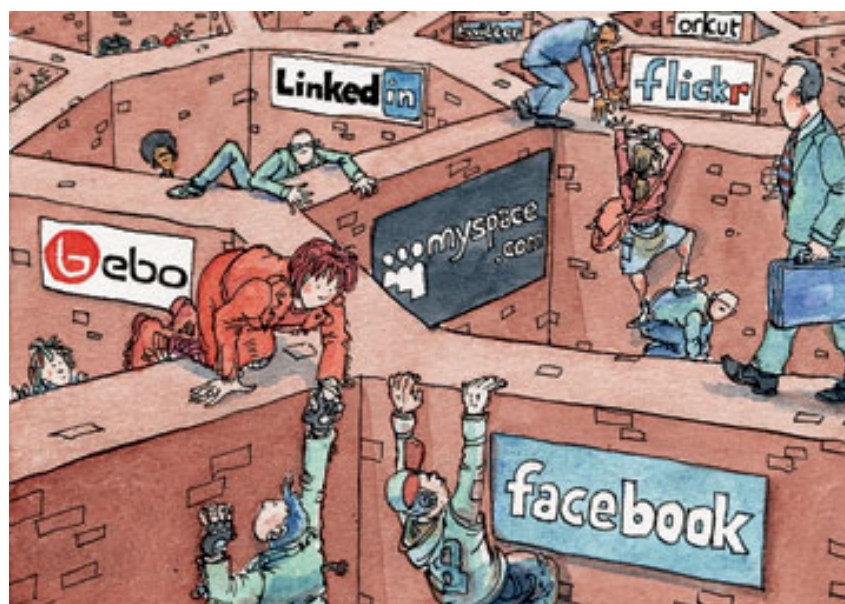
already possess billions of pieces of useful data that can infer so much about what's happening – and what is going to happen – in consumer society.”

## 1.2 Criticism of traditional social networking platforms

Many people think that ads are sufficient to generate revenue. However, a 2009 comScore study showed that only 16% of all visitors ever click on an ad, and 8% account for 85% of all clicks [2]. Today, more and more social networking sites try to profit from licensing their users' data. In 2016, Twitter reported a third-quarter revenue of \$616 million, of which \$71 million was generated from data licensing [65]. Because of the vast amount of data, social networking sites are an attractive target for many organizations who need large amounts of user data. As research has shown, it can be used to positively impact business decision making [39]. An assumption frequently made by social networks is that anonymous data is sufficient to protect privacy. Backstrom et al. [8] and Narayanan and Shmatikov [59] have shown that it is still possible to de-anonymize data, even when all names were removed from the data. Often, this data is used for good intentions. For instance, while still controversial, the law enforcement agencies in The Netherlands have used social networking data to investigate organized crime [47]. Unfortunately, some of these organizations have malicious intentions, for example identity theft, targeted scams, stalking and more [42]. Many of these criminals acquire their data through crawling. Since social platforms have acquired a poor reputation for privacy, crawling is surprisingly easy, as Bonneau et al. [22] demonstrated for Facebook. The quickest way of collecting data is by crawling Facebook's public profile listings, which consist of a person's name, picture, group memberships and links to eight friends. Note that these listings are *public*, i.e. you do not need an account to view them. Although the listings are optional, less than 1% of users change the default setting, which enables the listings [21]. Even better for crawling is making use of valid Facebook accounts. This way, you can see *all* friends of users who use the default privacy settings. Even when not using the defaults, Facebook has been notorious for a large disparity between desired and actual privacy settings. For every piece of content you share, the uploader can choose who is able to access his content. Facebook offers five granularities: only the uploader, a specific list of friends, all friends, all friends of friends, and everyone. Despite this perceived control of information, a survey conducted by Liu et al. [54] show that 39% of users that have changed their default settings, are having difficulty maintaining their privacy settings correctly. More importantly, half of the users do not care and keep using the default settings (share with everyone).

Another problem of modern day's society is a phenomenon called “The Filter Bubble”. It was first described by Eli Pariser [60], and although the problem is not unique to social networks, it does play a large role in how we experience social networking platforms. In his book, he googles the exact same keywords on two different laptops, and retrieved vastly different results. This is caused by *personalization*, a core strategy for almost every top site on the Internet. Using data collected by the platform, relevant new data can be generated (e.g. personalized ads). This data is mostly collected without the user knowing how, which leads to popular conspiracy theories. For instance, some believe the Facebook app is listening to your conversations and storing important keywords [56]. Personalization shapes what we buy, but does more than that. It starts to orchestrate our lives. 36% of Americans under thirty exclusively use social networking sites to catch up on news [60]. The algorithms running on these sites try to personalize what you see, which results in a unique universe of information for each of us. It fundamentally changes the way we encounter ideas and information. What the algorithm presents to you, is biased, subjective, and not always true. But from within the bubble, it is difficult to notice how biased it actually is. As Pariser says: “It is a cozy place, populated by our favorite people and things and ideas”. Inside the bubble, chance encounters that bring insight and learning are rare. Coming into contact with paradigms that completely change the way we think about the world and ourselves, becomes near impossible if personalization is too acute. Unfortunately, we usually do not have a choice to enter the bubble.

An additional problem are the growing number of information silos. The information that is stored on one traditional social network, cannot easily be transferred to another social network. This information includes



**Figure 1.3:** Drawing which illustrates the silo problem. People are stuck within particular social networking platforms. They would like to benefit from other social networking platforms and share the data from one platform with people on a different platform. (Source: David Simonds, [www.Economist.com](http://www.Economist.com), 2008)

messages, friends, photos, personal details, and more. The two platforms can be completely different. For example, you could want to use your facebook name and pictures to sign up for a dating website. When using a different social network, one might want to reuse everything that is stored on another social network, but few robust mechanisms to do so exist. This problem is illustrated in Figure 1.3. Fitzpatrick and Recordon [36] summarized this as follows: “People are getting sick of registering and re-declaring their friends on every site.” The main cause of this development is the fact that it is simply much easier to manage, publish and search large amounts of similarly structured content using a centralized platform [27]. The information that is stored in the silo, can be used by the silo owners for whatever they want, as long as it corresponds with the agreed policies [89]. This is usually stated in the end user license agreement, which is unfortunately seldom read. Furthermore, they reserve the right to change their EULA at any time. Therefore, many people do not know what data is being used, and for what purposes. Most social networks provide the option to deactivate your account, however it is often not possible to completely erase all personal information from the site [5]. With fewer than 1% of the servers serving over 99% of the content, the current phenomenon of information silos goes against the original intention of the Web to become a decentralized platform [27].

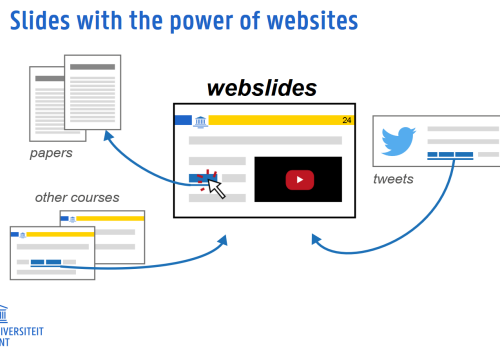
### 1.3 Open Webslides

A goal of this thesis is to study and develop a decentralized solution for annotations. Since annotations are posted and observed by multiple users, it is a social application. In particular, the solution will be tested on Open Webslides’ annotations<sup>3</sup>. This free platform enables teachers to create interactive slides. For instance, one can easily embed a YouTube video into a slide. The teacher can also flip the classroom by asking questions to the students, which they can answer by using an app<sup>4</sup>. In fact, anything on the Web can be added to the slides by using iframes, which display the content live, as if you were on the website itself, as shown in Figure 1.4. The platform also allows easy co-creation of slides, paving the way for a good cooperation between teacher and students. This is illustrated in Figure 1.5. Since the slides are Web-based, they are easy to share and can be opened on any device with a Web browser. Updates

<sup>3</sup><http://openwebslides.github.io>

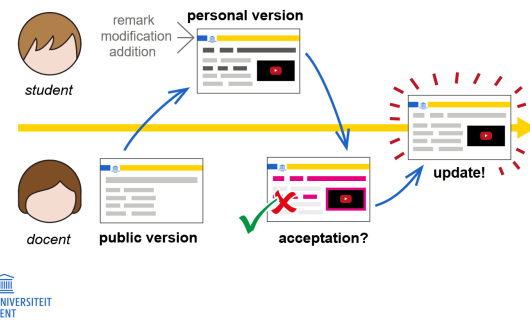
<sup>4</sup>For example: [www.menti.com](http://www.menti.com)





**Figure 1.4:** An example of an Open Webslide which shows some of its possibilities. (Source: [https://openwebslides.github.io/cocos\\_kickoff/#webslides](https://openwebslides.github.io/cocos_kickoff/#webslides))

### Co-creation via open-source



**Figure 1.5:** The process of co-creation, as shown on an Open Webslide. On the website itself, this slide is animated in multiple steps. (Source: [https://openwebslides.github.io/cocos\\_kickoff/#open-source](https://openwebslides.github.io/cocos_kickoff/#open-source))

are retrieved upon a page refresh, and since each slide has its own URL, you are able to link to specific slides. The slides are created using HTML. In case one does not know how to write HTML, an editor is available. The advantage of HTML is the concept of content before layout. Content can for instance easily be transformed from slides into course notes, without having to rewrite your HTML code.

The annotations will be able to precisely point to the relevant content. They will be stored as Linked Data, following the Solid conventions. In the rest of this thesis, these concepts will be covered extensively.

## 1.4 Structure of the thesis

The thesis starts with a review of articles that influence this work. It includes an introduction to the technologies that will be used. The next chapter describes the problem and research questions. The hypotheses are stated at the end of this chapter. Then, a largely theoretical analysis of the differences between file-based and query-based access techniques is given. These differences are then put to the test in a chapter about the various setups that were developed. Using the knowledge that was gathered during all previous chapters, a new chapter evaluates the differences in the context of annotations and evaluates whether decentralized annotations are a viable alternative to centralized annotations. The chapter ends by comparing annotations to social applications in general. Finally, a SWOT analysis for decentralization using Linked Data is given, along with a conclusion.

## Chapter 2

# Related literature

In this chapter, a survey of relevant publications is given, most of which is crucial to understand the rest of the thesis. It is important to have knowledge of previous contributions to the fields of decentralized social networks, Linked Data and annotation systems in order to understand the position of this thesis in these fields.

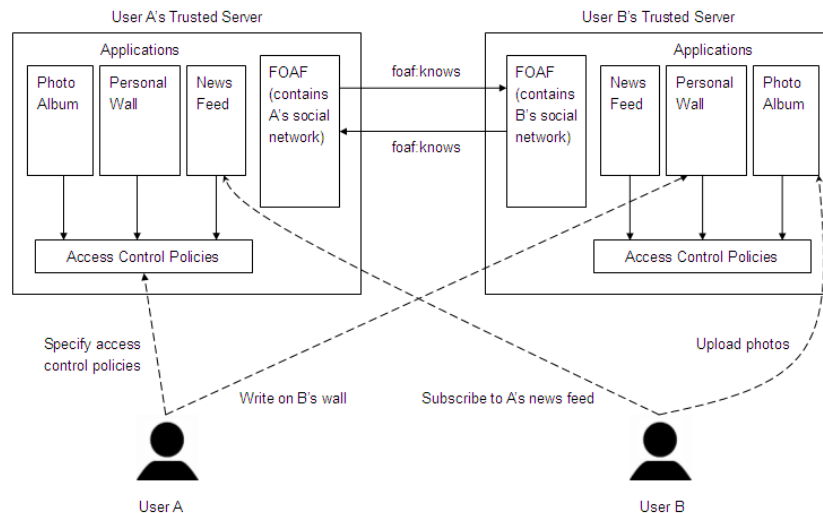
### 2.1 Decentralized social networking platforms

Some of the problems stated in section 1.2 can be avoided by using decentralized technologies. The definition of decentralization according to Rohit Khare [46] is as follows: “*A decentralized system is one which requires multiple parties to make their own independent decisions.*” In this context, the information that is used by social network platforms, will not be stored on the platform provider’s servers. Instead, it could for instance be stored on a server of your choice, or fragments of your information can be stored on multiple different servers. There are no restrictions on what you want to store. Figure 2.1 shows a simple illustration of a decentralized framework. According to Yeung et al. [89], users will be given back control in following three aspects thanks to decentralization:

- **Privacy:** Users can decide who has access to their information.
- **Ownership:** If the information is stored on a trusted server (e.g. your local computer), users are the sole owners of their information. Your data cannot legally be licensed to other parties, unless you give consent.
- **Dissemination:** Data can be disseminated however a user wants.

Decentralization does come with multiple technical challenges. For instance, in a centralized social network, all profile pictures are stored in the same format. However, in a decentralized social network, users have the freedom to store their profile picture in whatever format they want. There is no central authority that enforces a specific format. This means that decentralized social applications should be able to handle multiple formats, and use a default profile picture if no compatible image is found. Verifying reverse links is another challenge. It is easy for someone to store a relationship on his profile, saying he is friends with another person. However, this might not necessarily be true. Robust verification mechanisms should be provided to see if the friendship is mutual.

Besides the issues that concerns decentralized social network networks, decentralization in itself comes with some risks [38]. For instance, centralized servers are usually very strong at defending against attacks (e.g. DDoS attacks), while smaller servers (as is typically the case for decentralized servers) are much more prone to attacks, due to the fact that they are often run by hobbyists with small budgets. Another issue that could arise is the illusion of control when people blindly trust their data to e.g. a friend’s server. The fact that it is possible to move around your data, does not mean it is safe to do so. Illusion of control



**Figure 2.1:** A framework of decentralized online social networking (Source: Yeung et al. [89] )

can also be an issue in the context of sharing data. Websites like Facebook are still able to connect the dots and consume your data (although not necessarily in a legal manner), which remains problematic when oversharing.

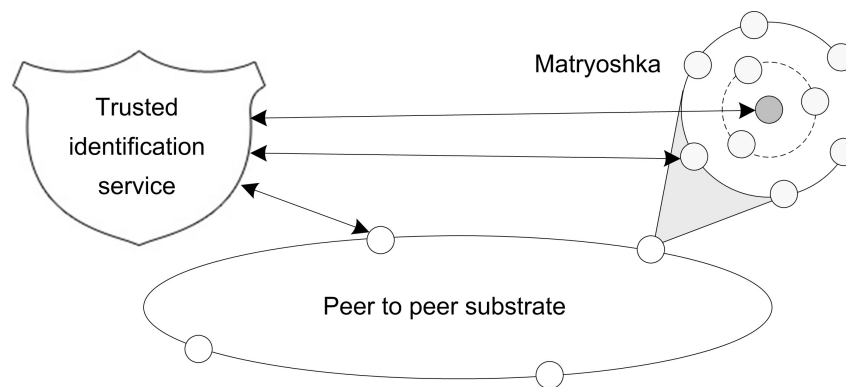
### 2.1.1 Trusted personal server

Diaspora [16] was one of the first decentralized social networks. It was launched in November of 2010. It differentiates itself from traditional social networks by allowing users to store their profile on any Diaspora server, or *pod*, they want. Some users desire complete control and host their own server, while others join an existing server. The server makes use of *push* design to communicate with other servers. For instance, sharing a message on your wall will prompt the server to push this message to the servers of all your friends. Although there is no central server, the security and integrity of your information is still in the hands of the server administrators. Because of the push mechanism, your information is also pushed to other servers, meaning that you should trust both your own server's administrators, as your friend's server's administrators. This could potentially lead to a privacy leak. Besides, many people do not have the resources to set up their own server and choose to join an existing server. Well-known servers with reliable uptime are the best candidate-pods, and this comes with consequences. On April 2014<sup>1</sup>, the three largest pods hosted over 92% of all users, with the largest pod hosting over 75% of all users. This unbalanced population combined with the fact that pods are application-specific makes clear that, although Diaspora had decentralized intentions, it is still quite similar to a centralized social network.

A different approach to decentralization was taken by Seong et al. [75] in the development of PrPI, short for private-public. It is based on a person-centric architecture. Similar to the pods in Diaspora, each individual stores his digital assets in a *Personal-Cloud Butler* service. The main difference between PrPI's Butler and Diaspora's pods is that PrPI only allows one user per Butler. One can choose to run this service on his own server, or contact a paid or ad-supported vendor of his choice. It is also possible for an individual to store his data on multiple Butlers, either duplicated, or disseminated. For authentication, the decentralized OpenID [64] management system is used. One of their main research goals was to create an API that allows the same application to be hosted on multiple domains, while still using the same, decentralized data. The data itself is stored in a format based on RDF (Resource Description Framework<sup>2</sup>, see section 2.2) and is optionally encrypted. It can be queried by using their new database query language SocialLite, which is an extension of Datalog. PrPI was a proof-of-concept and was not

<sup>1</sup><https://web.archive.org/web/20140409064653/http://pods.jasonrobinson.me:80/>

<sup>2</sup><https://www.w3.org/RDF/>



**Figure 2.2:** Components of Safebook (Source: Cutillo et al. [32])

optimized for speed, nor was it publicly deployed. Nevertheless, experiments proved that their approach on decentralization is viable. The platform succeeds in all three pillars of decentralized social networks: privacy, ownership and dissemination.

### 2.1.2 Peer-to-peer architecture

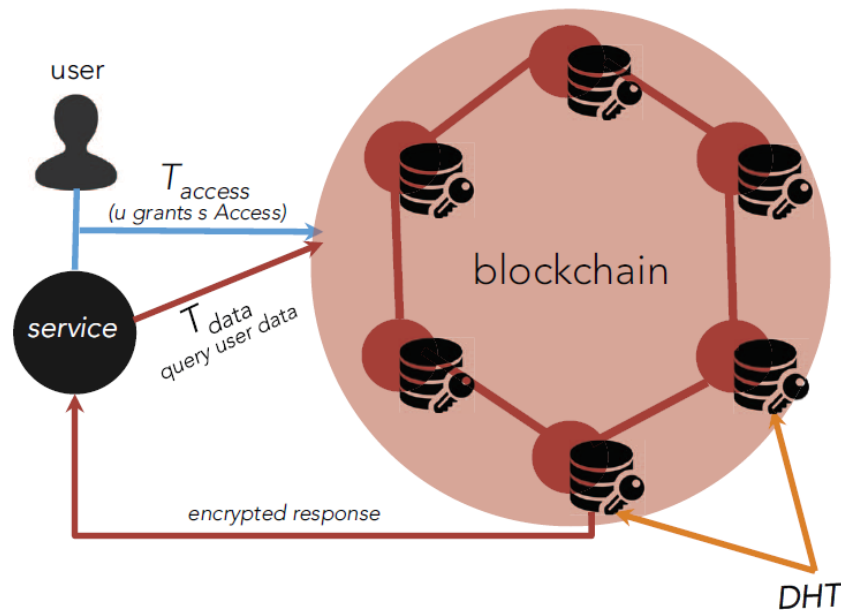
Previous models rely on trusted servers to host their data, but still abide to classic client-server techniques. A fully decentralized platform should not provide an option for users to trust a third-party to store their data. In a peer-to-peer (P2P) architecture, there are no servers, instead every user (or peer) hosts his own data and possibly caches data for other users. Any architectural design for social networks comes with multiple security requirements [32, 42, 61]:

- End-to-end confidentiality is required to make sure that only the requesting and responding parties can access the exchanged data. Peers along the path should not be able to conduct a man-in-the-middle attack.
- Access control requires proper authentication of members to avoid impersonation attacks.
- Privacy should be ensured to keep personal information confidential.
- Data integrity and availability are the other two rules in the CIA triad<sup>3</sup>. The former guarantees that profile data cannot be tampered with. The latter ensures that the stored data is accessible at any time.

Safebook is a P2P social network first described by Cutillo et al. [31] and meets the aforementioned requirements by design. Safebook consists of three components (Figure 2.2) [32]. First, Matryoshkas are layered structures composed of various nodes in the P2P network. Every user is the core node of his own matryoshka, but can be part of other user's matryoshkas. The first layer (*inner shell*) around the core node corresponds with the nodes belonging to the trusted contacts of the core node's user. The next layer are all nodes that are trusted by the users in the previous layer, and so on. To provide availability, nodes on the inner shell are able to cache the core node's data, and provide it if the core node is offline. A peer's identity is only known by his trusted contacts, since they are the only ones that are directly linked to his core node, and thus know his IP address. To join Safebook, you need to be invited by an existing member in order to build up your own matryoshka. The peer-to-peer substrate is the second component of Safebook and comprises all nodes and provides data lookup services such as a DHT<sup>4</sup>. Finally, the trusted identification service (TIS) provides each node with a unique pseudonym and identifier, along with related certificates, to protect against impersonation attacks. Data can either be private, protected or

<sup>3</sup>Confidentiality, integrity and availability. See <http://resources.infosecinstitute.com/cia-triad/>

<sup>4</sup>Distributed Hash Table



**Figure 2.3:** Basic principles of the decentralized privacy model described by Zyskind et al. [91]. A service can be a mobile application requesting user information for personalized ads. The information itself is stored in a distributed hash table (DHT), while the (hashed) keys to this information are stored in the blockchain itself. (Source: Zyskind et al. [91])

public. The core nodes will keep their private data, while duplicating protected and public data to the inner shell's nodes. Data can be retrieved from outside the inner shell by routing according to the standard P2P protocol. Performance-wise, three to four shells is sufficient for a matryoshka. In this case, and with 23 trusted contacts (i.e. nodes in his inner shell), an accessibility of 90% is obtained. Also, the data lookup delay was simulated to be below 13.5 seconds for 90% of all requests. Unfortunately, the slow and unreliable nature of this network is a major drawback.

### 2.1.3 Blockchain

Recently, Bitcoin has become increasingly popular. This cryptocurrency was built on blockchain technology, which can be described as a distributed database, or more commonly as a distributed ledger. Besides cryptocurrencies, blockchain has enormous potential for anything that desires independence from an authority. Indeed, thanks to blockchain's design, a third party becomes unnecessary.

It can also be used in the context of decentralized social network applications. One question that is immediately clear, is how to solve the problem of privacy. If everyone can see everything on the ledger, our private messages would be visible to the world. An attempt to solve this problem was made by Zyskind et al. [91] by using a combination of blockchain and off-blockchain storage. Data ownership is ensured by storing the data in an off-blockchain database of your choice. Access rights to this information is stored on the blockchain. Upon a data request by anyone (a user or a service), the blockchain checks if the entity has access rights. If so, it returns an encrypted response to the requester containing the location of the information. Only entities with access rights are able to decrypt the location. Figure 2.3 illustrates this principle.

For blockchain technology to be tamper-free, a sufficiently large network of peers is required. Because of the principal blockchain properties, it is not possible for adversaries to fake their identity or corrupt the network. In this model, it is also not possible for an adversary to learn anything from the blockchain, since the location of information is encrypted. Even when a node controls a database where the actual information resides, the adversary is still unable to learn anything, since the information is encrypted as well. Unfortunately, blockchain does come with drawbacks. By using the Proof-of-Work (PoW) consensus

model [58], a node's voting strength is solely based on computational resources. The consequences of this are excessive energy consumption, high latency and potential sybil<sup>5</sup> attacks. These drawbacks show that running blockchain nodes is not cheap. Social networks that rely on blockchain technology should thus come up with incentives for people to run these nodes. Also, to run a responsive social network, blocks should be added to the blockchain at a high rate. Unfortunately, traditional blockchains (e.g. Bitcoin and Ethereum) are limited to three transactions (not blocks!) per second [3]. Therefore, different consensus models than PoW can be used. One algorithm is called Delegated Proof of Stake (DPoS) [51]. Steem<sup>6</sup> [52] is an example of a blockchain-based social network that uses DPoS instead of PoW. In functionality, it is a message board comparable to Reddit<sup>7</sup>, which means that millions of changes (or transactions) happen each hour. Thanks to DPoS, a new block is added to the blockchain every three seconds, with each block containing many transactions. As the network is scalable to support 10,000 transactions per second, the Steem blockchain already contains more transactions than both Bitcoin and Ethereum.

### 2.1.4 Solving the silo problem

All of the solutions stated above have one thing in common: the data is still only compatible with its corresponding social network platform. The silos in Figure 1.3 remain a problem that we would like to deal with. Information should only be stored once, and be usable by all social network platforms. Tramp et al. [81] envisions this solution as an ocean, with big and small islands, and a tight network of bridges and ferries connecting these islands. Linked Data technology can be used to solve this problem, and is the main topic of this thesis. Therefore, the concept of linked data is explained in the next section, followed by a section about recent advancements in building semantic social networks.

## 2.2 Linked Data

### 2.2.1 Semantic Web

To understand the concept of Linked Data, one should first become familiar with Berners-Lee's vision about the Semantic Web [15]. In this extension of the current web, the new Web 3.0 gives meaning to information on the Internet such that not only humans, but also computers are able to understand it. This requires a language that expresses structured collections of information and inference rules to enable automated reasoning. The Resource Description Framework (RDF) is a fundamental component of the Semantic Web. It expresses meaning by encoding triples of subject, predicate and object (see Figure 2.4). For example, cats are fond of humans, so it could be structured as following triple:

- **Subject:** Cat
- **Predicate:** Likes
- **Object:** Human

However, good RDF triples require subject and object to be a Universal Resource Identifier (URI) [48]. This is similar to a link you would find on a web page. In fact, such links are called URLs (Uniform Resource Locator) and are the most common type of URI. Predicates are always URIs, as they have to express a clear relationship that can be used in multiple triples. A good RDF triple would thus be:

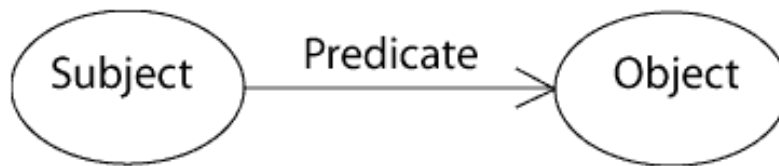
- **Subject:** <http://dbpedia.org/ontology/Cat>

---

<sup>5</sup>In a sybil attack, a peer majority is obtained by forging many identities. In the context of PoW blockchains, it means that an attacker has over 50% of the computational power.

<sup>6</sup><https://steem.io/>

<sup>7</sup><https://www.reddit.com/>



**Figure 2.4:** Graphical representation of an RDF triple. (Source: Klyne and Carroll [48])

- **Predicate:** <http://contextus.net/ontology/ontomedia/ext/common/trait#likes>
- **Object:** <http://purl.org/biotop/biotop.owl#Human>

The main advantage of expressing nodes and their relationships as URIs is to make it easier for computers to draw unambiguous conclusions. For instance, the word “break” has 75 different meanings<sup>8</sup>, such as a pause from doing something, the act of breaking something, a separation, or even a score consisting of winning a game when your opponent was serving in tennis. For each of these different meanings, a different URI should be used. To demonstrate this problem of ambiguous terms, we could use “break” in a triple (Prison guard, like, break). Prison guards do like breaks (as in a small pause from doing work), but they also hate breaks (as in a prison break). Besides having the need to discriminate between meanings, the URIs should also be chosen such that two terms with the same meaning use the same URI, such that computers are able to reason about the data. Even more reasoning is possible when inference rules are also added to the RDF document. For instance, if a person has two parents and a (full) brother, it can be inferred that the parents of the brother are the same people. With these mechanisms in place, an algorithm is able to read a web page without having to use advanced artificial intelligence techniques to understand what is stated. This offers multiple advantages: web search results are improved by looking at concepts instead of ambiguous keywords, and queries can be executed to quickly find answers to questions that require some reasoning (e.g. knowing the address, what is its state code?). Even more complicated questions can be tackled when pages are linked to each other. This way, an answer that does not reside on just one page, can still be found by following links to other web pages, and reasoning on all the collected data. However, unambiguous reasoning on different data sets (from various authors) is only possible if the selected URIs for the same concepts are identical over all datasets. For this reason, the concept of vocabularies<sup>9,10</sup> was introduced. They define the concepts and relationships concerning a specific topic. Multiple datasets can then use these standard terms to resolve ambiguities. To help in finding the right URI (if any exist), Linked Open Vocabularies (LOV, <http://lov.okfn.org/dataset/lov>) provides a search tool. A representation of the vocabularies in LOV is displayed in Figure 2.5. The most common vocabulary is *dcterms*, which is short for Dublin Core Metadata Initiative (DCMI) Metadata terms [44]. It is a set of terms that describe web resources (e.g. images, web pages, videos) and physical resources (e.g. books, CDs). The initial version, the Dublin Core Metadata Element Set, contained just 15 terms, including *title*, *creator*, *subject*, *description*, etc. Their broad suitability explain why these metadata elements are so often used in RDF documents.

## 2.2.2 Web of Data

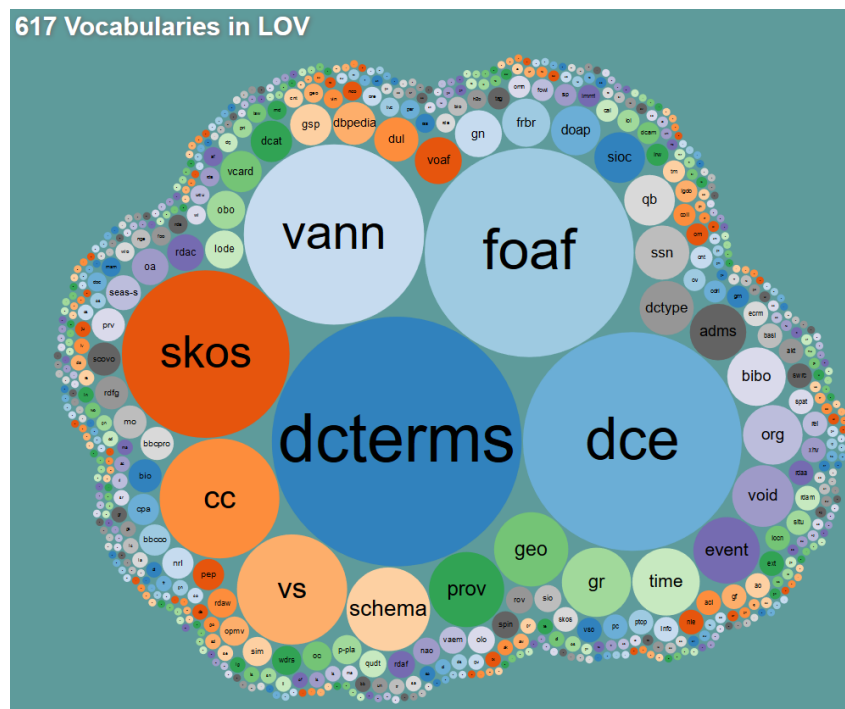
What we have today, is a web of HTML documents, interconnected by links. What we really want, is to also have all data available in a standard format (RDF), with links between other data and documents, creating a machine-readable *Web of Data*. This collection of interconnected datasets is also called *Linked Data*<sup>11</sup>. To use another definition for Linked Data, it is a set of techniques for publication on the Web

<sup>8</sup><https://muse.dillfrog.com/meaning/word/break>

<sup>9</sup><https://www.w3.org/standards/semanticweb/ontology>

<sup>10</sup>The words *vocabulary* and *ontology* are often used to described the same concept.

<sup>11</sup><https://www.w3.org/standards/semanticweb/data>



**Figure 2.5:** A representation of the importance of all the vocabularies in LOV (Linked Open Vocabularies). A big surface means that the vocabulary terms are often used. (Source: <http://lov.okfn.org/dataset/lov>)

using standard formats and interfaces, as well as data that conforms to those techniques [88]. The big advantage of Linked Data is that it is easily combinable with other Linked Data to generate new knowledge. This stands in sharp contrast with the traditional database systems, where data is stored on individual silos, using different storage techniques. One problem that could occur when attempting to combine two such databases, is the following. Let us assume these databases are simple Excel tables, with multiple columns, all of them with their respective column title. If the first database uses `Temp` for its temperature column, and another uses `Temperature`, you will have to manually intervene and state that they are the same. But even then, it might not be clear if the temperature is in Kelvin, Celsius or in Fahrenheit. Linked Data solves this problem by using unambiguous URIs. It is self-documenting, meaning that you can figure out what a term means by resolving it on the Web. However, just like traditional database system, Linked Data can still be of low quality and suffer from service failures. There are no inherent mechanisms in Linked Data that will protect you from this.

An example of a good Linked Data application is the following. The BBC uses Linked Data from various sources to enhance information presented about certain topics. Musical artist details are collected from MusicBrainz<sup>12</sup>, an open music encyclopedia. The World Wildlife Fund and DBpedia are some other sources that are consulted. The results can be seen in Figure 2.6. The BBC did not coordinate with its sources, which is why Linked Data is said to enable cooperation without coordination.

The Web of Data is built on the general architecture of the Web [45] and can be seen as an extension on the classic document Web. It shares many of its properties [18].

- The Web of Data is generic and can contain any type of data.
- It is also open for anyone to publish data.
- The publishers can choose which vocabularies to use.

<sup>12</sup><https://musicbrainz.org/>





**Figure 2.6:** DBpedia is generated by looking at the info boxes of Wikipedia articles. The BBC then uses content from DBpedia (and other sources) to create beautiful webpages. (Source: Wood et al. [88])

- The data is connected through RDF links, which results in a global data graph, spanning all sources and allowing discovery of new sources.

Analogous to hypertext links, RDF links are provided by RDF triples where the subject and object are both URI references, but of different datasets [17]. Because URIs are used for resource identification, it is possible to use the HTTP protocol as retrieval mechanism for URIs that use the *http://* scheme (i.e. URLs).

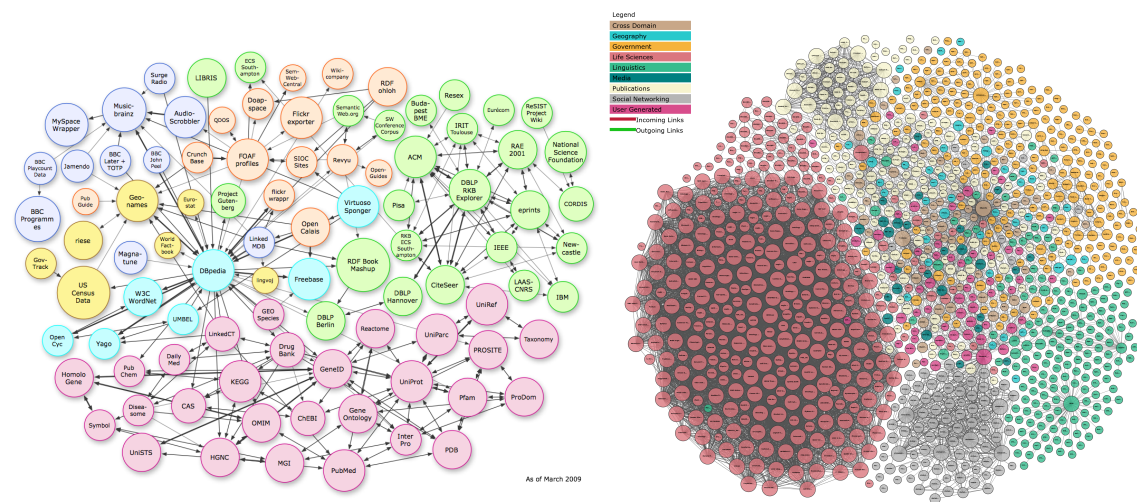
The result is a World Wide Web with structured data that can be combined with other people's data. Note that, just like on the actual World Wide Web, not all data is public. Linked Data techniques can still be deployed from within a corporate private network. The Linked Open Data (LOD) project is a community effort that started in 2007 to make data freely available to everyone. In more detail, the goal of the project is to identify existing data sets that are available under open licenses, converting them to RDF while staying true to the Linked Data principles, and publishing them on the Web. The result is a collection of all kinds of data that can be reached by the same API, and it is called the *LOD cloud*. Nowadays, the cloud is so big that generating a visual representation of this cloud is not easy. The latest attempt by <http://lod-cloud.net/> is shown in Figure 2.7b. For comparison, the visual representation from 2009 is shown as well in Figure 2.7a. As of November 2017, the cloud consists of 150 billion triples stored in 3000 datasets<sup>13</sup>.

Linked Data is not just RDF. It separates itself by four principles [13]:

- URIs are used to name things unambiguously.
- To allow people to look up these names, resolvable HTTP URIs are used.
- When looking up these URIs, useful information should be provided (in RDF standards).
- Links to other related resource URIs should be included, so that people can discover other things.

Only then is your data truly “linked” and are people able to surf the Web of Data. For instance, you could open [http://dbpedia.org/resource/Ghent\\_University](http://dbpedia.org/resource/Ghent_University) with a browser of your choice (e.g. Tabulator [14], Piggy bank [43] or simply your regular web browser), see that it is located in the *dbo:*

<sup>13</sup><http://stats.lod2.eu/>



(a) Linking Open Data cloud diagram 2009  
(Source: Bizer et al. [18])

(b) Linking Open Data cloud diagram 2017, by Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch and Richard Cyganiak. <http://lod-cloud.net/>

**Figure 2.7:** The growth of the LOD cloud.

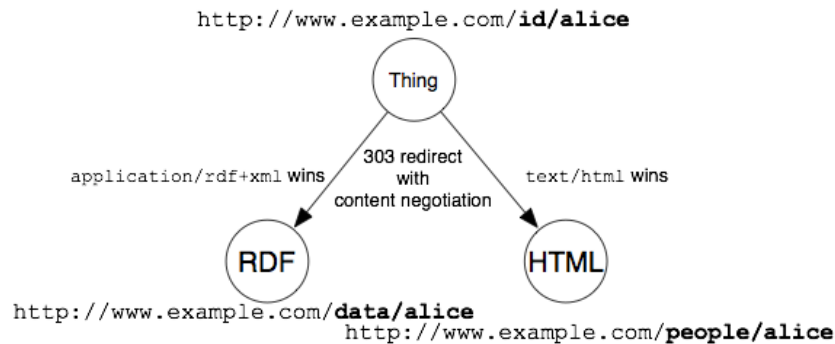
country **Belgium** (<http://dbpedia.org/resource/Belgium>), of which the motto (<http://dbpedia.org/property/englishmotto>) is “Strength through unity”.

### 2.2.3 Publishing Linked Data

So far, we know there is a Web of Linked Data available to us. But how do we contribute to this? Suppose we have a dataset that we want to share with the world as Linked Data, where do we get started? According to Bizer et al. [18], preparing your Linked Data for publishing involves three steps.

The first step is to assign URIs to all entities that are described in your data set. These URIs should also be dereferenceable over the HTTP protocol as to stay true to the design principles of Linked Data. As a provider, you can choose between two HTTP URI usage patterns: hash URIs and 303 URIs [73], or even combine them. Hash URIs contain a *fragment*, which is a part of the URI that is separated from the rest by a hash symbol (“#”). This way, you can point to a resource that is not a self-contained document, but rather a part of a document. An example would be <http://www.example.com/about#alice>, where information about alice is stored. This information resides on the same document as information about other things, such as <http://www.example.com/about#bob>. Upon requesting such a hash URI, the HTTP protocol automatically strips the fragment and returns the document URI, which is <http://www.example.com/about> in our example. The RDF framework can then use this document to identify the information about Alice, as specified in the fragment. The second option is to use 303 URIs. The name refers to the 303 See Other HTTP status code. It is used to indicate that a requested resource is not a normal Web document, and a Location header can be added to provide the URL of the document that represents the resource. Depending on the Content-Type header, a link to the matching document will be provided. This is illustrated in Figure 2.8. Different URIs can be used to identify the same entity. This is especially common when two providers do not know about each others datasets. One example are the URIs uses to identify Berlin. DBpedia uses <http://dbpedia.org/resource/Berlin>, while Geonames uses <http://sws.geonames.org/2950159/>. Both URIs refer to the same entity and are therefore called URI aliases. To ensure that these providers describe the same entity, RDF triples with the predicate *owl:sameAs* are used to link to known other URI aliases.

In the second step, RDF links to other related entities in different data sets should be included. This is not straightforward, since your dataset might contain a large amount of information. Therefore, (semi-)automated



**Figure 2.8:** Redirecting to the correct page by using 303 HTTP status codes. (Source: Sauermann et al. [73])

approaches are used to generate these links. When data is inherently connected by standardized naming schemes, making connections may be easier. For instance, publications typically have an ISBN number. If the target dataset also contains ISBN numbers, entities from your dataset and the target dataset can be linked. Problems arise when no such naming schemes exist. In this case, the similarity of entities within both datasets is used to determine whether two entities are the same. To compute this similarity metric, many research papers have already been written. Winkler [86] gives an overview of record linkage techniques, Elmagarmid et al. [34] is a survey about duplicate detection, and Euzenat et al. [35] is a book about ontology matching. In the context of Linked Data, Raimond et al. [63] have developed an algorithm to set RDF links between artists in the Jamendo and Musicbrainz data sets [18]. For this, they used a similarity metric that uses the names of artists, songs and albums.

Finally, you have to provide metadata about your published data. Metadata includes information such as its creator, creation date, creation method, and more. The Dublin Core terms (`dcterms`) provides most of the basic provenance. Adding metadata is necessary for consumers to assess the quality of your published data, and see if they trust it.

Optionally, you can use abbreviations for the namespace of common ontologies. This can significantly decrease the size of triple files, while also making them more readable. For instance, when an RDF file contains many links to entities on the `http://www.example.com/` namespace, they can be abbreviated by prefixing the entity-name by a defined prefix, such as `ex: <http://www.example.com/lion>` and `<http://www.example.com/tiger>` can then be written as `ex:lion` and `ex:tiger` respectively. In this thesis, prefixes will often be used. An overview of the most common prefixes can be found in the list of namespaces at the beginning of this thesis.

A great example of publishing process is the community effort to extract Linked Data from Wikipedia. The result is a huge dataset with connections to many of the most popular open datasets. In 2014, DBpedia consisted of just over 3 billion RDF triples<sup>14</sup>. To extract all this data, the dumps of the crucial relational database tables are used [6, 7]. These are made public<sup>15</sup> by Wikipedia on a regular basis. Following the success of DBpedia, The Wikimedia Foundation gave the DBpedia project access to its live feed which instantly reports all changes [19]. First, the relationships that are stored in this database are mapped to RDF. Then, additional information is extracted directly from the article texts and their info boxes. To link entities with entities in other data sets, many of the outgoing links are generated using simple identification schemata, such as comparing ISBN numbers, although others were more difficult. Chemical compounds are compared to entities in other bio-informatics data sets by looking at their gene, protein and molecule identifiers. Links with GeoNames and MusicBrainz were found by respectively looking at similarities in locations (similar name, coordinates, country, population, etc.) and bands (similar name, albums, members, etc.).

<sup>14</sup><http://wiki.dbpedia.org/about/facts-figures>

<sup>15</sup><http://download.wikimedia.org/>

## 2.2.4 RDF serialization formats

Tabular data is often serialized as CSV (comma separated values) or TSV (tab separated values) [82]. Here, every row is an entry in the database, with its properties separated by a comma, semicolon or tab. The titles of these properties (or columns) are explicitly added to the first row, but this is optional. The parser will have to be informed about this manually. Adding a new but sparsely used property to the table, results in extra empty spaces. Since RDF is highly flexible, we need different formats. Some of the most popular ones are discussed here. The main advantage of using RDF is that adding new data is as easy as adding new triples. There is no need to alter any existing data (or structure).

### RDF/XML

The semantic web as envisioned by Berners-Lee was introduced in 2001. At that time, XML was really popular, which is the reason why RDF was first serialized as RDF/XML. It can be parsed by any XML parser, but unfortunately, its verbosity is a huge downside. To encode the graph in XML, the nodes and predicates have to be represented in XML terms (element names, attribute names, element contents and attribute values) [10]. This makes it quite hard for humans to read, so other serialization formats have been proposed.

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
  <rdf:Description rdf:about="http://me.lukasvanhoucke.be/">
    <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
    <foaf:name>Lukas Vanhoucke</foaf:name>
    <foaf:workplaceHomepage rdf:resource="http://www.ugent.be/">
    <foaf:title xml:lang="nl">Ir.</foaf:title>
    <foaf:title xml:lang="en">Msc.</foaf:title>
  </rdf:Description>
</rdf:RDF>
```

**Listing 2.1:** Example of an RDF/XML document. (Based on Lanthaler and Gütl [50])

### Turtle

When human inspection is required, Turtle [9] is a better choice. As an extension of N-Triples [28], it is a compact and natural text form to write down an RDF graph. It is compatible with, and is a subset of, Notation 3 [11] (N3) and can be used in systems that support N3. The triples are represented as a sequence of subject, predicate and object, separated by whitespace and terminated by '.' after each triple. If multiple statements only differ in their object, the , symbol can be used to avoid repeating subject and predicate. Similar, the ; symbol is useful when you only want to avoid repeating the subject. Listing 2.4 shows an example of an RDF document that was written in Turtle.

### RDFa

In 2009, Google introduced rich snippets<sup>16</sup>. These enhanced search results are generated by looking at structured data on a web page. The data is added to the page through a form of Linked Data called

<sup>16</sup><https://developers.google.com/search/docs/guides/intro-structured-data>

RDFa, short for RDF in Attributes [1]. RDFa embeds triples into HTML. Most of the time, this structured data was already present on the page, although the trip this data takes from its database (an information silo) into HTML for display is often lossy. Usually, a human knows how to read the document and its metadata, but the meaning is lost on machines [67]. On a shopping website, an example of this could be a picture of a specific product. If you explicitly declare this picture as an image of the product, the computer knows what to display in search results. RDFa also allows adding data that should not be displayed on the webpage, but might be useful for computers. Using the example of pictures, you could include details about the photo creator, camera setting information, resolution, location, etc. To make sure that these machines understand the metadata terms, a shared language is needed. Fortunately, some of the most popular search engines (Google, Bing, Yahoo!) have created Schema.org, a shared vocabulary focusing on popular concepts. It cannot be an ontology for everything, but it is broad enough to not focus on one specific area. However, there does seem to be a slight bias towards commercial and search engine related terms<sup>17</sup>. Not everyone was happy with the Schema.org, since they fully defined their own vocabulary (including already existing terms), resulting in vocabulary lock-in [77]. The rich snippets are generated by providing markup for specific Schema.org terms. A product could for instance be marked up with RDFa Lite, a minimal subset of RDFa that can be used for most simple markup tasks [78]), as shown in listing 2.2.

Lastly, Although RDFa is most commonly used in HTML documents, it is a specification to express structured data in any markup language with attributes.

```
<div vocab="http://schema.org/" typeof="Product">

<span property="name">Dell UltraSharp 30" LCD Monitor</span>

<div property="aggregateRating" typeof="AggregateRating">
  <span property="ratingValue">87</span>
  out of <span property="bestRating">100</span>
  based on <span property="ratingCount">24</span> user ratings
</div>
...
</div>
```

**Listing 2.2:** An example of HTML marked up with RDFa Lite and the Schema.org vocabulary. (Source: <http://schema.org/docs/datamodel.html>)

## JSON-LD

Web developers often prefer working with JSON<sup>18</sup> data. Just like RDFa can build on HTML, web developers want to use a successful syntax, JSON in this case, and add a semantic layer on top of it. The result is JSON-LD, a lightweight serialization format for Linked Data documents that is fully compatible with all traditional JSON tools[50]. For this reason, it is also commonly used to store Linked Data in JSON-based storage engines [30]. JSON-LD was designed such that RDF and Semantic Web knowledge is not strictly required. In fact, the keywords `@context` and `@id` are all a developer needs to know. With just these keywords, meaning can be added to existing JSON documents without disrupting their operations. At the same time, JSON-LD is sufficiently expressive to support all major RDF concepts. As graphs are supported, it is in fact more expressive than Turtle. The JSON-LD version of listing 2.1 is shown in listing 2.3.

<sup>17</sup>As can be seen in the full hierarchy of the vocabulary: <http://schema.org/docs/full.html>

<sup>18</sup>JSON is short for JavaScript Object Notation, which is a lightweight and language-independent format for data interchange.

```

{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/",
    "title": "foaf:title",
    "name": "foaf:name",
    "homepage": {
      "@id": "foaf:workplaceHomepage",
      "@type": "@id"
    }
  },
  "@id": "http://me.lukasvanhoucke.be",
  "@type": "foaf:Person",
  "title": [
    {"@value": "Ir.", "@language": "nl"},
    {"@value": "Msc.", "@language": "en"}
  ],
  "name": "Lukas Vanhoucke",
  "homepage": "http://www.ugent.be/"
}

```

**Listing 2.3:** Example of a JSON-LD document. It is the JSON-LD version of listing 2.1. (Based on Lanthaler and Gütl [50])

## 2.2.5 Querying RDF: SPARQL

Just like regular databases, RDF datasets can be queried using a specialized query language: SPARQL [62]. An example of a SPARQL query is given in listing 2.5. The query consists of a set of triple patterns (or *basic graph patterns*) that are just like RDF triples, with the exception that each of the subject, predicate and object can be a variable (indicated by the question mark in front of the variable's name). A subgraph of the RDF data matches the basic graph pattern if valid terms of this subgraph can be used to substitute the variables of the basic graph pattern. The result of executing this query will be a list of matching values for the requested variables. For the data given in listing 2.4, two results are returned. These can be found in table 2.1.

```

@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Johnny Lee Outlaw" .
_:a foaf:mbox <mailto:jlw@example.com> .
_:b foaf:name "Peter Goodguy" .
_:b foaf:mbox <mailto:peter@example.org> .
_:c foaf:mbox <mailto:carol@example.org> .

```

**Listing 2.4:** Sample RDF data in Turtle notation. (Source: [62])

| name                | mbox                       |
|---------------------|----------------------------|
| "Johnny Lee Outlaw" | <mailto:jlow@example.com>  |
| "Peter Goodguy"     | <mailto:peter@example.org> |

**Table 2.1:** Query results after executing the SPARQL query in listing 2.5 on the data in listing 2.4. (Source: [62])

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE
  { ?x foaf:name ?name .
    ?x foaf:mbox ?mbox }
```

**Listing 2.5:** SPARQL query to return the names and mailboxes of people that have a name AND a mailbox. (Source: [62])

### Querying the Linked Data Web

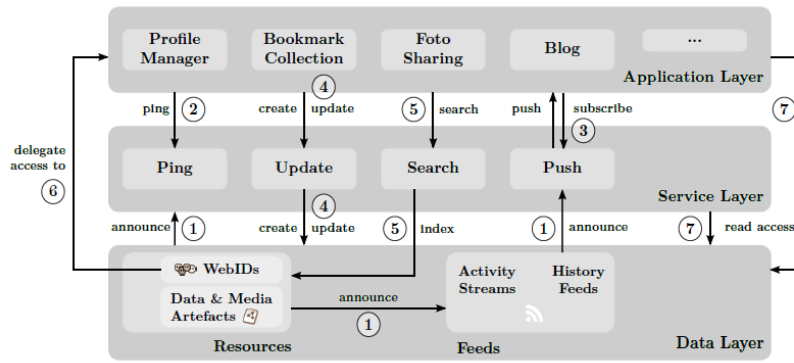
Instead of querying known RDF documents, we can also query the whole Linked Data Web. This Web of RDF links can be understood as a single, globally distributed dataspace [41]. The main advantage of querying this dataspace is the ability to combine data from various data sources to obtain a more complete view. However, since the dataspace is open, it is impossible to know all relevant data sources to answer the query. Hartig et al. [41] presented an approach to discover data that might be relevant for the query. For this, they used asynchronous traversal of RDF links during the query execution. The query engine then executes the query over a continuously growing set of relevant RDF documents. Note however that we still cannot assume to find all relevant data to answer the query. An example SPARQL query for "Return phone numbers of authors of ontology engineering papers at ESWC09" can be found in listing 2.6. In general, their approach performs better when the Web of Linked Data contains more links, since more relevant data might be discovered.

```
SELECT DISTINCT ?author ?phone WHERE {
  <http://data.semanticweb.org/conference/eswc/2009/proceedings>
    swc:hasPart ?pub .
  ?pub swc:hasTopic ?topic .
  ?topic rdfs:label ?topicLabel .
  FILTER regex( str(?topicLabel), "ontology_engineering", "i") .

  ?pub swrc:author ?author .
  {?author owl:sameAs ?authAlt} UNION {?authAlt owl:sameAs ?author}

  ?authAlt foaf:phone ?phone
}
```

**Listing 2.6:** SPARQL query to return the phone numbers of authors of ontology engineering papers at ESWC09 (Source: Hartig et al. [41])



**Figure 2.9:** Architecture of a distributed semantic social network. The various components are explained in section 2.3. (Source: Tramp et al. [81])

## 2.3 Semantic social networks

The previously mentioned decentralized social networking solutions were still lacking in certain aspects. Preferably, the application logic of a social network should be fully decoupled from the data to avoid a vendor lock-in [27]. Interoperability can only be maximized if well-defined protocols and standards are developed. At this point, social platforms become views over your data instead of fully contained applications [84]. Another desire is the principle of *freedom of expression*. In order to allow anyone to say something about anything, we need a way of identifying everything. This could for example be realized by linking content through dereferenceable URIs.

### 2.3.1 Designing a distributed semantic social network

An architecture of a distributed semantic social network (DSSN) was presented by Tramp et al. [81]. Everyone in the DSSN can set up their own DSSN node, or choose a trusted provider to host their data, much like the pods presented in section 2.1.1. This allows a maximal amount of control over privacy and ownership. Overall, the data is also more secure since it is difficult to steal large amounts of data when the data resides all over the web. Similarly, all kinds of cyberterrorism (e.g. denial-of-service attacks) and censorship become substantially harder in a decentralized context. Because the architecture describes social networks that use semantic resources without requiring a specific schema, full extensibility and interoperability is guaranteed.

The architecture is based on three principles:

- The Linked Data principles are followed for data publishing, retrieval and integration. Therefore, all information is stored as RDF triples.
- User data is decoupled from services and applications. Your own data is stored in your own DSSN node, while data generated by services, data you do not own, remains at the service's servers.
- Protocols should focus on their main task: communicate RDF triples between nodes. A specific workflow or an interpretation of the data should be avoided.

Figure 2.9 shows the architecture (without protocol layer) of which the components will be discussed in the following sections.

### 2.3.2 Web ID

Crucial in a social network is the ability to represent yourself. In a semantic context, this is usually done through a WebID [69]. It is a digital ID of the user, stored as a dereferenceable RDF document. Therefore,



any agent can be uniquely identified by its URI. Apart from identification, it is also used to describe its owner. For instance, many WebIDs contain links to the WebIDs of their friends. For this, the FOAF (Friend of a Friend) vocabulary is most commonly used, but users are free to use any vocabulary they want. FOAF is especially useful in a social context, since it describes basic information about people, as well as offering specialized terms to describe Social Web internet accounts [23]. Rich social data can be expressed using the SIOC (Semantically-Interlinked Online Communities) vocabulary [20], which is an extension of FOAF. Authentication by means of SSL client certificates is also included in the WebID platform [81]. The WebID document must be available in turtle serialization, but may also be offered in other serialization formats (through content negotiation). An example of a WebID document is shown in listing 2.7.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<> a foaf:PersonalProfileDocument ;
    foaf:maker <#me> ;
    foaf:primaryTopic <#me> .

<#me> a foaf:Person ;
    foaf:name "Bob" ;
    foaf:knows <https://example.edu/p/Alice#MSc> ;
    foaf:img <https://bob.example.org/picture.jpg> .
```

**Listing 2.7:** WebID example (in Turtle representation) (Source: Sambra et al. [69])

### 2.3.3 Storage

A higher degree of decentralization is achieved when more people have access to a data space of their own. Typically, servers are hosted in a computing center, and can be used for a small charge. Hosting your RDF data on such a server is not 100% safe, since the computing center staff can still access the data. Another option is to use software like FreedomBox<sup>19</sup>, which is able to run a personal server on a low-end device in a trusted area (e.g. your own house) that runs 24 hours a day [81]. This is more secure, but more expensive. In the future, smartphones have the potential to become your private data storage host. If battery issues and connection stability can be solved, smartphones will be able to be connected for 24 hours a day.

WebBox [83] is an example to realize the concept of “Socially-Aware Cloud Storage” [12], with the intention to separate data from apps. It is a personal information management server (PIM) that either runs on the user’s device (e.g. a smartphone), or on a virtual host (e.g. an Amazon S3 instance). It acts as a generic data-storage layer, allowing the user to store and manage its own data instead of at cloud service silos. It serves as a repository for data objects: structured data like tweets and Facebook likes, and unstructured data such as photos and videos. To support both formats, an optimized RDF query store is used alongside a filesystem-based linked data store. It offers multiple access control levels and can notify people when shared entities are changed.

### 2.3.4 Services

In order to have a decentralized social network with the same capabilities as traditional social networks, multiple services are required.

---

<sup>19</sup><http://freedomboxfoundation.org/>

## Update queries

First of all, there needs to be a service that allows queries to modify and create user resources. SPARQL is by far the most popular choice.

## Ping service

To facilitate friending between two users, or notifying the user when someone places a comment on his blog post, a ping service is required.

**Semantic Pingback** Tramp et al. [80] developed the Semantic Pingback mechanism, which allows bidirectional links between documents. A lightweight RPC<sup>20</sup> service is advertised in the document using a `ping:to` relation, allowing immediate feedback and therefore facilitating social interactions. For instance, friending is a process of establishing symmetric `foaf:knows` relations between two users [81]. To accomplish this, following steps are taken:

- Alice adds a `foaf:knows` relation to her WebID. Bob's WebID is the object of this relation.
- Using Bob's `ping:to` endpoint for pingback, Alice informs Bob about this statement.
- Bob receives this message, and can now choose whether he wants to publish his own `foaf:knows` relation to Alice.
- If Bob chooses to be friends, a ping is sent back to Alice, informing her about Bob's updated WebID.

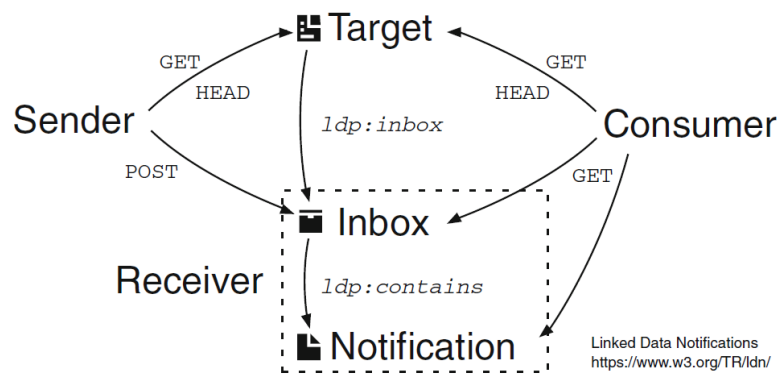
The content of these notifications are the source and destination URIs, indicating the relations between web resources without containing information about the relation itself. For this reason, these suite of protocols<sup>21</sup> are often called linkbacks, and are only useful when used in the context of Web publishing [26].

**Linked Data Notifications** According to Capadisli et al. [26], a Web notification protocol should ideally conform to all Linked Data design principles. Therefore, they developed the Linked Data Notifications (LDN) protocol, which allows the notifications to be shared and reused across applications. The notifications are considered resources and are identified with a unique dereferenceable URI. Any resource may be the target of a notification: it can be addressed *to* a target, or contain information *about* a target. Target resources are not required to have a receiving service, and can instead advertise an endpoint/inbox for its incoming notifications. In other words, where Semantic Pingback had a receiver and a sender role, LDN adds a consumer role to allow applications to read and use notifications without ever being concerned about sending or storing them. Of course, the consumer can still take up the role of receiver. The inbox stores a collection of notifications, which are retrievable resources that return RDF data. Because these notifications are reusable and accessible by all applications, there needs to be a way for consumers to autonomously find this inbox. Therefore, the inbox should announced by using a `ldp:inbox` triple. Using the notifications across different applications also means that a shared vocabulary should be used, which is not something that this protocol enforces. It is however in accordance with Linked Data principles to use appropriate vocabularies. As the notifications are persistent, a RESTful architecture is used for CRUD operations. It also includes a paging mechanism to support multiple page responses when the amount of notifications becomes too large to handle. Although the notifications are reusable, this does not mean that they may never be deleted. The receiver retains the right to manage the notifications in whatever way it wants, for instance by removing notifications that are too old, or marking the notification as *read* (using an update operation), or even filtering notification that match a specific pattern.

---

<sup>20</sup>Remote Procedure Calling

<sup>21</sup>Webmention and Provenance Pingback are similar protocols.



**Figure 2.10:** Graphical representation of the Linked Data Notifications protocol. (Source: Capadisli et al. [26])

## WebSub

Some applications request that notifications are sent to them immediately or at intervals. This push mechanism is also known as publish/subscribe. Using the three actors from Linked Data Notifications, the notifications are pushed from senders to receivers, and pulled from receivers by consumers [26]. An example of a publish/subscribe protocol is WebSub (previously also known as PubSubHubbub before being adopted as a Candidate Recommendation by W3C) [37]. Establishing a connection requires multiple steps. First, a feed advertises a hub service for anyone to subscribe to. If the feed changes, the hub service is notified, which in turn broadcasts these changes to all of its subscribers. In the DSSN architecture, activity feeds and history feeds are the most important types of feeds [81]. Activity feeds publish all activities with a specific user as the actor, and history feeds are used for resource synchronization (e.g. keeping caches up-to-date).

## Search and index services

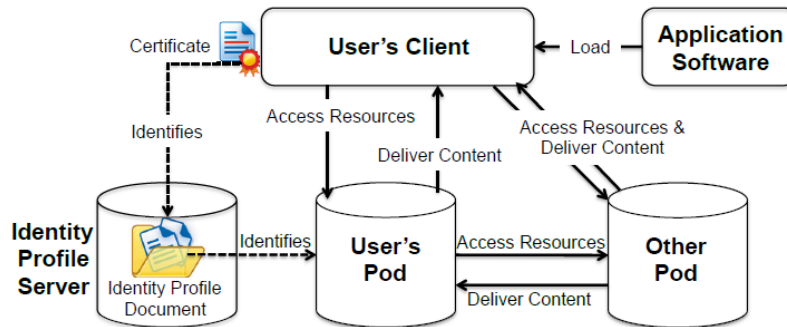
In social networks, it is usually not difficult to find a friend's profile since the servers are centralized. However, in the context of a DSSN, you cannot simply query the whole Linked Data Web and expect to find your friend in a reasonable amount of time. Instead, semantic search engines such as Swoogle<sup>22</sup>, the first of its kind, can be used. These search engines use similar techniques as standard Web search engines (e.g. Google, Bing) and are therefore a lot faster at finding relevant RDF documents. Similarly, a private search service is often used as a fast resource cache for a user's own private data [81].

## 2.4 Linked Data Platform

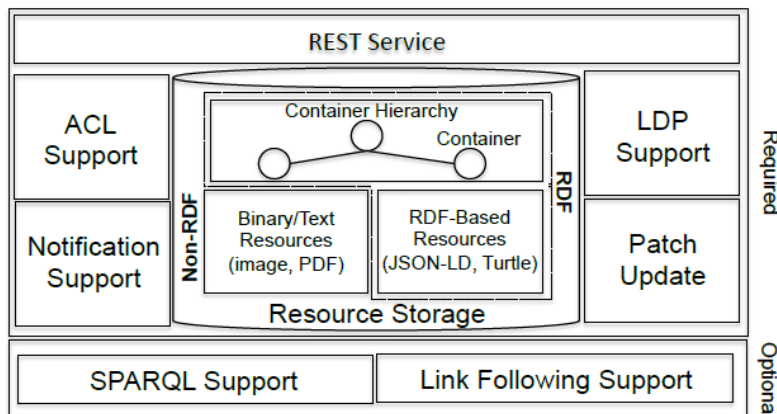
The W3C specification for Linked Data Platform<sup>23</sup> (LDP) [4] describes HTTP as a means to implement CRUD operations (Create, Read, Update, and Delete) on Linked Data resources. An important concept in this specification is the definition of an LDP resource (LDPR). This can be an RDF or non-RDF HTTP resource that conforms to certain patterns and conventions. A special kind of LDPR are LDP containers (LDPCs), which are able to group LDPRs into a single collection. Since these containers are LDPRs themselves, they can be grouped into another LDPC, resulting in a nested hierarchy. This is similar to how a file system works [70]. Note that all LDP resources (including containers) have their own URI, as the Linked Data standards prescribe. Resources can then be found by either advertising their URIs, or by following links from other items. The main advantage of LDP is the ability to write resources in a generic,

<sup>22</sup><http://swoogle.umbc.edu/>

<sup>23</sup><https://www.w3.org/TR/ldp/>



**Figure 2.11:** Solid architecture showing decentralized authentication and pod servers. (Source: Sambra et al. [70])



**Figure 2.12:** Pod architecture in Solid. (Source: Sambra et al. [70])

standard and RESTful way, without having to rely on APIs, which are less flexible. In this thesis, Solid will be used extensively.

## 2.4.1 Solid

### Overview

MIT has been working on a new project led by Tim Berners-Lee. It is called Solid<sup>24</sup>, an abbreviation for “social linked data”, and it aims to be a set of conventions and tools to build decentralized social applications based on Linked Data Principles. It is modular, easily extensible, and it attempts to rely only on W3C standards and protocols, such as the Linked Data Platform standard. The resulting applications have multiple advantages: a high degree of interoperability, portability of data between servers, and easy sharing of data and social graphs between applications [55]. An important factor in realizing this is the way data is stored. Just like Diaspora [16], it is stored in pods, short for personal online datastore. The main difference between these pods is that Diaspora uses application-specific pods, while Solid is application-agnostic and uses the LDP recommendation to store all kinds of data. Finally, authentication is realized through WebID, although other solutions could potentially be used instead. An overview of Solid’s architecture can be found in Figure 2.11.

## Solid pods

Pods distinguish between structured data (RDF) and unstructured data (images, videos, etc.), can store data in a hierarchical manner (using directories), and supports authorization through access control lists (ACLs). A user can have multiple pods. Note that Diaspora's definition of "pod" corresponds with Solid's definition of a "pod server" [70]. Data can be accessed using basic LDP operations (HTTP GET, POST, DELETE, OPTIONS and HEAD), but Solid also extends this recommendation with some extra features. For instance, it is now possible to use "\*" as a wildcard to find all resources that match the indicated pattern. Note that this aggregation process is not recursive. SPARQL support can optionally be included as well. There are two types of queries: local and link-following queries. The first type only accesses predicates that are located in the user's pod. The second type is able to follow links between a user's pod and other users' pods. To store RDF data, there are multiple possible implementations. A first option is to use the file system. All files (i.e. both RDF and non-RDF resources) will be stored as files, which is sufficient for simple applications that rely on resource-level manipulations, such as reading or writing a document. For more intensive RDF usage, various efficient RDF engines are available. For fine-grained access control, the `WebAccessControl`<sup>25</sup> (WAC) ontology is used. Each resource or container can have its own access control list, otherwise the parent container's ACL is used. To make apps more responsive, notification support when a resource has been modified is also included. An overview of the pod's architecture can be found in Figure 2.12.

## Development of applications

Solid provides multiple libraries to support development of decentralized social applications. To handle RDF resources, the `rdflib.js` library (Tabulator's core library [14]) is available. Specifically for solid, `solid.js` abstracts some complex operations, which makes development of Solid application easier. Modules for authentication and sign-up are also available. The number of libraries and Solid components is still continuously increasing. Data created by one app could be used by another app, following the Linked Data principles. Solid allows users to grant access rights to applications, such that for instance one application can read your data, while another application can also edit and delete your data [68]. Data can also be federated between applications by passing notifications using the personal data stores.

## 2.5 Annotation systems

Annotations are pieces of information that are attached to certain locations of interest. In the context of this thesis, they concern comments of users (in any form) about a certain component of an Open Webslide, e.g. the title of the third slide, or the video on the fifth slide. As a side-goal of this thesis is to decentralize annotations on the Open Webslides platform, it is interesting to look at some of the most relevant existing annotation systems. However, note that these are not the only annotation systems. For instance, Google Docs, Medium.com and Authorea all offer a centralized annotation system with the possibility of comment threads, but these systems are proprietary and limited in functionality. They are called *sticky note* systems, because they are limited to annotating text.

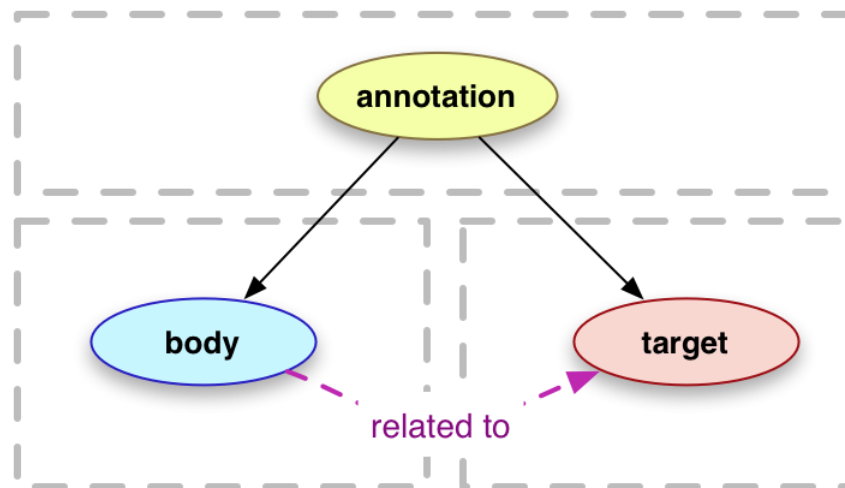
### 2.5.1 W3C Web Annotation Data Model

To support a standardized model for annotations, W3C created the Web Annotation Data Model recommendation [72]. It provides a JSON format for storing, sharing and consumption of annotations in a software-independent manner. It is simple enough for the most basic annotation tasks (*sticky notes*), but allow far more complex operations because of the rich vocabulary. Annotations consist of a body and a

---

<sup>24</sup><https://solid.mit.edu/>

<sup>25</sup><https://github.com/solid/web-access-control-spec>



**Figure 2.13:** The Web Annotation Model visualized graphically. (Source: Sanderson et al. [72])

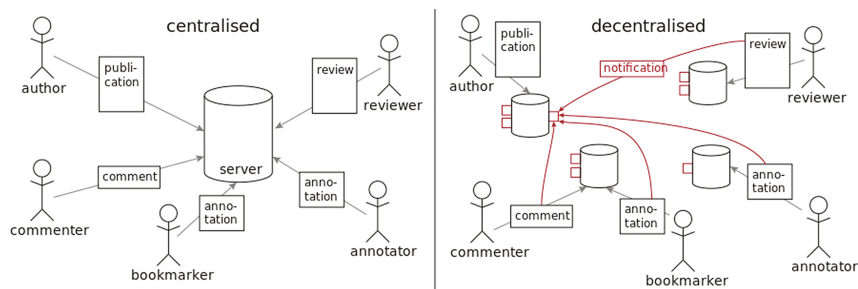
target that it relates to, as illustrated in Figure 2.13. The specification states that there should be zero or more bodies<sup>26</sup>, and one or more targets per annotation. The body will often contain text (optionally marked up, e.g. by HTML), but allows any content to be embedded (e.g. an external Web resource) and can contain styling hints to help clients render the annotation correctly. The target and body contain a *type* parameter (e.g. *Text* or *Video*), which helps determining whether downloading the body is useful or not. Targets can often be referenced by using IRIs with a fragment component, but in other cases, specific selectors are required. This W3C recommendation describes many additional selectors, such as the CSS Selector, XPath Selector, Text Quote Selector, etc. An example of an annotation using the `XPathSelector` is shown in listing 2.8.

```
{
  "@context": "http://www.w3.org/ns/anno.jsonld",
  "id": "http://example.org/anno22",
  "type": "Annotation",
  "body": {
    "type": "TextualBody",
    "value": "Comment text",
    "format": "text/plain"
  },
  "target": {
    "source": "http://example.org/page1.html",
    "selector": {
      "type": "XPathSelector",
      "value": "/html/body/p[2]/table/tr[2]/td[3]/span"
    }
  }
}
```

**Listing 2.8:** Annotation in JSON-LD with XPathSelector (Source: Sanderson et al. [72])

An interesting property for this thesis is that annotations can be stored in a machine-readable format (i.e. RDF) with a standardized vocabulary [90]. The standard includes the Annotation Protocol [71], which is a use of the Linked Data Platform. This means that a Solid server can be used to store annotations. The

<sup>26</sup>Annotating without a body is usually called highlighting.



**Figure 2.14:** Difference between architectures of centralized (left) and decentralized (right) authoring, annotation and notification system. (Source: Capadisli et al. [27])

specific Annotation Containers are an extension of normal LDP Containers in the sense that they are also ordered (i.e. they are a subclass of `OrderedCollection`<sup>27</sup>).

## 2.5.2 Hypothesis

Hypothesis<sup>28</sup> is a free and open client-side annotation plugin that works on all Web pages, without needing any code on the underlying site. On any page, you can open the browser plugin (or bookmarklet) and view all annotations. After signing up, you can also highlight and annotate text yourself, or reply to an existing annotation. The body of the annotations can contain links, images, videos and marked up text. The annotations can be made public, private or a combination of both by posting them to a certain group you belong to. It is built on top of Annotator.js<sup>29</sup> and tries to follow the Web Annotation Data Model. It is however limited to page and text annotations and is thus not able to annotate videos and images. Hypothesis stores its annotations centralized<sup>30</sup>, which is something we want to change in this thesis.

## 2.5.3 dokieli

Currently, the closest related project is dokieli<sup>31</sup>, a decentralized tool to publish, author and annotate articles [27]. It is a client-side application that is able to interact with Linked Data anywhere on the Web. It is completely independent from server-side software. Therefore, it transforms the current “Read-only” web into a “Read-Write Web”. To participate, users can use their WebID profile, and store annotations in their own personal on-line storage (pod), which complies with the Solid protocol. By making use of Linked Data Notifications (LDN), communication can happen in a fully decentralized manner. dokieli’s architecture compared to a centralized architecture is made clear in Figure 2.14.

With a single click, a user can create a new dokieli document. Likewise, the reader of a dokieli document can choose to save a copy in his own pod. Therefore, dokieli is said to be *self-replicating*. Semantics are added to dokieli documents by embedding triples as RDFa, which avoids duplication of data and allows the document to be parsed as a graph without needing dokieli as a dependency. The appearance of the document can easily be changed, since it is simply a matter of applying a different CSS3 stylesheet. Publishing a document is done in the same way as traditional Web pages, i.e. by using Web servers. The documents can easily be edited, since dokieli uses the Linked Data Platform for its CRUD operations.

All users can interact with the document, for instance by liking or commenting on the document as a whole. Furthermore, they can also interact with selections of the document, i.e. an annotation. These interactions are stored in the user’s own personal storage, which ensures that the user effectively owns their comments etc. For users that do not have their own pod, the document author can point to a storage

<sup>27</sup><https://www.w3.org/TR/activitystreams-core/#collections>

<sup>28</sup><https://web.hypothes.is/>

<sup>29</sup><http://annotatorjs.org/>

<sup>30</sup>Their API can be found at <https://hypothes.is/api/>

<sup>31</sup><https://dokie.li/>

service of his own, which can be used at the cost of less ownership. The author of the document is then notified using the LDN protocol, and can optionally take action.



## Chapter 3

# Problem description

This thesis covers the differences between file-based and query-based access, specifically in the context of annotations. The end results of this thesis may help the reader choose an access technique for his decentralized social application.

### 3.1 Accessing Linked Data

The centralization of current-day's social networks results in two large consequences: data cannot be reused across applications, and the user loses ownership over the data. These problems can be solved by making use of Linked Data technologies. As stated before, Linked Data allows different applications to work with the same data. By storing this data in a personal data pod, the end users can choose where their data is kept.

Although Linked Data uses the same syntax (i.e. RDF), triples can be accessed in multiple ways. A first option is to store each set of coherent triples as a file, resulting in multiple files. Accessing a triple then requires collecting the whole file in which the triple resides. This is a file-based access technique. Such a file contains triples that describe an entity or concept. A hierarchic directory structure allows closely related and similar files to be placed in the same directory, which is useful for applications that want to find all similar files. For instance, a messaging application could store all its messages in the `/messages/` directory, allowing a different messaging application to easily find all the messages without having to look at every single file in the user's datapod. At the time of writing, Solid uses this access technique by default and will therefore be used as this technique's implementation during the course of this thesis.

A second option is to store all triples in the same graph on a SPARQL endpoint, and use queries to collect the relevant triples. To achieve the same basic functionality as Solid, custom middleware needs to be inserted between the data store and the data-requesting user. The user should not be able to write SPARQL queries himself, but instead let the middleware generate these. The query is determined by the endpoint of the user's request. For instance, when a user wants to request all information about his friends, he would send a GET request to the RESTful `/friends` endpoint. In this thesis, this type of access will be referred to as graph- or query-based access.

Alternatively, a combination of the two techniques is another possibility. Here, all triples that would belong to a file when using file-based access techniques, are instead stored in their own graph. In other words, instead of storing all data in a single graph, a graph is created per file. To access the triples, queries can be executed to return specific data or to send the whole graph. This type of access technique is not covered in this thesis, as it is very similar to file-based access.

## 3.2 Goal of the thesis

All of these techniques can be used to implement the Linked Data Platform (LDP) specification. Solid implements this by design, but is only available for file-based access at the time of writing. In the future, Solid should also be able to work with other access techniques (e.g. SPARQL endpoints). The custom middleware for query-based access will implement the LDP features that are necessary to demonstrate the differences between file-based and query-based access techniques, along with more features that are not LDP-spec compliant.

During this dissertation, the goal is to find a comprehensive answer to all of the following research questions:

- Q<sub>1</sub>: What are the advantages and limitations of using graph-based techniques over file-based techniques in the context of decentralized collection of annotations?
- Q<sub>2</sub>: How does the number of annotations influence their collection time when using graph-based and file-based access techniques?
- Q<sub>3</sub>: How do these access techniques compare on user-friendliness and difficulty of implementation?
- Q<sub>4</sub>: How are these techniques reusable in social application contexts other than annotations?

Using the gathered knowledge, following hypotheses can then either be confirmed or falsified:

- H<sub>1</sub>: In the context of annotations, query-based access has faster retrieval of annotations
- H<sub>2</sub>: File-based access techniques can reuse existing filesystem-based techniques to make certain tasks easier, such as watching for updates using directory watchers or changing data using common text editors.
- H<sub>3</sub>: File-based access techniques can easily be reused in other social contexts, while this is harder for query-based access techniques.
- H<sub>4</sub>: The differences between file-based and query-based access techniques are still present when they are used for decentralized social applications other than annotations.

To test these hypotheses, we implemented decentralized annotations, as they are considered to be social applications. Here, annotations have to be stored on the datapod of the user instead of a central server. By providing the link to the annotation to, for instance, the website, the annotation data can be collected. To assure compatibility between different applications, annotations should be stored as Linked Data. This decentralized annotation system should be able to work on top of the Open WebSlides platform. It should also be possible to effortlessly adapt the system to work with other Web-based platforms.

## Chapter 4

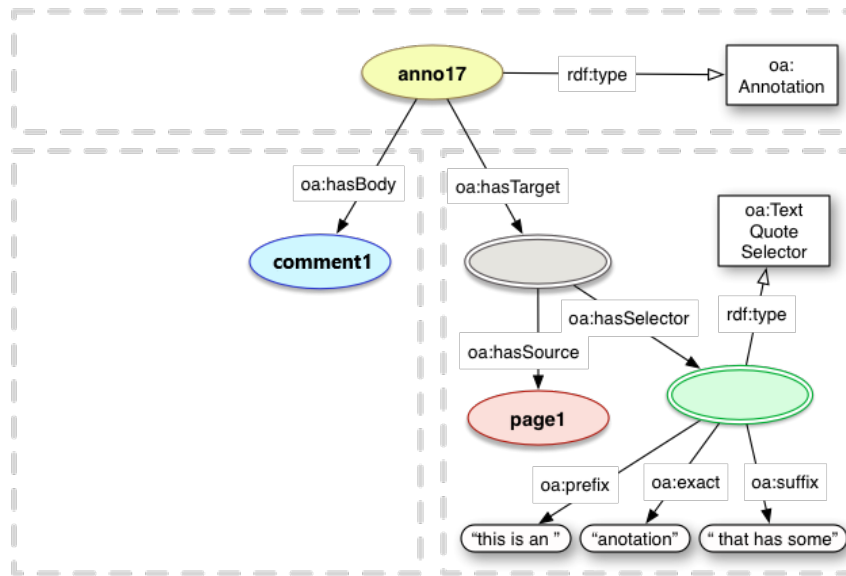
# Analysis

In this chapter, file-based and query-based access are compared in a theoretical manner in order to assess their differences and similarities. In most cases, decentralized annotations will be used in examples. In some cases, a test setup was implemented in order to showcase the possibilities. These setups are discussed in chapter 5.

### 4.1 Decentralized Annotations

#### 4.1.1 Annotations as Linked Data

Most websites that implement some sort of annotation system, store annotations on their own servers. In this thesis, they have to be stored in the user's datapod instead. To allow interoperability between different applications, the annotations will be stored as Linked Data. Fortunately, the Web Annotation Data Model, a W3C recommendation for annotations as was discussed in section 2.5.1, also includes a vocabulary to use annotations in a Linked Data context. The vocabulary has its namespace at `http://www.w3.org/ns/oa#` and is commonly prefixed as `oa:`. This model defines many ways to select annotations, and allows multiple properties. Implementing an annotation system that is able to deal with the whole specification would take a long time and is not part of this thesis. Instead, a basic structure is extracted that can be used in many occasions. At its core, it uses the `TextQuoteSelector` to select the highlighted text and its prefix and suffix. The basic structure of such an annotation is displayed in Figure 4.1. The body of the annotation can be left empty to represent a simple highlight of some text. It can also have a `TextualBody` for which the value represents a comment that is linked to the highlighted text, e.g. a sticky note. In the end, an annotation should look like the one found in Listing 4.1. In popular annotation systems, it is often possible to attach additional comments to annotations. In this case, the selector would point to an existing annotation. This has not been implemented for this thesis, since comments are conceptually very close to annotations.



**Figure 4.1:** Basic annotation structure using the TextQuoteSelector. Here, the word “annotation” would be highlighted. (Source: <https://www.w3.org/TR/annotation-vocab/#textquoteselector>)

```

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix oa: <http://www.w3.org/ns/oa#>.
@prefix terms: <http://purl.org/dc/terms/>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

<> a oa:Annotation;
  terms:created "Thu, 13 Feb 2018 10:00:00 GMT";
  terms:creator <https://lukas.vanhoucke.me/profile/card#me>;
  rdfs:label "Lukas Vanhoucke created an annotation"@en;
  oa:hasBody <#comment1>;
  oa:hasTarget <#target>;
  oa:motivatedBy oa:commenting.

<#comment1>
  a oa:TextualBody;
  rdf:value "tl;dr"@en.

<#target>
  a oa:SpecificResource;
  oa:hasSelector <#fragment-selector>;
  oa:hasSource <http://www.example.com/blog/page1> .

<#fragment-selector> a oa:FragmentSelector;
  oa:refinedBy <#text-quote-selector> .

<#text-quote-selector>
  a oa:TextQuoteSelector;
  oa:exact "anotation"@en;
  oa:prefix "this is an "@en;
  oa:suffix " that has some"@en.

```

**Listing 4.1:** Example Turtle file containing the data for an annotation.

### 4.1.2 Writing and reading annotations

Annotations will typically be generated by some kind of annotation system/application, which can be provided by the website or by a client-side plugin. The annotation is always stored in a data pod of the user's choice. This can either be his/her own datapod, a shared datapod or a datapod that is announced by the website<sup>1</sup>. If the user's datapod has a hierarchical structure, the annotations should be saved in the same directory, which exclusively holds annotations. This directory is usually announced by using the `oa:annotationService` predicate. This could for example be added to the user's WebID, or to the website by embedding it as RDFa.

To "upload" an annotation, the website needs to be notified that an annotation for the website was created. This can be accomplished by sending a Linked Data Notification to website's `ldp:inbox`. This inbox should be announced by the website (again, preferably through the use of RDFa). Upon receiving the notification, the website will first check whether the target source is indeed a URL that resides on the website's domain. If this is the case, the annotation is NOT stored on the website's server but instead a link is added to the website's list of annotations.

Annotation applications could also offer the option to keep an annotation private. For instance, a discussion group could exchange annotations with each other, without anyone else being able to see them. In this case, the notification should not be sent to the server's inbox, but to an inbox of the server where the discussion group's list of annotations (i.e. `oa:annotationService`) resides.

To display the annotations, they first have to be collected. For this, two options are available. The first option is to let the website collect the annotations and dynamically generate HTML code to display the annotations. This is especially useful because it allows caching, as will be discussed in section 4.7. The second option is to expose a list of relevant annotations and let the users collect the annotations by themselves. This option could be more useful for third-party client-side annotation plugins such that the website does not have the need to implement an annotation system of their own. A website could also implement the two options to give users a choice between the website's own annotation system and any third-party annotation system the user would like to use instead. When collecting private annotations, only the second option is viable, since the website does not have access to the annotations. In the case of discussion groups, the list of annotations should be requested from the owner of the discussion groups. Ideally, the third-party annotation system should allow multiple URLs to such lists to be stored, and collect all annotations at once.

When a user decides to react to an annotation, he creates his own annotation and targets the annotation he wants to react to. Optionally, a Linked Data Notification can be sent to the owner of the targeted annotation, or even to all users that have reacted to this annotation, notifying them of a new comment. As this notification is sent to the user's inbox, the datapod server can process the notification and take actions. It could for instance send an e-mail to the owner notifying them of a comment, or it could update the original annotation so that it includes links to all comments.

To demonstrate the effectiveness of the theory above, an annotation plugin was created. Since the implementation relies on items that will be discussed in the following sections, the plugin's implementation is explained later in section 5.5.

## 4.2 Data manipulation

Both the file-based and query-based setup expose data in a RESTful way, although they are fundamentally different in the way data is manipulated. In the following sections, some of the largest differences related to data manipulation are discussed.

---

<sup>1</sup>Note that in the last case, the user is no longer guaranteed to be the rightful owner of the annotation.

### 4.2.1 Retrieval of data

File-based access techniques are only able to collect whole files. Querying the data usually happens client-side, after collecting the files. There are some exceptions to this rule, with the most common one being ACL, as this is often stored as a `WebAccessControl` RDF file. To interpret it, the server will have to query this RDF file. ACL will be extensively discussed in section 4.6. File-level granularity has both advantages and disadvantages. The most important drawback is the fact that if files are large, the whole file still needs to be sent over the network, which might lead to network congestion in extreme cases. This is especially cumbersome when only a few fields are required by the application. Some applications require collection of multiple files, which often takes a couple seconds if the number of files is large. In some cases, the client-side querying process may take some time, depending on what query needs to be executed. Client-side query engines (such as `rdflib.js`) may be less efficient at querying compared to specialized SPARQL endpoints. However, client-side querying is also an advantage, since it reduces stress on the server, resulting in better availability and lower server costs. Public SPARQL endpoints suffer from frequent downtimes [24]. This is caused by clients requesting the server to run complex queries, for instance by including multiple `OPTIONAL` statements. Not having to run these queries server-side is a good argument towards using a file-based access technique. However, note that query-based access does not allow the users to write their own queries, so the back-end can be configured to use more optimal queries [66]. Also note that there are other alternatives that solve this problem but are out of scope for this thesis. Triple Pattern Fragments [85] is for instance a good middle ground between server-side and client-side complexity. Another strong point for file-based techniques is that social applications often make use of self-contained RDF resources, and use most of the information a file provides. In the case of annotations, each annotation is represented as a single file, and contains practical information so that an annotation system can use this information to dynamically generate corresponding HTML code. Not all information in the annotation's RDF file might be used. However, as Linked Data is meant to be usable for multiple end-applications, providing all fields gives developers the opportunity to choose which fields to use.

In contrast, query-based techniques use SPARQL `SELECT` queries to retrieve certain predefined fields. An example of such a query is given in Listing B.2, where it is stored in variable `show_all_annotations_query`. Since only field values are returned, less information is available compared to file-based techniques. However, application will only use these relevant fields anyway, so more information is not necessary. An example of the server's response is shown in Listing 4.3. Retrieving complete files (i.e. file-based access) is simple as it requires the exact same process for each file (e.g. check ACL and send file if authorized), while requesting a query-based resource (i.e. query-based access) requires a specific query for each request. This query is determined by the server, the end-user will never have to write queries himself. This means that the server code has to be aware of what the files look like in order to expose an interface to the data. In other words, the server needs application knowledge in order to have meaningful queries. Some resources are more complex than others and will therefore also require more complex server code. For instance, annotations are composed of a tree-like structure as shown in Figure 4.1. A query to retrieve all annotations on a specific website would then look like the one shown in Listing 4.2. Since only fields are returned, the structure is lost in the response, i.e. the client will not be able to reconstruct the structure in Figure 4.1 without knowing in advance how the server stores annotations. If triples are really required, `CONSTRUCT` statements could be executed instead, but this requires an additional server interface. Sending only the relevant fields will result in less network traffic. The largest advantages of using query-based techniques, is having the possibility of executing complex queries (e.g. unions) and easily applying filters. This will become clear in the following sections. It does come with the price of more stress on the server, which is more expensive and may result in less availability. However, as clients are not able to write (potentially extremely complex) queries themselves, the risk of overloading the server is reduced.

```
prefix oa: <http://www.w3.org/ns/oa#>
prefix dcterms: <http://purl.org/dc/terms/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select DISTINCT ?annotation ?target ?text ?creator ?date ?exact ?prefix ?suffix
where {
  ?annotation a oa:Annotation .
  ?annotation oa:hasTarget ?target .
  ?target oa:hasSource <{{URL}}> .

  ?annotation dcterms:creator ?creator .
  ?annotation dcterms:created ?date .
  ?annotation dcterms:created ?date .

  OPTIONAL {
    ?target oa:hasSelector ?selector .
    ?selector a oa:FragmentSelector .
    ?selector oa:refinedBy ?refinedselector .
    ?refinedselector a oa:TextQuoteSelector .
    ?refinedselector oa:exact ?exact .
    ?refinedselector oa:prefix ?prefix .
    ?refinedselector oa:suffix ?suffix .
  }

  OPTIONAL {
    ?annotation oa:hasBody ?body .
    ?body a oa:TextualBody .
    ?body rdf:value ?text .
  }
}
```

**Listing 4.2:** Query to retrieve all annotations on a specific website. The `{{URL}}` tag is replaced by the actual URL of the website.

```

{
  "head": {
    "vars": [ "source", "text", "creator", "date", "exact", "prefix", "suffix" ]
  },
  "results": {
    "bindings": [
      {
        "source": { "type": "uri", "value": "https://example.com/blog/42.html" },
        "text": { "type": "literal", "xml:lang": "en", "value": "This is a simple annotation." },
        "creator": { "type": "uri", "value": "https://lukas.vanhoucke.me/profile/card#me" },
        "date": { "type": "literal", "value": "Fri, 13 Apr 2018 12:49:31 GMT" },
        "exact": { "type": "literal", "xml:lang": "en", "value": "text to highlight" },
        "prefix": { "type": "literal", "xml:lang": "en", "value": "text before highlighted text" },
        "suffix": { "type": "literal", "xml:lang": "en", "value": "text after highlighted text" }
      }
    ]
  }
}

```

**Listing 4.3:** JSON response that is received by the client after querying for a specific annotation that resides in the single-graph SPARQL endpoint. Only selected fields are returned.

Although the responses are different, both server types' resources are accessed in the same way, i.e. with a standard GET request. However, when query-based access is used, more than one annotation can be returned at once.

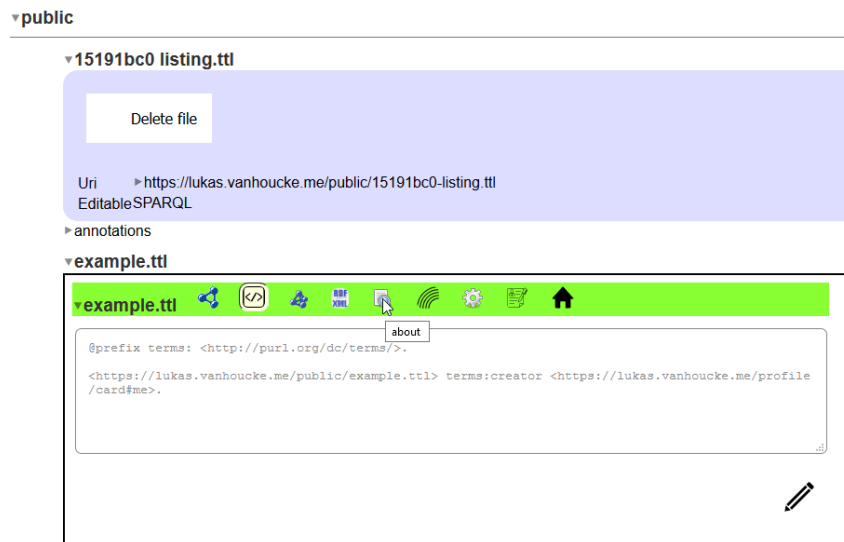
Some resources may have links to resources on other servers that have to be collected, or an application might request resources from multiple locations. For instance, a website might provide the user with a list of annotations URLs, expecting the user to collect the annotations by himself. In this case, the annotations could reside in many different datapods all over the world. In a scenario where some datapods may be offline, or some are slower than others, data should be collected and processed asynchronously in order to prevent a bottleneck.

## 4.2.2 Manually manipulating data

Sometimes, it can be useful to manually edit some information. For instance, you've been using a contact-list application, but it does not offer editing functionality. If you want to edit a contact's information, you will have to manually edit the RDF triples in your datapod. If your datapod stores data as files, changing the triple is really easy. If the datapod is hosted on your local computer, you can simply browse to the location of the triple file, and edit the triples using any text editor. If it's hosted on a remote computer and you can remotely log in, the same steps have to be followed. If it's hosted remotely and the server does not provide a login, you will have to rely on the file-based access technique's data editing interface. This interface may offer the same basic functionality as the file-system itself, while simplifying more difficult tasks, such as changing the ACL of a file. Some capabilities of Solid's interface are demonstrated in Figure 4.2.

Editing information that is stored in a single SPARQL endpoint's graph can be more difficult. There is no inherent mechanism (such as a file-system) to easily change a triple. However, an interface like the one Solid provides, is still a valid option. Implementing the interface might require some extra work, since SPARQL queries have to be generated for all of its functionality. For instance, deleting a file by making use of the operating system's file system, is simpler than deleting it using SPARQL. Indeed, SPARQL requires you to specify exactly which triples to delete, and therefore you will first have to collect all the relevant triples that would otherwise be bundled in the same file.





**Figure 4.2:** Solid's editing interface. The first file in the Public directory can easily be removed, the third file demonstrates editing capabilities.

### 4.2.3 Hosting public resources

It can occur that a user wants to host RDF data without having access to a datapod. With file-based access, this is easy. Files that are uploaded and stored on Dropbox, or any other file storage cloud platform, can just as easily be accessed as files in a datapod. This is very useful for public resources, since cloud platforms (as opposed to datapods like Solid) do not offer access control functionality, as they are unable to process ACL RDF files. There is also no certainty about the data's origin or author, whereas data that resides in a user's public datapod implicitly indicates that the owner of the datapod is the author of the file.

Hosting resources as RDF data in a public SPARQL endpoint's graph is not a possibility, since there simply are no public SPARQL endpoints that you can write to. Even if they would exist, they would need to expose an interface to the data through application-specific queries. In conclusion, hosting the data as files is a lot easier.

### 4.2.4 Non-RDF data

Some of the most popular social media platforms are Facebook, Instagram and Snapchat. These three networks are largely driven by photos and videos, which cannot be represented as RDF resources. LDP allows both RDF and non-RDF resources and treats them in the same way, except for the fact that non-RDF resources cannot express their state as RDF. Therefore, HTTP methods such as GET, PUT, etc., have different requirements. For instance, a GET request's response headers may include different representation formats (e.g. turtle, RDF/XML), while non-RDF resources only have a single representation format. Storing non-RDF data is easy on file-based setups such as Solid. Since they can also be requested by the usual HTTP methods, they are treated as regular LDP resources (if LDP is used) and can be stored in the same directory as RDF resources. A unique URI is generated for the file, so it can be included in other RDF documents.

Storing non-RDF data in a SPARQL endpoint's graph is harder. It's not possible to store these resources in the graph without for instance transforming them by using Base64 encoding. This is not optimal. It is better to include links to non-RDF resources, which means that the resources have to be hosted somewhere else. A hybrid architecture where non-RDF data is stored as files, and RDF data is stored inside the graph, could be a practical solution.

### 4.2.5 Current Web

Both the file-based and the query-based access setups are implemented in a RESTful way, meaning that all resources are accessed with their own URI, and accessing them twice in a row will always result in two identical responses. Therefore, they are both equally compatible with many current Web servers. For instance, caching<sup>2</sup> requests is possible for both access types. However, file-based access follows the same paradigm of simple file-servers (e.g. Apache, NGINX). Although there exist more types of servers, file-servers are the most common case, i.e. they mirror the file system over HTTP. Due to the vast amount of knowledge surrounding this type of server, implementing a new file-based access server is easier compared to setting up a query-based access server.

## 4.3 Detecting updates

In some cases, it can be useful to have a way of knowing when a resource is updated. In particular, the Linked Data Notifications specification (LDN) [26] would benefit from being able to process new or updated resources as they enter the server's inbox. This inbox can be discovered by looking for a website's or profile's `ldp:inbox` predicate. For instance, when someone sends a friend request to some user, a Linked Data Notification is sent to this user. This in turn could trigger an app notification on the user's phone. However, this requires knowing that a notification was added to the inbox. Detecting this can be accomplished in 3 ways. First, the state of the inbox can periodically be polled, which is a possibility for both server types. Depending on the polling rate, processing updates might happen with a significant delay. A better option would be to use a subscription model. The server can inspect all incoming packets and see whether they are being watched by the user, for instance when the target directory is the user's inbox. In this case, the corresponding event handler is called. This is possible for both access techniques. The query-based test setup uses Jena, which can capture updates using the `SparqlUpdate` module<sup>3</sup>. File-based servers can use mechanisms such as ActivityPub or WebSub. The Solid specification supports live updates through WebSockets<sup>4</sup>, to which clients can subscribe and receive changes in real-time. The final option is to attach a file or directory watcher to the resource you want to track. This is only possible for file-based servers. This option is perhaps easier for users that use a server without support for the other options. It could also potentially offer more flexibility/granularity to the end user, again depending on the server's implementation. It is showcased later in Section 5.2.

## 4.4 Directories

### 4.4.1 Hierarchic structure

A user's datapod can contain large amounts of data. To avoid losing track of what data belongs together, some kind of structure is necessary. That is why modern operating systems implement a hierarchical file system. Linked Data is machine readable, which implies that a machine should also know where to find the data. That is why related data is usually stored in the same (nested) directory. The exact location can be set in stone by some protocol (e.g. all messages have to be stored in the `~/messages/` location), which is not flexible, and is problematic when the same directory is used for two different protocols. A better option is to announce a link to the relevant directory. This principle is used for LDN's inbox and for `WebAnnotationModel`'s annotation service. These two specifications require triples with respective predicates `ldp:inbox` and `oa:annotationService` to be added to a user's profile, or to be announced by a web page (e.g. embedded as RDFa). The objects of the triples then point to the correct location where respectively notifications and annotations have to be stored. Some additional metadata information can be

<sup>2</sup>More information about caching can be found in section 4.7.

<sup>3</sup><https://github.com/SmartDataAnalytics/jena-sparql-api/blob/master/doc/SparqlUpdate.md>

<sup>4</sup><https://github.com/solid/solid-spec/blob/master/api-websockets.md>

added to the directories, such as only allowing a specific type of resource to be added to the directory (e.g. `ldp:constrainedBy` when LDP is used), or what access control rules to apply. An additional advantage of using explicit structure is the ability to use relative URLs in your triples. The LDP Best Practices and Guidelines [57] recommend this, because relative URLs are shorter than absolute URLs. They also make the resource more portable, since the hostname is omitted. This is especially convenient in development, where a development (with `localhost`) and production (with the actual domain name) setup is often used. The difference between relative and absolute URLs is demonstrated in Listing 4.4 and Listing 4.4.

```
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix ldp: <http://www.w3.org/ns/ldp#>.

<http://example.org/container1/>
  a ldp:Container, ldp:BasicContainer;
  dcterms:title "A very simple container";
  ldp:contains
    <http://example.org/container1/member1>,
    <http://example.org/container1/member2>,
    <http://example.org/container1/member3>.
```

**Listing 4.4:** Using LDP to achieve a directory structure. (Source: [57])

```
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix ldp: <http://www.w3.org/ns/ldp#>.

<> a ldp:Container, ldp:BasicContainer;
  dcterms:title "A very simple container";
  ldp:contains <member1>, <member2>, <member3> .
```

**Listing 4.5:** Using relative URLs to represent the same data as in Listing 4.4. (Source: [57])

#### 4.4.2 Directories in file-based and query-based servers

Servers that implement their Linked Data as files can use the operating system's file system for organization. Here, query-based servers are at a disadvantage, since they initially do not have any inherent structure. All data is stored in the same graph as an ocean of triples. Still, through queries, relevant triples can be retrieved. For instance, finding all annotations for a specific URL in this unstructured graph can be done by using the query in Listing 4.2. This is not possible in a file-based scenario, since you cannot efficiently query all files at once. These query types make it so that a directory structure is not strictly necessary. However, sometimes it can be a requirement, like when trying to implement the interface of section 4.2.2. If you want this interface to mimic Solid's interface – a directory structure with hierarchic ACL – an artificial directory structure inside the graph will be necessary. Where Solid already implements LDP's directory specification by design, a query-based server can also implement this manually. In this case, collections of triples should be stored as Linked Data Platform Resources (LDPRs). This is the equivalent of Solid's files. Each LDPR is then stored inside an LDP container, which is the equivalent of solid's directories. Again, to obtain a hierarchic structure, LDP containers can contain other LDP containers. The container stores its content by keeping links to its LDP resources and containers by using the

`ldp:contains` predicate. This is shown in Listing 4.4. An implementation of a directory structure for the single-graph server will be discussed later in Section 5.3.

## 4.5 Intersections

### 4.5.1 File-based intersections

When a file-based server is used, each annotation is stored in a separate file. This allows ACL at annotation-level granularity, makes it easier for the user to modify specific annotations, and a single link per annotation results in only collecting annotations that are relevant (as seen in section 4.1.2). Once an annotation is created for a certain webpage, the webpage should add the annotation URI to its list of annotations. In this case, collecting all relevant annotations is the simple and efficient process of requesting a list of annotations that are linked to the webpage, and collecting them. But what if an application wants to find annotations that have a specific field value? For instance, finding annotations for which the comment value contains certain words. In this case, all annotations will have to be collected and queried. In case the number of annotations is large, collecting all of them takes a long time (typically over 50ms per annotation<sup>5</sup>). After the collection process, the annotations still have to be queried. There are two possibilities. For simple queries like the “searching” example, querying each annotation separately is an option. If this happens frequently, indexing can be a solution, but is not always applicable. In short, indexing uses an additional file which contains links to all annotations, along with their specific field value, allowing the user to query this single file instead. As annotations are self-contained (i.e. they do not contain links to other annotations), simple queries are often sufficient. However, sometimes complex intersections have to be performed. Assume for example that someone wishes to collect all annotations for which the creator’s age is over fifty years. In such cases, it is more efficient to collect all relevant information (e.g. all annotations and all WebID profiles), aggregate them into a single graph, and run a query over this graph. Although client-side querying is slower than using specialized SPARQL endpoints, collecting the information is still the bottleneck.

### 4.5.2 Intersections over a single graph

When all information is stored in a single graph on a SPARQL endpoint, executing intersecting queries becomes a lot easier. First of all, there is no need to send many different files over the network, since queries are executed server-side. More importantly, having all data in the same graph means that SPARQL can be used to easily query ALL data without having to construct a new graph. For instance, using the same “searching” example of previous section, all results can be collected using the query in Listing 4.6. This query needs only a couple milliseconds to find all results.

---

<sup>5</sup>Using a remote server and a 50 Mbps connection. See also section 6.1.2.

```
prefix oa: <http://www.w3.org/ns/oa#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

select ?annotation ?text where {
  ?annotation a oa:Annotation .
  ?annotation oa:hasBody ?body .
  ?body a oa:TextualBody .
  ?body rdf:value ?text .
  FILTER regex(?text, "{{CONTAINS}}") .
}
```

**Listing 4.6:** Query to find annotations of which the body value contains a specific word or sentence. `{{CONTAINS}}` is replaced by the search terms.

The implementation of a test-setup for intersections over file-based and query-based graphs is further discussed in Section 5.4.

## 4.6 Access control

### 4.6.1 File-based ACL

File-based querying is less flexible than querying a single SPARQL endpoint. A great example of this is the way access control can be implemented. Solid uses `WebAccessControl` (WAC) to allow or deny access to resources. This uses an ontology with simple rules to determine whether users or groups, identified by their WebID, can read a resource, write or append to it, or a combination of the three. Admins or resource-owners also have control rights, which means that they can edit Access Control List (ACL) resources. Although each file is bounded by an access control list, it is not required to generate a new ACL file for each resource. Instead, WAC uses a hierarchic access control structure, which requires at least a root ACL. The corresponding ACL will be the most specific ACL, e.g. if such a resource was created for a single file, it will take have priority over any other ACL. If it was created for a specific directory, all resources in this directory will use this ACL instead of the root ACL (except if there are more specific ACLs). An example of such a resource can be found in Listing 4.7. It affects all files in a user's Solid inbox. Just like an e-mail inbox, all users except for the owner can only send new files to the inbox. Clearly, the granularity of WAC is one file. Sometimes, it can be useful to have an even finer-grained ACL where access to specific fields can also be controlled. For instance, logged in users could be able to see the textual content of an annotation, while guests would only see highlighted text.

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

<#owner>
  a acl:Authorization;
  acl:agent
    <https://lukas.vanhoucke.me/profile/card#me>;
  acl:accessTo <./>;
  acl:defaultForNew <./>;
  acl:mode
    acl:Read, acl:Write, acl:Control.

# Public-appendable but NOT public-readable
<#public>
  a acl:Authorization;
  acl:agentClass foaf:Agent; # everyone
  acl:accessTo <./>;
  acl:mode acl:Append.
```

**Listing 4.7:** WebAccessControl ACL file for a typical Solid inbox.

## 4.6.2 Graph-based ACL

WebAccessControl can also be implemented within a single graph on a SPARQL endpoint that stores all relevant data. For hierarchical ACL to work, a directory structure needs to be present. An implementation of this can be found in section 5.3. Otherwise, a single root ACL controls access to all resources in the graph. Therefore, using a SPARQL endpoint has no disadvantages concerning ACL compared to using Solid. However, having all data in a single graph allows more complex queries, which can be used to implement more complex ACL rules as well. For instance, one could request all annotations except for annotations that have a specific admins-only page as target. When the WebID content of users is also cached in the graph, even more is possible, such as disallowing people to see your annotations when they are friends with someone you dislike.

## 4.6.3 Development of an ACL extension

For this thesis, an experimental extension to WebAccessControl was developed to demonstrate these two types of additional ACL rules. In the **first phase**, the ACL is tested on a graph which contains user information and relations between users. In the **second phase**, the new ACL is adapted to work with annotations, which is more complex. In the experimental implementation, a single ACL for the whole graph is used. In future work, this could be extended by, for instance, implementing a directory structure or by adding triples that point to the correct ACL. In the examples, two different ontologies and their respective prefixes will be used. The first one is `acl`, which is the well known WebAccessControl ontology. The second one is `new-acl`, which is a new, custom ACL ontology that extends the WAC ontology.

### Filtering fields

A graph consists of many triples, each of which is considered a field. The type of field is determined by the predicate, which defines a relation between the subject and the object. With file-based access techniques, files are returned as a whole. The content inside this file is not modified when sending it

to the user, i.e. all fields are accessible by anyone with access to the file. With query-based access techniques, a selection has to be made. One could argue that querying for all (*?s*, *?p*, *?o*) triples returns all data, but this quickly becomes unmanageable when the graph grows and contains data of multiple applications. Therefore, endpoints such as `/user-information` usually generate a query to collect all users and return the relevant fields. In some cases, it can be useful to determine which fields can be accessed by the user or application. For instance, a phonebook application should only be able to see the names and phone numbers of your friends, and a birthday reminder app should only see the names and the birthdays of your friends.

In the graph of the first phase (Listing 4.8), simple relations with mostly literal field values are used. The graph contains the data of four different people that are all related to the first person (Lukas). A query to receive the information about these persons would then return a name, address, phone number and birth date. To show the server that these are the fields that need to be returned, the corresponding ACL (Listing 4.9) lists each of them as a `new-acl:possibleField`. Since these fields are explicitly mentioned, it is easy to determine which fields specific users may access. For this, there are two approaches: either all fields can be accessed by default, except for some blocked fields that are mentioned in the ACL, or no fields can be accessed by default, except for the fields that were mentioned in the ACL. The first option is more useful when all users can access most of the fields, while the latter option is more restrictive and assumes that users do not have a lot of access by default, which is also the case in this example. In this thesis, it was opted to use the latter option as explicitly giving permission to see fields is easier to implement. In the example, the owner can see all fields, which is shown as following triple in the ACL: `#owner new-acl:allowsField new-acl:allFields`. Other users can only see the `:hasName` field by default, and the person with WebID `https://random.person.me/profile/card#me` can see both names and birth dates. The server will generate different queries for each type of user, selecting only the user's allowed fields. The query for default users is displayed in Listing 4.10, which clearly selects nothing but the names of all persons.

### Filtering records

Entire results can also be filtered. For instance, instead of sending the information of Lukas, Sarah and Anne-Laure, only the information of Lukas and Anne-Laure is sent. This is horizontal filtering, whereas in the previous case vertical filtering was used. Using the graph of the first scenario, one could for instance disallow a certain user to see persons who have `Sarah` as name. Things really start to get complex when this user is also not able to see people that are a Tinder date of Lukas. This truly shows the advantage of using query-based access over file-based access, since in the latter case, all persons have a file of their own, and filtering out the Tinder dates would require accessing the file of Lukas beforehand. Even more complex rules might be very impractical when using file-based querying, for example all persons that share their name with another person are filtered when some user requests the information of all people. This would require collecting all persons to determine the persons that should be filtered. In contrast, using SPARQL to query a single graph is straight-forward and efficient. Very complex rules like the last one are not supported by the ACL extension shown in this thesis, but serves merely as an example of what is possible with future ACLs.

Again, the choice between inclusive and exclusive filtering needs to be made. For this thesis, the assumption is made that the number of returned results is usually close to the total number of possible results, and only few records are filtered away. In other words, exclusive filtering will be used. This is realized by adding restrictions to the ACL. In the first phase, these are rather simple and consist of a predicate (`new-acl:restrictPredicate`) and either a subject (`new-acl:restrictSubject`) or an object (`new-acl:restrictObject`). For instance, say we want to exclude persons of which the name is `Sarah`. In this case, the predicate will be `:hasName` and the object will be `"Sarah"`. When the server generates a query, following rule will be added to the query: `FILTER NOT EXISTS { ?s :hasName "Sarah" }`. Similarly, a restriction that is defined by predicate and subject is also added to the default ACL in Listing 4.9, and results in filtering out all subjects that are a Tinder date of Lukas (i.e. `:tinderGirl`). In the future, this ACL

could be extended to also allow a restriction to have just a single `restrictSubject`, `restrictPredicate` or `restrictObject`, in which case two new rules would be added. For instance, if a restriction has a `restrictPredicate` of `:hasTinderDate`, the following two rules would be added: `FILTER NOT EXISTS { ?s :hasTinderDate [] }` and `FILTER NOT EXISTS { [] :hasTinderDate ?s }`, where `[]` is a blank node.

```
PREFIX : <http://example.org/#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX acl: <http://www.w3.org/ns/auth/acl#>
PREFIX new-acl: <https://vanhoucke.me/ontology/new-acl#>

<https://lukas.vanhoucke.me/profile/card#me> a foaf:person, foaf:agent ;
    :hasName "Lukas" ;
    :hasAddress "Some Address";
    :hasPhoneNumber "0800-SOME-PHONE";
    :hasBirthday "05-July-1994";
    :hasFamily :sarah ;
    :hasFamily :laure ;
    :hasTinderDate :tinderGirl .

:sarah a foaf:person, foaf:agent ;
    :hasName "Sarah" ;
    :hasAddress "Some Address";
    :hasPhoneNumber "0800-SOME-PHONE";
    :hasBirthday "30-April-1992" .

:laure a foaf:person, foaf:agent ;
    :hasName "Anne-Laure" ;
    :hasAddress "Some Address";
    :hasPhoneNumber "0800-SOME-PHONE";
    :hasBirthday "27-May-1996" .

:tinderGirl a foaf:person, foaf:agent ;
    :hasName "Tinder-Date" ;
    :hasAddress "Some Address";
    :hasPhoneNumber "0800-SOME-PHONE";
    :hasBirthday "01-January-1970" .
```

**Listing 4.8:** Graph content in the first scenario: data of four persons excluding ACL.



```

<#this> new-acl:possibleField
  [new-acl:predicate :hasName; new-acl:label "name"],
  [new-acl:predicate :hasAddress; new-acl:label "address"],
  [new-acl:predicate :hasPhoneNumber; new-acl:label "phone"],
  [new-acl:predicate :hasBirthday; new-acl:label "birthday"] .

<#owner> a acl:Authorization;
  acl:agent <https://lukas.vanhoucke.me/profile/card#me>;
  new-acl:allowsField new-acl:allFields .

<> a acl:Authorization; # Blank nodes for new users.
  acl:agent <https://random.person.me/profile/card#me>;
  new-acl:allowsField :hasName, :hasBirthday ;
  # Restrictions:
  new-acl:hasRestriction [
    new-acl:restrictPredicate :hasName ;
    new-acl:restrictObject "Anne-Laure" ;
  ].

<#default> a acl:Authorization;
  acl:agentClass foaf:Agent;
  new-acl:allowsField :hasName;
  # Restrictions:
  new-acl:hasRestriction [
    new-acl:restrictPredicate :hasName ;
    new-acl:restrictObject "Sarah" ;
  ];
  new-acl:hasRestriction [
    new-acl:restrictSubject <https://lukas.vanhoucke.me/profile/card#me> ;
    new-acl:restrictPredicate :hasTinderDate ;
  ].

```

**Listing 4.9:** Graph content in the first scenario: corresponding ACL.

```

SELECT ?name
WHERE {
  ?s a foaf:person .
  OPTIONAL { ?s :hasName ?name }
  FILTER NOT EXISTS {
    <https://lukas.vanhoucke.me/profile/card#me> :hasTinderDate ?s }
  FILTER NOT EXISTS { ?s :hasName "Sarah" }
}

```

**Listing 4.10:** The query for default users that was generated by the server. It will return the names of Lukas and Anne-Laure when provided the graph content of the first scenario.

#### 4.6.4 Adding advanced ACL to the SPARQL server

This ACL was implemented to demonstrate how the SPARQL server should create personalized queries. The actual code is written in JavaScript (using the same Express based back-end as in the test setups of

Chapter 5) and can be found at <https://gist.github.com/Lukkie/83e5d25fe2dbce8c66199a74a7eb6b2d>. A reader-friendly pseudocode version is found in Algorithm 1. It consists of four steps.

First, all queryable predicates are collected, along with suggested labels. For instance, the ACL of Listing 4.9 contains four possible fields, one of which has `:hasName` as predicate and `name` as label. In the final query, this label is used to define an output variable `?name`, as seen in the generated query in Listing 4.10.

The second step is to collect all the predicates that the user is allowed to see. This process starts at line 5. If the ACL does not contain any specific rules for this user, the default predicates are used. Collecting these requires executing another query. If the ACL does contain rules for the user, there are two options. Either the user can access all fields (indicated by a `new-acl:allFields` triple), and the same predicates as the ones collected in step 1 are used. Otherwise, the user can only access the fields that are specified in the ACL.

Then, the restrictions are collected at line 14. If in the previous step the default predicates were collected, the default restrictions will now also be used. Otherwise, restrictions that are specific for this WebID are collected instead. In both cases, an additional query has to be performed.

Finally, using the data of the previous three steps, a personalized query can be composed at line 20 of the pseudocode. The labels are used to generate variables (e.g. `?name`), the allowed fields are used to determine which variables will be added to the query along with the `OPTIONAL` lines of Listing 4.10. Lastly, the restrictions are applied in the form of `FILTER NOT EXISTS` lines, as can again be seen in Listing 4.10. This query is then executed by the SPARQL endpoint, and the results are returned to the user.

---

**Algorithm 1:** Pseudocode to apply advanced ACL. Here, `query()` executes a query at the SPARQL endpoint.

---

**Input** : WebID

```

1 possible_fields ← query(get all possible fields)
2 labels ← getLabels(possible_fields)
3 all_predicates ← getPredicates(possible_fields)
4
5 personalized_predicates ← query(get allowed fields for this WebID)
6 if personalized_predicates is empty then
7   | allowed_predicates ← query(get default allowed fields)           ▷ Default scenario
8 else if personalized_predicates contains new-acl:allFields then
9   | allowed_predicates ← all_predicates
10 else
11   | allowed_predicates ← personalized_predicates
12 end
13
14 if personalized_predicates is empty then
15   | restrictions ← query(get default restrictions)                   ▷ Default fields and restrictions
16 else
17   | restrictions ← query(get restrictions for this WebID)           ▷ Restrictions can be empty
18 end
19
20 personalized_query ← composeQuery(allowed_predicates, restrictions, labels)
21 results ← query(personalized_query)

```

**Output:** Personalized results of query

---

### 4.6.5 Adapting the ACL extension for annotations

The ACL extension works great for simple scenarios, such as the first one. The next scenario is a case study in which the ACL will be applied for annotations. In this use case, an endpoint returns the information of all annotations, but filters may be applied for certain users. Some example filters are: basic users cannot see annotations for an admins-only page, or they are not able to see the text values of the annotations.

Here, the endpoint will again generate a query to collect all annotations and return the relevant fields. An annotation contains hierarchic links and therefore requires complexer server-side code that implements application-specific knowledge. Querying annotations will for example require knowing that a resource has an `rdf:type` of `oa:Annotation`, and that annotations can have a target (`oa:hasTarget`), and the target then usually has a selector (`oa:hasSelector`), etc. This raises a first problem with the new ACL's initial implementation. The current ontology only supports predicates that have `?s` as subject to be a `possibleField`. For instance, when `dc:creator` and `rdf:value` are the two `possibleFields`, then a query to return all values of the possible fields would look like Listing 4.11:

```
SELECT ?creator ?text WHERE {
  ?s a oa:Annotation .
  OPTIONAL { ?s dc:creator ?creator } .
  OPTIONAL { ?s rdf:value ?text} .
}
```

**Listing 4.11:** Query to demonstrate the problem with `possibleField`. Prefixes are hidden for conciseness. It should return the creator and the textual content of all annotations. Due to the complex structure of an annotation, the textual content is always empty.

For a valid annotation, this will never return a value for `?text`, since the `rdf:value` is coupled to an annotation's body, and not to the annotation directly. In order to get this value, the query should instead look like Listing 4.12:

```
SELECT ?creator ?text WHERE {
  ?s a oa:Annotation .
  OPTIONAL { ?s dc:creator ?creator } .
  OPTIONAL {
    ?s oa:hasBody ?body .
    ?body rdf:value ?text .
  } .
}
```

**Listing 4.12:** Query showing the solution to the problem with `possibleField`. Prefixes are hidden for conciseness.

In other words, in order to allow filtering on fields, the server needs to know how an annotation is structured, and how the possible fields are related to this structure. This is a large drawback, since it cannot easily be extended to applications other than annotations. Arguably, one could make an ACL for each possible subject, i.e. an ACL for the annotation, and another ACL for its body, etc. An additional triple for each of these subjects would then point to the corresponding ACL. This solution requires less server knowledge, but it might bloat the graph with numerous complex ACL triples. The proposed solution in this thesis uses just one ACL for all annotations as this improves query performance.

The ACL extension also allows horizontal filtering. This again raises a problem when applied to annotations. In fact, the reason why it does not work is the same as for the previous problem. When a restriction has specified a `restrictPredicate` and a `restrictObject`, the generated query cannot always use `?s` to fill in the subject field of the corresponding `FILTER NOT EXISTS` clause. The query in Listing 4.13 demonstrates this problem.

```
SELECT ?text WHERE {
  ?s a oa:Annotation .
  ?s oa:hasBody ?body .
  ?body rdf:value ?text .
  FILTER NOT EXISTS { ?s rdf:value "SPAM" } .
}
```

**Listing 4.13:** Query to demonstrate the problem with restrictions. Prefixes are hidden for conciseness. It should return the textual content of all annotations in the graph for which this content does not equal “SPAM”. Due to the complex structure of an annotation, this restriction does not work.

Annotations do not have an `rdf:value` field by default, and more importantly, the `rdf:value` field is located in the annotation’s body. Therefore, no annotations will be filtered away. Successful filtering requires the subject to be `?body`, as shown at the bottom of Listing 4.14.

```
SELECT ?text WHERE {
  ?s a oa:Annotation .
  ?s oa:hasBody ?body .
  ?body rdf:value ?text .
  FILTER NOT EXISTS { ?body rdf:value "SPAM" } .
}
```

**Listing 4.14:** Query showing the solution to the problem with restrictions. Prefixes are hidden for conciseness.

This can again be solved in two ways: either the server is intelligent enough to know which subject or object to use, or the ACL gives a hint to the server by providing the variable name (e.g. `?body`) which would be used in the server’s generated query. Unfortunately, both options are tightly coupled to the application and how the server generates the queries. In this thesis, the second option was implemented since it requires less server-side coding. The ACL is however not comprehensible for other servers that generate the queries in a different way. Here, the hints are provided by adding two new terms to the ontology: `restrictSubjectField` and `restrictObjectField`. These define the variable to be used in generating the restriction’s corresponding `FILTER NOT EXISTS` clause. To show how this works in practice, an example graph content and query is attached in Appendix A.

As the solution is application-dependent and not easily reusable, it is not useful in practice. However, it would be more effective when an annotation’s structure is set in stone and included in the graph, so that applications know exactly what the annotation looks like. Restrictions could then reference a specific node in this structure, so that the application knows which variable to use. A current standard of applying a constrained structure to entities is through the use of SHACL [49]. It uses RDF to represent shapes, which are conditions that the data structure has to confirm to. Each node is then described by such a shape, which can be referenced by the ACL to specify the subject or object of the restriction.

Since annotations are self-contained (i.e. the annotation file contains all of its information<sup>6</sup>), there is usually no need for a complex ACL such as the demonstrated one. If some fields do need to be filtered, it would also be better to use an ACL on a per-annotation basis, preferably by allowing inheritable ACL through the use of some directory structure. Although vertical filtering is not possible in file-based access techniques (i.e. the server can only send whole RDF files), `WebAccessControl` is sufficient in almost all situations.

---

<sup>6</sup>Comments on annotations might be an exception, as a link to this comment can be added to the original annotation. This is however irrelevant, since the presented ACL is for a single datapod, i.e. comments (which are annotations as well) in other user’s datapods will have their own ACL.

## 4.7 Caching

Caching is the act of keeping copies in order to speed up the system. A drawback of decentralization is that resources have to be collected from all around the globe, which takes a long time and also stresses the network. Instead, some entity could keep copies from multiple sources and send all copies at once. Doing so is possible for both file-based and query-based access, since both servers are implemented as RESTful endpoints. This implies that successive<sup>7</sup> GET requests with the same parameters and URLs always return the same result, which is the only requirement for caching. The file-based cache holds files, while the query-based cache holds field values.

Specifically for annotations, the server that is announced by the website's `as:annotationService` could make a copy of all foreign annotations that are listed in its annotation list. The link of the cached version would then be sent to the clients, so that only one server has to be accessed during the collection process. Each cached annotation needs to be refreshed periodically, which can be realized by using a “valid-until” time stamp. Manual refreshes can be announced by sending a Linked Data Notification to the server's inbox, announcing an update. Alternatively, the server could also send all relevant annotations at once so that the client does not have to collect them one by one. This however requires complexer server-code, and is further discussed in section 5.5.5.

The corresponding ACL files have to be included in the cache, otherwise plenty of requests still have to be sent. The query-based ACL (including the advanced ACL from section 4.6.3) is a lot harder to cache. When the ACL is stored inside the foreign server's graph (and not as a file), it cannot be sent to a different server. Especially in cases where advanced ACL is used, the ACL might depend on lots of information that is only stored on the foreign server. When simple ACL is used, caching can still be possible. The ACL is then transmitted in the form of several fields, containing enough information for the receiving server to reproduce the ACL statements internally. Caching ACL implies that the data owners allow their ACL files to be public, which is definitely not always the case! A possible solution would be to cache all annotations that are announced to be public by the creator, and only keep a link to the annotation's location when the creator announces it not to be public.

## 4.8 Versioning

In some cases, it can be useful to keep a history of your data's content. For this, version control systems are used. Git and Subversion are some of the most popular technologies. They keep track of changes in files, which makes them immediately applicable for file-based datapods. In theory, it should be possible to request old versions of Linked Data Resources by specifying a date. These technologies cannot be used in combination with query-based access techniques, since they're simply not available for SPARQL endpoints. Instead, custom middleware could be written to keep track of changes to the graph, and to recreate the content at any point in time. This is however very complex and is still an active topic of research. OSTRICH [79] is an example of such a system. It stores the initial version of the graph as an immutable snapshot, while all future versions are stored as time-stamped lists of triples that need to be removed or added relative to the initial version or intermediate snapshots.

---

<sup>7</sup>No data modifications (e.g. POST or PUT requests) in between the successive GET requests.

## Chapter 5

# Implementation

In order to objectively compare file-based and query-based access, we have developed multiple tools. Most of these tools are aimed at decentralized annotations. This chapter starts with setting up the servers and ends with our implementation of a decentralized annotation plugin.

The code used in the implementations is available in the following UGent GitHub repositories:

- <https://github.ugent.be/lbvhouck/vanhoucke.me>
- <https://github.ugent.be/lbvhouck/DecentralizedAnnotations>
- <https://github.ugent.be/lbvhouck/QuickSPARQLEndpoint>
- <https://github.ugent.be/lbvhouck/LinkedDataNotificationsWatcher>

### 5.1 Test setup

Multiple test setups have been developed in order to support the theoretical differences between file-based and query-based access techniques. In general, most setups are backed by a demonstrative website. Because Solid is written in JavaScript (both server and client libraries), and it is easier to use the same language for both front-end and back-end, the programming language of choice is JavaScript. Annotations are often used to add comments to a website, e.g. the Open Weblides platform, which again is an argument in favor of JavaScript. The demonstrative web pages are hosted on the <https://vanhoucke.me> domain, to be able to show the results to anyone and anywhere.

#### 5.1.1 Setup with file-based access

As mentioned before, file-based access will use Solid as the reference implementation. First of all, a Solid server needs to be set up. This implements the LDP specification on top of the operating system's file system. Technically, Solid can also implement LDP over other data storage techniques, such as SPARQL endpoints, but this is not the default setting. Although Solid does not yet have a large documentation to rely on, setting up a server is easy when following the steps on `solid-server`'s GitHub page. The hardest part is to configure a WebID with which you are able to authenticate on other servers over TLS. This requires a valid certificate which originally had to be generated by making use of HTML's `<keygen>` element, but this has been deprecated by major browsers. Instead, manually creating this certificate requires following steps<sup>1</sup>:

1. Create an asymmetric key pair (e.g. use the `ssh-keygen` tool)

---

<sup>1</sup>Steps are taken from <https://github.com/dindy/solid-resources/blob/master/webid-tls.md>, which documents the process of manually creating a certificate in detail.

2. Create a certificate (e.g. use the `openssl` tool)
3. Import the certificate in your browser
4. Add the certificate to your WebID profile

The datapod that was used for this thesis can be accessed at <https://lukas.vanhoucke.me>, and the WebID profile at <https://lukas.vanhoucke.me/profile/card#me>.

To use data that resides on a Solid server, the client-side JavaScript library `solid-client`<sup>2</sup> can be used, often in combination with `rdflib.js`<sup>3</sup> to manipulate and query data.

### 5.1.2 Setup with query-based access

While the setup for file-based access already has an implementation ready to be used, there is no equivalent setup for query-based access. Therefore, an experimental setup with only the necessary features was developed. Just like Solid, it exposes resources in a RESTful way. Solid uses LDP by design, and in a way, the query-based setup is similar to LDP (HTTP, RESTful), but still differs in many ways (no directory structure, no LDP resources). All data is stored in the same graph. To store triples, Apache Jena Fuseki is used. This is a simple SPARQL endpoint accessible over HTTP. The server runs behind a closed firewall so as not to expose the raw data to the users. Instead, the data is queried by a middleware application. For this, Express is used for routing, and Node.js to implement the necessary middleware code, which uses either a dynamically generated SPARQL query, or a static query, to send a query request to the SPARQL endpoint. To demonstrate this, we provide a simple pseudocode example, shown in Algorithm 2. The actual Express and Node.js implementations have been added to Appendix B. There, the Express routing code is shown in Listing B.1, and the corresponding Node.js server code to handle the GET request to `/annotations` is shown in Listing B.2.

---

**Algorithm 2:** Pseudocode to apply advanced ACL. Here, `query()` executes a query at the SPARQL endpoint.

---

**parameter:** SPARQL endpoint URL

**parameter:** One or more routes with a corresponding query *Routes*

**Input** : Request

```

1 foreach route  $R \in Routes$  do
2   if  $R$  matches request URL then
3     Send query associated with  $R$  to SPARQL endpoint
4     if Response  $Res$  retrieved then
5       return  $Response(status: 200, body: Res.body)$ 
6     else
7       return  $Response(status: 500)$ 
8     end
9   end
10 end
11 if No routes matched then
12   return  $Response(status: 404)$ 
13 end

```

**Output** : Response

---

The middleware supports two types of query-based access. As explained in chapter 3, annotations can all be stored in the same graph, or they can each be stored in a separate graph representing different

<sup>2</sup><https://github.com/solid/solid-client>

<sup>3</sup><https://github.com/linkedin/rdf-lib.js>

## Annotation generator

Generates annotation and stores them in your solid datapod and the server's SPARQL equivalent. Timing results are outputted to console.

Pressing the second button will also collect the stored annotations solely for timing purposes.

Note that your solid datapod requires an annotation container that is announced by an `oa:annotationService` triple in your profile. Press following button to create a directory and add the triple to your profile.

Set up annotation directory

Store all annotations in your own directory?

Number of users:  10

Number of websites:  10

Number of annotations:  50

Generate annotations    + Load annotations

**Figure 5.1:** The annotation generator found at <https://vanhoucke.me/browser-tests/solid-generator/generator.html>.

files, just like file-based access. Both options have been implemented and will respectively be referred to as the single-graph SPARQL server and the multi-graph SPARQL server. When not explicitly stated, the single-graph SPARQL server will be assumed. Why “SPARQL” server? Because it uses a SPARQL endpoint internally.

### 5.1.3 Generator

Since creating many annotations manually is a tedious task, an annotation generator was developed. First of all, the user has the option to either save all annotations in his own annotation directory, which requires an `oa:annotationService` triple in his profile. If this is not the case, the generator also facilitates a button to create an annotation directory and add this triple to his profile automatically. In the other case, if the user does not want the annotations to be stored in his own annotation directory, fake users are created, each of them having their own annotation directory. These fake users are implemented as directories in the standard annotations directory. The number of users is configurable. Annotations are linked to websites, so the generator generates fake URLs and randomly selects one of these URLs per annotation. The number of fake URLs can also be configured, as well as the the number of annotations. Finally, some performance measuring tools are included in the code, such as timing benchmarks, and a button to load all annotations just to see how long it would take. The generated annotations are then stored in the logged-in user's Solid datapod, the single-graph SPARQL server and in the multi-graph SPARQL server.

The generator can publicly be accessed at <https://vanhoucke.me/browser-tests/solid-generator/generator.html>. In case the page is offline, a screenshot is provided in Figure 5.1



## 5.2 Implementation of an inbox listener for file-based access

When an annotation is added to a site, a Linked Data Notification should be sent to the site's inbox in order to announce the location of this new annotation. We decided that this notification file requires just a single triple: `<> as:Announce <annotation_URL> .` Here, `as` is the prefix for the activity streams ontology<sup>4</sup>. To process these notifications server-side, a Node.js script was used. A listener is attached to the server's inbox directory, and each added file triggers a mechanism to analyze the file. First, the file needs to contain an `as:Announce` triple, to make sure that it is a notification. Then, the annotation at `<annotation_URL>` is collected in order to get its source URL. Finally, a triple of the form `<source_URL> as:items <annotation_URL>` is added to the list of annotations that is stored in the server's public directory. Now, all users can collect and query this list to retrieve all annotations that are targeted at the user's current website.

## 5.3 Implementation of a directory structure for the single-graph query-based server

We created a simple additional test setup to provide a basis for future query-based servers that would like to implement directories. The API allows the creation of an annotation inside the annotations directory. Any directory can be queried to request its content. Finally, information about files in the annotation directory can be retrieved. The setup's capabilities are rather minimal, but it was developed only to prove the point that directories can easily be supported.

To store a resource, first it needs to be checked that the directory exists. This can be accomplished using the query in Listing 5.1. Optionally, if the directory does not exist, a new directory can be created (or multiple directories, in which case this process is recursively repeated). Then, the resource can be added to this directory by creating the resource and adding its URI to the directory's `ldp:contains` triples. This is demonstrated in Listing 5.2. To get the contents of a directory, all subjects of the directories `ldp:contains` triples are requested. This is shown in Listing 5.3. To request a file inside a directory, first the existence of this directory has to be confirmed. The same process is followed to see if the file is indeed part of this directory. Therefore, the triple pattern of Listing 5.1 is replaced with `{{directory_name}} ldp:contains {{file_name}}`. Once it is known that the resource exists, it can be queried in the usual way.

```
ASK WHERE {
  {{directory_name}} a ldp:BasicContainer .
}
```

**Listing 5.1:** Query to check if a directory exists. `{{directory_name}}` is replaced by the actual directory's name. Prefixes are hidden for conciseness.

---

<sup>4</sup><http://www.w3.org/ns/activitystreams>

```

INSERT DATA {

  <annotations> ldp:contains <annotations/{{annotation_id}}> .

  <annotations/{{annotation_id}}> a oa:Annotation;
    oa:hasBody [
      a oa:TextualBody ;
      rdf:value "Sample comment with id {{annotation_id}}"@en ;
    ] ;
    oa:hasTarget [
      a oa:SpecificResource ;
      oa:hasSource <www.example.com/some_page.html> ;
    ] ;
    rdfs:label "A sample comment for demonstrational purposes"@en .

};

```

**Listing 5.2:** Query to insert a sample annotation into the annotations directory. `{{annotation_id}}` is replaced by an ID provided by the user. Using a different query, it is first asserted that an annotation with this ID does not already exist. Prefixes are hidden for conciseness.

```

SELECT ?item {
  <annotations/{{directory_name}}> a ldp:BasicContainer ;
  ldp:contains ?item .
}

```

**Listing 5.3:** Query to retrieve the contents of a directory. `{{directory_name}}` is replaced by the name of the directory. First, it is checked whether the directory exists by executing the query in Listing 5.1. Prefixes are hidden for conciseness.

## 5.4 Demonstrative setup for intersections

A demonstrative setup was developed in order to show the impact of intersections on both the file-based and query-based techniques. Just like in Section 4.5, the use case is to search for certain words in all annotation bodies. The file-based setup<sup>5</sup> uses Solid. After logging in, all annotations in the user's `oa:AnnotationService` directory are collected and stored into a single graph. The JavaScript code to collect annotations is shown in Listing 5.4. Since `rdflib.js` does not have regex functionality as far as we know, the querying code in Listing 5.4 has to rely on JavaScript's `indexOf` to manually look for occurrences of the words. Clearly, a lot of client-side processing is necessary, along with a separate GET request for each annotation.

<sup>5</sup><https://vanhoucke.me/browser-tests/solid-intersection/demonstrator.html>

```

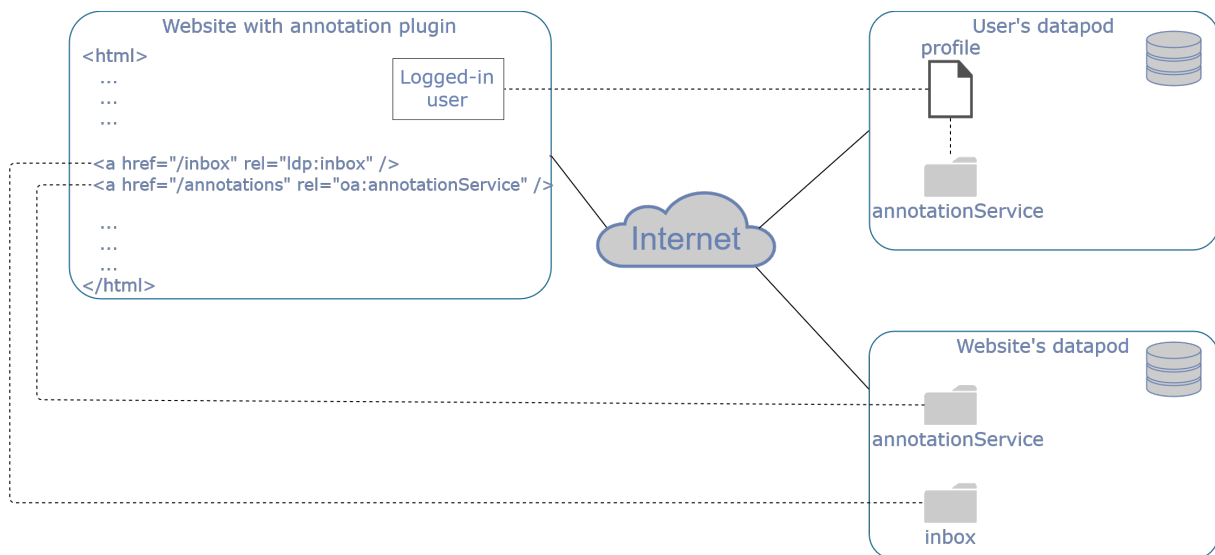
1  function collectAnnotationLocations(profile) {
2    let save_location = profile.find(vocab.oid('annotationService'));
3    return new Promise(function(resolve, reject) {
4      // Get all annotation locations
5      solid.web.get(save_location)
6        .then(function (container) {
7          resolve(Object.keys(container.resource.resources));
8        })
9    });
10 }
11
12 function collectAnnotations(locations) {
13   return new Promise(function(resolve, reject) {
14     let graph = null;
15     // Collect the annotations and store in graph
16     let counter = 0;
17     for (let i = 0; i < locations.length; i++) {
18       let location = locations[i];
19       solid.web.get(location)
20         .then(function (response) {
21           // Merge graph with current graph
22           if (graph === null) graph = response.parsedGraph();
23           else graph.add(response.parsedGraph());
24           // If last location: Return the graph
25           counter++;
26           if (counter == locations.length) resolve(graph);
27         }).catch(function(err) {
28           reject(err);
29         });
30     }
31   });
32 }
33
34 function queryAnnotations(graph, stringToBeContained) {
35   return new Promise(function(resolve, reject) {
36     let results = [ ];
37     if (graph === null) {
38       resolve(results);
39     }
40     // example query: Find all strings that contain 'Lorem ipsum dolor'
41     let comments = graph.statementsMatching(undefined, vocab.rdf('value'), undefined);
42     comments.forEach(function(comment) {
43       let value = comment.object.value;
44       if (value.indexOf(stringToBeContained) !== -1) results.push(comment);
45     })
46     resolve(results);
47   });
48 }

```

**Listing 5.4:** Three client-side JavaScript functions that are necessary to collect and query all annotations when using file-based access. The first returns a list of all annotations in the user's annotation directory. The second function collects each annotation and stores it in a single graph. The third function finds all annotations for which the comment value contains specific words.

Collecting the search results when a SPARQL endpoint is used, requires less client-side processing. In fact, just a single GET request is all that is needed. The equivalent JavaScript code is given in Listing 5.5. At the server's side, the query in Listing 4.6 is executed and the results are sent back.

Clearly, file-based intersecting queries are more complex and take longer to execute. When many intersections are required, it is best to switch to query-based storage. Alternatively, solutions such as indexing



**Figure 5.2:** Architectural diagram showing the connections between the annotation plugin and the datapods. Note that the website, the user's datapod and the website's datapod may reside on different servers.

can be applied, but this is not always a possibility.

```

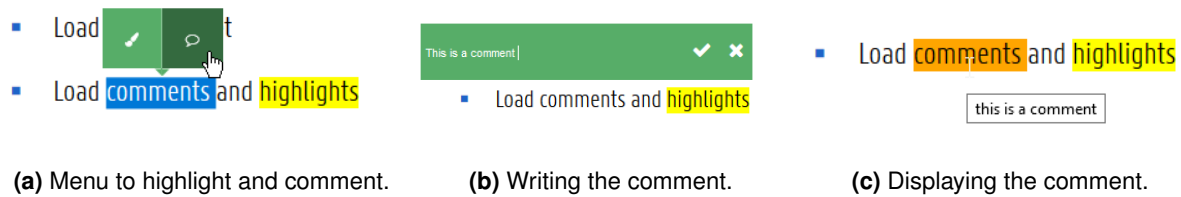
1 function collectFilteredAnnotations(filter) {
2   return new Promise(function(resolve, reject) {
3     let sparql_endpoint = 'https://vanhoucke.me/sparql-2/annotations/filter';
4     request
5       .get(sparql_endpoint)
6       .query({filter: filter})
7       .end((err, res) => {
8         if (!err) resolve(JSON.parse(res.text));
9         else { reject(err); }
10      });
11   });
12 }

```

**Listing 5.5:** Client-side JavaScript code to collect the filtered annotations. This is the query-based variant of Listing 5.4.

## 5.5 Annotation Plugin

The theory behind decentralized annotations was put to the test by implementing it. To be able to test it on any web page, a bookmarklet was used to inject the necessary Javascript code. The initial version was written to make use of Solid (i.e. file-based access). The plugin should first read the website's `oa:AnnotationService` field, which is able to list all the (public) annotations for this page. Since few sites actually have such a service, the plugin uses its own service instead. Similarly, a link to the website's `ldp:inbox` should also be present, but was instead replaced by the plugin's hard-coded inbox due to the fact that there's no point in sending notifications to someones inbox when they have no idea what to do with it. An ideal architecture is shown in Figure 5.2.



**Figure 5.3:** GUI of the annotation plugin. MediumEditor is used to show the buttons, FontAwesome provides the button icons and Rangy is used to highlight text.

### 5.5.1 Inspiration

The decentralized editor dokieli (see section 2.5.3) was an inspiration for this annotation plugin. Although dokieli offers annotation functionality, it does not fully use Solid’s capabilities (e.g. dokieli manually checks whether the user’s WebID certificate is valid.) and the code is bloated with other functionality. The presented annotation plugin aims to be a clean and easy-to-comprehend alternative to dokieli, especially for cases where only annotations are required.

### 5.5.2 User interface

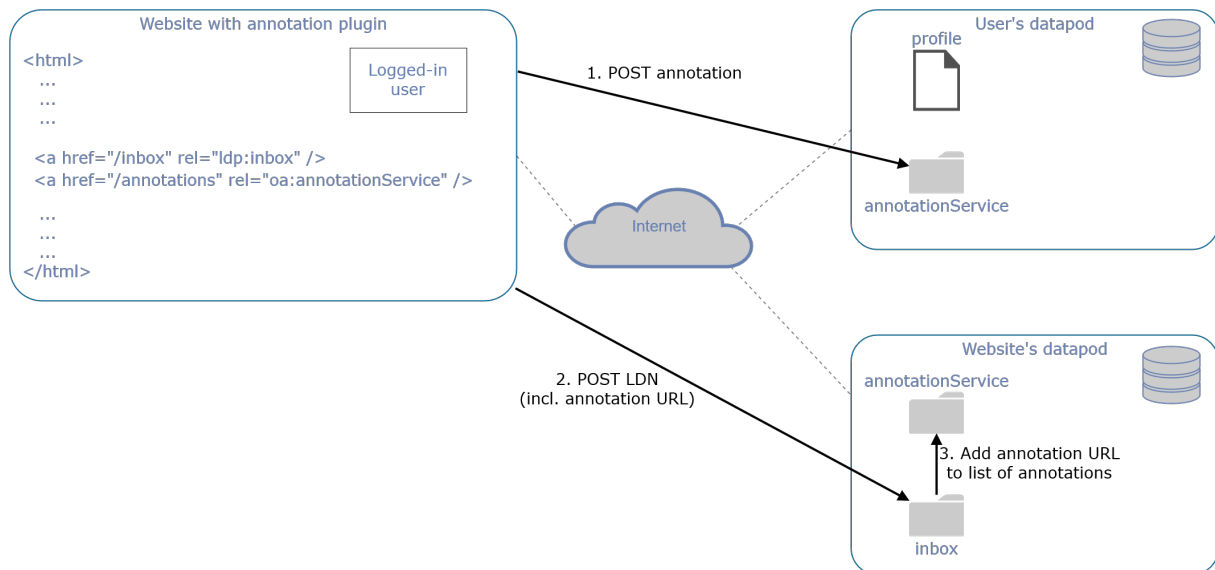
For the UI, MediumEditor<sup>6</sup> was used. This clone of Medium.com’s editor is mainly used as an editor (e.g. apply bold or underline tags to text), but can be edited to suit your needs, like doing non-editor actions. The editor pops up when text is selected, but the creator of the plugin has to define what text is selectable. In the case of Open Webslides, any text within a container of the `slide` class, is considered selectable text. In this case, upon selecting some text, two buttons appear: one to highlight text and one to comment. Text is highlighted by the use of Rangy’s Classapplier module<sup>7</sup>. The buttons and rangy’s highlights are demonstrated in Figure 5.3. At the top right, a button can be clicked to retrieve all annotations.

### 5.5.3 Highlighter

When the highlight button is clicked, the currently selected text is highlighted by Rangy. It also attempts to store an annotation to the user’s annotation directory. Note that this annotation only requires a target but not a body. To specify what has to be selected, the `oa:TextQuoteSelector` is used, which defines the exact text that has to be highlighted, as well as the text immediately before and after this highlighted text. In some cases, the combination of exact text, prefix and suffix is not unique (e.g. when an article contains the same sentence twice). This is a weakness of this type of selector and can be solved by using a different kind of selector, such as the `TextPositionSelector`, which uses integer values (offsets) to define the exact position in the text. However, the `TextQuoteSelector` is often more reliable since it can deal with small changes to the website, which is preferred in the case of Open Webslides. The annotation contains more fields, such as the author (i.e. his WebID which can easily be retrieved by Solid), a title, a date and the URL of the targeted website. All of these fields are added to a graph similar to the one shown in Figure 4.1. The graph is then converted into Turtle serialization. Since only RDF data is included in the graph, the output will actually conform to the Turtle format as well. The annotation is then posted to the user’s annotation directory, which is announced in his profile as `oa:AnnotationService`. Upon success, the inbox of the website is notified about this new annotation (using the mechanism of section 5.2), so that it can add the annotation URL to its list of annotations. This is depicted in Figure 5.4.

<sup>6</sup><https://github.com/yabwe/medium-editor>

<sup>7</sup><https://github.com/timdown/rangy/wiki/Class-Applier-Module>



**Figure 5.4:** Diagram showing how an annotation is posted on the user's datapod, and then added to the website's datapod's list of annotations.

### 5.5.4 Comments

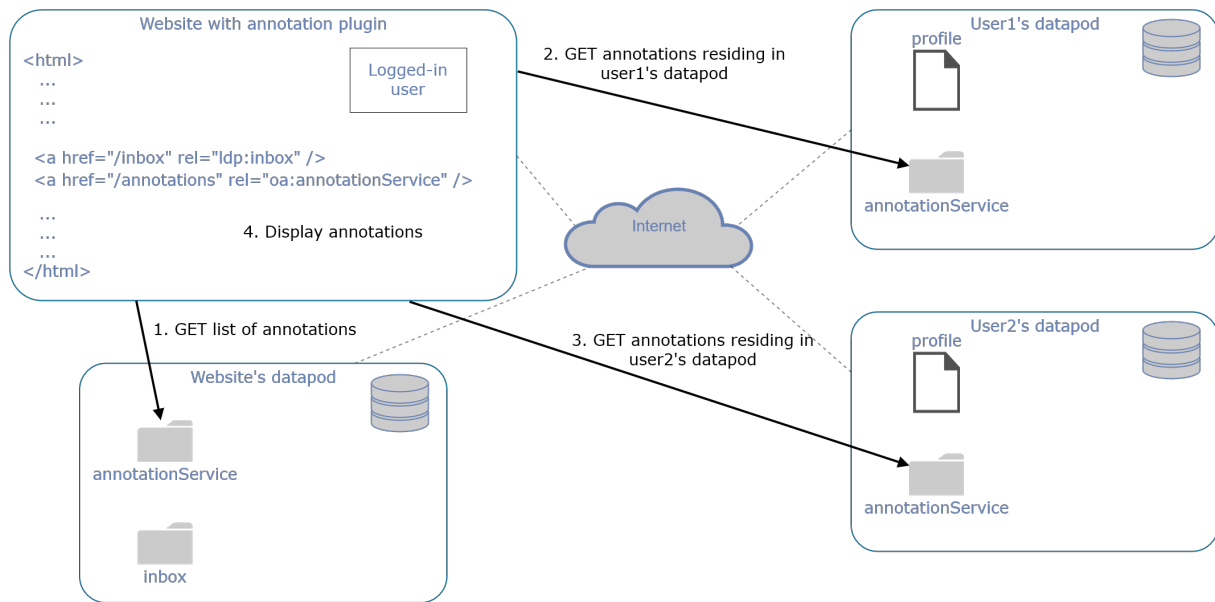
The comment button is functionally very close to the highlight button. The only difference is that upon clicking the button, the MediumEditor changes into a prompt for the comment's text. Here, the annotation does have a body, specifically a `oa:TextualBody` with an `rdf:value` equal to the inputted text. The comment annotation is saved in the exact same way as for highlight annotations.

### 5.5.5 Loading annotations

The load button in the top-right will collect all highlights and comments. In reality, this button is not required since annotations are usually loaded from the moment you open the page. As this is a demonstrative plugin, the annotations are loaded on-demand because it is easier to show the process of loading annotations. Specifically, there are two options to load annotations: either let the client collect a list of annotations and generate the corresponding HTML code client-side, or let the server collect his own list of (potentially cached) annotations and generate the HTML code upon loading of the page.

#### Client-side collection of annotations

For the first option, the client will first request the list of annotations to the server's `annotationService`. Note that private annotations can also be collected in the same way by contacting the private `annotationService` instead. If the server's `annotationService` returns a list of all annotations (instead of just the annotations that have the current page as target), the relevant annotations might have to be looked for first. This does not scale well, especially not when the list of annotations covers many pages. In this case, all annotations have to be collected when in reality, only a few are relevant. A more complex service is then preferred. All annotations on the (filtered) list are then collected, and the necessary fields are queried. In Solid, this can be accomplished by the `graph.any(subject, predicate, object)` function, in which an unknown field is indicated by assigning `undefined` to the field's value. The result of the query can then be stored in a variable. Since highlights and comments differ only in the presence of a `oa:hasBody` triple, the type of the annotation can easily be determined and the appropriate HTML code can be generated. This process is demonstrated in the JavaScript code in Listing 5.6. Graphically, it is shown in Figure 5.4.



**Figure 5.5:** Graphical representation of how annotations are loaded. First, the list of remote annotations is collected. Then, for each annotation on the list, a request to collect the annotation is sent to the corresponding datapod. When using query-based access, only a single request per datapod is required.

```

1 solid.web.get(annotationService).then(function(response) {
2   graph = response.parsedGraph();
3
4   current_url = window.location.href.split('#')[0];
5   graph.each(rdf.sym(current_url), vocab.as('items'), undefined)
6     .forEach(function(annotation_url) {
7
8     solid.web.get(annotation_url.value).then(function(response) {
9       let annotation_graph = response.parsedGraph();
10      let target = annotation_graph.any(annotation_url, vocab.oe('hasTarget'), undefined);
11      let selector = annotation_graph.any(target, vocab.oe('hasSelector'), undefined);
12      let text_quote = annotation_graph.any(selector, vocab.oe('refinedBy'), undefined);
13
14      let prefix = annotation_graph.any(text_quote, vocab.oe('prefix'), undefined).value;
15      let exact = annotation_graph.any(text_quote, vocab.oe('exact'), undefined).value;
16      let suffix = annotation_graph.any(text_quote, vocab.oe('suffix'), undefined).value;
17
18      // See if annotation is highlight or comment
19      let body = annotation_graph.any(annotation_url, vocab.oe('hasBody'), undefined);
20      if (body) {
21        // comment
22        let comment_value = annotation_graph.any(body, vocab.rdf('value'), undefined);
23        showComment(prefix, exact, suffix, comment_value);
24      } else {
25        // highlight
26        applyHighlight(prefix, exact, suffix);
27      }
28    });
29  });
30 }).catch(function(err) { /* Process error */ });

```

**Listing 5.6:** Client-side collection of annotations using Solid's JavaScript library.

## Server-side collection of annotations

The second option is to let the server collect all annotations. Doing so, the server will need to look at its local list of annotations, and then request all foreign annotations. A list of (potentially formatted) annotations can then be sent in a single packet to the client. This increases server-side complexity and processing time. However, the biggest advantage over client-side collection is the fact that the server can cache annotations of other users. Implementing this would significantly improve the performance of collecting annotations, which was one of the main disadvantages of using file-based servers for annotations. It also reduces overall network load. ACL is unchanged: each annotation's ACL still needs to be checked by the server, as was also the case for client-side collection. Solid was used for the file-based server, but it does not yet support advanced application-specific caching mechanisms. Different file-based LDP servers would also need to implement such mechanisms, which is against the purpose of being application independent. Implementing this was therefore skipped for now, but should definitely be considered in future research.

### 5.5.6 Query-based annotation plugin

The original file-based version (using Solid) was adapted to now use queries for data retrieval. For this, the single-graph back-end is used.

To create an annotation, a request is sent to a specific endpoint for annotations. The annotation's details are passed as body parameters. Following body keys are required: `creator`, `title`, `exact`, `prefix`, `suffix` and `source`. Optionally, a `body` key can also be attached to represent a comment's text. In the query-based server, a query is then generated by using the values of these fields. For this, the template in Listing 5.7 is used. Note that the server needs to replace newline and tab characters by explicit `\n` and `\t` characters to obtain a valid UPDATE statement.

Loading all annotations requires the same steps as in the file-based version, but there is one major difference: all annotations for the same webpage that reside in a single datapod can be queried at once, i.e. in a single request. This has several implications. First, after storing an annotation, its location should be announced to the server (or any private annotation listing). Instead of storing a link per annotation, now a link per datapod (typically one per user) is stored. Collection of all annotations typically requires less requests, significantly reducing the required time. The results are given as JSON, which requires minimal processing to extract the annotation's information. Where the file-based version takes a couple seconds to retrieve all annotations and apply the highlights, the query-based version is able to do the same in the blink of an eye.

As the query-based server does not yet support authentication, a user can fake its identity, and all annotations are public. This is however something that can be realized in future versions, and is not a limitation of query-based access techniques.



```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX oa: <http://www.w3.org/ns/oa#>
PREFIX dc: <http://purl.org/dc/terms/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

INSERT DATA {
  <{{annotation_identifier}}>
    a oa:Annotation;
    dc:created "{{date}}";
    dc:creator <{{creator}}>;
    rdfs:label "{{title}}"@en;
    oa:hasBody <{{annotation_identifier}}#body>;
    oa:hasTarget <{{annotation_identifier}}#target>;
    oa:motivatedBy oa:commenting.

  <{{annotation_identifier}}#body>
    a oa:TextualBody;
    rdf:value
      "{{body}}"@en.

  <{{annotation_identifier}}#fragment-selector>
    a oa:FragmentSelector;
    oa:refinedBy <{{annotation_identifier}}#text-quote-selector>.

  <{{annotation_identifier}}#text-quote-selector>
    a oa:TextQuoteSelector;
    oa:exact "{{exact}}"@en;
    oa:prefix "{{prefix}}"@en;
    oa:suffix "{{suffix}}"@en.

  <{{annotation_identifier}}#target>
    a oa:SpecificResource;
    oa:hasSelector <{{annotation_identifier}}#fragment-selector>;
    oa:hasSource <{{source}}>.
}

```

**Listing 5.7:** Template to store an annotation using SPARQL.

### 5.5.7 Possible extensions

The annotation plugin was developed in scope of a Master's thesis in order to demonstrate the workings of decentralized annotations. If the plugin were to be used in a production environment, and the developers aim to offer complete Annotation Data Model support, all five default selectors have to be implemented, among other requirements such as multi-language support, right-to-left text direction, etc.

The vocabulary can easily be extended. For instance, instead of only being able to annotate text, it could also be able to annotate images or even selected frames of a video. The Web Annotations Vocabulary can easily be extended to, for example, include 3D coordinates [90].

Besides extending the vocabulary, the plugin can be extended to allow editing annotations, allow multiple `annotationService` providers, show or hide annotations, have comment thread functionality, provide administration tools, and so on.

# Chapter 6

## Evaluation

In this chapter, the implementation of decentralized annotations is evaluated. First, some performance tests are performed to measure the timing differences between manipulating annotations on file-based and query-based servers. Then, the annotation plugin is tested by applying it to the Open Weblides platform and the pros and cons of the access techniques for annotations are assessed. Finally, decentralized social applications in general are discussed.

### 6.1 Performance

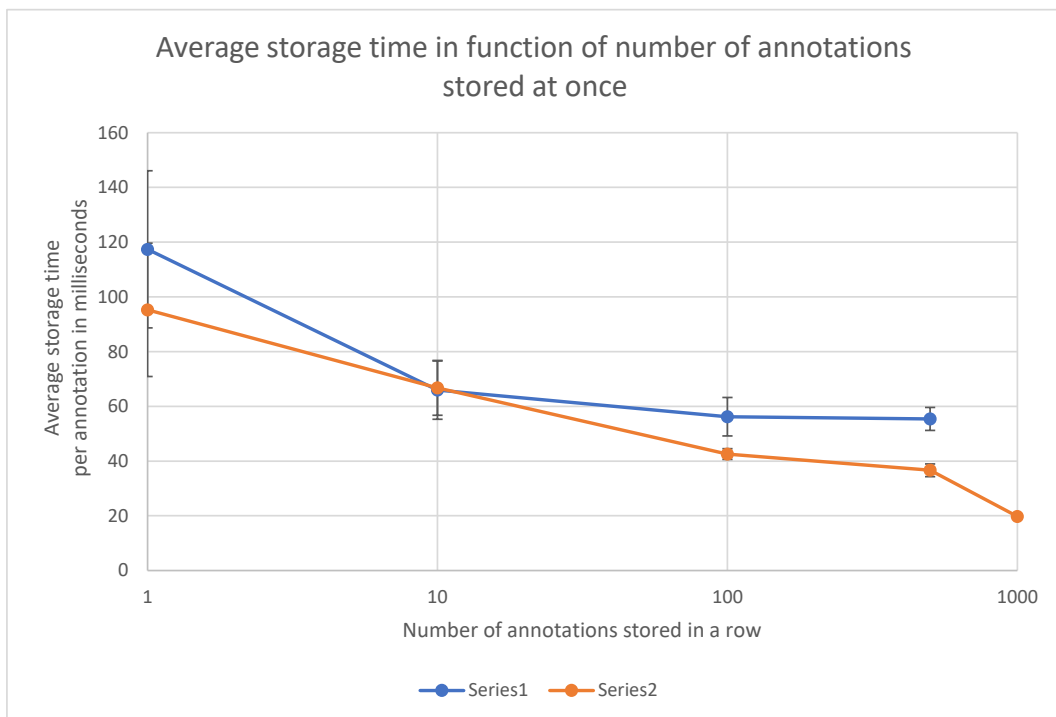
File-based and query-based access may potentially differ in performance. To assess these differences, we have created a test setup. It is able to collect every single annotation in the data pod, or only the ones that have a specific website as target. It can do this for both file-based and query-based techniques. Note that all tests are performed on the same datapod. This is not entirely realistic, as usually annotations from multiple users (and thus multiple datapods) have to be collected. However, the results are still meaningful, since caching techniques (see section 4.7) can be used such that the data is stored in a single datapod. The annotation generator of section 5.1.3 was also adapted to include timing measurements of storing annotations. All tests are executed on a laptop with 8 GB RAM and an Intel® Core™ i7-7500U CPU @ 2.70GHz. The annotations are stored on a remote server, for which a DigitalOcean droplet with a single vCPU and 2 GB RAM is used. Disk speed was not a bottleneck. The local laptop is connected to the Internet by a 50 Mbps connection. The results are available at <https://github.ugent.be/gist/lbvhouck/b6131e9f25ace00ce6b0bec00de34af5>.

#### 6.1.1 Storing annotations

In the first scenario, a number of annotations is stored on the server. The time between sending the first request and receiving the last confirmation is divided by the number of annotations to calculate the average storage time. Since requests happen asynchronously, the average time should decrease as the number of annotations increases. The test was performed for 1, 10, 100 and 500 annotations, and was executed 5 times each. It was attempted to do this test for 1000 annotations, but the Solid server could not handle this many requests. The SPARQL server however had no problem dealing with this. All previous data is cleared before running the test, and then a dry-run with a single annotation is performed. The results of this test can be found in Table 6.1 and is visualized in Figure 6.1. Clearly, storing annotations is slightly faster when using query-based techniques. However, the storage times for both access techniques are within the same order of magnitude. This is because each annotation is sent as a single request, and these communication delays are bottlenecking the system. Since the storage times for a single annotations are not very different, it can still be concluded that there are no extreme differences between the two techniques concerning storage.

**Table 6.1:** Time required to store a number of annotations using the file-based and query-based test setups.

| Number of annotations | Average time per file-based annotation (ms) | Average time per query-based annotation (ms) |
|-----------------------|---|--|
| 1                     | 117.3799589                                 | 95.26691644                                  |
| 10                    | 65.93104481                                 | 66.75438742                                  |
| 100                   | 56.21330136                                 | 42.57094193                                  |
| 500                   | 55.40218087                                 | 36.62920585                                  |
| 1000                  | No results                                  | 19.76281547                                  |

**Figure 6.1:** Results of the storage test visualized using a graph with corresponding standard deviations over 5 runs. The x-axis uses log-scale.

**Table 6.2:** Time required to collect a number of annotations using the file-based and query-based test setups.

| Number of annotations | Total file-based collection time (ms) | Average time per file-based annotation (ms) | Total query-based collection time (ms) | Average time per query-based annotation (ms) |
|-----------------------|---------------------------------------|---|--|--|
| 1                     | 466.9363043                           | 466.9363043                                 | 71.39485892                            | 71.39485892                                  |
| 10                    | 1181.116469                           | 118.1116469                                 | 78.77931478                            | 7.877931478                                  |
| 100                   | 7229.702002                           | 72.29702002                                 | 212.9854295                            | 2.129854295                                  |
| 500                   | 28647.38246                           | 57.29476492                                 | 253.2886921                            | 0.506577384                                  |

**Table 6.3:** Time required to collect a specific number of annotations out of a total of 200 annotations using the file-based and query-based test setups.

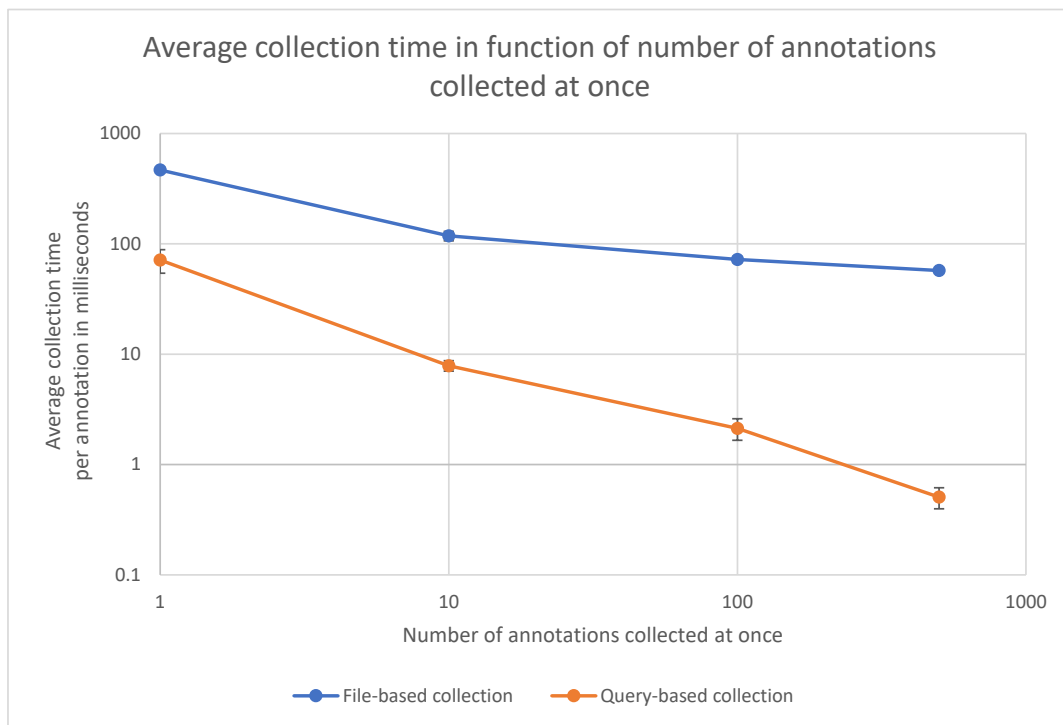
| Total number of annotations | Number of annotations for specific page | Average time per file-based annotation (ms) | Average time per query-based annotation (ms) |
|-----------------------------|---|---|--|
| 200                         | 1                                       | 11974.45299                                 | 82.37654375                                  |
| 200                         | 10                                      | 1195.653798                                 | 15.39008294                                  |
| 200                         | 50                                      | 241.8082521                                 | 3.701904918                                  |
| 200                         | 100                                     | 117.0484017                                 | 2.524178083                                  |

### 6.1.2 Loading annotations

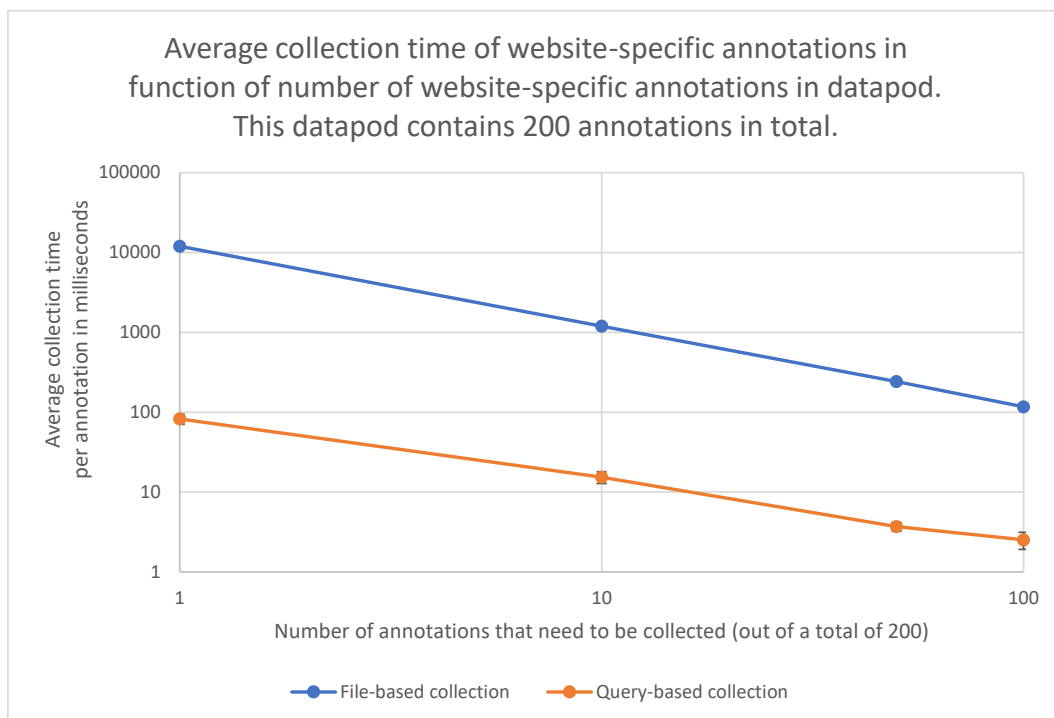
After having stored these annotations, they can now be loaded. In this test, all annotations for all websites are collected at once. This scenario happens when the server's `annotationService` contains links to annotations that are cached on the server's datapod, and there is a different annotation directory for every single page on the website. Again, the test is executed for 1, 10, 100 and 500 annotations. The time includes client-side parsing into a readable format. The results are shown in Table 6.2. Clearly, there is now a large difference between file-based and query-based retrieval. This time, the communication delay is less of a bottleneck for query-based access, since all annotations can be retrieved using a single request. Meanwhile, the number of requests that has to be sent when using a file-based server is equal to the number of annotations. When a website contains tens of annotations, loading these annotations starts taking a long time, which drastically worsens the user experience. Columns with the total retrieval times were added to show that the query-based collection is still dependent on the number of annotations, but scales considerably better than file-based collection. The results are visualized in Figure 6.2. SPARQL endpoints typically use database storage techniques and querying happens server-side using specialized query engines. This is not the case for file-based servers, which results in slower data retrieval and query performance.

### 6.1.3 Loading specific annotations

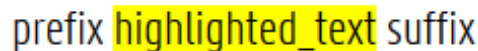
This final scenario assumes a datapod with 200 annotations, out of which a specific number of annotations have to be collected (including client-side parsing). This scenario occurs when a webserver uses the same `annotationService` (i.e. same annotation directory) for multiple of its independent webpages. Since the server does not query annotation files, and the files are located in the same directory without any structure, collecting the annotations that correspond to the page you are on requires collecting all annotations and querying them individually. Meanwhile, the query-based server exposes an endpoint to collect all relevant annotations using a single query. The test is executed for 1, 10, 50 and 100 specific annotations, and each test is executed five times. The results in Table 6.3 confirm again that query-based access vastly outperforms file-based access. It is visually shown in Figure 6.3.



**Figure 6.2:** Results of the loading test visualized using a graph with corresponding standard deviations over 5 runs. Both axes use log-scale.



**Figure 6.3:** Results of the specific loading test visualized using a graph with corresponding standard deviations over 5 runs. Both axes use log-scale.

The image shows the text "prefix highlighted\_text suffix" where the words "highlighted\_text" are highlighted in yellow. This illustrates the concept of a highlight being preceded by a prefix and followed by a suffix.

**Figure 6.4:** A highlight preceded by its prefix and succeeded by its suffix.

#### 6.1.4 Conclusion

A SPARQL endpoint in combination with query-based access is easily able to outperform file-based access techniques such as Solid. The fact that SPARQL endpoints typically use more advanced database storage techniques than a file-system could explain this. The most important reason is however that queries are able to collect all annotations at once, reducing communication delays tremendously. Collecting 100 annotations in over 7 seconds is really bad for the user experience. In the storage test, each annotation was sent as a separate message in both setups. Storing multiple RDF files at once is usually not something that happens in the context of annotations, but might happen in other social contexts, such as uploading a photo album with tags and other metadata. The query-based setup could then be adapted to allow a list of photo metadata to be sent in a single message, again reducing the communication delays. The same could theoretically be done for a file-based setup, although Solid implements LDP so only a single resource can be created at once.

## 6.2 Annotation plugin for Open Weblides

The annotation plugin described in section 5.5 was tested on the Open Weblides platform. As mentioned, the annotation was implemented as a bookmarklet. In order to use it as an actual annotation plugin supported by the website itself, some changes have to be made. First of all, instead of injecting the code, it should be loaded by default by including the JavaScript code and CSS stylesheets in the source of each Open Weblides document. Users should then be able to load annotations, but since the platform currently does not announce an `oa:AnnotationService` tag, users are unable to find where annotations are stored. Therefore, the source should have a working back-end (either file-based or query-based) and link towards the list of annotations by including the `AnnotationService` triple as RDFa. At the moment of writing, the bookmarklet does not yet parse RDFa, but `rdflib.js`, a library that is already included in the code, can be used to easily collect the location of the list of annotations. Preferably, there is a unique `AnnotationService` for each document. All annotations for all slides are then loaded at once. A triple pointing to the server's inbox should also be present in order to allow users to send a notification when a new annotation is added. The back-end should be able to deal with this. Note that an inbox can be shared among all Open Weblides on a domain if they all use the same back-end. When these two triples are present, the Linked Data infrastructure should work.

The plugin uses MediumEditor to select text and add annotations. This framework allows specifying a *selector* to make sure that only specific parts of the text can be selected. In the case of Open Weblides, we decided to allow selecting all elements that inherit the `.slide` class, as this is the actual content of the slides. When implementing this annotation plugin on a different platform, a different selector is probably required. Another design choice is the *context length*, which determines the length of the highlight's prefix and suffix. In other words, all highlights are preceded and succeeded by some extra text to put the highlight in the correct context. This is demonstrated in Figure 6.4. This context is also required for the `TextQuoteSelector` (see section 4.1.1), which is then used to highlight the correct words when the annotation is loaded. Dokieli uses a context length of 32 characters, which seems to work well.

## 6.3 Comparing the access techniques in the context of annotations

With all the knowledge that was obtained in this thesis, it is time to determine whether the two access methods are well suited for annotations.

### 6.3.1 Difficulty of implementation

First, implementation should be easy. File-based access is able to use the file system which is available on all modern operating systems, while query-based servers have to set up a SPARQL server to store data. File-based access makes use of simple retrieval of files, while query-based access techniques require custom queries for each type of request. Both of these reasons are arguments for using file-based access when implementation difficulty is important. It is especially relevant when a user's datapod should also be used for other decentralized applications. File-based implementations such as Solid are able to deal with all kinds of RDF documents due to its simplicity of only manipulating at file-level granularity. Meanwhile, in order to implement query-based access to new types of data, one or multiple new access endpoints have to be programmed since each request requires a new custom SPARQL query. This confirms hypothesis  $H_3$ .

### 6.3.2 Retrieval performance

Another important aspect of a good access technique in the context of annotations is retrieval performance. When loading annotations, users prefer to have them show up as fast as possible. By using a query to retrieve all annotations with a specific source at once, it is possible to load 100 annotations in about 200 milliseconds, provided that they are stored on the same datapod. This is a negligible delay. The user perceives this as if it was loaded instantly. By contrast, collecting this number of annotations takes over 7 seconds when a file-based access technique is used. This is a huge difference and is definitely noticeable by the users. Query-based access has the same delays when the annotations are stored on many different datapods since these separate graphs cannot be queried together. Clearly, file-based access has user experience flaws when a large amount of annotations have to be collected. It is possible to reduce this delay by implementing application-specific caching mechanisms, such as caching all annotations for a web page in the same file and returning this file upon a request to the `AnnotationService`. This is however not something Solid supports by default. In the case of a realistic amount of annotations, say between 1 and 10 annotations per page, and a separate `AnnotationService` is used for every page, the delay of about 1 second is still noticeable but is less of a problem. Another potential solution to this problem is to order the list of annotations that is provided by the `AnnotationService` so that annotations that appear at the top of the page are loaded before the ones at the bottom. In conclusion, hypothesis  $H_1$  can be confirmed, although file-based access can negate performance differences in some scenarios.

### 6.3.3 State of research

At the moment of writing, file-based access technologies have been researched thoroughly and have ready-to-use implementations such as Solid's server and client. Field-tested authentication (using WebID) and ACL (using WAC) technologies are included in these implementations. SPARQL endpoints have existed for a long time, but using them as an interface to a datapod is rather new. To the best of our knowledge, query-based access implementations for datapods are not yet being developed, so one will have a harder time setting up a query-based back-end.

### 6.3.4 Implementing update detection

Detecting updates is easier when using file-based access, since you can simply attach a listener to the directory you would like to watch. This can be done independent from what file-based back-end implementation you use. Doing the same on top of a SPARQL endpoint requires inspecting packets upon retrieval, which needs to be implemented in the query-based back-end (e.g. by the use of an additional Express layer). Detecting updates is something that is used in the context of annotations, so the simplicity of directory watchers is an argument for file-based access.

Just like how Linked Data Notification watchers for file-based notifications can easily be implemented using standard directory watcher APIs, versioning can be made easy by using versioning systems such as Git, and raw graph data can be edited with your standard text editor, or any file-based API. These are all examples of technologies that already exist and can be reused in the context of file-based access, but not when using query-based access, confirming hypothesis  $H_2$ .

### 6.3.5 Weighing the pros and cons

So which one should be used? Query-based access outperforms file-based access by a large margin. However, these delays may become less and less noticeable in certain scenarios. First, if all annotations are cached within the `AnnotationService`'s datapod, query-based access seems to be faster. However, file-based access endpoints can be adapted to return all annotations on a list with just a single request. This requires modifying server-side code, and goes against the rules of the LDP standard. The Solid specification includes this as an extension to LDP. It allows wildcards (\*) to be placed in a request, upon which a Solid-compliant server will return all resources that match the pattern. Officially, this is called "globbing"<sup>1</sup>. An even better solution is to cache all annotations for a webpage in the same file, and this file is then returned when requesting annotations from the `AnnotationService`. This can be done in a single request and is fully LDP-compliant. A second scenario is the following. When a small number of annotations per user are stored, and the annotations' ACL states that they are not publicly viewable by all users, and the ACL itself is not public either, the annotations cannot be cached! In this case, the performance is determined by communication delays, and thus the number of users. A last scenario occurs when each user stores a large amount of annotations with the same ACL rules as previous scenario. In this case, query-based access easily outperforms file-based access. However, note that the users' datapods can still implement good caching mechanisms, in which case the performance differences will again become negligible.

Clearly, performance is initially a reason to choose for query-based access, but file-based access can compete by using an efficient caching design. Meanwhile, file-based access techniques are further developed, are easily reusable in other contexts, and are very easy to set up compared to query-based access techniques. Therefore, it is recommended to use file-based access techniques when implementing a decentralized annotation system.

## 6.4 Decentralized social applications in general

During this dissertation, most experiments and analyses used decentralized annotations to compare file-based and query-based access techniques. It was assumed that decentralized annotations are representative for all decentralized social applications, as stated by hypothesis  $H_4$ . Now that most differences have been examined, it becomes clear that annotations were indeed representative. For instance, assume a social network application where images are shared between people. Conceptually, this is still similar to annotations: adding an annotation to a specific website becomes posting an image and a description to the application's website. The image and its metadata are still stored on your own server, while the website keeps track of all URIs. It can still use advanced caching mechanisms, but the usefulness depends entirely on the application. Caching all annotations so that many of them are loaded at the same time, is for instance much more useful than caching all photos and their description, as applications like Instagram only display one or two photos at a time, making delays less noticeable. Now, imagine a decentralized version of Reddit where tens of thousands of comments are added to a single post. Loading all comments one by one is still unfeasible, and caching mechanisms should be used to improve the user experience. Restrictive ACL is however still a bottleneck, as this means only the URI can be cached, and not the content of the graph(s).

---

<sup>1</sup><https://github.com/solid/solid-spec/blob/master/api-rest.md#globbing-inlining-on-get>



## Chapter 7

# Conclusions

### 7.1 SWOT analysis for decentralization using Linked Data

Now that decentralized annotations, and social network applications in general, have been excessively analyzed, a SWOT analysis is presented to demonstrate what exactly decentralization by means of Linked Data can offer us, and what it cannot.

#### 7.1.1 Strengths

The biggest motivation for decentralization of social network applications is data ownership. People choose where they store their data, who can access it, and who can purchase their data. Data is readable by all applications, which stops people from being bound to a single application because they do not want to lose their photos, tweets, etc. In fact, they only have to create their data once, even when multiple applications use it. Advanced ACL opens up many privacy opportunities, and there is usually no single point of failure.

#### 7.1.2 Weaknesses

Collection of data across multiple sources can still be quite slow, which may drastically worsen the user experience while increasing network load. All users need to store their data in a user pod of their own choice, which will probably cost money. Users also need to be more tech-savvy in order to maintain their data, although good interfaces will only make this easier as time passes by. Finally, file-based storage is intuitive to the user, but is also considerably slower compared to using specialized database technologies.

#### 7.1.3 Opportunities

Linked Data incentives like Solid makes implementing decentralized applications relatively easy, although it is still very much a work in progress. When such a decentralized application breaks through, and many people decide to create their data pod, the number of other decentralized applications will increase rapidly thanks to the open-source community.

#### 7.1.4 Threats

People will likely remain using centralized applications, as they come free and people will have trouble letting loose of their current data. Setting up a datapod, and working with tools like Solid, is still a barrier that cannot easily be overcome. People also do not yet care enough about their privacy, although the

recent Facebook allegations made people more aware of the issue<sup>1</sup>. People are free to hire datapods wherever they want, however there is a significant chance that free datapod providers will surface who compensate their costs by selling their users' data. Finally, for an application creator, there are less financial incentives to create a decentralized app over a centralized one.

## 7.2 Relevance

The Cambridge Analytica scandal has recently sparked conversation about what social networks are allowed to do with your data. This company used the data of about 87 million Facebook profiles to manipulate elections<sup>2</sup>. Although Facebook did not explicitly sell this data, it allowed an app to harvest this data by not offering enough protection for the users. The app requests the permission of the users to acquire their data, but at that time, this included all data of their friends as well. The ability to easily access such a large amount of data, is inherent to centralization. When a decentralized datapod with `WebAccessControl` would have been used, each user would have had to give the app permission to access their data. This would have reduced the number of victims to about 270,000 Facebook users. While currently many social applications offer the ability to only share a specific part of your data, you are still dependent on the platform's integrity to not share anything else. Also, the granularity of these access rights is usually fixed. For instance, you can only give an application access to all of your photos, not a specific subset of photos. These problems are solved by using your own datapod, granted you use a sufficient ACL system. Note that using a SPARQL endpoint with query-based access is probably the best choice, since it allows the most control over your data. File-based access can for instance not filter out certain fields, as it operates at file-level granularity.

With the arrival of the General Data Protection Regulation (GDPR)<sup>3</sup>, which was enforced on 25 May 2018, some of the privacy issues concerning traditional centralized social networks have been resolved. The legislation applies to all companies that process EU citizens' information. They risk a large fine when they do not obey to the following rules. First, the users have the right to know exactly what is done with their data, and they have to agree with it. Therefore, the terms of use have to be readable for people without a law degree. Nevertheless, often people still cannot continue to use the social platform without accepting their (more readable) terms of use, and the platforms may still process your data for various reasons. People also have the right to access their data (i.e. data transparency) or remove all their data from the database. In the context of this thesis, data portability is the most important right given to the users. This allows people to request data in a machine readable format which can be used in other contexts, such as a competing social media website. However, only the data that was provided by the data subject can be retrieved, not the data that was additionally generated by the platform. Also, there is no incentive for the companies to represent this data as something that is easily reusable by a competitor.

## 7.3 Conclusion

During this thesis, both file-based and query-based access was explored by using a decentralized annotations set-up. We proved that annotations are easily decentralizable. While query-based access outperforms file-based access when collecting many annotations, this difference can sometimes be neglected through a good caching design. File-based access techniques are able to reuse existing file-based technologies such as Git and directory watchers. It also makes manually manipulating data easier, and implementing a directory structure is straightforward since it is inherent to a file-system. File-based access relies on a simple API (e.g. the LDP specification) which allows easy reusability in other contexts. Meanwhile, the back-end to handle query-based access currently requires a new specific implementation

---

<sup>1</sup><https://www.cmo.com.au/article/640531/facebook-ad-revenues-up-year-on-year-despite-data-privacy-scandals>

<sup>2</sup><https://www.wired.com/story/facebook-exposed-87-million-users-to-cambridge-analytica/>

<sup>3</sup><https://gdpr-info.eu/>

for each endpoint. As there is already a file-based LDP implementation, Solid, it is easier for a developer to set up a decentralized application. To the best of our knowledge, there are no existing datapod technologies that allow query-based access. While there are many advantages to using file-based access, there is one feature that is exclusive to query-based access: performing complex intersections in a single query. This may, depending on the application, be a good reason to use query-based access over file-based access. In fact, through its better performance, query-based access can still compete with file-based access, especially when it is supported by easy-to-use user interfaces that abstract away the differences for end users.

Some of the topics that were handled during this thesis can still require some additional research. First, while advanced caching may solve some performance issues that occur with file-based access, it lacks an actual implementation. Furthermore, when a datapod's ACL is not publicly visible, it is not always possible to cache the datapod's contents. This is a problem for both file-based and query-based access techniques. Future work can explore ways to increase performance without having to share the datapod's ACL. Also, the advanced ACL for query-based access to annotations currently has the flaw of being application dependent. We proposed to use SHACL to solve this problem, but this was not implemented. Future research may explore other options, or implement SHACL in order to prove that it solved this problem. Finally, one of the biggest flaws of query-based access is the fact that each new endpoint requires some new endpoint-specific code. There may be ways to simplify this task, potentially even automating it.

This thesis proves that decentralization of social applications is perfectly feasible. In the future, a utopian internet in which everyone has their own datapod and where applications are reduced to interfaces of our data [84] may become reality. Nevertheless, some bridges still need to be crossed. Potentially the biggest reason why people will hesitate to make the switch, is because decentralization does not come for free. Applications may charge the users to use their interface, and hosting your own datapod consumes power. As not everyone is willing or is competent to set up their own datapod, there will be a need for good service providers who will host your data for a small fee. However, once most people grasp the potential of a decentralized social ecosystem, and decide to switch, the application developers will follow. In the end, the competition for the best interfaces – now based on service quality instead of data ownership – will result in something that is essentially not very different from what we have now, but without many of its disadvantages.

# Bibliography

- [1] Adida, B., Birbeck, M., McCarron, S., and Pemberton, S. (2008). Rdfa in xhtml: Syntax and processing. *Recommendation, W3C*, 7.
- [2] Anderson, C., Wolff, M., et al. (2010). The web is dead. long live the internet. *Wired Magazine*, 18.
- [3] Antonopoulos, A. M. (2014). *Mastering Bitcoin: unlocking digital cryptocurrencies*. " O'Reilly Media, Inc."
- [4] Arwe, J., Malhotra, A., and Speicher, S. (2015). Linked data platform 1.0. W3C recommendation, W3C. <http://www.w3.org/TR/2015/REC-ldp-20150226/>.
- [5] Aspan, M. (2008). How sticky is membership on facebook? just try breaking free. *The New York Times*, 11:2008.
- [6] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. (2007). Dbpedia: A nucleus for a web of open data. *The semantic web*, pages 722–735.
- [7] Auer, S. and Lehmann, J. (2007). What have innsbruck and leipzig in common? extracting semantics from wiki content. In *European Semantic Web Conference*, pages 503–517. Springer.
- [8] Backstrom, L., Dwork, C., and Kleinberg, J. (2007). Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography. In *Proceedings of the 16th international conference on World Wide Web*, pages 181–190. ACM.
- [9] Beckett, D., Berners-Lee, T., and Prud'hommeaux, E. (2008). Turtle-terse rdf triple language. *W3C Team Submission*, 14(7).
- [10] Beckett, D. and McBride, B. (2004). Rdf/xml syntax specification (revised). *W3C recommendation*, 10(2.3).
- [11] Berners-Lee, T. (2005). Notation 3 logic. <https://www.w3.org/DesignIssues/Notation3.html>.
- [12] Berners-Lee, T. (2009). Socially aware cloud storage. *Notes on web design*, Aug, 17. <https://www.w3.org/DesignIssues/CloudStorage.html> (visited: 27-11-2017).
- [13] Berners-Lee, T. (2016). Linked data design issues. <https://www.w3.org/DesignIssues/LinkedData.html>.
- [14] Berners-Lee, T., Chen, Y., Chilton, L., Connolly, D., Dhanaraj, R., Hollenbach, J., Lerer, A., and Sheets, D. (2006). Tabulator: Exploring and analyzing linked data on the semantic web. In *Proceedings of the 3rd international semantic web user interaction workshop*, volume 2006, page 159. Athens, Georgia.
- [15] Berners-Lee, T., Hendler, J., Lassila, O., et al. (2001). The semantic web. *Scientific american*, 284(5):28–37.

- [16] Bielenberg, A., Helm, L., Gentilucci, A., Stefanescu, D., and Zhang, H. (2012). The growth of diaspora—a decentralized online social network in the wild. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*, pages 13–18. IEEE.
- [17] Bizer, C., Cyganiak, R., and Gauß, T. (2007). The rdf book mashup: from web apis to a web of data. In *Proceedings*, volume 1.
- [18] Bizer, C., Heath, T., and Berners-Lee, T. (2009a). Linked data—the story so far. *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227.
- [19] Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., and Hellmann, S. (2009b). Dbpedia—a crystallization point for the web of data. *Web Semantics: science, services and agents on the world wide web*, 7(3):154–165.
- [20] Bojars, U. and Breslin, J. G. (2007). Sioc core ontology specification. *Member Submission, W3C*. <https://www.w3.org/Submission/sioc-spec/> (visited: 27-11-2017).
- [21] Bonneau, J., Anderson, J., Anderson, R., and Stajano, F. (2009a). Eight friends are enough: social graph approximation via public listings. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, pages 13–18. ACM.
- [22] Bonneau, J., Anderson, J., and Danezis, G. (2009b). Prying data out of a social network. In *Social Network Analysis and Mining, 2009. ASONAM'09. International Conference on Advances in*, pages 249–254. IEEE.
- [23] Brickley, D. and Miller, L. (2014). Foaf vocabulary specification 0.99. <http://xmlns.com/foaf/spec/> (visited: 27-11-2017).
- [24] Buil-Aranda, C., Hogan, A., Umbrich, J., and Vandenbussche, P.-Y. (2013). Sparql web-querying infrastructure: Ready for action? In *International Semantic Web Conference*, pages 277–293. Springer.
- [25] Campbell, S. (2010). How do social networks make money. <http://www.makeuseof.com/tag/how-do-social-networks-make-money-case-wondering/>.
- [26] Capadisli, S., Guy, A., Lange, C., Auer, S., Sambra, A., and Berners-Lee, T. (2017a). Linked data notifications: a resource-centric communication protocol. In *European Semantic Web Conference*, pages 537–553. Springer.
- [27] Capadisli, S., Guy, A., Verborgh, R., Lange, C., Auer, S., and Berners-Lee, T. (2017b). Decentralised authoring, annotations and notifications for a read-write web with dokieli. In *International Conference on Web Engineering*, pages 469–481. Springer.
- [28] Carothers, G. and Seabourne, A. (2014). Rdf 1.1 n-triples. *W3C Recommendation*, 25(2).
- [29] Casteleyn, J., Mottart, A., and Rutten, K. (2009). How to use data from facebook in your market research. *International Journal of Market Research*, 51(4):439–447.
- [30] Consortium, W. W. W. et al. (2014). Json-ld 1.0: a json-based serialization for linked data.
- [31] Cutillo, L. A., Molva, R., and Strufe, T. (2009a). Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine*, 47(12).
- [32] Cutillo, L. A., Molva, R., and Strufe, T. (2009b). Safebook: Feasibility of transitive cooperation for privacy on a decentralized social network. In *World of Wireless, Mobile and Multimedia Networks & Workshops, 2009. WoWMoM 2009. IEEE International Symposium on a*, pages 1–6. IEEE.
- [33] Dumon, P. (2008). Facebook groter dan myspace. *De Morgen*, (24 June):25.

- [34] Elmagarmid, A. K., Ipeirotis, P. G., and Verykios, V. S. (2007). Duplicate record detection: A survey. *IEEE Transactions on knowledge and data engineering*, 19(1):1–16.
- [35] Euzenat, J., Shvaiko, P., et al. (2007). *Ontology matching*, volume 18. Springer.
- [36] Fitzpatrick, B. and Recordon, D. (2007). Thoughts on the social graph. *bradfitz.com*, 17.
- [37] Genestoux, J., Fitzpatrick, B., Slatkin, B., and Atkins, M. (2017). Websub. *Editor's draft, W3C*. <https://w3c.github.io/websub/#title> (visited: 27-11-2017).
- [38] Guy, A. (2017). *The Presentation of Self on a Decentralised Web*. PhD thesis, University of Edinburgh. <https://rhiaro.github.io/thesis/> (accessed: 16-12-2017).
- [39] Hajli, N. and Lin, X. (2016). Exploring the security of information sharing on social networking sites: The role of perceived control of information. *Journal of Business Ethics*, 133(1):111–123.
- [40] Harris, R. (2009). Social media ecosystem mapped as a wiring diagram.
- [41] Hartig, O., Bizer, C., and Freytag, J.-C. (2009). Executing sparql queries over the web of linked data. *The Semantic Web-ISWC 2009*, pages 293–309.
- [42] Hogben, G. (2007). Security issues and recommendations for online social networks. *ENISA position paper*, 1:1–36.
- [43] Huynh, D., Mazzocchi, S., and Karger, D. (2005). Piggy bank: Experience the semantic web inside your web browser. *The Semantic Web-ISWC 2005*, pages 413–430.
- [44] Initiative, D. C. M. et al. (2004). Dcmi metadata terms. <http://dublincore.org/documents/dcmi-terms/>.
- [45] Jacobs, I. and Walsh, N. (2004). Architecture of the world wide web.
- [46] Khare, R. (2006). Definition of decentralization. <http://isr.uci.edu/projects/pace/decentralization.html> (accessed: 11-12-2017).
- [47] Klerks, P. (2001). The network paradigm applied to criminal organizations: Theoretical nitpicking or a relevant doctrine for investigators? recent developments in the netherlands. *Connections*, 24(3):53–65.
- [48] Klyne, G. and Carroll, J. J. (2004). Resource description framework (rdf): Concepts and abstract syntax. w3c recommendation, 2004. *World Wide Web Consortium*, <http://w3c.org/TR/rdf-concepts>.
- [49] Kontokostas, D. and Knublauch, H. (2017). Shapes constraint language (SHACL). W3C recommendation, W3C. <https://www.w3.org/TR/2017/REC-shacl-20170720/>.
- [50] Lanthaler, M. and Gütl, C. (2012). On using json-ld to create evolvable restful services. In *Proceedings of the Third International Workshop on RESTful Design*, pages 25–32. ACM.
- [51] Larimer, D. (2014). Delegated proof-of-stake (dpos). *Bitshare whitepaper*.
- [52] Larimer, D., Scott, N., Zavgorodnev, V., Johnson, B., Calfee, J., and Vandeberg, M. (2016). Steem an incentivized, blockchain-based social media platform. *March*. *Self-published*.
- [53] Lee, J., Lee, M., and Choi, I. H. (2012). Social network games uncovered: Motivations and their attitudinal and behavioral outcomes. *Cyberpsychology, Behavior, and Social Networking*, 15(12):643–648.
- [54] Liu, Y., Gummadi, K. P., Krishnamurthy, B., and Mislove, A. (2011). Analyzing facebook privacy settings: user expectations vs. reality. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 61–70. ACM.

- [55] Mansour, E., Sambra, A. V., Hawke, S., Zereba, M., Capadisli, S., Ghanem, A., Abounaga, A., and Berners-Lee, T. (2016). A demonstration of the solid platform for social web applications. In *Proceedings of the 25th International Conference Companion on World Wide Web*, pages 223–226. International World Wide Web Conferences Steering Committee.
- [56] Martinez, A. G. (2017). Facebook’s not listening through your phone. it doesn’t have to. <https://www.wired.com/story/facebooks-listening-smartphone-microphone/>.
- [57] Mihindikulasooriya, N., Gutiérrez, M. E., and Burleson, C. (2014). Linked data platform best practices and guidelines. W3C note, W3C. <http://www.w3.org/TR/2014/NOTE-ldp-bp-20140828/>.
- [58] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.
- [59] Narayanan, A. and Shmatikov, V. (2006). How to break anonymity of the netflix prize dataset. *arXiv preprint cs/0610105*.
- [60] Pariser, E. (2011). *The filter bubble: What the Internet is hiding from you*. Penguin UK.
- [61] Poller, A. (2008). Privatsphärenschutz in soziale-netzwerke-plattformen. *Fraunhofer SIT Survey*, [www.sit.fraunhofer.de](http://www.sit.fraunhofer.de).
- [62] Prud, E., Seaborne, A., et al. (2006). Sparql query language for rdf.
- [63] Raimond, Y., Sutton, C., and Sandler, M. B. (2008). Automatic interlinking of music datasets on the semantic web. *LDOW*, 369.
- [64] Recordon, D. and Reed, D. (2006). Openid 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management*, pages 11–16. ACM.
- [65] Roberts, J. J. (2016). Twitter, surveillance, and the challenges of selling social data. <http://www.fortune.com/2016/12/09/twitter-social-media-data-surveillance/>.
- [66] Rogers, D. (2013). The enduring myth of the sparql endpoint. <https://daverog.wordpress.com/2013/06/04/the-enduring-myth-of-the-sparql-endpoint/>.
- [67] Ronallo, J. (2012). Html5 microdata and schema.org. *Code4Lib Journal*, 16.
- [68] Sambra, A., Guy, A., Capadisli, S., and Greco, N. (2016a). Building decentralized applications for the social web. In *Proceedings of the 25th International Conference Companion on World Wide Web*, pages 1033–1034. International World Wide Web Conferences Steering Committee.
- [69] Sambra, A., Story, H., and Berners-Lee, T. (2017). Webid 1.0: Web identity and discovery. *Editor’s draft, W3C*. <https://dvcs.w3.org/hg/WebID/raw-file/tip/spec/identity-respec.html> (visited: 27-11-2017).
- [70] Sambra, A. V., Mansour, E., Hawke, S., Zereba, M., Greco, N., Ghanem, A., Zagidulin, D., Abounaga, A., and Berners-Lee, T. (2016b). Solid: A platform for decentralized social applications based on linked data. [http://emansour.com/research/meccano/solid\\_protocols.pdf](http://emansour.com/research/meccano/solid_protocols.pdf) (accessed: 28-11-2017).
- [71] Sanderson, R. (2017). Web annotation protocol. W3C recommendation, W3C. <https://www.w3.org/TR/2017/REC-annotation-protocol-20170223/>.
- [72] Sanderson, R., Young, B., and Ciccarese, P. (2017). Web annotation data model. W3C recommendation, W3C. <https://www.w3.org/TR/2017/REC-annotation-model-20170223/>.
- [73] Sauermaun, L., Cyganiak, R., and Völkel, M. (2007). Cool uris for the semantic web.

- [74] Scott, J. (2017). *Social network analysis*. Sage.
- [75] Seong, S.-W., Seo, J., Nasielski, M., Sengupta, D., Hangal, S., Teh, S. K., Chu, R., Dodson, B., and Lam, M. S. (2010). Prpl: a decentralized social networking infrastructure. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, page 8. ACM.
- [76] Shirky, C. (2011). The political power of social media: Technology, the public sphere, and political change. *Foreign affairs*, pages 28–41.
- [77] Sporny, M. (2011). The false choice of schema.org. *The Beautiful, Tormented Machine*. <http://manu.sporny.org/2011/false-choice/> (accessed: 12-12-2017).
- [78] Sporny, M. (2015). RDFa lite 1.1 - second edition. W3C recommendation, W3C. <http://www.w3.org/TR/2015/REC-rdfa-lite-20150317/>.
- [79] Taelman, R., Vander Sande, M., and Verborgh, R. (2018). Ostrich: Versioned random-access triple store. In *Companion of the The Web Conference 2018 on The Web Conference 2018*, pages 127–130. International World Wide Web Conferences Steering Committee.
- [80] Tramp, S., Frischmuth, P., Ermilov, T., and Auer, S. (2010). Weaving a social data web with semantic pingback. *Knowledge Engineering and Management by the Masses*, pages 135–149.
- [81] Tramp, S., Frischmuth, P., Ermilov, T., Shekarpour, S., and Auer, S. (2014). An architecture of a distributed semantic social network. *Semantic Web*, 5(1):77–95.
- [82] Van Hooland, S. and Verborgh, R. (2014). *Linked Data for Libraries, Archives and Museums: How to clean, link and publish your metadata*. Facet publishing.
- [83] Van Kleek, M., Smith, D. A., Shadbolt, N., et al. (2012). A decentralized architecture for consolidating personal information ecosystems: The webbox.
- [84] Verborgh, R. (2017). Paradigm shifts for the decentralized web. <https://ruben.verborgh.org/blog/2017/12/20/paradigm-shifts-for-the-decentralized-web/>.
- [85] Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., and Colpaert, P. (2016). Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. *Journal of Web Semantics*, 37–38:184–206.
- [86] Winkler, W. E. (2006). Overview of record linkage and current research directions. In *Bureau of the Census*. Citeseer.
- [87] Wolf, M. (2009). Will social networks on the web ever make money? [https://www.forbes.com/2009/02/19/facebook-myspace-twitter-linkedin-opinions-contributors\\_zuckerberg\\_internet.html#7d7641291774](https://www.forbes.com/2009/02/19/facebook-myspace-twitter-linkedin-opinions-contributors_zuckerberg_internet.html#7d7641291774).
- [88] Wood, D., Zaidman, M., Ruth, L., and Hausenblas, M. (2014). *Linked Data*. Manning Publications Co., Greenwich, CT, USA, 1st edition.
- [89] Yeung, C.-m. A., Liccardi, I., Lu, K., Seneviratne, O., and Berners-Lee, T. (2009). Decentralization: The future of online social networking. In *W3C Workshop on the Future of Social Networking Position Papers*, volume 2, pages 2–7.
- [90] Young, B., Ciccarese, P., and Sanderson, R. (2017). Web annotation vocabulary. W3C recommendation, W3C. <https://www.w3.org/TR/2017/REC-annotation-vocab-20170223/>.
- [91] Zyskind, G., Nathan, O., et al. (2015). Decentralizing privacy: Using blockchain to protect personal data. In *Security and Privacy Workshops (SPW), 2015 IEEE*, pages 180–184. IEEE.



## Appendix A

# Using advanced ACL for annotations

```
PREFIX : <http://example.org/#>
PREFIX acl: <http://www.w3.org/ns/auth/acl#>
PREFIX new-acl: <https://vanhoucke.me/ontology/new-acl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX oa: <http://www.w3.org/ns/oa#>
PREFIX dc: <http://purl.org/dc/terms/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

<https://lukas.vanhoucke.me/public/annotations/annotation-1>
  a oa:Annotation;
  dc:created "Thu, 29 Mar 2018 08:55:56 GMT";
  dc:creator <https://lukas.vanhoucke.me/profile/card#me>;
  rdfs:label "Lukas Vanhoucke created an annotation"@en;
  oa:hasBody <https://lukas.vanhoucke.me/public/annotations/annotation-1#body>;
  oa:hasTarget <https://lukas.vanhoucke.me/public/annotations/annotation-1#target>;
  oa:motivatedBy oa:commenting.

<https://lukas.vanhoucke.me/public/annotations/annotation-1#body>
  a oa:TextualBody;
  rdf:value
    "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut"@en.

<https://lukas.vanhoucke.me/public/annotations/annotation-1#fragment-selector>
  a oa:FragmentSelector;
  oa:refinedBy <https://lukas.vanhoucke.me/public/annotations/annotation-1#text-quote-selector>.

<https://lukas.vanhoucke.me/public/annotations/annotation-1#text-quote-selector>
  a oa:TextQuoteSelector;
  oa:exact "text to highlight"@en;
  oa:prefix "text before highlighted text"@en;
  oa:suffix "text after highlighted text"@en.

<https://lukas.vanhoucke.me/public/annotations/annotation-1#target>
  a oa:SpecificResource;
  oa:hasSelector <https://lukas.vanhoucke.me/public/annotations/annotation-1#fragment-selector>;
  oa:hasSource <https://www.somesite.com/index#introduction>.
```

**Listing A.1:** Graph content in the second scenario (annotations): the first annotation.

```
<https://lukas.vanhoucke.me/public/annotations/annotation-2>
  a oa:Annotation;
  dc:created "Thu, 30 Mar 2018 11:10:42 GMT";
  dc:creator <https://lukas.vanhoucke.me/profile/card#me>;
  rdf:label "Lukas Vanhoucke created an annotation"@en;
  oa:hasBody <https://lukas.vanhoucke.me/public/annotations/annotation-2#body>;
  oa:hasTarget <https://lukas.vanhoucke.me/public/annotations/annotation-2#target>;
  oa:motivatedBy oa:commenting.

<https://lukas.vanhoucke.me/public/annotations/annotation-2#body>
  a oa:TextualBody;
  rdf:value "consectetur adipiscing elit, sed do eiusmod"@en.

<https://lukas.vanhoucke.me/public/annotations/annotation-2#fragment-selector>
  a oa:FragmentSelector;
  oa:refinedBy <https://lukas.vanhoucke.me/public/annotations/annotation-2#text-quote-selector>.

<https://lukas.vanhoucke.me/public/annotations/annotation-2#text-quote-selector>
  a oa:TextQuoteSelector;
  oa:exact "text to highlight"@en;
  oa:prefix "text before highlighted text"@en;
  oa:suffix "text after highlighted text"@en.

<https://lukas.vanhoucke.me/public/annotations/annotation-2#target>
  a oa:SpecificResource;
  oa:hasSelector <https://lukas.vanhoucke.me/public/annotations/annotation-2#fragment-selector>;
  oa:hasSource <https://www.somesite.com/index#admins-only>.
```

**Listing A.2:** Graph content in the second scenario (annotations): the second annotation.

```
<> new-acl:possibleField
  [new-acl:predicate dc:created; new-acl:label "created"],
  [new-acl:predicate dc:creator; new-acl:label "creator"],
  [new-acl:predicate rdf:value; new-acl:label "text"],
  [new-acl:predicate oa:exact; new-acl:label "exact"],
  [new-acl:predicate oa:prefix; new-acl:label "prefix"],
  [new-acl:predicate oa:suffix; new-acl:label "suffix"],
  [new-acl:predicate oa:hasSource; new-acl:label "source"] .

<#owner> a acl:Authorization;
  acl:agent <https://lukas.vanhoucke.me/profile/card#me>;
  new-acl:allowsField new-acl:allFields .

<> a acl:Authorization; # Blank nodes for new users.
  acl:agent <https://random.person.me/profile/card#me>;

  # Cannot see the creator
  new-acl:allowsField new-acl:allFields ;

  # Cannot see any annotations for admins-only
  new-acl:hasRestriction [
    new-acl:restrictSubjectField '?target' ;
    new-acl:restrictPredicate oa:hasSource ;
    new-acl:restrictObject <https://www.somesite.com/index#admins-only> ;
  ] .

<#default> a acl:Authorization;
  acl:agentClass foaf:Agent;
  # Cannot see the creator
  new-acl:allowsField dc:created, rdf:value, oa:exact, oa:prefix, oa:suffix, oa:hasSource ;

  # Cannot see any annotations for admins-only
  new-acl:hasRestriction [
    new-acl:restrictSubjectField '?target' ;
    new-acl:restrictPredicate oa:hasSource ;
    new-acl:restrictObject <https://www.somesite.com/index#admins-only> ;
  ] .
```

**Listing A.3:** Graph content in the second scenario (annotations): corresponding ACL.

```
SELECT ?source ?suffix ?prefix ?exact ?text ?created
WHERE {
  ?s a oa:Annotation .
  ?s oa:hasTarget ?target .
  ?target oa:hasSelector ?selector .
  ?selector oa:refinedBy ?quoteselector .
  ?s oa:hasBody ?body .
  OPTIONAL { ?target oa:hasSource ?source }
  OPTIONAL { ?quoteselector oa:suffix ?suffix }
  OPTIONAL { ?quoteselector oa:prefix ?prefix }
  OPTIONAL { ?quoteselector oa:exact ?exact }
  OPTIONAL { ?body rdf:value ?text }
  OPTIONAL { ?s dc:created ?created }
  FILTER NOT EXISTS {
    ?target oa:hasSource <https://www.somesite.com/index#admins-only>
  }
}
```

**Listing A.4:** The query for default users that was generated by the server. Prefixes are hidden for conciseness.

## Appendix B

# JavaScript code for routing and handling requests

```
1 var sparqlAnnotationsController = require('../controllers/sparqlAnnotations.server.controller');
2
3 module.exports = function (app) {
4   app.route('/annotations').post(sparqlAnnotationsController.storeAnnotation)
5     .get(sparqlAnnotationsController.getAllAnnotations);
6   app.route('/annotations/:user').get(sparqlAnnotationsController.getUserAnnotations);
7 };
```

**Listing B.1:** Demonstrational Express code which routes POST requests to `/sparql` to a function that stores an annotation, routes GET requests to `/sparql` to a function that returns all stored annotations, and routes GET requests to `/sparql/<username>` to a function that retrieves all annotations that were posted by a specific user.

```
1 var rp = require('request-promise');
2 const sparqlEndpoint = 'http://localhost:3000/dataset/';
3 const showAllAnnotationsQuery = '
4 prefix oa: <http://www.w3.org/ns/oa#>
5 select ?annotation ?source where {
6   ?annotation a oa:Annotation .
7   ?annotation oa:hasTarget ?target .
8   ?target oa:hasSource ?source .
9 }
10 '
11 exports.getAllAnnotations = function (req, res) {
12   let query = showAllAnnotationsQuery;
13   query = encodeURIComponent(query);
14   let queryUrl = sparqlEndpoint + 'sparql?query=' + query;
15   rp(queryUrl)
16     .then(function (body) {
17       res.send(body);
18     }).catch(function (err) {
19       res.status(400).send({ message: err });
20     });
21 };
```

**Listing B.2:** Demonstrational server code which implements the `getAllAnnotations` function that is used to retrieve all annotations.



